Project Number	AC026
Project Title	Secure Electronic MarketPlace for Europe SEMPER
Deliverable Security Class	Public
CEC Deliverable Number	AC026/SMP/CT2/DS/P/010/b1
Contractual Date of Delivery	September 31, 1998
Actual date of Delivery	March 15, 1999
Title of Deliverable	Deliverable D10 Advanced Services, Architecture and Design
Contributing Workpackages	WP2.1, WP2.3, WP2.4, WP2.5
Nature of the Deliverable	Specification
Author(s)	N. Asokan (IBM), B. Baum-Waidner (R3), T. Beiler (Univ. Saarbrucken), J. Brauckmann (Univ. Saarbrucken), L. Fritsch (Univ. Saarbrucken), M. Gratziani (Cryptomathics), M. Mazoue (SEPT), R. Michelsen (Sintef), S. Mjolsnes (Sintef), M. Nassehi (IBM), B. Patil (GMD), T. Pedersen (Cryptomathic), J. L. Abad Peiro (IBM), B. Pfitzmann (Univ. Saarbrucken), S. Prins (KPN), L. Salvail (CWI), M. Schunter (Univ. Dortmund), T. Schweinberger (IBM), M. Steiner (IBM), J. Swanenburg (KPN), M. Waidner (IBM), K. Zangeneh (GMD)
Editor(s)	Michael Steiner, IBM Research Zurich
Keywords	Architecture, Electronic Commerce, Security

SEMPER Consortium:

Advanced Services, Architecture and Design

Deliverable D10 of ACTS Project AC026

Public Specification

March 15, 1999

SEMPER is part of the European Commission's ACTS Programme (Advanced Communications Technologies and Services). Funding is provided by the partner organisations, the European Union, and the Swiss Federal Department for Education and Science.

The members of the SEMPER consortium are Commerzbank (D), Cryptomathic (DK), DigiCash (NL), EUROCOM EXPERTISE (GR), Europay International (B), FOGRA Forschungsgesellschaft Druck (D), France Telecom / CNET (F), GMD - German National Research Center for Information Technology (D), IBM La Gaude (F), IBM Zurich Research Lab (CH), INTRACOM (GR), KPN Research (NL), Otto-Versand (D), r3 security engineering (CH), SINTEF Telecom and Informatics (F), Stichting Mathematisch Centrum / CWI (NL), University of Dortmund (D) (having replaced University of Hildesheim (D)). University of Freiburg (D), University of Saarbrucken (D). Banksys (B), Banque Générale du Luxembourg (L) and Telekurs (CH) are associated with SEMPER.

For more information on SEMPER visit

<http://www.semper.org>

or contact

Gerard Lacoste	Michael Waidner
IBM La Gaude	IBM Research Division
Centre d'Etudes et Recherches	Zurich Research Laboratory
Le Plan du Bois	Säumerstrasse 4
F-06610 La Gaude, France	CH-8803 Rüschlikon, Switzerland
Phone + 33 92 11 48 07	Phone +41 1 724 8220
Fax +33 93 24 45 45	Fax +41 1 710 3608
Mail lacoste@vnet.ibm.com	Mail wmi@zurich.ibm.com

MANAGEMENT SUMMARY

This report is the final publicly available technical deliverable of *SEMPER*, "Secure Electronic Marketplace for Europe." It describes the main results on the security architecture.

The main objective of *SEMPER* is to develop, implement, trial and evaluate an open security architecture for electronic commerce over open networks (e.g., the Internet). This was done in several steps: in the first one and a half years a first set of basic services were designed and implemented. A subsequent trial of the basic services and its evaluation led to a second phase in the remainder of the project. The result of the evaluation led to a revision of the basic architecture. Some additional advanced services were added as well This revised implementation was again evaluated in a trial.

This report describes the *SEMPER* the detailed service architecture and how it was implemented. Details on the integration of this implementation in the trials can be found in Deliverable D04 ("Basic Services, Prototype and Internet Trial") and D08 ("Prototypes and Trials (Draft)"), while more information on the trials can be found in Deliverable D05 ("First Year Surveys and Evaluation"), D09 ("Evaluation of Phase II Trials") and eventually in D11 ("Surveys and Evaluation") and D16 ("Evaluation of the SME Trials").

The architecture of *SEMPER* defines a set of service layers. The lowest layer contains supporting services (archiving, cryptographic functions, communication, user interface). On top of this, a transfer layer provides services for electronic payment and secure transfer of documents. The next layer, the exchange layer, combines transfers into fair exchanges (e.g., contract signing, certified mail.) These services are used by the commerce layer to provide more complex business primitives (e.g., "send order" for filling in a pre-defined order template, signing it, and sending it to the business partner.)

The architecture is open: New service modules can be integrated via generic API's (e.g., new payment systems by adapting them to a generic payment API). New business applications (e.g., an on-line auction) can be supported via dynamically loadable scripts that use the primitives of the commerce layer.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Program Interface
CORBA	Common Object Request Broker Architecture
EDI	Electronic Data Interchange
GPSF	Generic Payment Service Framework
GUI	Graphical User Interface
IBM	International Business Machines
JCC	Java Commerce Client
JEPI	Joint Electronic Payments Initiative
MIME	Multipurpose Internet Mail Extensions
OBI	Open Buying on the Internet
OTP	Open Trading Protocol
PDU	Protocol Data Unit
PIN	Personal Identification Number
SAP	Service Access Point
SECA	Secure Electronic Commerce Agreement
SEMPER	Secure Electronic MarketPlace for eURope
SET	Secure Electronic Transactions
SPI	Service Provider Interface
TINGUIN	Trusted INteractive Graphical User INterface
U-PAI	Universal Payment Application Interface
UML	Unified Modeling Language
WWW	World Wide Web

1 Introduction

(M. Waidner / ZRL)

As in a real marketplace, the main purpose of an electronic marketplace is to bring potential **sellers** and **buyers** together. Sellers **offer** their goods and buyers **order** these goods; together this is a two-party **negotiation**, often ending with a **contract.**

This may require that buyer and seller have some relations already established, e.g., to banks or government agencies. For instance, a buyer might only want to buy from sellers that are accredited with a well-known payment system provider (so that they can use a certain payment instrument) or from sellers approved by a consumer organisation. This requires **registration** and **certification**.

After the negotiation, the seller **delivers** the goods and the buyer **pays**; this is a two-party **(fair) exchange**. Goods should here be understood very generally. E.g., the buyer might receive a ticket that subsequently enables **conditional access** to certain services (such as subscription to a journal).

In this scenario third parties, such as **notaries** and **financial institutions**, may be needed in the actual business transaction.

An architecture covering all aspects of electronic commerce must not only be able to deal with the above generic services in an ideal situation. It should also be able to cope with technical problems as well as disputes between buyer and seller (in short, **exception** and **dispute handlers** are necessary).

The architecture described in this document covers the following typical situations which include the above scenario as a special case:

- **On-line purchase** A retailer accepts electronic orders and payments, based on digital or conventional catalogues, and delivers physical or digital goods (possibly copyright protected). An interesting example of this is when a user buys a ticket that can be used by a well-defined set of users (e.g., a family) to access a certain service for some time period or a certain number of times.
- **Subscriptions**: An organisation offers services on a subscription basis, e.g., subscription to news services, database services, or journals. Another similar scenario is subscription to an insurance.
- **Electronic Mall**: An organisation offers services for several service providers, ranging from *directory services* (“index”) over *content hosting* to *payment services.*
- **Auctions**: Users participate in an auction, maybe anonymously, and with the usual fairness requirements. This situation is atypical as more than one buyer will participate
- **Secure communication**: Secure transfer of electronic documents offering confidentiality, non-repudiation services, etc. This also includes secure electronic mail.

- **Contract Signing**: Two or more parties exchange signed copies of the same statement.

There will be several different implementations of the services offered by the marketplace fulfilling different functional requirements and different security requirements, and following different business models. For instance, payment can be designed in many different ways (e.g., pre-paid or post-paid), and for each design many different actual implementations may be available. Another example is registration which may be performed by different organisations using different registration procedures and different types of certificates. The *SEMPER* architecture is open in that different designs and implementations may be integrated provided they have a suitable API.

1.1 Services

The definition of the *SEMPER* architecture is based on the above scenarios envisaged to be useful in a marketplace as well as the functionality required by the three service providers EUROCOM, FOGRA and Otto Versand.

SEMPER identified and addressed following security requirements in the architecture and the design: Authentication (identification), Integrity, Accountability, Payment, Confidentiality and Fairness.

1.1.1 Authentication

Authentication (reliable verification of identity) is frequently required by both the merchant and the customer at various stages in electronic commerce. The first example of this requirement arises at the initial point of contact between merchant and customer, the electronic shop-front. The customers entering this electronic shopping environment may need confirmation, that he/she is genuinely in contact with the company he/she has selected and not an electronic impostor.

Authentication of the customer to the merchant is dependent on the specifics of each customer merchant relationship. Otto for example does not need to know the identity of a customer until an order is placed, while FOGRA or EUROCOM may restrict access to certain information to only those customers that they have authenticated. EUROCOM, for example, frequently deals with students attending prepaid courses, who should only be granted access to their services for the duration of that particular course. FOGRA requires controlled access because they offer different conditions for members and non-members.

1.1.2 Integrity Protection

Two of the service providers will be delivering their products on-line and it is essential that the customer receives the correct service and that it be received intact. Both EUROCOM and FOGRA require confirmation that their services have been delivered to the customer.

1.1.3 Fairness

Fairness is a requirement for both the service provider and its customer: Both parties require that if they fulfil their obligations in a commercial transaction that other party will fulfil its share: if e.g., a customer of Otto orders a pair of shoes and pays online she wants to be sure to eventually receive the shoes, Otto on the other hand needs the assurance that delivered shoes are eventually paid. In Section 5.1 we'll present protocol which guarantee such a requirement when both parties don't trust each other.

1.1.4 Accountability

The requirement for certification of the identity of merchant and customer mentioned above applies in particular to three aspects of electronic transaction, i.e. offer, order and receipt. At each of these points in the electronic transaction both merchant and customer may require a secure and non-repudiable electronic exchange of information, e.g., the customer must be able to treat the merchant's offer as binding, the merchant must be able to regard the customer's order as binding and (as already mentioned) there must be proof of delivery/receipt of the "product". In the worst case these proofs should hold as evidence in case of disputes at court. To give signatures this strong functionality we require secure registration (see Chapter 5.2) and explicit treatment of liabilities (see SECA (Secure Electronic Commerce Agreement) in the upcoming deliverable D13).

1.1.5 Payment

The trials should support traditional payments by means of invoice but also include the secure transfer of payment card data. In addition, there should be the opportunity to test implementations of electronic money. All service providers in the trial require electronic invoices/receipts for their customers, regardless of whether they prefer to settle their accounts on-line or off-line.

1.1.6 Confidentiality

All the service providers perceive a potential customer requirement for confidentiality. For example customers seeking information or advice in respect of competitively sensitive areas of business, where the client might not necessarily wish outsiders to be aware of an active business interest in the particular field.

1.2 Overview

This report describes the *SEMPER* architecture and the implementation supporting the services identified above.

Chapter 2 gives an overview over the architecture. It presents the model of electronic commerce upon which the *SEMPER* architecture will be developed. Among other issues this chapter describes the actors in the electronic marketplace: seller, buyer and third parties enabling the required services. This chapter also discusses how electronic purchases can be considered a sequence of transfers and exchanges. A typical example

is when digital goods are delivered in exchange of a payment. It also takes a brief look at other projects dealing with general architectures for electronic commerce and discusses their relevance to *SEMPER*.

Chapter 3 then describes the methodological aspects of the framework. The central issue introduced here is service blocks. A service block consists of a manager and a number of modules. The manager provides the required services using one of the modules (e.g., the payment block will use existing payment systems as modules). A manager provides a generic interface such that several modules (possibly through an adapter) can be used in *SEMPER*. This is the basis of the independence of specific implementations of the modules — one of the key points of *SEMPER*.

Chapter 4 describes for each block in the architecture introduced in Chapter 2 the detailed design and the lessons learned.

Chapter 5 describes protocols of particular importance: *Fair Exchange Protocols* guarantee that the fairness requirements of transaction parties are maintained even in the presence mutual mistrust and a secure *registration protocol* is mandatory to achieve accountability. Finally we investigate also the issue of *credentials*.

Chapter 6 looks in the aspects of anonymity: Anonymous communications and credentials are considered. Additionally the impact of anonymity on the overall architecture is explored.

Chapter 7 presents as an application of the SEMPER architecture and as an outlook on what might expect us in the future a tool for peer-to-peer manual sale: the Fair Internet Trader (FIT).

Appendix A contains the collected references, Appendix B contains a glossary defining the most frequently used terms and Appendix C contains the index.

The architecture is implemented in JAVA and documented using JAVADOC. Appendix D gives guidelines for where and how to retrieve this documentation.

2 Introduction to Framework

(M.Waidner / ZRL)

2.1 Model

This chapter presents a model for electronic commerce, by introducing the key players and the properties that we want to focus on.

2.1.1 Players of the Electronic Marketplace

The services listed in the introduction involve several **players**, i.e., types of assumed real-world people or bodies participating in the marketplace. Figure 1 lists the essential ones in an electronic marketplace.

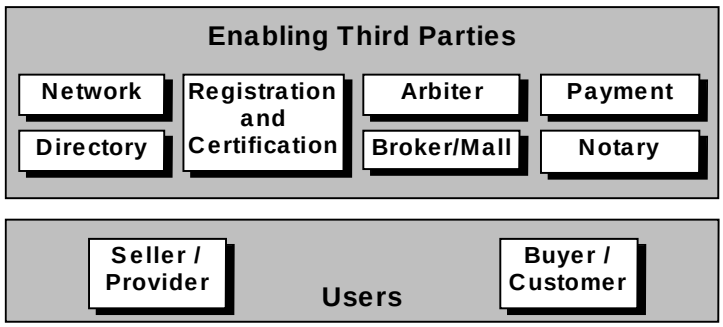

Figure 1: Players of an electronic marketplace

From the perspective of a single business session (e.g., a purchase), the participants in a marketplace are acting as

- **users** (of the marketplace) i.e., buyers and sellers, also called customers and providers, and
- **third parties**, who enable the session between the users, e.g., network and payment system providers.

Note that these are just convenient names of ***roles*** (i.e., we use "player" as a synonym of "role", the words "participant", "party", and "person" are left for less restricted usage). There is no partitioning of all real participants into such classes: A bank might act as the *provider* of a home banking service at one time, and at the next time as a *third party* that guarantees a payment of the customer.

Figure 1 lists those enabling third parties we expect to be present on most electronic marketplaces. We do not use the term *Trusted Third Party (TTP)* since:

- Most authors do not distinguish between *TTPs* and *TTP Services.* Any party can provide the latter, but whether a specific provider of such services is a TTP, i.e., trusted, is completely up to each individual user to decide. There is no such thing as a TTP in an objective sense.

- The main objective of security protocols is to limit the trust necessary in other parties. In fact, most services provided by so-called TTPs could be implemented such that not much trust is required at all.[1]
- Conversely, the term TTP suggests that a user does not need to trust those third parties not called TTP. Unfortunately this is not true: For instance, a user has almost no chance to verify that her PC works correctly, i.e., she has to trust the manufacturer of this device. But usually nobody calls manufacturers of computing equipment TTPs.

The purpose of the third parties mentioned in Figure 1 is:

Network: The network describes the providers of communication infrastructure used on the electronic marketplace.

Directory: A directory is a (possibly distributed) database that maps a name of a participant to a directory entry and its attributes such as the name holder's description, addresses, certified public keys, etc.

One might consider physical *phone books* the analogue of directory services. Relevant standards are, e.g., *X.500*, *LDAP*, *WHOIS++* and the Internet *DNS*.

Registration and Certification: A *registration and certification authority* (abbreviated RA and CA, respectively) links digital and real identities of a user who wants to register. For this, it verifies the user's real identity (registration) and certifies a public key of the user (certification), so that the user can now act under the digital identity.

One might consider physical *passports* the analogue of certificates. The most important standards for certification schemes are *PKCS #6*, *X509* [*X509*], and the *UN/EDIFACT Certificates* [*EDIFACT*].

Arbiter: An *arbiter* evaluates the meaning of digital evidence based on protocol specific *decision procedures*. The decision procedure of a protocol defines which messages serve as evidence, and defines their semantics, i.e., what they are evidence for.

Note that properties like *non-repudiation* or *disputability* do not make any sense without well-defined decision procedures. Since dishonest behaviour cannot be prevented in many cases, requirements on who would win a dispute in a certain situation is often the only way to express specific security properties.

[1] First, most services can be implemented using several third parties such that one honest third party among all of them is sufficient, i.e., trust is *distributed*. Secondly, TTPs can be made *accountable*. Sometimes, this has to be based on the physical world. For instance, the certification of public keys for signatures can be made disputable if a written contract between CA and user is required, and the user's public key is included in that contract. The CA can still issue false certificates, but in case of a dispute the user can prove that the CA has certified the wrong key, and thus the user is not held responsible for the CA's fault.

Ideally, everybody should be able to evaluate a decision procedure, i.e., the arbiter's decisions should be predictable. One might consider an *expert witness* the analogue of an *arbiter*.

Broker: Like a *directory*, a *broker* is a database of entities of the marketplace, but it enables more intelligent requests, e.g., a characterisation of a service requested.

One might consider physical *Yellow Pages* as being the analogue of a *broker*.

Mall: A *mall* provides a common interface to several service providers. This includes

- *service gateways* that translate and redirect requests from buyers to sellers and the responses back to the buyers,
- *content hosting*, where the mall provides the services on behalf of the sellers.

Payment: This means the *payment system provider* as well as all other financial institutions (*issuing banks*, *acquiring banks*) that participate.

Notary: Notaries will provide or support trustee services like

- notarizing/time-stamping,
- contract signing,
- fair exchange of values, and
- long term archiving.

If necessary, we will add specific third parties for these services later.

2.1.2 Commerce as a Sequence of Exchanges and Transfers

Our model describes business sessions in terms of sequences of *transfers* and *exchanges* (similar to the *dialogues* of interactive EDI).

In a **transfer**, one party sends an item, called a *container*, to one or more other parties. The sending party can define certain security requirements, such as confidentiality, anonymity, non-repudiation of origin, non-repudiation of delivery.

An **exchange** consists of several combined transfers: two or more parties have the *assurance* that if they transfer something specific to the others, they will also receive something specific.[2] Note that we consider a transfer with non-repudiation of submission or delivery as a "transfer", and not as an "exchange."

[2] We require a *guarantee* of fairness for exchanges. If no such guarantee is required, we model such a conversation by several transfers.

Begin and end of a transfer or exchange will be clear for each party locally, but there is not necessarily global synchronisation, i.e., different parties might recognise the same logical events at different times or in different orders.

The actual sequence of transfers and exchanges in a business session can either be determined directly by the user, or it can be described by a protocol for such business sessions. Several exchanges and transfers may then be linked via a common context. Of course, a protocol may branch, i.e., allow more than one sequence.

After each transfer or exchange, the parties are either

- **satisfied** and thus willing to proceed with a certain number of other transfers or exchanges or
- **dissatisfied**, in which case an *exception* or *dispute* is raised (which might end up at a real court if all else fails).

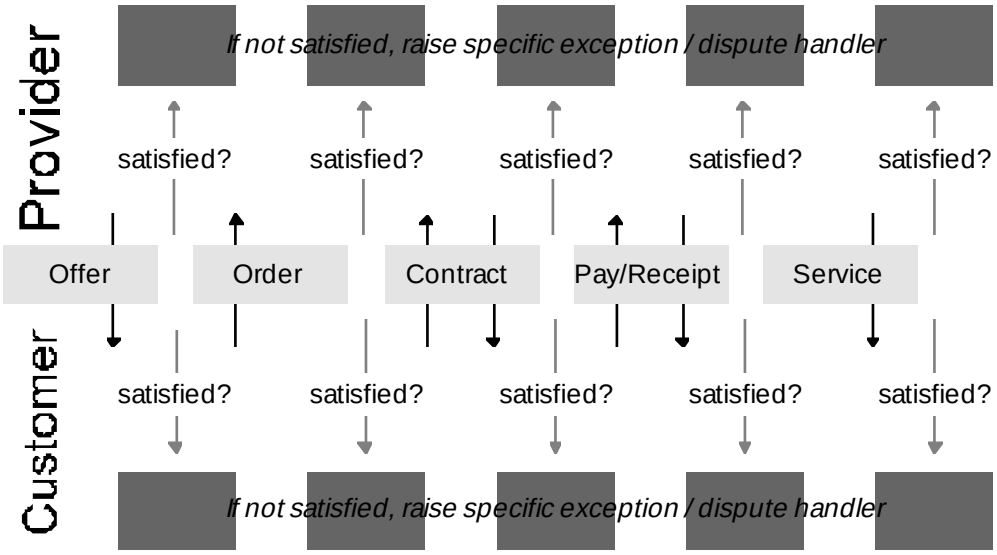

Figure 2: Simple example of a sequence of exchanges and transfers

A **container** is a general data structure for what can be transferred and exchanged. It may contain several **primitive types**:

- signed documents, such as
  - certificates;
  - receipts;
  - signed statements;
- information, such as
  - digital goods;
  - information necessary to access a service (e.g., an address and password, or a cryptographic key that protects a video stream);
  - information necessary to access physical goods;
- electronic money.

Information means something that is not interpreted for the purpose of a specific transfer or exchange, i.e., it is just transferred as a bit-string. (Of course, it *is* interpreted in the protocol for the business session, e.g., as good or bad digital goods.) In contrast, the transfer of a signed document primarily means that the recipient must get a correct signature. Money, as it occurs in the business session, is primarily a description, such as "350 ECU", and the implementation of a transfer of money will typically consist of several internal transfers of payment details for a given payment protocol.

Figure 3 gives an overview of the possible exchanges of primitive types. Transfers are included as exchanges of "something" for "nothing".

Party A sends → Party B sends ↓	Money	Signed document	Information
nothing	Payment	Certificate transfer etc.	Information transfer
Money	Fair money exchange	Fair payment with receipt	Fair purchase
Signed document		Fair Contract Signing	Fair conditional access
Information			Fair information exchange

Figure 3: Transfers (first row) and exchanges of primitive types

We expect that the listed transfers and exchanges are sufficient in order to build all relevant commerce services. Obviously, the matrix of Figure 3 is complete with respect to pairs, but there may be different security requirements in detail.

Our initial investigation is two-party centred. The same considerations can be applied to the multi-party case. For instance, more than two parties might wish to sign a joint contract, or one sender might want to send a certified mail to several recipients.

An important goal of later sections is to define a toolbox (e.g., as a set of generic services, or as a configuration / script language) for transfers and exchanges, so that business sessions can easily be composed out of a few generic building blocks.

2.1.3 Exception and Dispute Handling

All services have to define what happens if one of the parties is dissatisfied at a certain stage, i.e., how an **exception** is handled. Exceptions may be raised because of

- a party having insufficient privileges,
- an unintentional error, e.g., a lost message, incompatible software versions, or a typing mistake by the user,

- malicious behaviour by others, e.g., abuse of the credit card number of someone else.

Obviously, there is no well-defined difference between the second and the third possibility. In particular, the reason for the exception is not known at first, e.g., when the dissatisfied party notices an incorrect transaction on its credit card statement. Therefore one starts with **exception handling** in the narrow sense, which is based on the assumption that all parties are honest. If this optimistic approach fails, the parties are in a real **dispute,** and more pessimistic approaches must be used, usually involving **arbiters** (and finally courts).

Exception handling requires all parties to keep sufficient **audit trails** of what has happened so far, so that they can try to agree on the current state. For a real dispute, the audit trail has to contain evidence (e.g., non-repudiation tokens) that can be shown to and verified by the arbiter using a **decision procedure**.

For legal certainty, the result of a decision procedure must be well-defined and predictable. But there may be protocols where the result is not necessarily "correct" in the sense that it restores a previous state where all honest parties are satisfied. For instance, in current mail-order/telephone-order transactions using a credit card, disputes between buyers and sellers about an order cannot be resolved unambiguously. In these cases, additional rules define how to resolve such ambiguous situations, e.g., they might say that the buyer is never liable for such orders.

Other disputes (e.g., due to typing mistakes or incompatible software) may not be solvable until a case has been in court and a judge has interpreted the law.

Exception handling might require to switch to another communication protocol, e.g., from http to asynchronous electronic mail, but should remain within the digital system as far as possible. Actions outside, like phone calls or sending letters with hand-written signatures, should be minimised, but may remain necessary as a last resort.

2.2 Service Architecture Overview

The *SEMPER* architecture consists of four *layers*, see Figure 4.

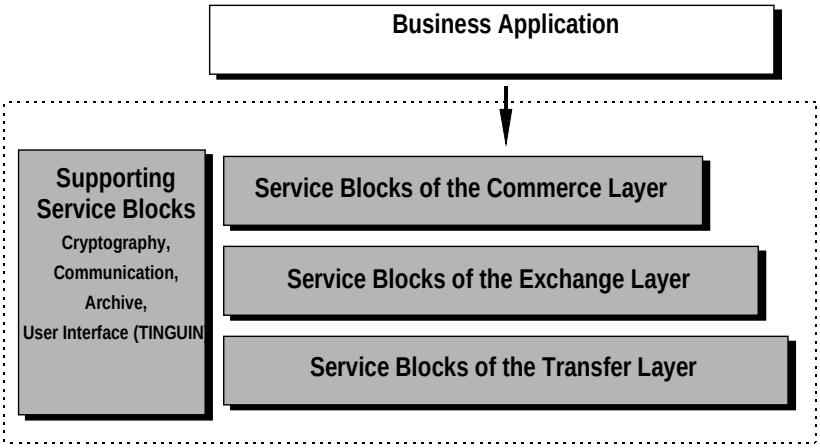

Figure 4: Architecture of *SEMPER* — Overview

2.2.1 The Business Application

The Business Application Layer is the environment where the *SEMPER* library is used. Currently, http servers are implemented to embed the *SEMPER* library into the World-Wide-Web (see Chapter 3.6 or D04 for details).

A business application is implemented using the services of the lower layers, primarily the commerce layer. Access control is used to prevent unauthorised access to services of the layers below the commerce layer. Note that this may allow certain applications to access services of layers below the commerce layer. An example where this is useful is when a user is being registered (this is a business application in *SEMPER* and is most easily handled by letting the application access the certificate service block directly).

2.2.2 The Commerce Layer

The Commerce Layer provides services that directly implement protocols of business sessions, e.g., how specific merchants or types of merchants handle customer registration and offering, ordering, payment, and delivery of goods. An experimental secure downloading mechanism has also been designed to enable clients to interact with formerly unknown server commerce services and still retain security. Naturally, this involves some kind of code certification.

However, the commerce layer not only offers such entire protocols, but also building blocks that may be of more general use, in particular for buyers. In particular, these standardised services may offer template services, e.g., to display and fill-out standardised order-forms and construct containers from them.

Currently, only these basic building blocks are implemented whereas the control flow is done by the business application.

2.2.3 The Exchange Layer

The Exchange Layer provides services for the fair exchange of containers. “Fair” means that, if A and B agreed on what to exchange, Party *A* receives *B*’s container if and only if *B* receives *A*’s container. (Without fairness, this would be done with two transfers.)

2.2.4 The Transfer Layer

The Transfer Layer provides send, receive, and composition services for containers in order to transfer information between client and server according to given security attributes.

As mentioned in Section 2.1.2, everything that can be transferred by the transfer layer is encapsulated in one data type called container. Each container can contain several basic items, such as signed documents, information, and money. Moreover, the transfer or the container may have **security attributes**. For instance, the sender (more precisely: the invoking entity) specifies that the transfer of the container should be with non-repudiation of origin; then the sender’s entities of the transfer layer, with the

help of the supporting services, attach a suitable signature and send everything off, and the recipient's entities of the transfer layer verify the signature and tell the recipient something like "you received this container, and the sender cannot deny the transfer".

This layer contains the **transfer manager**, the **payment service block**, the **statement service block**, and the **certificate service block**. Basically, methods of the transfer manager decompose containers to be transferred, and let signed documents and information be handled and transferred by the statement service block and payments by the payment service block. The peer transfer manager reassembles the received parts of a container to the original tree-structure. The certificate block helps in the administration of the necessary certificates and performs user-registration procedures.

In addition to the actual transfers, the transfer layer also offers invoking entities to manage these transfer services, e.g., to manage the contents of an electronic purse, to set up relations to enabling third parties or to configure the policies for purse selection. In contrast to the actual transfers, such services do not need to go via the transfer manager, e.g., the payment service block can immediately be used by higher layers for purse management.

2.2.5 Supporting Services

The Supporting Services provide support to all the other layers.

The **Cryptographic Services** provide the cryptographic services needed for data security. Examples for such services are message encryption and decryption, hash functions, message authentication codes, digital signatures, and their key generation. The crypto block is entirely used as a supporting service and does not provide a transaction class allowing the creation of crypto sessions.

The **Communication Services** support communication between different entities. The majority of the services offered are independent of the network and protocols actually used: for example, only the object that creates a communication end-point needs to know what protocol is to be used for the communication; objects which use (read from or write to) an already established communication channel need not be aware of the protocol being used underneath.

The **Archive Services** provide local storage and archiving of all persistent data. Examples are digitally signed messages, certificates, cryptographic keys, and transaction or evidence objects. The archive includes features for secure storage of data by, e.g., encrypting the data or archiving in tamper-resistant memory (e.g., on a smart card).

The **Trusted Interactive Graphical User Interface Services** support the interaction between services and the human user. The basic idea as that by providing a distinguished user interface to the security services, the user is aware that all inputs through this user interface are critical and that the "usual" business application user-interface (e.g., www-browser) must not be used for any critical input.

Naturally, the security services user-interface must be under control of the user's own device. For instance, if a buyer verifies the identity of a seller, the message "the verification of the identity was correct" cannot come with the seller's html-pages

dependably: Any crook could simply provide this message ready on its html-page! Instead, the users own device must show this message in a way so that the user can recognise it as a message from her own device. Ideally, the TINGUIN would reside on a mobile user interface, such as a PDA [*PPSW97*].

Moreover, ergonomics has to be considered in the design of the TINGUIN even more carefully than usual, because any misunderstanding by the user may destroy security.

The **Preference Services** provide a uniform way of handling preferences for each service block. Each manager must define the available options and the user selects the preferred ones through the TINGUIN. In addition, the preference block stores the actually installed configuration, i.e., all information specific to a particular installation like the target system, the screen type or any other hardware devices currently supported.

Access Control Services provide all services needed to implement access control on the methods of the various blocks of the *SEMPER* architecture. Access control ensures that potentially dangerous, compromising or undesirable operations cannot be performed by unauthorised users or without the user's explicit consent. Access control is *capability* based. Each block must declare a capability for each action for which unconditional access is undesirable. Such a capability must be verified each time the corresponding method is called. The Access Control Block provides the necessary services to manage such capabilities based on roles. By the way, we assume that they only try to access each other nicely by their services, i.e., that the operating system does not allow them to tamper with each other's code or memory.
The roles are currently unlocked after appropriate user authorisation. Later, this should be based on a policy fixing how and at what times the user device makes sure that the correct user is there, *identification* (also called *user access control*), and at what times the user must be asked for an OK, *authorisation*, in particular whether higher layers can ask for authorisation for many actions, e.g., many small payments, in advance, that would have to be authorised by the user if they were carried out individually.

The service blocks described so far are shown in Figure 5.

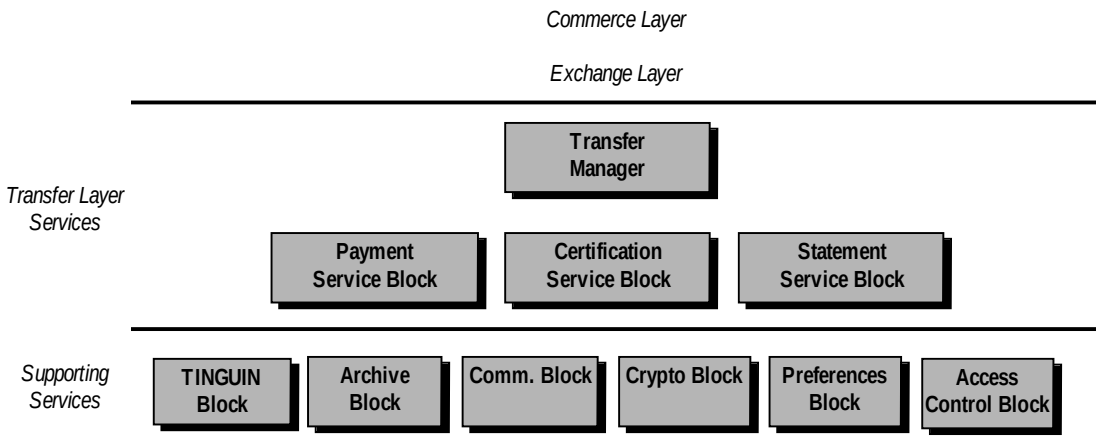

Figure 5: Service blocks of Supporting and Transfer Services

In addition, the actual implementation also contains a **utility block** with various utility services for, e.g., for logging and debugging.

The whole architecture has to be considered specifically to provide for **trusted hardware** on small mobile devices (ranging from a smart card that can *only* be used together with another device of the same user, to a PDA that is used on its own). Of course, if trusted hardware is available at all, as much as possible should be implemented on it, but there are limits in program storage, computing capability, and the quality of the user interface.

3 Implementation architecture

This chapter discusses general notions needed to define the *SEMPER* architecture.

3.1 Services, Interfaces and Protocols

(M. Waidner / ZRL)

SEMPER defines a layered service architecture. Since we make a very informal description, we do not need a formal semantics of "layer" and "service". But in principle we use these and related terms in the sense of the ISO OSI model [ISO 7498]. We note however that this layering is not strict and higher layers are allowed to bypass lower layers; the architecture is therefore best seen as a staircase:

The active elements in such an architecture are called **entities**. Each entity resides in one device, where **device** means a physical object, such as a personal computer. The term **local** means "in the same device". Usually, a device belongs to the player (or set of players) who relies on it.

A **layer** is a (virtual) collection of entities in all devices that perform functions of a similar degree of abstraction. More basic functions are collected in lower layers, and higher layers perform more complex functions using the services provided by the lower layers. Layers are usually numbered 1, 2, …, starting with the lowest one. The interface between two layers is called a **layer interface**. It is the union of all interfaces of the entities of this layer. Entities within the same layer providing the same service on different devices are called **peer entities**.

The joint functionality that a layer offers at its upper interface, i.e., to the higher layers, is called the **services** of this layer. A layer can offer several services.

An entity can access the services of any lower layer at a **service access point (SAP)**, see Figure 6. A **service interface** of a service is a subset of the layer interface, consisting of service access points and a syntactical definition of the **service primitives** that can be exchanged via these service access points. An entity accessing (or using) a service is called an **invoking entity**.

A **service** is described by characterising the possible behaviour on the whole service interface, i.e., how inputs and outputs at the different service access points are related, but *without* describing how these relations are implemented.

A **protocol** describes how a service is provided by means of interactions of peer entities via lower layer services. Typically, the same service can be provided by many different protocols, and some of these different implementations may even be present in the same system. A protocol that implements services of Layer *i* is based on the services of the lower layers 1,...,*i*-1 and can also invoke other services of its own layer. Thus access is not restricted to the layer immediately below, even though protocols will mainly use services of the layer immediately below.

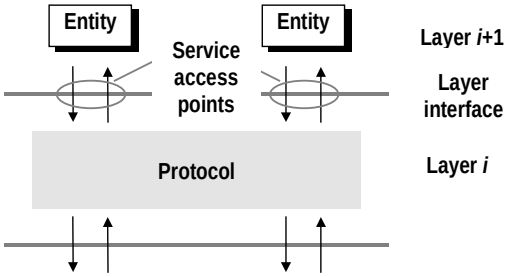

Figure 6: Layered service architecture

3.2 Service Blocks containing Managers and Modules

(M. Waidner/ ZRL)

Generic services are provided by **service managers** and several **service modules** (see Figure 7). The union of a manager and its modules is called a **service block**.

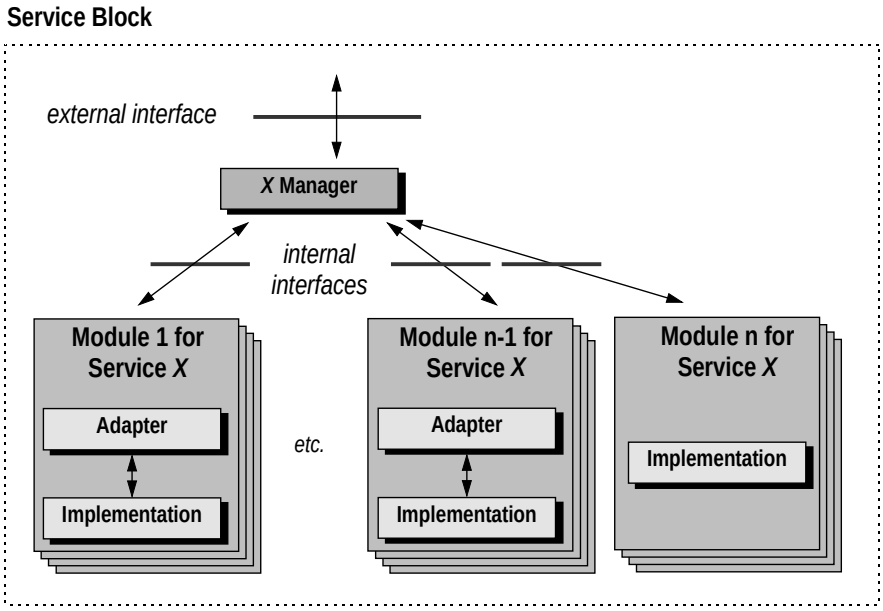

Figure 7: Managers, modules and adapters[3].

The **service block** provides the generic, unified service, e.g., service = "payment". We call the interface supported by the entities of the block the **external interface**[4] of the service block.

These external interfaces provide a **generic service**. The generic service is based on a model of the service which should cover a broad range of protocols implementing this service, i.e., we will have one generic interface for a whole class of services.

For instance, the external interface of the payment manager is based on a *generic payment service* that will cover all kinds of *payment protocols* (or at most at a small number of generic payment services, one for each payment model). Note that this does *not* mean that we will support only a few specific payment protocols, but that the interface definition is so general that *any* reasonable payment protocol can be accessed via that interface. If, e.g., company *XYZ* comes up with a new payment system *abc*, all they would have to do in order to link *abc* into our architecture is to map the service interface of *abc* to the internal payment interface. This guarantees the desired openness of the marketplace.

[3] The last module is an example of module written specifically for SEMPER and needs no adapter

[4] This is also often called an Application Programming Interface (API).

A **service manager** provides a common interface to several modules together with methods for negotiation and selection of an appropriate module.

A **service module** corresponds to a protocol implementing the service, i.e., it more or less implements one entity of such a protocol. Its interface to the service manager is called an **internal interface**[5] of this service. Each generic service model may define several internal interfaces based on the **type** of the expected service of its modules.

Having several modules per service allows different types of service and different protocol implementations by different manufacturers. Service modules are said to be of the same type if their behaviour at the internal interface is the same. Examples of types of internal payment interfaces are “cash-like”, and “account-based”. Modules could be “SET”, and “e-cash” where “SET” implement the account-based model whereas “e-cash” implements the cash-like model.

As *SEMPER* wants to build on existing products as far as possible, we cannot assume that they all fit the same internal interfaces originally. Therefore, the interface of an existing implementation is enhanced by a **service adapter** so that the resulting module supports the required service.

The *SEMPER* architecture describes a fixed set of service managers. The set of service modules is not fixed, i.e., service modules can be dynamically attached to managers.

[5] This is also often called a Service Provider Interface (SPI)

3.3 Transactions

(M. Schunter / SRB)

3.3.1 Transaction Objects and Context

Services of a service block in *SEMPER* can either be used directly or through a so called transaction object. A transaction object is managed by the manager of the service block and when instantiated it defines a context for all services (possibly by negotiating with the peer entity - see Figure 8). Thus the advantage of using the services through a transaction object is that many parameters of the service are fixed by the context. The manager is notified during instantiation of the new transaction object and keeps track of all ongoing transactions of the entity.

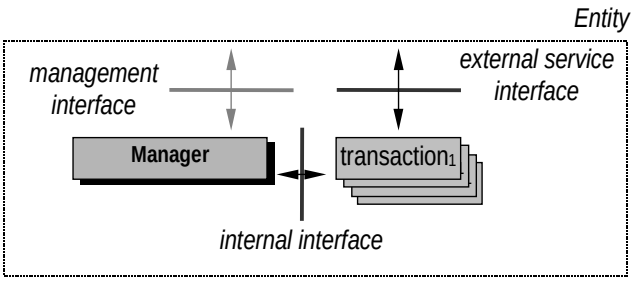

Figure 8: Entities (managers or modules) and their transactions.

3.3.2 Fault-Tolerance

We now describe the concept how service blocks can implement fault-tolerant **transactions** in the object-oriented model. A *SEMPER* transaction is a sequence of method invocations which implement a state transition from one consistent state to another where the **state** of a transaction is the set of all its objects which are stored persistently. A payment transaction should, for example, ensure that it either ends in a state before or after a payment, i.e., if it is interrupted during the execution of a payment protocol, it should either complete the protocol or clean-up by undo the effects of the protocol. To achieve this property in the presence of faults, every transaction must implement so called recovery mechanisms to get into a consistent state after the failure.

Since protocols of a transaction may use other services of other transactions, transactions are ordered hierarchically. The transaction started by another transaction is called its **sub-transaction** (see Figure 9). For example, making a payment may involve transactions for the actual payment and for a transfer of a signed statement.

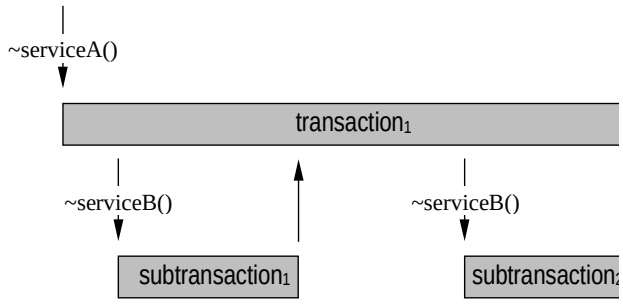

Figure 9: Transactions and sub-transactions.

During execution or when starting sub-transactions, each transaction stores persistent states at so called recovery points.

After a crash as in Figure 10, all ongoing transactions need to be recovered, i.e., the recovery procedure should either “roll-back” into the last consistent state or redo parts of the transaction to reach the next consistent state. The recovery is performed top-down: The topmost manager recovers its transactions by re-instantiating the transaction objects and calling the method which has been interrupted. The transaction restores the last recovery point which may involve instantiating sub-transactions and either undoes all actions and reports a failure or continues (see Figure 11 and Figure 12).

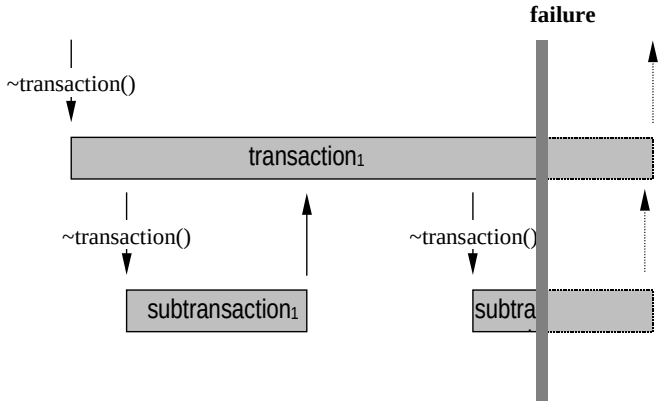

Figure 10: Transactions with failure.

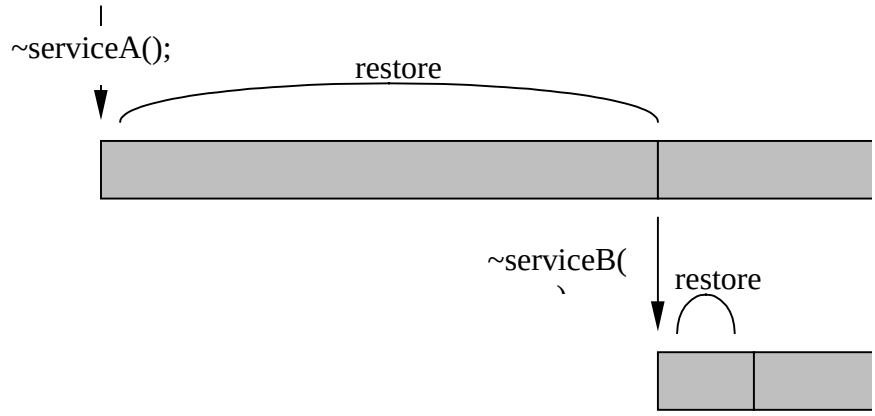

Figure 11: Restoring the last recovery point.

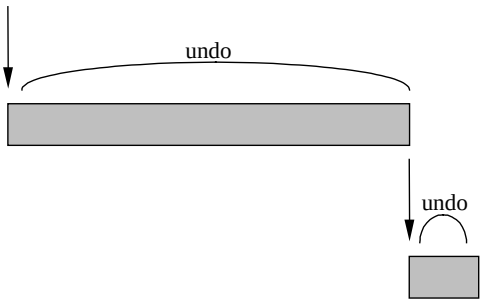

Figure 12: Transaction Undo[6]

Since recovery of each transaction depends on the details of the algorithm, the recovery is implemented by each transaction object which must make sure that all objects needed for recovery are stored persistently in the archive to enable consistent recovery.

3.3.3 Design

We now describe the interfaces needed to support transaction management, which includes the requirements on transaction objects in order to be recoverable. In our view, the notion “recoverable transaction“ does only apply to transactions that were interrupted by exterior influences, e.g. a crash of the local or remote computer. We do not consider normal failure situations that can appear in transactions, like incorrect parameters.

Transaction objects are required to provide all methods for transaction management and recovery. Transaction objects are created by the service blocks implementing the service delivered by the transaction. A transaction object creates all subtransaction objects it needs itself. The knowledge how to do recovery is local to each transaction,

6 If the transaction is not continued, all persistent changes which would represent an inconsistent state should be undone.

i.e., each transaction is required to implement its own recovery which is triggered by the manager of the block using the non-terminated persistant transaction object as input. We assume that each transaction has a state, i.e., a subset of its attributes which are sufficient to reinstatiate all attributes.

Every transaction and its subtransactions are assigned a unique transaction identifier, so that in later recovery runs the transaction factories of a block can learn which transaction objects belong together.

We assume that each ongoing transaction object stores before and after critical actions its state in so called transaction records. When a transaction finishes successfully, all transaction records of the transaction and all subtransactions can be removed unless some kind of undo mechanism should be provided.

In case of a crash the manager of a block (or dedicated transaction factories) reinstantiates the registered transaction objects with the appropriate transaction identifier. The transaction object reinstantiates all of its subtransactions and tells them to start recovery.

3.3.3.1 Services needed for Transaction Management

An object that wants to provide Transaction Management for a given service is created by a transaction factory:[7]

- `Transaction` ← `TransactionFactory.GetTransaction()`

All further operations are then performed on the output transaction object.

In general, transaction objects must offer the following methods:

- `Transaction.start()`
  to actually start and perform the transaction. All parameters and attributes that are relevant to the transaction must be fixed before the transaction object is started. The transaction and its subtransactions must take care of storing their transaction records regulary that later enables recovery or undo.
- `Transaction.recover()`
  to recover the transaction after a crash. This includes recovering all subtransactions. There are two possibilities: Either a recovery run consists of the transaction finishing itself, starting at the point where it was interrupted before and continuing with all subtransactions that were not performed in the first run; or some subtransactions require the transaction to undo everything that was performed before the crash. The information which subtransactions can not be recovered after a crash and require the whole transaction to be undone must be stored by the transaction object itself.
- `Transaction.undo()`
  to allow for undoing a transaction after a successful run. Note that it may not be possible to implement this method for all kinds of transaction.

---

[7] Note however that the names mentioned here are not strictly used through SEMPER. So in the payment block `Purse` is the `TransactionFactory` where `startTransaction` is the equivalent of `GetTransaction.`

We need some kind of crash detection mechanisms:

- `TransactionFactory.recover()` ()
  This will be a method which is invoked by the block Manager for all its transaction factories at bootstrap time and which should check for transactions that were interrupted. If such a transaction are found, the stored transaction record are loaded and the transaction object are reinstantiates and recovered.

To implement `recover()`, a constructor has to be provided that creates a transaction object from a `TransactionRecord`:

- `Transaction` ← `Transaction(TransactionRecord)`

3.3.3.2 How to Use Recoverable Transactions

When the object that implements the original tasks is already instantiated and has all its attributes set, all what is needed is a method invocation to create the transaction object and a further call to the transaction objects `start()` method.

Additionaly a program that wants to use recoverable transactions has to recognize that it has crashed before and has to find the point and all necessary context to start a recovery run.

So what actually happens when SEMPER starts up is this:

1) In the `init()` method of `semper.util.bootstrap.library` after the initialization of all Managers the method `recover`() is called for each Manager with higher layer managers first.

2) `recover()` would try to recover all open transactions which are not yet recovered as part (subtransactions) of the recovery of higher layer transactions.

To allow for recovery on both sides, client and server, the managers must open a port and listen on it for recovery requests from the peers with which transactions were done. When such a request arrives, it has to retrieve the appropriate transaction records and perform recovery.

3.4 Extended Finite State Machine Design Pattern

(L.Fritsch , M. Krüger & M. Schunter/UDO)

3.4.1 Introduction

A distributed protocol is a collection of machines that cooperate by sending and receiving messages. Each machine takes messages as input, performs some local computations that may change the machine's state, and finally outputs messages. Examples for such protocols are communication protocols [Hogref89], payment protocols [AJSW97], cryptographic protocols [Schnei96], or other commerce protocols. Note that each protocol may have many steps of progress. The protocols described in [PfScWa98,BürPfi89], for example, each have up to eight states and the protocol has about fifteen possible messages, which have to be handled by the machines.

A major problem when designing such protocols is to specify exactly which messages exist, which messages are expected in a given state and finally what happens if an unexpected message occurs in any given state. Even if the specification is complete, in practice, most implementations of security-critical protocols have failures [Neuman95]. A common problem in the implementation results from incomplete handling of all possible and unforeseen messages in each state. As a result, the protocol is neither robust, nor does it follow its specification.

In order to simplify implementing correct and robust distributed protocols, we describe a pattern for distributed protocol machines in Java which has been used for payment and communication protocols in SEMPER. The main goal is to encourage a clear definition of all states and messages as well as the behavior when processing any message in any state. Our design reflects this goal by defining the known messages as well as default error handling in a superclass of all states. Each subclass then inherits the automated error handling while defining the intended behavior for expected messages. This way, a complete specification in one place guarantees robustness in the protocol implementation.

3.4.2 Anonymously Transferable Standard Values

As an example for the ease of use of the design pattern, we will now sketch the well-known "anonymously transferable standard values" [BürPfi89]. This on-line payment scheme can be based on any digital signature scheme together with an anonymous communication network.

A standard value is like an account with a fixed amount on it (say, 1 EURO). The bank then stores one public signature key for each of the standard values, which identifies its owner[8]. If the owner then signs another (newly generated) public key and

8 Note that this key pair is only used for this single standard value, i.e., for privacy, it must not be used for any other purposes.

send it to the bank, this new public key is stored as the new owner of this particular standard value, i.e., payment using standard values looks as follows:

- The bank knows pk_1 as the current owner of a given standard value.
- The payee generates a fresh key pair with a public key pk_2 and sends it in message "MsgRequestPayment" to the payer.
- The payer signs pk_2 using the secret key corresponding to pk_1 and sends the signature in message "MsgTransferValue" to the bank.
- The bank verifies the signature and stores the new public key pk_2 as the new owner of the standard value. Furthermore, it keeps pk_1 and the signature on pk_2 as evidence of this transfer in order to detect and prove double-transfer of the same value.
- The bank signs a receipt of this transaction including pk_2 and sends it in "MsgForwardReceipt" to the payer.
- The payer forwards the receipt in message "MsgComplete" to the payee.

The scheme guarantees untraceability since no key is used twice any we assumed that the communication network is anonymous.

Figure 1 shows the protocol and the three participating parties as a diagram.

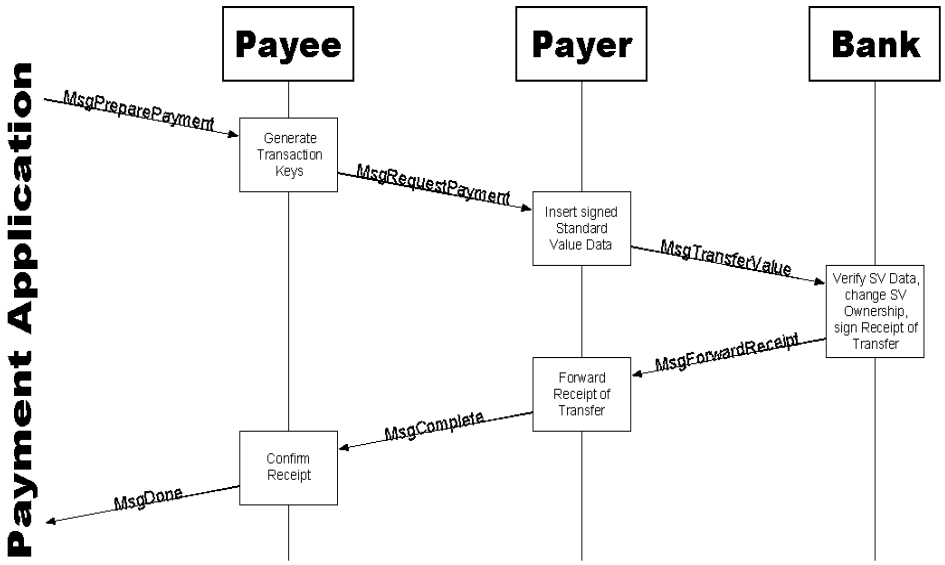

Figure 13: Protocol diagram for the payment protocol.

3.4.2.1 Automata for standard value payments

The standard value payment protocol is specified as two extended finite state machines. The first machine is the payer/payee machine. Starting from the *ProtocolCompleted* state, the first of the payment messages decides whether the automaton has to play the role of the payer or the payee. The payer/payee machine is shown in Figure 14.

The machine that implements the bank's protocol has only one state that is waiting for messages that cause transfers of standard values. Figure 15shows the bank machine.

Additionally, a *Broken* state is entered by each machine in case of critical errors or messages that cannot be processed.

Not addressed in the automaton transition diagrams are variables. The states of the automata do calculations and comparisons on state and message variables (e.g., checking whether the originator of a payment request really owns the amount of standard values that is to be transferred). If a computation on variables leads to a protocol failure, transactions can be aborted, the *Broken* state reached or the initial *ProtocolCompleted* state can be restored.

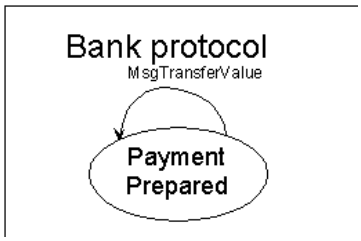

Figure 14: Automaton for bank protocol[9].

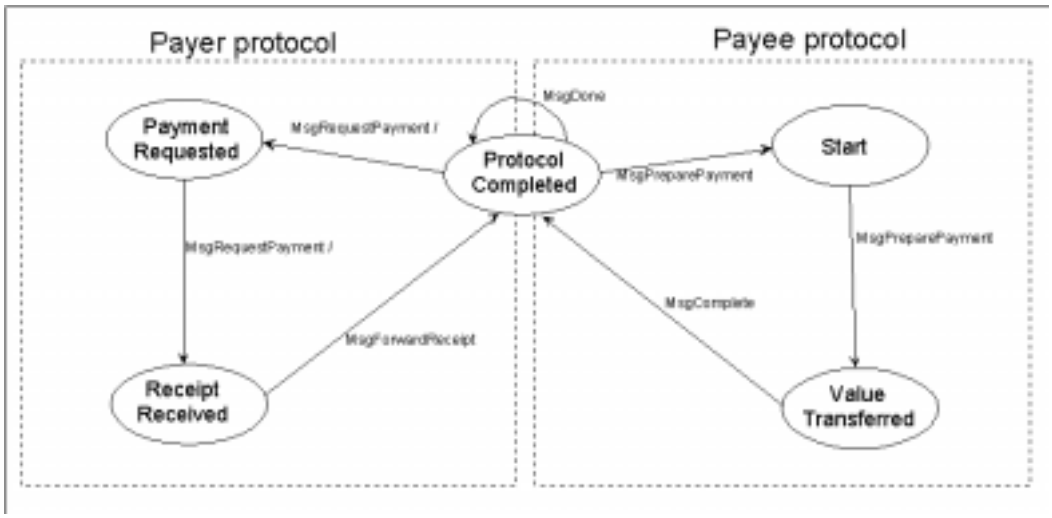

Figure 15: Automaton for payer/payee protocol[10]

3.4.3 The Pattern "Extended Finite State Machine"

3.4.3.1 Intent

Our goal is to provide a pattern for extended finite state machines that can be combined to distribute protocols. Requirements are automated enforcement of a well-defined reaction to all messages as well as ease of use.

9 Messages output by the bank are omitted

10 The *ProtocolCompleted* state is a waiting state where the machine stays at times of no activity. Messages output by automaton are omitted.

3.4.3.2 Motivation

An extended finite state machine [Lynch96] is a finite state machine[11] where each message and each state may contain a finite number of variables and the state transitions not only change to another state out of the finite set but also perform arbitrary computations on the variables contained in the state, the message received, and the message sent (this is the "extension"). The purpose of modelling the infinite state of a Turing machine as a combination of a small finite set of states together with an assignment of values to a given finite set of variables is to enable more intuitive protocol specifications: Each of the finite number of states and messages has an intuitive meaning whereas the variables make the model as powerful as Turing machines.

For a distributed two-party protocol, two such machines are then connected by sending all messages output by one machine into the other machine and vice versa. This communication between the participants is not considered part of the extended finite state machine since we aim at a protocol which is independent from the means of communication.

The implementation of a distributed protocol with communicating machines in Java requires an implementation of the machine and of the communication. While the possibilities of implementing communication are numerous, we want to focus on ways to enforce the protocol and minimize the chances for undetected protocol failures or other pitfalls for programmers and users of the protocol. Therefore, our requirements for a robust distributed protocol are:

- well-defined states,
- well-defined messages, and
- a well-defined reaction on any kind of message in any state.

3.4.3.3 Operation

First, instantiate a machine. Then, feed messages (possibly containing some meaningful data) to its input, and analyze the message returned as a response. In order to produce the output message, the state object internal to the machine will process the message, do computations on the state variables, analyze the data content of the messages, instantiate a message to be returned, and change the state.

3.4.3.4 Properties

The behavior of the machine is well defined by the extended finite state machine's states (that include the transition function). Message processing and error handling also happen in the state classes.

---

11 A finite state machine is a tuple (S, M, s_0, F, d) where S is a finite set of states, M is a finite set of inputs (i.e., messages), s_0 is the starting state, F is the set the final states, and $d: SxM \rightarrow SxM$ is the transition function which get a state and a message as input and produces a follow-up state and a message to be sent as output.

Protocol failures[12] are caught and can possibly be recovered if states of the extended finite state machine reflect recoverable protocol transactions and recovery has been properly specified.

The protocol can be modified or extended in a variety of ways as described in Section 3.4.4.5.

3.4.4 Java Representation of the Pattern

We now describe the Java representation of the extended finite state machine. In general, the class *Machine* contains the data of the machine in which the protocol computations take place. The subclasses of *Message* define all messages and their data content. The class *State* defines the default error handling. Its subclasses correspond to the states and define the behavior of the machine, i.e., the code executed for each message expected in each particular state of the machine.

3.4.4.1 The Extended Finite State Machine Pattern

The classes of the extended finite state machine and their relations are depicted in Figure 16. Each extended finite state machine has a state, as well as the data common to all states on which the machine performs computations while processing messages.

12 Protocol failures are the receiption of messages not expected in the current state and the receiption of machine-internal error messages (defined in the machine) .

Intuitively, the machine has some code for handling each specific message in each specific state .We structured these pieces of code after the current state of the machine, i.e., each subclass of *State* defines how to process the messages expected in this state. Unexpected as well as unknown messages[13] are handled by the *State* superclass.

We now describe the message and state classes and then how message processing and exception handling work.

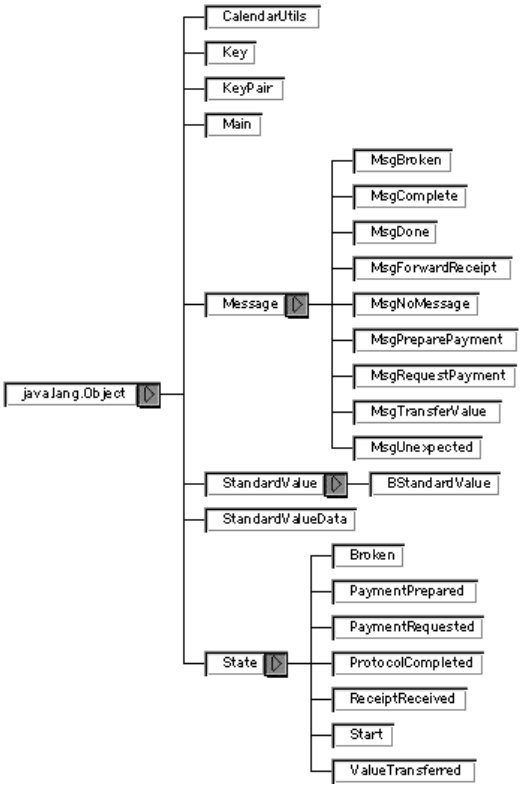

Figure 16: Class hierarchy of the payment protocol automata example.

3.4.4.2 Messages

Figure 17 shows an abstract message superclass and the message subclasses of the protocol example. The *Message* superclass defines protocol parameters such as message sequence numbers or the addressee of messages in multi-party protocols. An inherited subclass of *Message* may define message-specific parameters.

[13] "Unknown" messages are messages that have not been declared for processing in the *State* superclass.

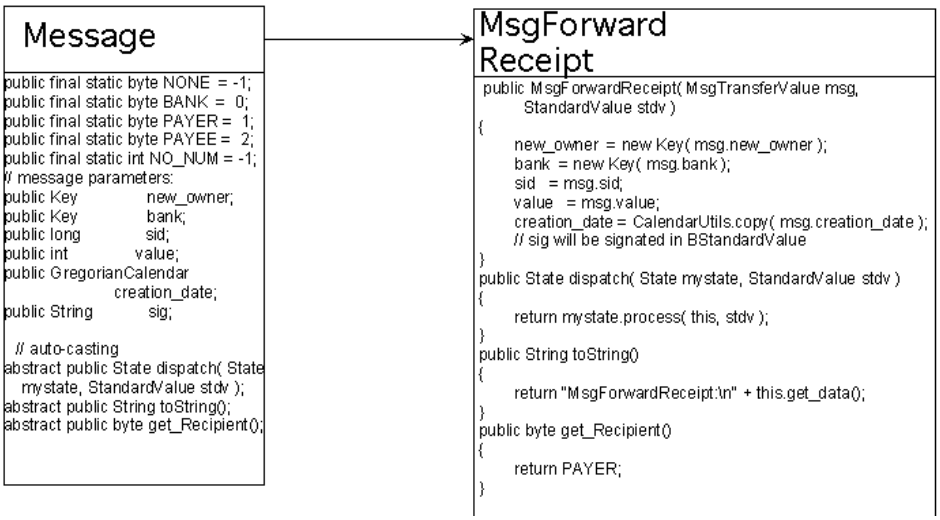

Figure 17: The Message superclass and one subclass of the payment protocol

Furthermore, the Message superclass defines an abstract method *dispatch()* for dispatching itself to the appropriate method of the current state. *dispatch()* is called by the machine after receiving a message.

Note that this method is overloaded with identical code in all subclasses. This has proven essential in Java because the language lacks run-time subclass awareness when subclasses are addressed under their superclass type. Without the subclass-local dispatcher, no automatic selection of the appropriate processing methods in the states is possible: A message read from a network is de-serialized as *Object* and then is cast to *Message* without knowing the subclass. Casting the message to its correct subclass is left to the message object's *dispatch()* method (which knows about its subclass type). That ensures our concept of subclassed messages where the environment of the machine need not be aware of the type of the message. See [GoJoSt96] for the Java method invocation procedure.

In our example, each message subclass has a method *getRecipient()*. As the protocol is a three-party-protocol, messages sent out by one of the machines have adressed to their corresponding party. E.g., the payer protocol will receive a request from the payee, and in reaction to that send a message to the bank.

3.4.4.3 States

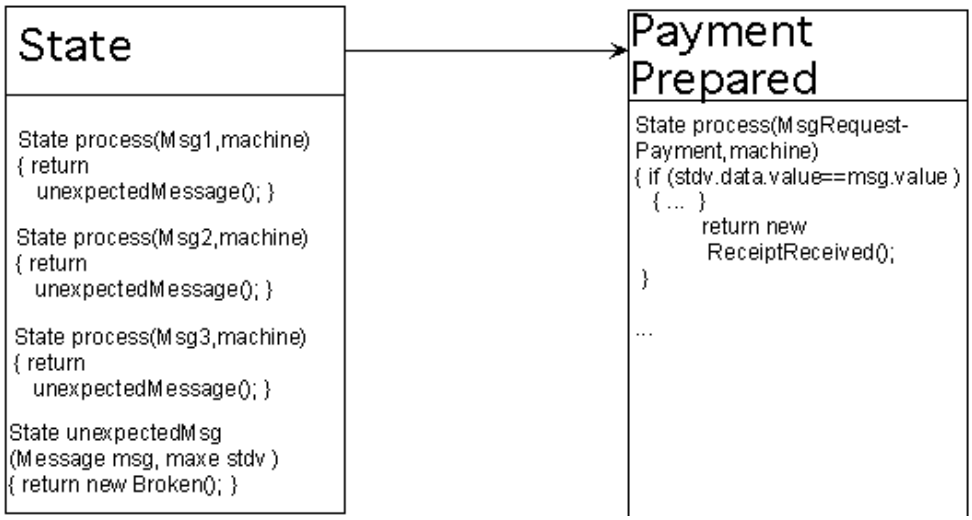

Figure 18: The State superclass and one of the subclassed states of the payment protocol.

The *State* class and its subclasses for the example depicted in Figure 18define the actual behavior of the machine. This class hierarchy contains all protocol states as well as the complete protocol behavior as it defines the automaton's transition function in its message processing methods.

Note that Java does not allow to override methods in subclasses (i.e., introduce new messages): Instead, when selecting the method whose signature fits the call, it seems that Java first searches for the most specific method in the *State* superclass and then checks whether this particular method has been overloaded without searching for further subclass methods that have a more precise or more general signature for the object to be passed into the state. As a workaround, all messages to be handled in any subclass have to be defined in the superclass of the states.

3.4.4.4 Message Processing, State Transitions, Default Error Handling

When the process method of the machine is called with a received message, the machine asks the message to dispatch itself to the current state for processing. This is done in the *dispatch()* method of the message. The indirection is needed since the type of the message *msg* sent into the machine is *Message*. So calling *process(msg)* will always invoke the *process(Message msg, Machine m)* method instead of the specific method for this subclass of message. If, on the other hand, *process()* is called by the message itself, the correct method of the current state is selected. While processing the message, the *process()* method can change the variables of the machine and return the subsequent state. Finally, the machine changes to the new state and outputs the message generated by the *process()* method.

The superclass *State* common to all states defines how messages are handled by default. Given a subclass of *State*, a message can be handled in the following ways:

- *Known and Expected messages:* For these messages, the state defines a *process()* method handling the messages. For accessing the machine's variables, it receives a reference to the machine as additional input. The method returns the subsequent state. If a message is required to be expected and processed individually by each state, its *process()* method must be declared *abstract* in the superclass. Depending on the intended scope of a process method, there are several ways to declare them:
  - *Processing a particular message in a particular state:* A method *process(MsgType mess, Machine m): Message* for the particular message type is declared in the particular state (such individual processing in each state can be enforced by declaring the method abstract in the state superclass).
  - *Identical processing of a particular message in all states:* A method *process(MsgType mess, Machine m): Message* for the particular message type is declared in the *State* superclass (different processing in subclasses can then be prevented by declaring the method final).
  - *Identical processing of all message in a particular state:* A method *process(Message mess, Machine m): Message* for the *Message* superclass is declared in the particular state.
- *Known but Unexpected Messages:* The superclass defines that all known but unexpected messages are handled by the method "*unexpectedMsg()*". If a given state wants to handle all unexpected messages differently, it may overload this method. Handling of some foreseen protocol errors does not involve unexpected messages - such errors belong to the case for expected messages.
- *Unknown Messages:* For messages which are now known to the protocol at all[14], the process method "*process(Message msg, Machine m)*" for general messages handles it. In our example, they are just ignored. Naturally, a state may overload this method to handle unknown messages differently.

In our example all known messages are processed as unexpected messages in the superclass and return the *Broken* state by default (this is done by the *unexpectedMsg()* method). Unknown messages are just ignored.

3.4.4.5 Design Options

We now describe some options for the pattern, which enable to adapt the pattern for different applications.

3.4.4.5.1 Extending the Machine

Adding a new state to an existing machine is done by just subclassing *State* and defining it as a follow-up state of another existing state.

[14] I.e., this message subclass has not been declared to the State superclass by defining a process(NewMsg msg) method in it but it has been defined somewhere else. If a state would define a method for a message without it being defined in the superclass, this method will not be found because the superclass processes all undeclared messages.

Adding new-defined message subclasses is done by defining a *process()* method in the superclass *State* and adding a *process()* method for this message type to all states in which this message is expected. Only messages that subclass *Message* are permitted.

3.4.4.5.2 Time-Outs

Many protocols define time-out periods on events. Time-outs are easy to add to our Java extended finite state machine: First, initialize a timer object with the time-out period and a reference to the machine (this could be a thread with a simple *Thread.sleep(n)* call). Then, after the timer reached the time-out, it sends a *MsgTimeout* message to the machine. Every state that cares for time-outs has to provide processing for time-out messages.

3.4.4.5.3 Two-Party Protocols

In a protocol with only two participants exchanging messages, information about the messages' origin doesn't have to be available. A protocol could easily be driven by feeding one machine's output to the other machine as input and vice versa.

A disadvantage of the synchronous, message-driven approach is that the machine is only able to send messages as responses to other messages. If more powerful communication is required, the machine may run as a separate thread having communication links to all machines that deliver input:

The environment may also register communication observers at the machine, which are then used for sending and receiving messages. In this scenario, the machines may be threads that run and communicate independently from the environment.

3.4.4.6 Sample Code

First, we present the *State* superclass.

```
/**
    State superclass for the StandardValue machine.

    The process() methods define the messages accepted by and the behavior
    of the states.

    StandardValue references are passed into process() to enable machine variable access.
*/
abstract public class State {

        public final static byte STOPPED   = -1;
        public final static byte COMPLETED =  0;
        public final static byte RUNNING   =  1;

         // All usable messages must be declared with default handlers.
        // Here, by default all messages are unexpectedMessage. This is handled by the method "unexpectedMessage"
```

```
        // Alternatively, each message may be handled by a different method.

        public State process( MsgPreparePayment msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        public State process( MsgRequestPayment msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        public State process( MsgTransferValue msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        public State process( MsgForwardReceipt msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        public State process( MsgComplete msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        public State process( MsgDone msg, StandardValue stdv ){
                return unexpectedMsg( msg, stdv );
        }

        // This method processes all messages unknown to the protocol.
        public State process( Message msg, StandardValue stdv ){
                        System.out.println( "Message " + msg +
  " not declared in State.java. Ignoring it!" );
                return this;
        }

        // This code handles messages that is not cared for in a particular State instance.
        // By default, it changes to the broken state.
        public State unexpectedMsg( Message msg, StandardValue stdv ){
                stdv.status = STOPPED;
                System.out.println( "Message " + msg + " not handled by the current
                                     state. Protocol broken." );
                return new Broken();
        }

        public String toString(){
                return new String( "State superclass." );
        }
}
```

3.4.4.7 Efficiency

One natural question is whether an object-oriented implementation, while being robust, is not efficient. Therefore, we compared our object-oriented implementation of the example with a direct implementation using nested `switch`-statements. The result on our machines (Sun Ultrasparc 10; Apple PowerMac 7600) was that the object-oriented variant is about half as fast as the direct implementation, i.e., the additional time needed for the object-oriented variant was at most $11\mu s$ per transition.

Since this test measures the overhead of the object-oriented state-*transitions* without computations, the overhead will be negligible in any protocol doing cryptographic computations. Therefore we feel that the robustness gained is well worth the performance loss.

Note that the pattern, for robustness, a new state object is created and initialized for each transition. This seems to be inefficient but our performance comparisons have shown that, for our example, this is as fast as returning references to existing state objects which are only created once.

3.5 Browser Framework

(B.Schoenmakers / DIC)

3.5.1 Introduction

The SEMPER client needs to facilitate several types of browsers. To accomplish a balanced approach for these different browsers, it was decided to implement a generic browser. The generic browser may be used for viewing (and possibly handling) SEMPER objects like payment transaction records, certificates, and more complex objects such as deals, which may consist of several transactions themselves.

3.5.2 Basic OO Design

The idea is to let all of the objects that are to be browsed implement the **`Browseable`** interface. The code that manipulates `Browseable` objects is concentrated in the **Browser** class. A SEMPER block that wishes to use the generic browser will first load the `Browseables` and then pass them to the generic Browser. From then on the user is able to browse these objects in a number of ways. To support the interaction with the user, a `Browseable` object provides methods, that make (part of) the contents of the object available to the Browser.

3.5.3 Scenario

The basic scenario for browsing looks as follows:

- The user launches the SEMPER browser by selecting it from the application menu on the Tinguin. It should be possible to select the type of objects to be browsed as well. This is currently done by having an entry for each type of objects, e.g., **payments**, **certificates**, **deals**, and so on. The user is allowed to have various of these browsing sessions open at the same time.
- For the selected type of objects, the responsible SEMPER module will be called to get a list of all relevant objects. These objects are all Browseable objects, which ensures that certain methods are available. An example of a basic method is `exportToHTML`(), which may produce a String formatted in HTML to be shown on the Tinguin. Another example is a method `exportToASCII`() that returns a String in ASCII. These methods will be called by the browser during the browsing process. The responsible module knows where and how the objects are stored and may return all of them in an array or Vector.
- Once the `Browseable` objects have been loaded by the browser, all of them or a suitable subset will be displayed in a main browser window. The browser window is equipped with several facilities that enable the user to walk through the objects, and to perform specific actions on them. The objects are sorted according to specified criteria, e.g., by date. This may give rise to more windows, or even completely separate sessions, that operate on user-selected objects.
- The normal way to stop browsing will be to quit the main browser window.

Other features of the browser are concerned with the export of the information to a file, such that the information can be used outside SEMPER. Future enhancements could support mechanisms that enable people to import SEMPER payment transactions into their financial (bookkeeping) applications.

The implementation of the Browser is simply using the Tinguin as is. However, it is relatively easy to replace this with an implementation that uses the Java AWT directly or other toolkits such as JFC.

In the latter case, the Tinguin may still be used to show possible error messages, hence even if advanced windowing is used, the Tinguin may still be used for some basic interactions (including starting up the browser application).

Technically the objects that implement a kind of browser for SEMPER should be subclasses of the browser class. Thus, when there is need for browsing through `Browseable` they can call the `interact` method from the `Browser` class. Using method overloading two options are offered concerning actions upon `Browseable` objects. The term actions includes various kinds of sorting, or several methods that depend on the `Browseable`. The two options are the following :

- All actions can be provided by the object that implements the `Browseable` interface. In that case the browser developer should call the method `public void interact(Browseable[])` of the `Browser` class. This is the preferred method.
- A different object that implements the `BrowseableActions` interface can provide the actions. In that case the browser developer should call the method `public void interact(Browseable[], BrowseableActions)` of the `Browser` class. This method is only provided in case the other method cannot be used.

3.6 Integration into the World Wide Web

(K.Tzelepis / INC)

The *SEMPER* trials will be based on the World Wide Web. Therefore, we have to define ways of how to integrate *SEMPER* into the existing infrastructure.

3.6.1 Client Side Integration

It is not feasible (manpower and skills) and not even desirable (trial users most likely already possess a browser they are used to and wouldn't like to learn a new one) that *SEMPER* builds its own general-purpose browser. Thus there is a need to integrate *SEMPER* in an existing browser.

There are several ways to solve this problem:

1. The *SEMPER* application is launched as a completely independent external MIME-viewer.
2. The *SEMPER* application is launched as an external MIME-viewer and communicates with the browser over a browser-specific client-interface.
3. The *SEMPER* application runs on the local host as a daemon (background process) providing an HTTP-server interface to communicate with the browser.
4. We integrate the *SEMPER* application/library inside a browser such as Netscape which provides a Java runtime environment.

Weighting generality (e.g., runs with every browser), portability (does not require fundamental implementation changes when switching platform), flexibility (e.g., does provide an easy way to pass back information to the browser) and ease of programming (problem of keeping data consistent when multiple *SEMPER* applications run in parallel) we came to the conclusion that approach 3) is the most promising.

3.6.2 Server Side Integration

Integration on the server side does not pose big problems. Most servers already provide an interface (in most cases CGI) which allows simple extension of a server's capabilities by forking a separate process. But as we already implement a minimal HTTP-server on the client side it's more natural to use the same approach on server side thus providing a very uniform environment.

3.6.3 The *SEMPER* Application

The decision taken in the last section leads to a SEMPER implementation architecture which is depicted in the following figure:

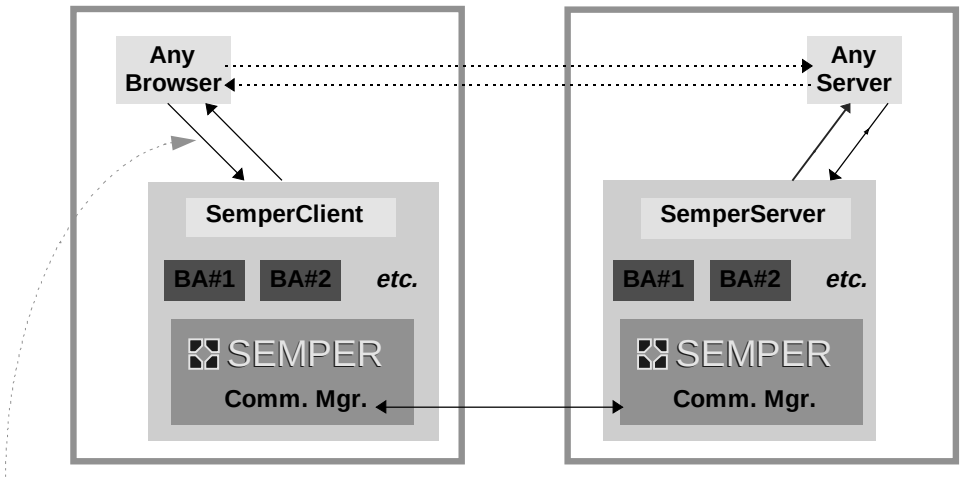

Figure 19: Integration into WWW

The main components of the SEMPER implementation architecture are the following:

- `SemperClient`
- `SemperServer`
- `Business Application`
- `SEMPER services`

3.6.3.1.1 The SemperClient Module

The `SemperClient` module provides http services to the Business Application framework. It starts an http server and accepts requests from an HTML client. The accepted requests are parsed in order to invoke the generic business application (BA) with appropriate information objects. The triggered business application is then executed in its own thread. Furthermore, `SemperClient` caters for the establishment of a communication framework (i.e. by making use of the SEMPER core services) which allows a local Business Application (BA) object and the corresponding BAobject on the server site to communicate. The `SemperClient` module consists of the following primary elements:

- the scheduler `SCTimer` class
- the monitor `SCMonitor` class
- the listener `SCDaemon` class
- several request parsers `SCHTTParser` class

Class: SCTimer

The main task of this object is to ensure that the duration of a continuos execution of a thread object will not exceed a given threshold. Furthermore this object offers some methods that allows a running thread to notify its state. The above state-notification messages are passed to the `SCMonitor` object.

Class: SCMonitor

The main task of the monitor is to initialise the `SCTimer, SCDaemon` and several `SCHTTParser` objects. Furthermore this class offers some methods that process the notification messages coming from the `SCTimer` object. The most important notification messages are:

- the current job has exhausted the execution duration
- the current job has run into a idle loop
- the current job hangs in an I/O call
- a new client request, via a HTTP browser, has arrived

Class: SCDaemon

This object is listening to a specific port on the local host in order to process a new client request, made by an HTTP browser to a specific Business Application. When a new client request is accepted, it notifies the `SCTimer` and stores the new created socket connection object to a common storage place.

Class: SCHTTParser

This object provides the following main functionality:

- read the HTTP request that comes from the socket stream
- parse the HTTP request made by the client (via the browser)
- call the BA Session Manager in order to get the requested BA object
- call the Commerce Layer and pass the HTTP request in order to invoke the appropriate BA on the server side.
- pass the new Commerce Layer object to the new local BA
- call the start-up method of the local BA

If no more pending requests are available the `SCTimer` object will be notified.

3.6.3.1.2 The SemperServer Module

The `SemperServer` module accepts http requests from the `SemperClient` and initiates the corresponding business application at the server site. The main task of the `SemperServer` is to establish an environment where a Business Application (BA) object on the client site and the corresponding local BA object at the server site are

able to communicate. The `SemperServer` module consists of the following primary elements:

- the scheduler `SSTimer` class
- the monitor `SSMonitor` class
- the listener `SSDaemon` class
- several request parsers `SSHTTParser` class
- several BA invokes `SSActionEngine` class

`SSTimer and SSMonitor do more or less the same as the corresponding classes in the client.`

Class: SSDaemon

This object is layered above the Commerce Layer in order to proceed a new client request for a specific Business Application. For a new client request this object notifies the `SSTimer` and stores the new created connection object to a common storage place.

Class: SSHTTParser

This object provides the following main functionality:

- parsing the HTTP-like request made by the client (via the new created Commerce Layer object)
- call the BA Session Manager in order to get the requested BA object
- store the parsing results and the BA object to a common storage place in order to be retrieved by a `SSActionEngine` object

If no more pending requests are available the `SSTimer` object will be notified.

Class: SSActionEngine

The `SSActionEngine` object is responsible for the execution of the requested Business Application object. Furthermore the new Commerce Layer object, created from the `SSDaemon`, will be passed to the new BA in order to allow peer to peer communication between the BA on the client site and the local BA (i.e. at the server site).

3.6.3.2 A SEMPER Scenario

3.6.3.2.1 Initiate a Request

A customer initiates a *SEMPER* business application (BA#1 in the figure) by issuing a *HTTP POST* request with the URL *http://localhost:1234/BA#1* where *1234* stands for a well-known TCP port number. This URL can either be stored in the browsers'

bookmark-file (e.g. for getting a list of outstanding transactions) or it can be part of an HTML-form which is prepared by the merchant to trigger a business transaction.

The SemperClient takes the HTTP request and initialize a communication channel via the `Deal` object (see Section 4.1 for details) to the `SemperServer`. The connection characteristics of the communication channel are preserved with the creation of a `Deal` object by the commerce layer. After the new connection is established, the HTTP request made by the customer is passed to the `SemperServer` site. The `SemperServer` and the `SemperClient` will parse the HTTP request and raise the appropriate BA locally. In order to allow the message interchange between Client-BA and Server-BA the previously established connection characteristics (i.e. `Deal` object) will be passed to both BAs.

3.6.3.2.2 Process a Request

Once started the business application can get all necessary information to initialise itself from the attribute-value pairs of the body of the HTTP POST (e.g. the name of the merchant, a description of the goods to buy etc.). The business application will then call *SEMPER* services to do the desired transactions. A simple application might for example call a `GetOffer` primitive from the commerce layer. The *SEMPER* core will then send this request with the help of the Communication Manager to the merchant. The merchant's previously registered business application (see previous section) will get this request from the *SEMPER* core, handle this request and send the offer back through the *SEMPER* core and the communication manager to the customer. The customers business application can now order the goods by calling `SendOrder` which will be processed in a similar manner. The final result can then be delivered back to the browser as HTTP reply (remember there is still a pending HTTP request).

For business applications, which have to be suspended or require additional browser interaction, the SemperClient will also allow to resume a transaction. This can be done by specifying a transaction-identifier (tx=23 in our example in the figure above). In that case the SemperClient will not launch a new business transaction but wakes up the suspended transaction and passes all parameters from the HTTP request to it.

3.7 Implementation Language

(G. Karjoth / ZRL)

As *SEMPER* has to run on a variety of platforms it is of utmost importance that all parts of *SEMPER* are as portable as possible. A clean abstraction mechanism to separate the different parts of the *SEMPER* architecture is a requirement for the implementation language as well as easy prototyping. The availability of (portable) standard libraries for graphics and communication is an additional criteria we measured an implementation language. Last but not least the implementation language should also offer sufficient performance.

Taking all of this into account in comparing languages such as C, C++, Python and Java we finally decided that, in order to create a uniform implementation environment, all new programs and applications for use in *SEMPER* will be written in the JAVA language [GoJoSt96].

As *SEMPER* uses JAVA simply as programming language the security problems of JAVA applets do not affect *SEMPER* at all.

The strong and clean (in comparison e.g., to C++) object-oriented language features of Java indeed proved to be very valuable in the project. Platform independence, in particular of the window subsystem plus a large number of existing libraries eased the development of the project. Unfortunately the implementations of JAVA seems still to be rather immature: The initial lack of good debuggers, numerous bugs and version upgrades, but in particular a lack of stability and quality in the platform independence of AWT (the window system) made our lives harder. Not surprisingly for a new interpreted language the performance was not very good and memory consumption showed to be a problem. The future will tell how Just-In-Time compilers and other compile time and runtime optimizations can overcome these problems.

4 Detailed Service Architecture

In the following sections the services and the design of different service blocks as introduced in Section 2 are described in details. The notation and conventions of the design descriptions are mostly based on UML [FowSco97].

4.1 Commerce layer

(S. Mjolsnes &R. Michelsen / STF)

4.1.1 Domain Description

The *SEMPER* commerce layer is located on top of the *SEMPER* core services. Trusted and not trusted business applications must use the commerce transaction service to implement concrete business scenarios and applications.

4.1.1.1 Commerce transaction service rationale

The rationale for the commerce transaction service in the *SEMPER* architecture is based on the following high level requirements:

Business commonalities Although the business applications will differ in layout and functionality, there will be common features. The commerce layer should provide these common business features. These common features will be offered as a set of abstract business services built on top of the lower layer services such as connection and transport services.

Business context One of the key common features of business applications is the requirement for maintaining a business context. The commerce layer transaction service should provide means for maintaining such a business context. This context generally comprises of an association between the parties involved in the business, the transactions exchanged between these parties, and private data stored by each party.

Secure service access point Both trusted and not trusted business applications will be built on top of the commerce transaction service. For untrusted applications the commerce transaction service represents the perimeter between the trusted and untrusted functionality of *SEMPER*, and a secure service access point must be provided. It must be impossible for untrusted business applications to use the commerce transaction service to perform actions that are not sanctioned by the user.

Enforce user security policy Business applications will request that the commerce transaction service performs security critical actions on behalf of the user. Business applications may request payments to be made or statements to be signed by the commerce transaction service on behalf of the user. The commerce transaction service must only perform such actions when they are in

agreement with the user's security policy. All critical actions must be authorised by the user either explicitly through a dialogue with the user or implicitly by giving applications rights to perform a limited set of such actions based on user preferences.

The basic *SEMPER* commerce transaction service will *not* provide services that assumes knowledge about specific business semantics such as contracts, orders, confirmations or invoices. Business rules will have to be implemented in the business applications built on top of the commerce services. We believe that it is impossible to standardise a small set of business services accommodating all kinds of businesses, and we justify this belief by looking at the messages defined for EDI. Currently, there are more than 130 different messages defined for EDIFACT and new messages are steadily defined, and still bilateral agreements on the precise interpretation of these messages must be established between the parties prior to exchanging EDI messages. This inflexibility is not fit for Internet commerce.Hence, our approach is to design a service based on very generic *commerce transactions* that can be extended, refined and interpreted according to the service provider's requirements.

We do not deny that providing specialised services with knowledge of the semantics of some common business scenarios will be useful. However, it must be possible to realise business scenarios without such advanced services in a truly flexible architecture. Such advanced services can be provided as an extension to the core *SEMPER* commerce transaction service. Such extensions can include transaction types for different common business scenarios such as purchasing applications, hotel booking or tendering for public contracts.

4.1.1.2 The commerce deal

To support the commerce transaction service we introduce the key concept of a *commerce deal* in the commerce layer. A deal encapsulates a business context and comprises a representation of the association between the participants, the transactions that haven been exchanged, and private data stored by each participant.

A deal can be accessed in two main states. An *active deal* establishes an active connection between the participants of the deal, and adding new commerce transactions to it can modify the deal. All participants must participate actively for a deal to be accessed in this state. A *suspended deal* has no such active connection, and it can only be used for inspection of the business context. Any participant can access a deal in suspended mode without the active participation of any other entities.

The active and suspended states of a deal are only two different ways of accessing a deal and do not influence the deal itself. An active deal is similar to opening a file in read-write mode, and a suspended deal is like opening a file in read-only mode. A deal can always be accessed in suspended state, and it can be activated or reactivated as long as the peer is also willing to do so. Several processes can access the same deal simultaneously, but only one process can access the deal in the active state at any time.

A deal is an abstract concept, and the commerce transaction service does not assume anything about the semantics of the deal itself or the transactions that it contains. A deal can be a long lived object representing for instance a bank account where the

transactions can be deposits and withdrawals to this account, or the deal can represent a single visit to an on-line store. It is up to the business application and the user to provide a meaningful interpretation of deals.

4.1.2 Requirements

In this section we will describe the functional and non-functional requirements to the commerce transaction service. We use use-case modelling as defined in the UML. First we identify the different actors that are using the commerce transaction service in some way, and then we describe the various use-cases. Use-cases are summarised in Figure 20. Finally, we discuss non-functional requirements to the commerce transaction service.

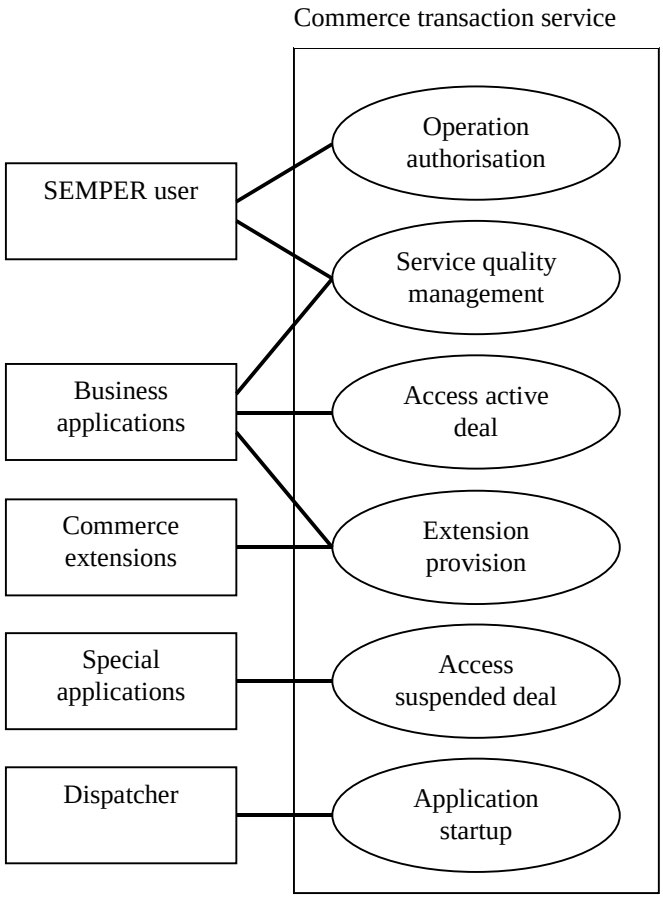

Figure 20: Use-cases for the commerce transaction service.

4.1.2.1 Actors

The actors that will use the commerce transaction service are:

Business applications Both trusted and not trusted business applications will use the commerce transaction service. Business applications will need to establish a session to their peers and to exchange commerce transactions.

Special applications Some special applications for management of deals will exist. One example of such an application is the deal browser.

The *SEMPER* user This is the actual person using the *SEMPER* system either as a service consumer or as a service provider. The user must have full control of security critical operations that is performed on behalf of the user.

Commerce extensions It is likely that extensions to the core commerce transaction service will be developed offering services that are more oriented towards specific business scenarios and with more assumptions about the semantics of commerce transactions.

Dispatcher The dispatcher module is responsible for accepting requests from a client side business application and starting the corresponding business application on the server side. The dispatcher interacts with the commerce transaction service to find out which business application to start.

4.1.2.2 Access to active deal

Normal business applications need access to active deals. Business applications must be able to create new active deals and to reactivate existing deals. All participants in a commerce deal must co-operate for an active deal to be successfully activated.

An active deal supports operations for inspection and modification of private application data. Participants can only access their own private data. Furthermore, an active deal supports operations for inspecting the transactions in the deal and to exchange new transactions.

4.1.2.3 Access to suspended deal

Special business applications, e.g. the deal browser, need access to suspended deals. A deal cannot be modified when it is in the suspended state. To perform such operations the deal must be activated as discussed in the previous section.

A suspended deal can be opened independently of any participant in the deal without the co-operation of any other participants. A suspended deal supports operations for inspection of deal data and inspection of the transactions that have been exchanged in the deal.

Deals are normally kept in the *SEMPER* archive. However, it is possible to access an externally stored deal in suspended mode. Deals can also be deleted.

4.1.2.4 Service quality management

Service quality settings can be set for deals and transactions. Both the business application and the *SEMPER* user have requirements to the service quality settings, and the quality settings that are actually used are the result of a negotiation between these two actors.

Service quality settings will influence authentication of participants, confidentiality of transactions with respect to third parties, non-repudiation of transactions and so on.

4.1.2.5 Operation authorisation

Some security critical operations require authorisation by the *SEMPER* user. Such authorisation can be given explicitly by carrying out a dialogue between the user and the *SEMPER* system. The system presents information about the operation in question, e.g. a payment transaction request, and the user authorises or refuses the operation.

Operations can also be authorised implicitly based on a combination of preferences set by the user and capabilities given to business applications based on the trust level assigned to the application.

4.1.2.6 Provision of extensions

The core *SEMPER* commerce transaction service provides only very generic transaction types with little assumption about business semantics. A framework is provided so that business applications and extensions to the core commerce services can extend this with transaction types tailored to specific business scenarios.

4.1.2.7 Application startup

The server side dispatcher accepts requests from a client to start a server side business application. Such requests are generated when the client activates a deal. The commerce transaction service provides functionality for the dispatcher to retrieve necessary information from the client about which business application to start on the server side.

4.1.2.8 Non-functional requirements

The commerce transaction service must provide a secure service access point for business applications to access the *SEMPER* services. It is particularly on the client side that the operation of the business applications must be checked. In the general case the user of the *SEMPER* system will not trust client side business applications. Such applications must be denied access to some of the services, such as inspection and deletion of random information. Other services, such as payment transactions, can be accessed after user authorisation.

Some business applications will be partially or fully trusted. The trust level can be based on signatures attached to business applications and user preferences. A trusted application is given capabilities to access more of the security critical services with less user interaction.

4.1.3 Design Overview

4.1.3.1 Class diagram

The realisation of the commerce transaction service is based on three main class hierarchies and some additional helper classes.

Deal state A class hierarchy for maintaining the state of a deal is provided. These classes are internal to the commerce transaction service realisation and cannot be accessed directly by applications.

Deal manager This class hierarchy provides functionality for accessing and managing deals. Deals can be managed in active or suspended mode.

Transactions Finally, a class hierarchy representing the different types of commerce transactions is provided. The core commerce transaction service provides only very general transaction types, but business applications and extensions can provide additional transaction types tailored to specific business scenarios.

A class diagram showing the most important classes in the realisation of the commerce transaction service is depicted in Figure 2.

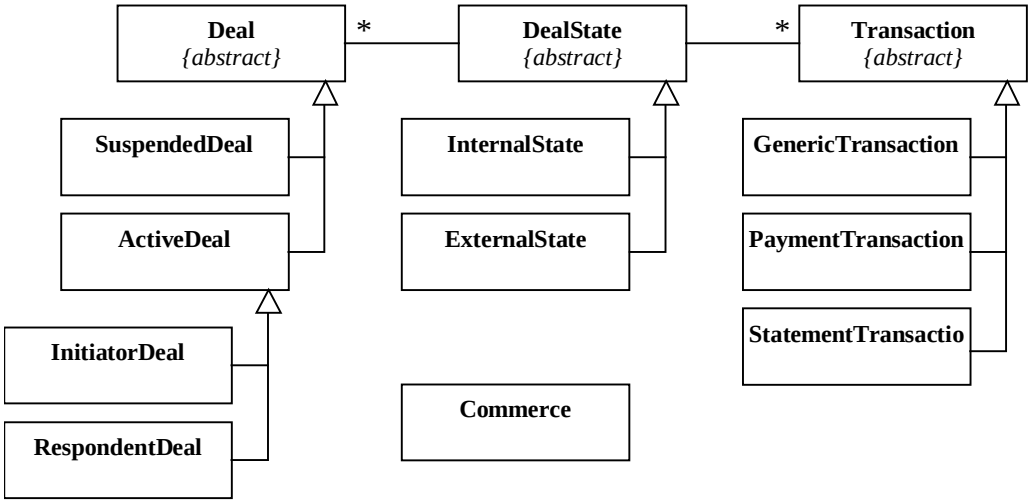

Figure 21: Class diagram for the commerce transaction service.

The hierarchy with the class **Deal** at its root constitutes the deal manager classes. A deal can be accessed in active or suspended mode by using the **ActiveDeal** or **SuspendedDeal** class respectively. An application must assume the role of initiator or respondent (client or server) when accessing an active deal. These roles are represented by the classes **InitiatorDeal** and **RespondentDeal** respectively.

The class hierarchy beginning with **DealState** constitutes the classes for managing the state of a deal. These classes are invisible to applications using the commerce transaction service and can only be accessed via the deal hierarchy. A deal state can be stored internally in the *SEMPER* archive or externally as a file. These two cases are managed by the classes **InternalState** and **ExternalState** respectively.

The **Transaction** class hierarchy encapsulates commerce transactions. Three very generic transaction types are defined by the core commerce transaction service.

GenericTransaction This transaction type can be used by applications for general data transfer between the client and server part of the application. No special processing of the data is performed by the commerce transaction service.

PaymentTransaction This transaction type is used for performing payments from one participant to another. Payments require authorisation by the user.

StatementTransaction This transaction type is used to send a signed statement or message from one participant to another. Statements provide non-repudiation of origin. Statements require authorisation by the user.

Applications or extensions to the core services can provide additional transaction types tailored to specific business scenarios. These new transaction types are subclassed from the existing transaction types.

The **Commerce** class is a static class providing some operations that don't fit naturally in any of the other classes. Some of these operations are only for internal use in the commerce transaction service, but this class also provides methods that can be used by the dispatcher to determine which business application to start when a client attempts to activate a deal.

4.1.3.2 Representation of a commerce transaction

An application specific transaction type is defined as a subclass of one of the predefined transaction types as shown in Figure 22. In this example the **ApplicationPayment** class is derived from the **PaymentTransaction** class which itself is derived from the **Transaction** class. Each level in this class hierarchy adds some attributes to the transaction.

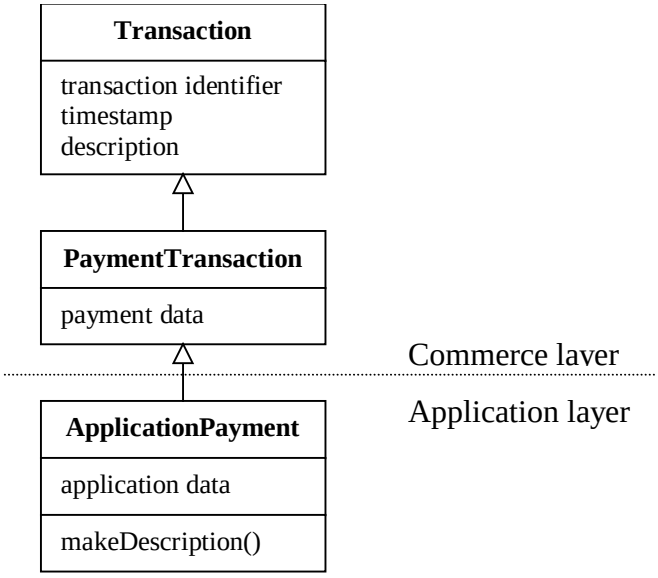

Figure 22: Class diagram for application specific transaction.

Only the **Transaction** and **PaymentTransaction** classes are parts of the commerce transaction service. Thus, the commerce transaction service cannot assume anything about the semantics of the attributes added by the **ApplicationPayment** class. However, these attributes must be presented to the user in a secure way for the user to interpret these attributes in the context of the business being performed and to authorise the transaction.

To achieve this the **ApplicationPayment** class should override the inherited method `makeDescription` that provides a textual interpretation of the transaction. This method will be invoked from the **Transaction** class and the result stored in the description attribute. This textual description can be presented to the user for

interpretation and authorisation, and it is the textual description that will be signed and constitutes the "contract" between the participants of the deal.

When the transaction is requested both the application data and the description are sent to data to description by invoking `makeDescription` and verify that there has been no tampering with the the receiver. The receiver must once again perform the transformation from application description or the application data. The receiver can then interpret the application data and act accordingly knowing that the peer has committed to this by signing the description.

4.1.3.3 Quality of commerce service

A commerce quality of service can be set for a deal. The current design is restricted to some security-related quality of service settings. The attributes that have currently been specified are summarised below.

Authentication This attribute ensures strong authentication of the parties involved in the deal. Authentication can be specified for the individual participants so both one-sided and mutual authentication is supported.

SECA[15] Specifies whether the parties are required to be committed to SECA, the secure electronic commerce agreement. SECA commitment can be specified for individual participants.

Confidential Specifies that all transactions exchanged in this deal will be protected by encryption for confidentiality.

Payment receipts This attribute ensures that a payer always receives a receipt for all payment transactions in the deal. The receipt can be shown to a third party in case of dispute and cannot be repudiated by the payee. The receipt only signifies that the payee *received* a payment. The receipt does not commit the payee in any other way.

Statement receipts This attribute ensures that the sender of a statement always receives a receipt. The receipt can be shown to a third party in case of dispute and it cannot be repudiated by the recipient of the statement. The receipt only signifies that the statement was received. It does not commit the recipient in any other way.

Statement and payment receipts are exchanged using the fair exchange mechanisms of the transfer layer. This is currently the only way that the fair exchange mechanism is used by the commerce transaction service. Support more general fair exchange (e.g., payment for goods) requires some more research.

Commerce quality of service is negotiated between the participants of the deal during the creation of the deal. This is a description of how we envision the interaction between users of the business application, the commerce layer client and server, and the users themselves by dialog windows.

[15] The concept of SECA is described elsewhere in this report.

The business application can, by a request to the deal constructor method of the application program, set the quality attributes.The deal constructor method will pop up a dialog window and ask the user to select the correct quality attributes required for this deal instance. This window will show the values that the business application client proposes, and it is now for the user to modify and acknowledge the parameter settings. The *SEMPER* client will use this user advised setting in the communicated request to the *SEMPER* server. The server will show the requested values in a dialog window, now for the user of this server to acknowledge that this is an acceptable choice of values, or else to reject this parameter value request as not acceptable. The server rejects by telling what the values should look like and returning this to the client by throwing an exception and closing the communications. The dialog window for specifying and acknowledging quality attributes can be replaced by pre-set user preferences.

4.1.3.4 Deal activation

A deal is a context association between a client and a server where commerce transactions can take place. Creating this association involves activating a new deal. Activating an existing deal amounts to re-establish the association between a client and a server. The client is the initiator in activating a deal. The activation includes connection establishment, opening the local archive, and negotiating the quality of service. This functionality is implemented in the constructors of **InitiatorDeal** and **RespondentDeal**, subclasses of **ActiveDeal**.

Activation of a deal is always initiated by the business application that “owns” the deal. A future enhancement of *SEMPER* is to also permit reactivation of deals from special applications such as the deal browser. This will require modifications to the reactivation protocol both in the commerce layer and in the business applications.

4.1.3.4.1 Client side

On the client side the following is happening in the creation of an active deal association/context. The client user is asked to fill in and accept the quality attributes advised for this deal instance. These commerce layer attributes are then mapped to the TX-layer attributes in the association establishment request. When done the client enters CONNECTED state.

In a re-activation of an existing deal, the user could merely be asked to indicate the name of the deal. However, this is not implemented due to the constraint that a deal context does not survive a communication session, so a new deal is created with every new communication association. A single message exchange between client and server is done in this respect.

After the creation of a new deal has taken place, the client sends a quality proposal to the server and enters NEGOTIATING state. The reply from the server will indicate whether the server could match the requested security policy. If match is possible, then the client enters DEALING state, establish the rest of the necessary local administration of the deal instance, and returns the control to the client business application. If policy mismatch occurs then the client closes down the association and returns accordingly to the business application.

4.1.3.4.2 Server side

The deal activation functionality of the server is implemented in the **RespondentDeal** object. The commerce layer **RespondentDeal** object receives the established **TXContext** as input, and enters CONNECTED State. Then it receives a message indicating which deal to activate from the archive, or whether a new deal is to be created for this association. As already explained in the client side section, in the current implementation the outcome of this interaction will always be a new deal created.

Next it receives the requested policy of security parameters. This request is displayed to the (server) user. The proposal can be matched if the user only fill in ANY values, that is, values that are not assigned by the client user. The server user states a mismatch if some of the assigned security parameter values have to be changed. Anyway, the resulting set of values is returned to the client. If the server cannot accommodate the requested policy, it closes down after returning the reply. If there is a match then it enters the DEALING state, establish the rest of the necessary local administration of the deal instance, and returns the control to the business application, that is, the business application server.

4.1.3.5 Transaction request and indication

Figure 23 shows the activity diagram for a transaction request. A transaction is requested by invoking the `request` method of the transaction object. A transaction description is then generated as described in Section 4.1.3.2. It is then checked whether the transaction requires explicit authorisation by the user or not. If authorisation is required, then a dialogue box with information about the transaction is displayed and authorisation is requested. Such a dialogue box will look like the example in Figure 24.

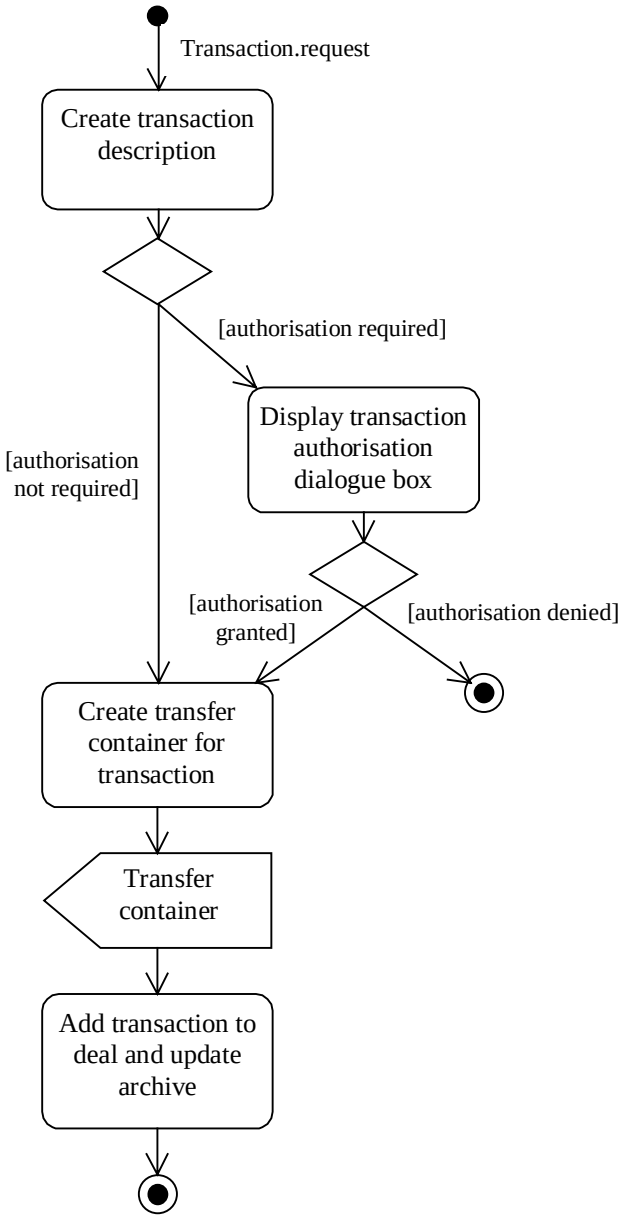

Figure 23: Activity diagram for a transaction request.

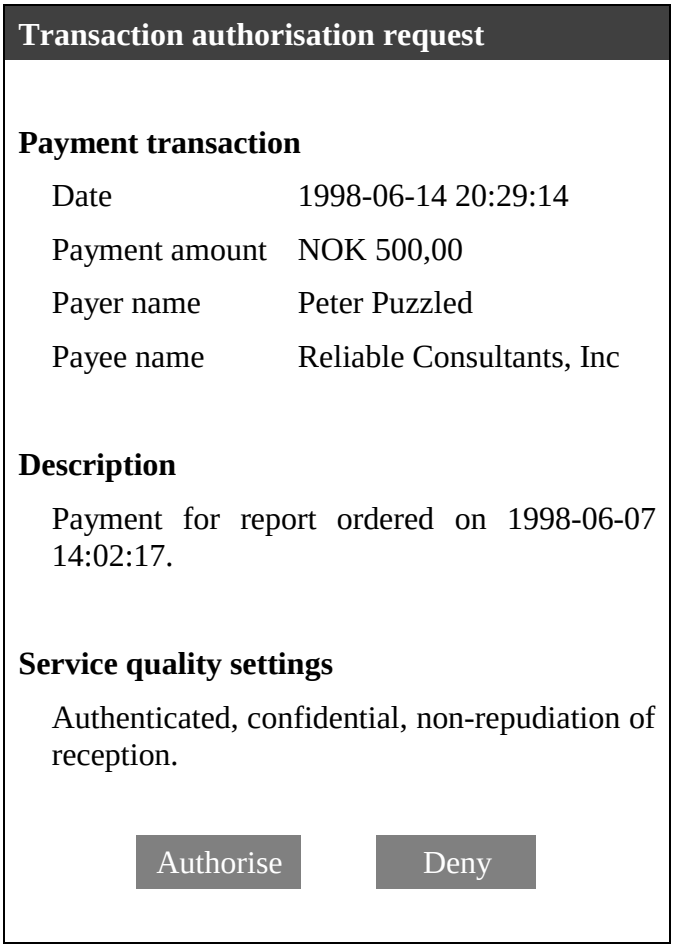

Figure 24: Dialogue box for authorisation of transaction.

Providing that authorisation is granted, a transfer container is created containing the transaction. (The object class of transfer container is part of the service of the txlayer, and is explained in Ch.4.2.) Quality attributes must be set according to the transaction type and the service quality attributes in effect for the deal. The transfer container is then sent to the peer. Finally the transaction object is added to the deal transaction history and the archive is updated.

A transaction request will manifest itself as a transaction indication to the peer. The activity diagram for a transaction indication is shown in Figure 25. A separate execution thread is constantly waiting for transfer containers sent by the peer. When a container is received, it is decoded into a transaction object and put into a transaction queue.

An application queries the commerce transaction service for transaction indications by invoking the `indication` method of the deal object. This method will retrieve the first transaction from the transaction queue and verify that service quality settings satisfy the settings for the deal and that the transaction description matches the description generated locally as described in Section 4.1.3.2.

If explicit acknowledgement of the transaction is required, a dialogue box with information about the transaction is displayed and acknowledgement is requested from the user. When the transaction is acknowledged it is added to the deal transaction history and the archive is updated.

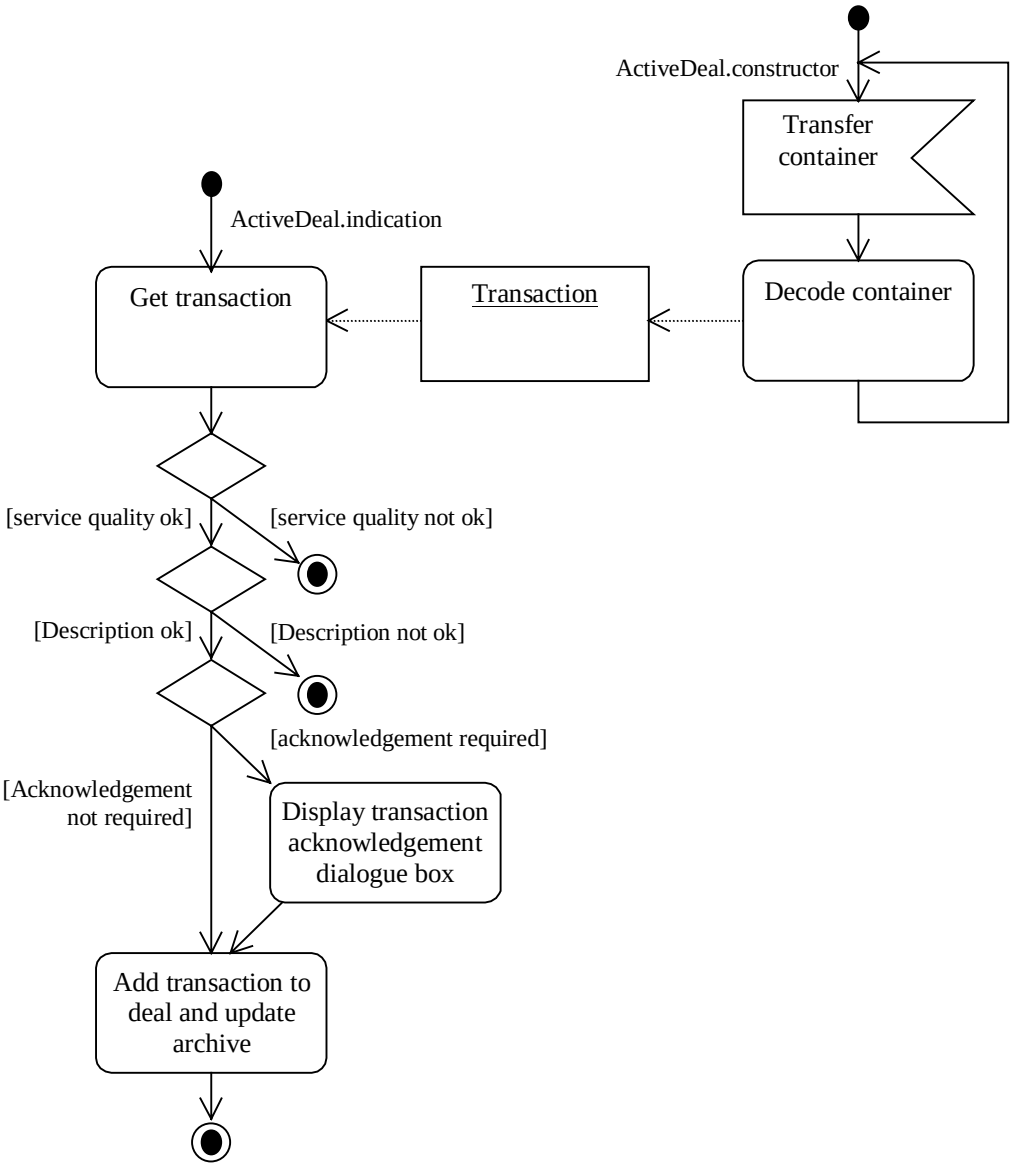

Figure 25: Activity diagram for transaction indication.

4.1.3.6 Relations to other modules

Figure 26 shows the relations between the commerce transaction service and other modules in the *SEMPER* architecture. Below these dependencies are briefly described.

Business applications Applications use the commerce transaction service to manipulate commerce deals and transactions.

Transfer layer The commerce transaction service uses the transfer layer as a transport service for the commerce transactions. The transfer layer provides a uniform service interface for transferring different types of transfer transactions such as generic data transfer and payment transfers.

Archive The archive is used for persistent storage of commerce deals and

transactions.

TINGUIN The TINGUIN provides a secure user interface that is used to display transactions and deals for authorisation and acknowledgement by the user.

Access control The commerce transaction service provides a secure service interface to trusted and not trusted applications. The capability based access control module is used to discriminate between trusted and not trusted applications.

Preferences The preferences manager is used to set policies for deal service quality and user authorisation and acknowledgement.

Dispatcher The dispatcher uses the commerce transaction service during initialisation of business applications on the server. The commerce transaction service tells the dispatcher which business application it is supposed to start to handle the request from the client.

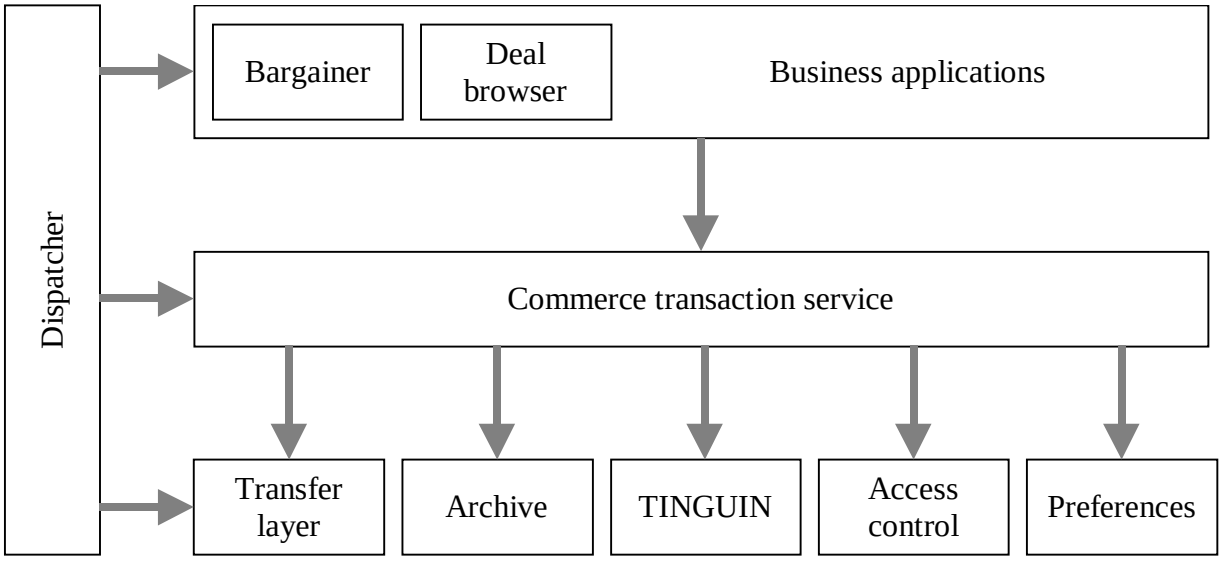

Figure 26: The commerce transaction service and relation to other modules.

4.1.4 Related Work

4.1.4.1 Java Commerce Client

The Java Commerce Client provides an infrastructure for hosting business applications that is somewhat similar to the *SEMPER* commerce services [Gold98, Sun98]. The services provided by the Java Commerce Client (JCC) are:

Wallet GUI This is a framework for a graphical user interface for user interaction. The *SEMPER* commerce service does not provide a similar service for graphical user interaction. The business application can use the browser to interact with the user and additionally the TINGUIN is used for trusted interaction between

the user and the trusted parts of the *SEMPER* architecture.

Business metaphors JCC intends to support a set of business metaphors such as purchase, home banking etc. These metaphors are rules for conducting business. Currently only purchase is supported. The *SEMPER* commerce service does not define such metaphors or business rules, but an infrastructure for adding such rules is provided. Rules can be provided as extensions to the basic commerce service either by business applications or as generic commerce layer extensions.

Secure API Both the JCC and the *SEMPER* commerce service provides a secure service access point for business applications.

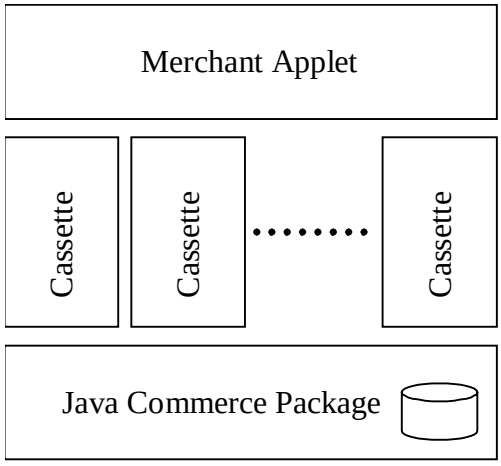

Figure 27: Simplified architecture of the Java Commerce Client.

A simplified view of the Java Commerce Client architecture is shown in Figure 27. This shows the merchant applet, which corresponds to *SEMPER* business applications, on top. This applet use the services provided by a number of installable *cassettes* and a set of basic services provided by the Java commerce package.

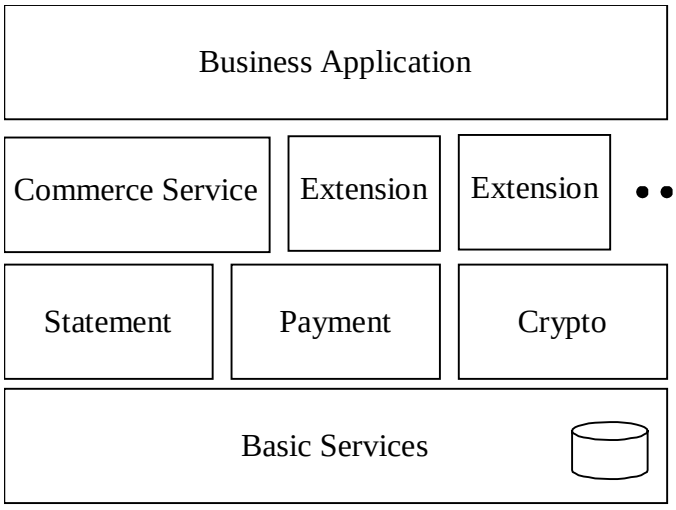

Figure 28: Simplified *SEMPER* architecture.

This can be compared to the simplified *SEMPER* architecture shown in Figure 28. The cassettes in JCC corresponds to the commerce service extensions and the specific payment systems attached to the payment manager. In the *SEMPER* architecture the

payment instruments are buried deeper in the layers than in JCC, and JCC merchant applets has a less abstract view of payment instruments than a *SEMPER* business application.

4.1.4.2 Other initiatives

Other initiatives for electronic commerce, such as OBI and OTP, concentrate mainly on electronic purchasing. This is a much narrower application than the goal of *SEMPER* and Java Commerce Client that aims to cover all kinds of electronic commerce including advanced services such as home banking, auctions, and games.

4.1.5 Self-assessment

This section briefly discusses to what extent the current design of the *SEMPER* commerce transaction service meets the requirements presented in Section 4.1.2. We perform this self-assessment by briefly commenting on each of the use-cases that were presented.

Access to active deals The commerce transaction service supports operations on active deals. These operations include creation and reactivation of deals, deal and transaction inspection, and transaction exchange between the participants of the deal.
Currently a deal can only incorporate two business partners. In a future version of the commerce transaction service deals with more than two active participants should be investigated. Note that the current design supports use of third parties, such as payment system operators and certification authorities, but these parties cannot take a primary role in a deal.

Access to suspended deals Operations on deals in suspended mode are supported. The primary operations on a suspended deal are inspection of the deal and transactions that it contains.

Operation authorisation The design of the commerce transaction service supports authorisation of security critical operations by the user. Discrimination between trusted and not trusted business application is based on the capability-based access control mechanism in *SEMPER*.
Although authorisation of operations is supported by the *design*, it is currently not supported by the actual *implementation* as described in Section 4.1.6.2.

Provision of extensions The commerce transaction service provides a class hierarchy that can be extended by business applications and special commerce extensions to provide support for transaction types with more knowledge about business semantics.

Application start-up Services are provided for the dispatcher to obtain information about which business application to start when a client connects to a *SEMPER* server.

Non-functional requirements The design of the commerce transaction service

specifies a secure service access point based on the use of the capability-based *SEMPER* access control module to restrict access to security critical operations. Although this requirement is supported by the *design*, it is not fully supported by the current *implementation* as described in Section 4.1.6.1.

The present self-assessment of the commerce transaction service was made without any real experience with actual use of the service. Business applications that will use the service are still being implemented, and the business applications that have been specified for the trial will not test the full set of features and the flexibility of the commerce transaction service.

4.1.6 Implementation notes and Recommendations

There are a number of discrepancies between the design of the commerce transaction service as described here and the actual implementation used in the *SEMPER* trials. These discrepancies are briefly summarised below.

4.1.6.1 Access control

The commerce transaction service offers a secure service access point to business applications, and it is a perimeter between trusted and not trusted parts of the *SEMPER* architecture. The commerce transaction service uses the *SEMPER* access control module to discriminate between trusted and not trusted business applications, and prevents not trusted applications from performing security critical operations.

Access control has not been implemented for the commerce transaction service in the current *SEMPER* implementation. Access control is not integrated in any of the other *SEMPER* modules and then it is impossible to fully support access control in the commerce transaction service at this point.

The service interface of the commerce transaction service has been specified with access control in mind, and access control capabilities are required as parameters in many of the methods. These capabilities are however not used in the implementation and even null references will be accepted.

When access control is lacking all business applications must operate on the same level of trust. For the trial only trusted business applications will be implemented, and hence the current implementation of the commerce transaction service assumes *all* business applications to be fully trusted.

4.1.6.2 User authorisation of critical operations

The design requires that the user authorises some critical operations that are requested by non trusted business applications. The commerce transaction service displays a dialogue box with information about the operation, and the user must either choose to permit or refuse the operation.

In the current implementation all business applications are considered fully trusted, and hence authorisation of critical operations is not necessary. The lack of integrated access control makes it impossible to discriminate between trusted and not trusted applications. Hence, the functionality for requesting authorisation of critical

operations is not implemented in the current *SEMPER* version.

4.1.6.3 Reactivation of deals

A deal can represent a long-term relationship between business partners, and a deal must be permitted to span several communications sessions. An existing but closed deal can be reactivated whenever new commerce transactions are to be exchanged between the participants of the deal.

In the *SEMPER* demonstrator it is only the Fair Internet Trader business application that will activate deals, and the Fair Internet Trader specifies the lifetime of a deal to correspond to the communications session between the business partners of that deal. Hence, in the trial deals will never be reactivated and this functionality has not been implemented.

4.2 1.1 Transfer & eXchange Layer

(T. Beiler/SRB, M. Schunter /UDO)

4.2.1 1.1.1 Domain Description

This section gives an introduction to TX-Layer. The TXLayer contains the two blocks Exchange Block and Transfer Block. For simplicity and since both blocks share some technical mechanisms, the logical distinction between the two blocks is not done strictly in this chapter. Moreover and to take this into account, we speak of services only instead as of blocks.

The basic command the TX-Layer evaluates is to transfer or exchange an item under consideration of a certain set of security relevant attributes. Examples of such items are payments (see Section 4.3), statements (see Section 4.8), and valuable digital data like images. An example of a security attribute to be considered within a TX-Layer service is *confidentiality*, meaning that the data involved in the service should only be accessible to an exactly determined group of parties. Another example is *anonymity*, meaning that a party wishes to stay unidentifiable to other parties.

The TX-Layer offers the two services transfer and exchange on behalf of a service *requester*. In SEMPER this is mainly the Commerce Layer. A *transfer* is the sending and receiving of one item from one party to another. It consists of two local algorithms, one for sending and one for receiving the item, and correspondingly, the participants have the roles *Sender* and *Receiver*. (Note that "sending" is in fact not always trivial, e.g., sending money means that a paymnent protocol must be executed.)

Within an *exchange*, two transfers of two items are carried out between two parties in opposite directions. The two transfers are coupled: each party starts to send only if the counter party has started its corresponding transfer, and in the end either each party either is satisfied with the received item or no party has received anything valuable. This property of an exchange is called *fairness*. For protocols which achieve this property even in the presence of mutual mistrust see Section 5.1. The two participants of an exchange have the roles *Originator* and *Responder*, respectively

A TX-Layer service operates on *items*[16]. An item can be any arbitrary object, it only has to be compatible to either TX-Layer service, i.e., it actively has to take part in certain steps of a service. Items a TX-Layer service operates on may be payments, statements, or digital images, and so on; the term item abstracts from all this. A TX-Layer service is intended to operate especially on a data structure called *Container*. With containers, through recursive nesting a tree of items can be built, so that a TX-Layer service can be applied to the whole structure at once.

In SEMPER, a transfer is more than just transmitting data in one go, a transfer is a *protocol* in the sense defined by [MeOV97]. The same holds for exchange. Thus, executing a TX-Layer service means usually running a certain protocol.

16 Note that *good* is very similar to item. The term item is chosen since the TX-Layer has no understanding of of commercial values, while the term good implies a commercial notion.

Before a transfer or an exchange actually takes place, the exact protocol and its parameters are to be chosen. Since there are two participants who have to agree on this, this is a *negotiation*. A negotiation is again a protocol. It follows that transfer and exchange have two phases, the negotiation and the actual service protocol execution. In the design presented here, these two phases are encapsulated in a high level view onto transfer and exchange protocols.

A TX-Layer service is seen as a *transaction*. A transaction is an action or a state transition from exactly one state into another, and with no intermediate visible states. This property of a transaction is called *atomicity*. More properties of transactions can be found in [LMWF94, GrRe93].

A TX-Layer service can be required to meet a number of security objectives within its execution. These objectives are expressed with *Security Attributes*, which are collected and passed to the service with an *Attribute Set*. Security attributes are bound to a party, and each involved party can determine attributes the respective party desires to be met. So far, the TX-Layer supports the SecurityAttributes authenticity, confidentiality and non-repudiation, while fairness and anonymity are processed but not evaluated. With attributes which concern modules below the TX-Layer, the layer enforces those modules to consider them; the attribute set is passed through (chain-of-responsibility pattern [GHJV95]). [17]

4.2.1.1 1.1.1.1Requirements

The overall requirements the TX-Layer has to meet are:

- The TX-Layer service should offer a general interface (API) to the service requester via the Manager-Module concept of SEMPER. The internal circumstances of a service execution should be kept as abstract and hidden as the requester desires.
- The TX-Layer should be extensible in such a way that new parts can be added without updates to the TX-Layer framework (SPI).

To fullfill these requirements the TX-Layer should be a framework, i.e., it provides (abstract) classes to derive from in order to combine external objects with functionality located in the framework, further classes to coordinate the behaviour of the framework services and support classes for common behaviour. For more on frameworks, see [GHJV95].

More of the requirements can be derived from the introduction in the previous chapter. The following two use cases of transfer and exchange give a functional view on the service requirements.

[17] It should be considered that passing the attribute set requires attention to keep security goals, regarding the visibility to the service modules. This has not exhaustively been done yet.

4.2.1.2 1.1.1.2Use Cases

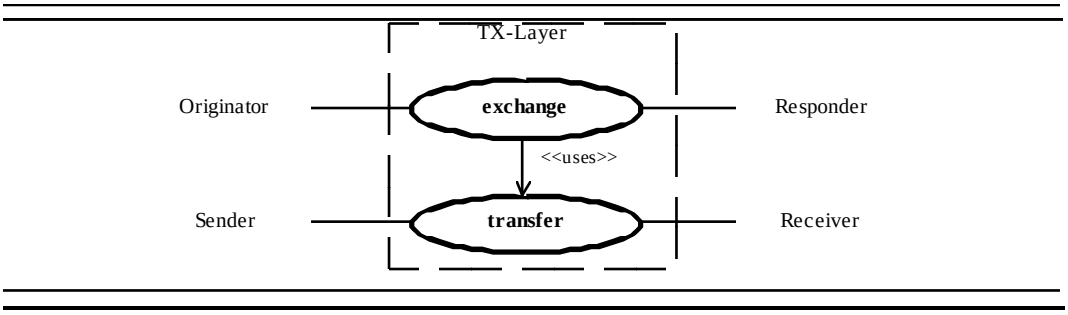

Figure 29: Use Case Diagram

This section contains descriptions of the two use cases of the TX-Layer, transfer and exchange. A transfer has two participants which behave differently and is therefore split into two views. An exchange is symmetric, hence only one side needs to be shown.

The abstract transaction framework that is also implemented in the TX-Layer is only presented in the design.

4.2.1.2.1 1.1.1.2.1 Use Case Transfer

The Sender‘s View

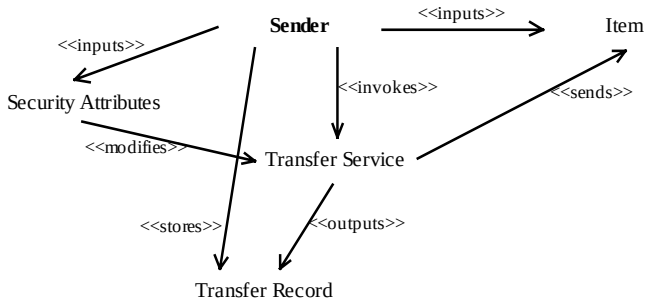

Figure 30: Use Case Transfer, the Sender’s View

These are the things a sender does: The sender has a reference to an item which it wants to transfer to the receiver. It might also desire a number of security attributes to be applied to the transfer. Thus, it invokes the transfer service and inputs the item’s reference and the attributes to the service. Then the sender expects the service to transfer the item on its behalf. As a result, the transfer service outputs a transfer record which contains data logged at the sender’s side and describing the sender’s view of the transfer process. The sender gets the resulting transfer record and stores it. The sender might want to show it as evidence in case of a dispute.

The Receiver‘s View

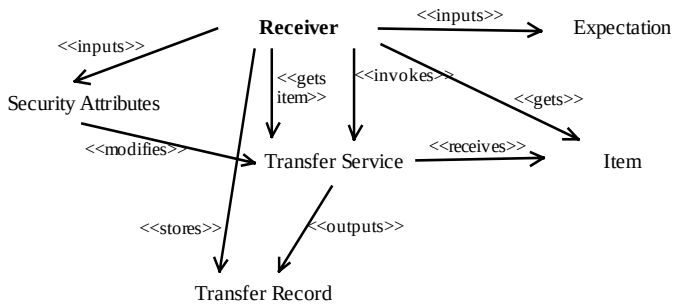

Figure 31: Use Case Transfer, the Receiver’s View

These are the things a receiver does: The receiver has an expectation of the item it wants to receive from the sender within the transfer. This expectation also expresses what the receiver does not want to receive. The receiver too might desire a number of security attributes to be applied to the transfer to meet certain security objectives, e.g., authenticity of the sender. To start the transfer service, the receiver inputs the expectation and the security attributes. Then, the receiver expects the service to receive an item on its behalf. As result, the transfer service outputs an item and a transfer record which contains data logged at the receiver’s side and describes the receiver’s view of the item transfer process. The receiver gets the resulting transfer record and stores it. The receiver might want to show it as eveidence in case of a dispute.

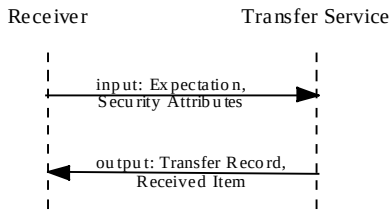

Figure 32: Interaction Sequence Diagram, Receiver and Transfer Service.

Use Case Exchange

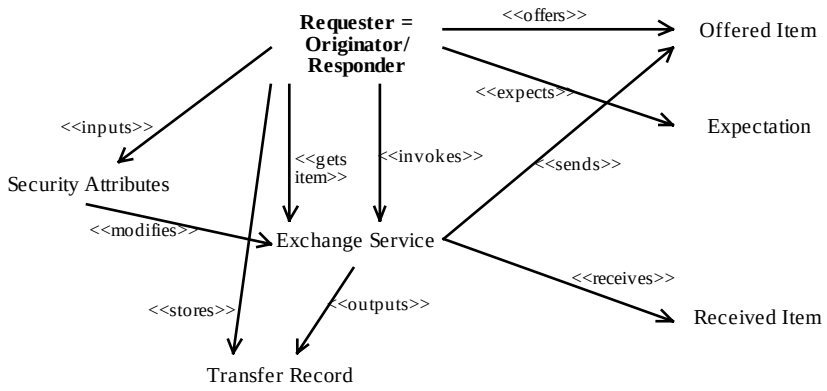

Figure 33: Use Case Exchange, the Originator's/Responder's View.

In this use case, the two main actors of an exchange are exchangeable. Thus, only one use case is depicted, which holds for both of them. The actor is called requester.

These are the things a requester does: He has an item to offer to the other party, and an expectation of the item to receive from the other side in exchange. Both are input to the exchange service. An additional input are the security attributes the requester desires to be applied within the exchange. After a successful exchange in which the offered item is transferred and an item meeting the expectation is received, the service outputs the received item and an exchange record to the service requester.

4.2.2 1.1.2 Design Overview

4.2.2.1 1.1.2.1 Overview of the Java Classes and Interfaces

The following figure shows the class diagram of the basic TX-Layer framework. Afterwards, the classes will be introduced in shorts.

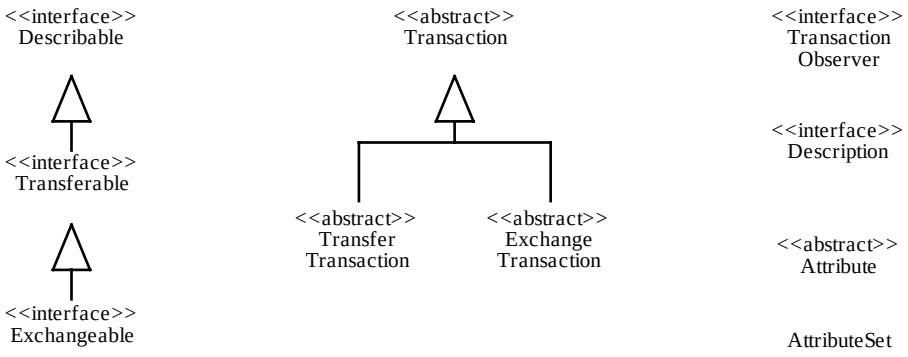

Figure 34: Class Diagram.

Transaction:

The abstract class `Transaction` serves as the base class for both `Transfer Transaction` and `Exchange Transaction`. It maintains the checkpoints and the concurrency of the objects accessed by the transaction. Following are the most important methods of `Transaction`:

begin: starts the execution of the transaction. The begin-method family contains blocking and non-blocking variants.

input, output: none.

commit: after execution of the transaction, this operation lets the transaction propagate its end state to the scope outside the transaction.

input, output: none.

abort: after this operation, the transaction will not run any longer. It will try to undo all state transitions that happened due to it.

input, output: none.

Further methods allow to query the momentary state of the transaction, set and query the transaction's unique identifier, and to add and delete an observer of the transaction. To preserve basic functionality on subclassing, this class offers template methods[18] which are called previous a begin, a commit or an abort.

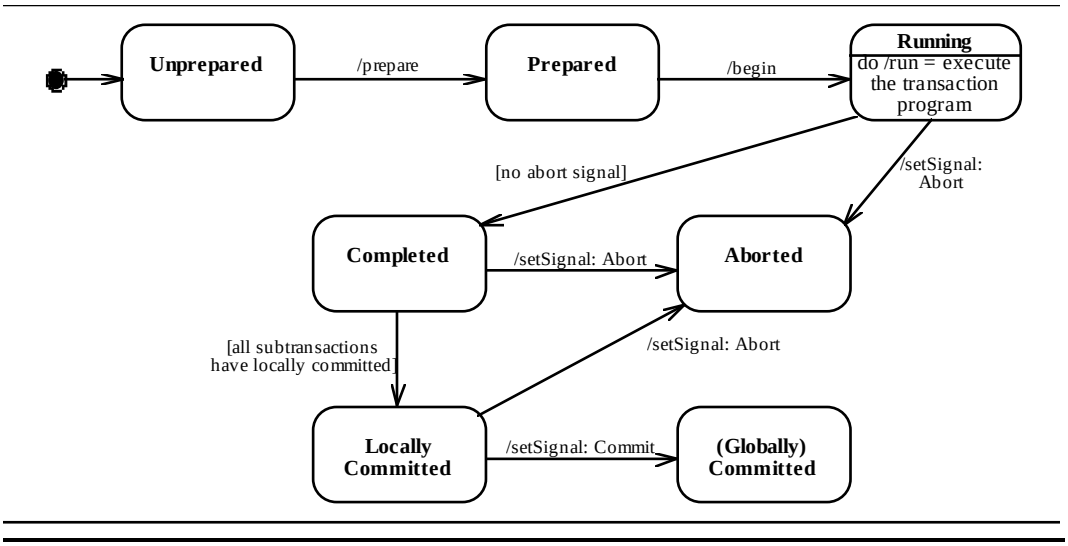

Figure 35: The Transaction Automaton

[18] For template methods, see [GHJV 95].

The initial state after creation of a transaction is *unprepared*. After the transaction is loaded with the necessary arguments, the transaction is *prepared*. This is done via the prepare-method of the transaction. When the transaction is started via a *begin*-method, the transaction‘s state is *running*; during this phase, the transaction program is executed. If the transaction receives an abort signal from a subtransaction, the transaction aborts and rolls back, and transits to the state *aborted*. If the transaction itself encounters an abort situation, it also transits to the abort state. In both cases, the abort signal is propagated upwards to the supertransaction. The same happens if the transaction has already completed, stopped running and has transitted to the state *completed*. Otherwise, if all subtransaction have completed successfully and are in the state *Locally Committed*, the transaction itself transits to the state *Locally Committed*. The resulting state of the accessed objects is propagated upwards. If the transaction receives a commit signal from the super transaction, it releases all global locks from the accessed objects and transits to the state *(Globally) Committed*. The whole transaction tree recursively commits this way.

TransactionObserver:

This type is implemented as a Java-interface. If an object of this type is registered with a transaction, the transaction calls certain methods of this object on particular transaction events. The caller of a transaction can use this mechanism to establish an event channel with the transaction which runs asynchronously to the caller.

The methods of this object which a transaction calls are: *onAbort*, *onBegin*, *onRollback*, *onSetState*, ... All these methods are named after the corresponding event, have the calling transaction as parameter (one observer might be connected with multiple transaction objects), and are called right after the occurrence of the event.

TransferTransaction:

This abstract class is a subclass of the transaction class; an object of this class is a concrete transaction which is specialised to do a transfer special to a certain item object. These are the most important methods of TransferTransaction, besides those inherited from Transaction:

prepareSender: with this method, the transfer transaction is locally configured to perform the Sender’s part/role of the transfer, and the role-dependent input parameters are set.

input: the address of the Receiver’s communication point.

output: none.

prepareReceiver: with this method, the transfer transaction is locally configured to perform the Receiver’s part/role of the transaction, and the role-dependent input parameters are set.

input: the address of the Sender’s communication point.

output: none.

getObject: this method returns the output of the transaction. input: none.

output: the transaction's result.

For subclassing, this class offers two methods, send and receive, to be overloaded with the Sender's functionality and with the receiver's functionality, respectively. To enable communication between the two participants of a transfer, a read and a write method, connected to the other party, is offered to be used by the subclasses send and receive method. Derived from this channel-based variant, a token-based variant is available for further subclassing. Token-based transfer transaction are used by the Generic Payment System Framework (see Section 4.3). Further examples, besides TokenBasedTransfer, of the use of this class via subclassing are the classes SerializingTransfer, VectorTransfer and ContainerTransfer, described ??.

ExchangeTransaction:

This class is a subclass of the transaction class; an object of this class is a transaction which is specialized to do an exchange. These are the most important methods of ExchangeTransaction, besides those inherited from Transaction:

prepareOriginator: with this method, the exchange transaction is locally configured to perform the Originator's part/role of the exchange, and the role-dependent input parameters are set.

input: the address of the Responder's communication point, and the local (i.e. the Originator's) expectation.

output: none.

prepareResponder: with this method, the exchange transaction is locally configured to perform the Responder's part/role of the exchange, and the role-dependent input parameters are set.

input: the address of the Originator's communication point, and the local (i.e. the Responder's) expectation.

output: none.

getObject: this method returns the output of the transaction.

input: none.

output: the transaction's result.

The exchange protocols as described in Section 5.1 provide the same service based on different assumptions. For the integration into SEMPER, they are subclasses of the exchange protocol base class (Figure 37). The exchange manager first negotiates which transfer properties are provided by the goods to be exchanged, and then selects the appropriate exchange according to these transfer properties. . Then, the particular exchange protocol is executed.

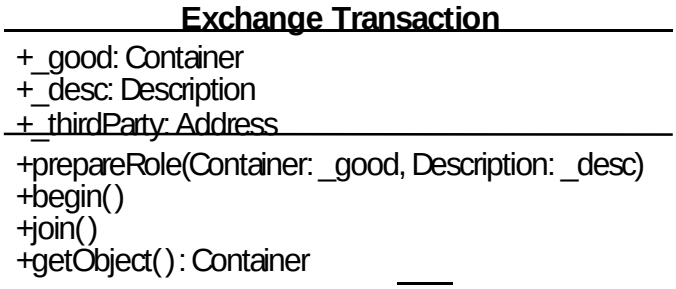

Figure 36: Exchange Transaction

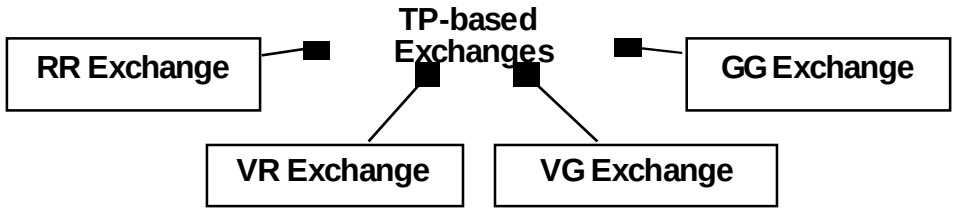

Figure 37: Protocol Classes

Description:

This Java-interface covers an object which specifies one or a set of items. The compare method tests whether an item is in the set of those specified, or it tests whether two descriptions specify the same set of items.

compare: a test with a boolean result.

input: Another description or an arbitrary object to compare this description to.

output: the comparison's result; either true or false.

Attribute:

This abstract class subsumes all the security attributes like Authenticity, Confidential, Non-Repudiation, etc.

AttributeSet:

An object of this class is a set whose elements are restricted to be attributes only. It offers the standard functionality of a set like *union*, *intersection*, *minus*, *isElement*, *isSubset*, *size*.

Describable:

This Java-interface reflects the ability of an object to generate a description of it. This interface contains the method:

getDescription: a factory method to generate a description from this particular object.

input: none.

output: a description of this object.

Transferable:

An object which implements this Java-interface can be subject to a transfer. Transferability implies Describability and thus this interface extends the Describable Interface. This interface contains the method:

getTransferTransaction: a factory method to obtain a transfer transaction object chosen by and customized to *this* object.

input: none.

output: a transfer transaction which can be executed to transfer this object.

Exchangeable:

This Java-interface indicates that an object can be subject to the exchange service. Since an exchange is composed from transfers, the object to be exchanged also has to be transferable; thus this interface extends the Transferable interface. This interface contains the method:

getExchangeTransaction: a factory method to obtain an exchange transaction object chosen by and customized to *this* object.

input: none.

output: a transfer transaction which can be executed to transfer *this* object.

Special Transfer Protocols

Three classes of transfer transaction are predefined, these are introduced now. All of them are extensions of TransferTransaction. The class *SerializingTransfer* implements a transfer protocol which performs the transfer by simply serializing/deserializing the object. The result is of type Serializable. The class *VectorTransfer* implements a transfer protocol suitable to transfer the Java utility class Vector. The result is a Vector. The abstract class *TokenBasedTransfer* is the base class to derive from when implementing a transfer protocol with a token based interface. It uses the sequential r/w interface of *TransferTransaction* to form the token-based interface, the sequential r/w interface is not to be used when extending this class.

Container

A container is the standard data structure to pack items into, it is similar to a real-life parcel. A container is a list in the usual sense, i.e., a number of items in sequential

order which can be indexed and accessed by non-negative integers, and multiple occurrence of items is allowed. The container class implements all three interfaces the TX-Layer expects for its two services: Describable, Transferable, Exchangeable. The Container chooses to be transferred by an extra transfer protocol, the Container Transfer, which maps the service down to the contained items. Consequently, a container assumes all items to implement the three required interfaces, too.

Non-Repudiation of Origin

Non-Repudiation is a protocol which is hooked into the transfer of the item. This is done via the Observer mechanism of transactions. In particular it is designed as follows:

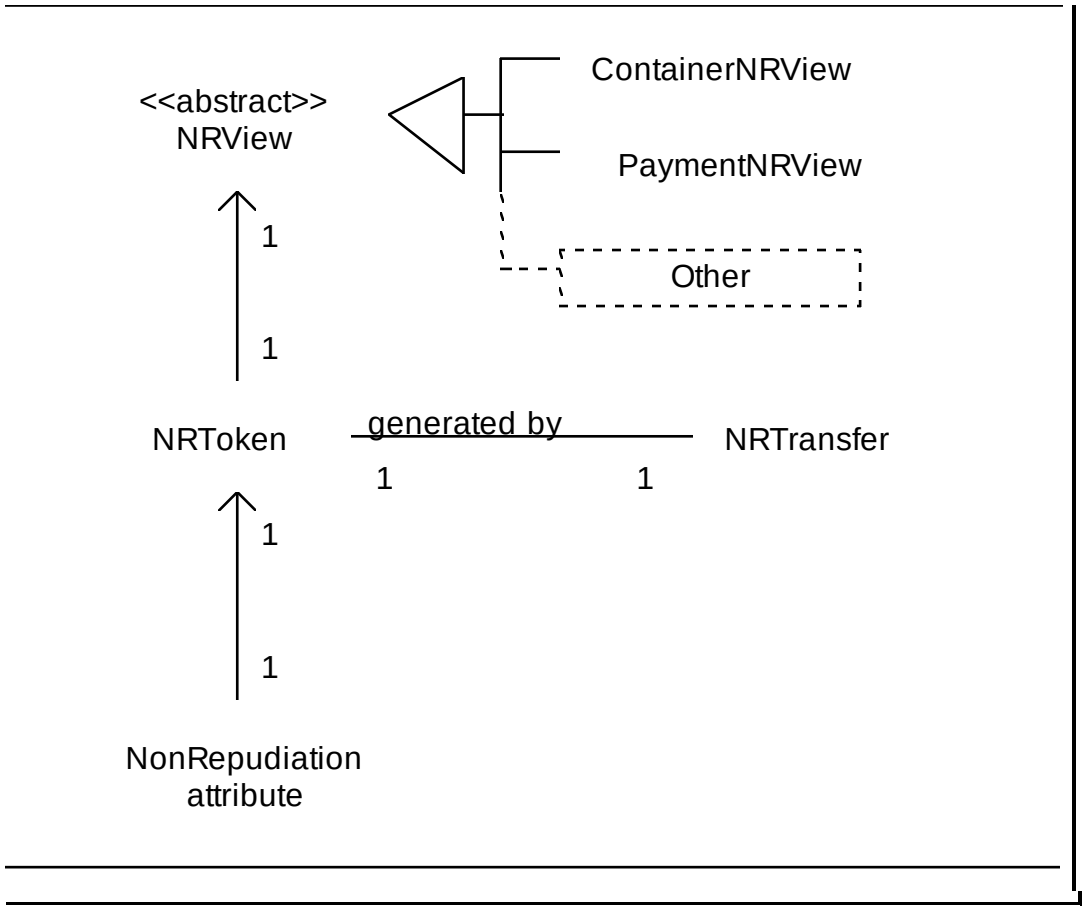

Figure 38: Class relations

There are four main classes that are relevant for the non-repudiation structure.

- `NRView`**:** Its task is to provide a container where TransferTransactions may store data about a transfer. NRView must be subclassed by all Transfer- and Exchangelayer extensions. Of course NRViews may be nested, e.g., an NRView of a ContainerTransfer will contain the NRViews of all the subtransfers of the items in the container.

- `NRToken:` Contains the NRView of the transfer and all necessary additional information, such as certificates (either key or attribute certificates), timestamps, and a signature over all this information.
- `NonRepudiation attribute:` The Nonrepudiation attribute may be set by a user when preparing a TransferTransaction. If the user requests non-repudiation, he must set it in the requested attributes, if he offers it he must set it in the set of offered attributes. If he requested non-repudiation, he can find an NRToken in the NonRepudiation attribute after the TransferTransaction ended.
- `NRTransfer:` There is a specialized TransferTransaction NRTransfer that is able to generate NRTokens. When a TransferTransaction has the NonRepudiation attribute set, it creates an NRTransfer. The NRTransfer then registers itself as an observer with the TransferTransaction that created it. At the end of the TransferTransaction it is notified and can then process its protocol, which may be either to generate and send an NRToken or to receive an NRToken. The received NRToken is stored in the NonRepudiation attribute that the user set in the TransferTransaction.

4.2.3 1.1.3 Functional View

4.2.3.1 1.1.3.1The Transfer Service

When transferring an item, the process strongly depends on the internal structure of the item. The transfer can be a simple one-message protocol with the object as message content, or it can involve more sophisticated mechanisms like sending the item in parts. For example, a payment usually involves a complex protocol with multiple messages and possibly an online bank as a third party to confirm the payment. With video the sequence of data may be chopped and transferred piece by piece. A transfer of a simple integer has a different structure than a transfer of tree-like data structure.

Items are objects, and objects are composed from other objects. This can be seen as a tree, with basic built-in objects as leaves. When transferring a tree structure, the node objects have to be transferred. The nodes can differ in their meaning in a transfer. The serialization[19] mechanism of Java works similarly, see, e.g., [GrKn97]. Serialization deals with copying of objects between Java applications, but is less general than the transfer concept of Semper.

The infrastructure of the cooperation of item and transfer service is designed as follows: The item knows and chooses a transfer service object adequate to transfer the item. According to the tree structure of composed objects, the service block provides basic transfer objects for the basic objects, i.e., the leaves of the tree. From these basic transfers, items can assemble new transfer objects regarding their internal structure and further special conditions. To build a transfer from a tree of objects, the objects

[19] Object serialization is new with Java 1.1. Objects can be serialized into a sequential flat stream in order to write them to flat media, e.g., files and networks.

are required to offer the Transferable interface. This interface indicates that an object knows to create a suitable transfer service object. If a basic transfer type is not suitable for a special reason, the basic transfer can be redefined for that special case. With this construction, a transfer request can recursively be mapped down.

The Participants of a Transfer and their Cooperation

In this section, a TX-Layer transfer is described in terms of the objects participating in a transfer and their cooperation, see Figure 39. The use of attributes is omitted in the figure.

The first step for the requester to perform a transfer is to get a service object from the item (1). This is done with a factory method, *getTransferTransaction*. This method is part of the Java-interface Transferable (2). On the call of this method, the item creates (3) the item-transfer object (4), and returns it to the requester. After creation of the transfer object, the item is registered at the item-transfer object. This is necessary, because the transfer has to access the item in order to transfer it.

The item-transfer object offers the TransferTransaction interface (5) to the requester which is inherited from the generic Transaction (6). With this interface, the transfer can be used as described. The requester can either run (7) the transfer directly, or via a transfer observer (8). Also the observer and the requester can control the transfer in cooperation (9), or they can even be identical. The relation between both is not specified and therefore depicted shaded. The requirement on an observer is that it implements the corresponding observer interface (10), and that the observer is registered at the item transfer object. When the transfer is running, it calls the observer on certain events (11), which can then react in a special way like, e.g., prompting a user for confirmation. Observer is a pattern described in more detail in [GHJV95].

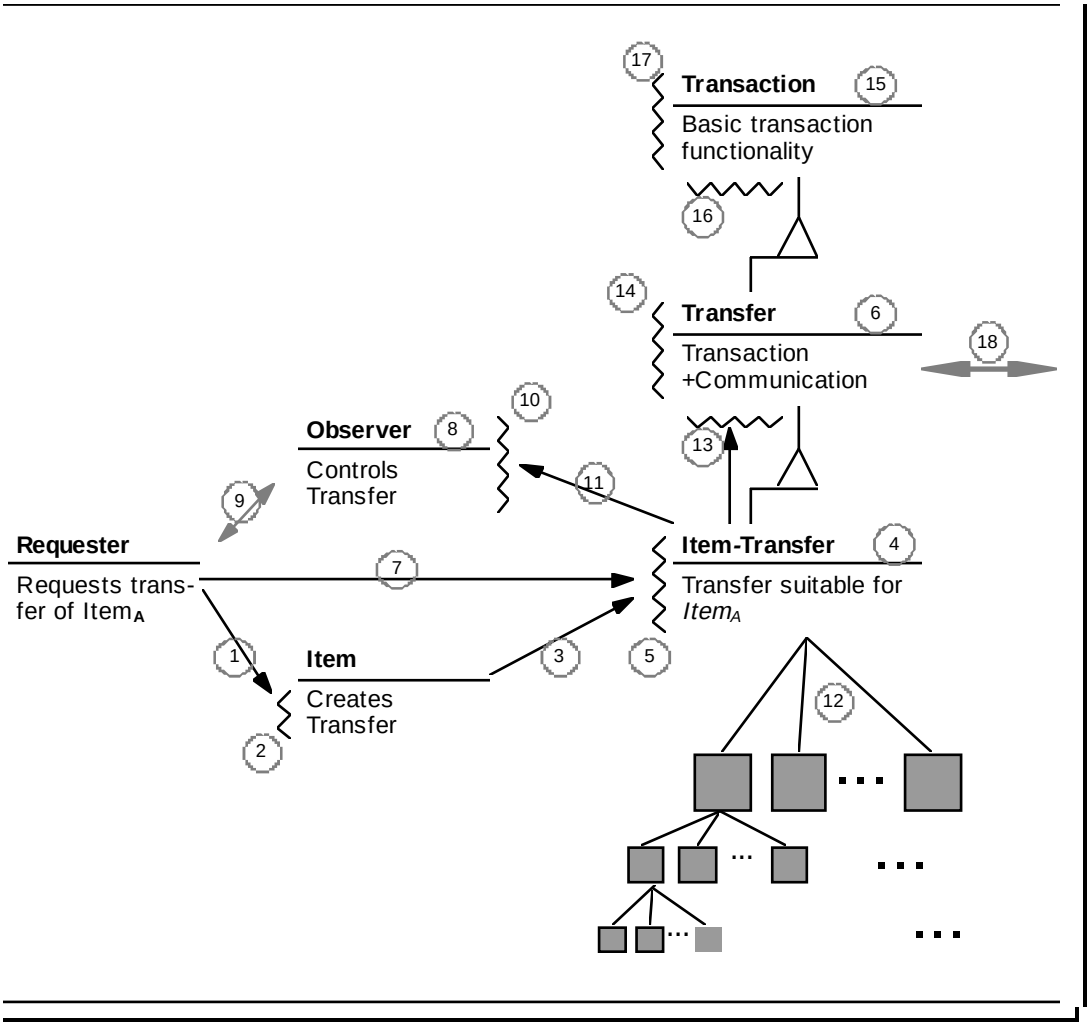

Figure 39: The participants in a transfer and their cooperation.

The item object is possibly composed from a number of objects which again are composed. In order to consider this, the item-transfer object invokes a number of subtransfers according to the encapsulated objects. This results in a tree of transfers; the transfers are nested. For reason of simplicity, the subtransfers are depicted as anonymous shaded boxes (12). Even not visible in the figure, the boxes are derived from the generic transfer transaction and offer its interface. So for them the same holds as for the item-transfer. For these subtransfers, the parent transfer has the role of the requester.

The item-transfer object inherits a number of classes. From the Transfer class (6) it inherits two interfaces (13), (14). One of these is the interface to the requester, with the prepare-methods customized for a transfer (14), e.g., to pass the location of the transfer partner. The other interface is for the item-transfer class itself to use the basic transfer functionality (13). This interface extends the internal interface from the Transaction class (16) with methods for communication.

The Transfer class (6) is abstract, or generic. It is a transaction (15) by inheritance. It inherits the transaction functionality and its interfaces (16), (17), and it extends Transaction with a means for communication (18) in form of a delegation, and customizes the preparation methods.

4.2.3.1.1 1.1.3.1.1 The Exchange Service

In the TX-Layer, exchanges are composed from transfers. This is due to the idea of exchange described in [AsScWa97] and the remarks on exchanges given in this work so far. As a consequence, an exchange service involves the item of the services less strongly than a transfer does: within an exchange an item takes part via its transferability. Analogously to the incorporation of transaction in the transfer service, the exchange service is an extension of transaction. Within the exchange transaction, a number of transfers are performed according to the logic of exchange.

A concrete exchange protocol, perfoming a certain interleaving of transfers, is chosen by the exchange service by evaluating so-called transfer properties (introduced in Section 5.1. Such a property determines the possible involvement of a third party in a transfer, which impacts the message flow of an exchange. A transfer property is bound to an item, and for the properties of the items on both sides there might be a negotiation on selecting an exchange protocol. In the implementation, properties are a kind of attributes.

In more detail: the properties of transfers may be implemented by additional transfer attributes: Each player playing a particular role (Third-party, sender, or recipient) instantiates a suitable attribute for the property to be exploited and inputs them into the transfer. There is one specific attribute for each role and property which provides the interface to this property for a given role. The role-attribute classes are abstract and define the common functionality of all properties. This is depicted in Figure 40, Figure 41, Figure 42 and Figure 43.

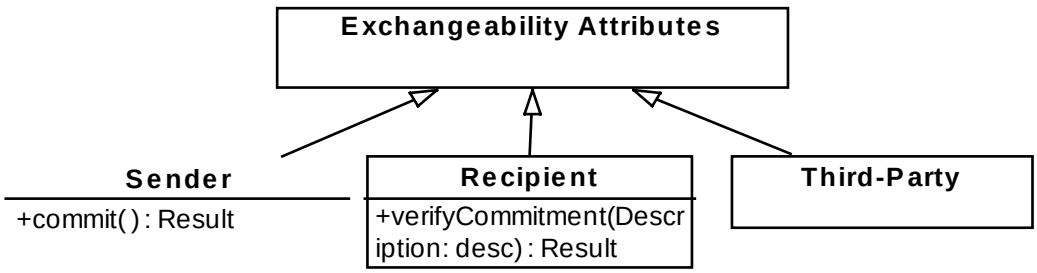

Figure 40: Class Hierarchy for Exchangeability Attributes.

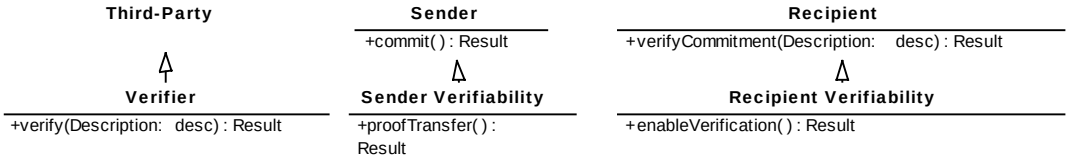

Figure 41: Attributes for Verifiability Roles.

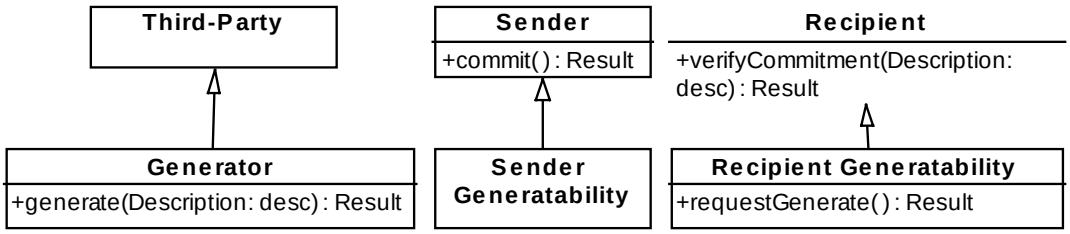

Figure 42: Attributes for Generatability Roles.

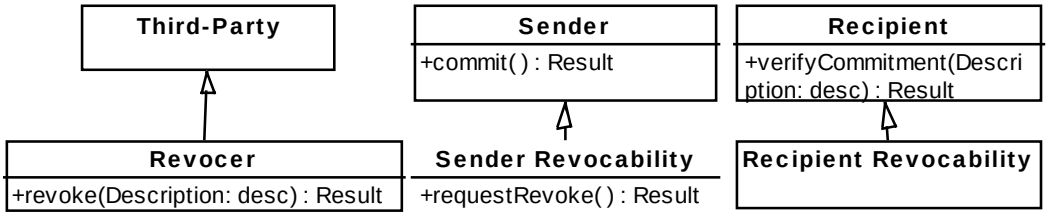

Figure 43: Attributes for Revocability Roles

4.2.4 1.1.4 Related Work

In [LMWF94], a theoretical system for nested atomic transactions is proposed, also including proofs of atomicity. Lynch et al. deal with so-called object automatons, i.e., objects formulated as automata and thus are accessible to a mathematical treatment.

4.2.5 1.1.5 Self-Assessment

It is not clarified if some of the functionality can be broken down to Corba service specifications. This would enable the TX-Layer to be more generic by allowing to plug in standard components for transaction processing and persistence.

Nevertheless, it has not been investigated how atomicity of a TX-Layer protocol execution differs from usual database-transaction concepts, the main difference lies in the trust model underlying both services.

It is not further investigated if the following assumptions hold:

- A transfer protocol and an exchange protocol always consists of two subprotocol: a negotiation protocol and a service protocol (the actual transfer or exchange) and not more. The negotiation protocol may be empty.
- The transaction record of an exchange has the same meaning as a receipt in a transfer with receipt.
- It is not investigated satisfactorily how the responsibilites of item and service module in a transfer are distributed.

4.2.6 1.1.6 Implementation Notes

The transaction class still is only a wire model, real fault tolerance and persistence is not implemented yet. The same holds for a concurrency-control mechanism. A correct treatment of real actions is not designed yet, also as the complete set of exchange protocols.

The implementation of Fair exchanges is quite experimental and not finished.

4.3 Payment block

(N. Asokan / ZRL)

4.3.1 Domain Description

Before we start with the requirements, let us introduce some terminology (See [AJSW97] or [AASW98] for further information) to describe the target domain. The purpose of payment systems is to **transfer** economic **value** between entities. Entities involved in such a transfer are called **players.** There are various types of players: a **payment** is a value transfer from a **payer** to a **payee.** The player which links the exchange of electronic payment messages to a real transfer of value is the **financial institution.** Ordinary players (such as payers and payees) need to have a business relationship with a financial institution in order to be able to make electronic value transfer. The financial institution associated with the payer is called the **issuer** and that associated with the payee is called the **acquirer.** A **payment system** is a collective name for one "way" of making a value transfer: it consists of protocols, contractual agreements, and data structures. Each participating player will have its own component of the payment system. A **payment instrument** is an instance of one player's component of a payment system: for example. A value transfer takes place from one payment instrument to another (the instruments involved in one value transfer transaction must belong to the same payment system). For example, a SET Mastercard is a payment instrument. A user may have several payment instruments. Multiple payment instruments of a user may belong to the same payment system (e.g., a user may have two SET Mastercard credit cards). The entity that makes use of a payment system is called the **application**. Applications act on behalf of the human user. Applications need to know (a) how to select a suitable payment instrument from the many that may be available to the user, and (b) how to use the selected payment instrument to actually carry out the value transfer.

In SEMPER, the "application" which uses the payment systems is the transfer-exchange layer. However, SEMPER has a staircase architecture: other entities, such as commerce layer transactions, and the business applications can also use the payment block directly, if they need to do so.

4.3.2 Requirements

Requirements can be stated at various levels of abstraction. In this document, we will first specify the basic overall requirement for the payment block, and illustrate this requirement with various use cases. Where possible, we use a standard approach and notation (Universal Modelling Language [FowSco97]) to describe the requirements and design.

The overall requirement for the generic payment service framework (GPSF) is to

> *Provide a framework enabling electronic commerce applications to use a variety of payment systems. The framework should*

- *to the extent possible, make application development independent of specific payment systems,*
- *preserve security services provided by payment systems; if possible, allow applications to supplement security services provided by payment systems,*
- *provide a means to choose a suitable payment instrument for a specific payment transaction,*
- *allow applications to specify requirements on the selection of a payment instrument for a transaction they request; this specification can be at various levels of abstraction, and*
- *allow users to configure the selection of payment instruments used for their transactions*

From the point of view of the user or an application acting on his behalf, these requirements are supplemented by the requirements on payment systems themselves. The basic functional requirement on a payment system is to carry out the transfer of economic value from one player to another. This transfer may be performed in a way that satisfies additional security requirements.

The only *security requirement* for the GPSF itself is that it must not undermine any security service provided by the underlying payment systems.

The *functional requirements* of the GPSF are illustrated by the following use cases. The use cases describe typical interactions between the GPSF and its users (human users as well applications acting on their behalf).

Use Case 1: Instrument Management: Inserting/configuring/removing payment instruments.

Use Case 2: Instrument Selection: Selecting a payment instrument.

Use Case 3: Value Transfer: Making a value transfer transaction.

Use Case 4: Information: Extracting information about payment instruments/value transfer transaction.

We do not include use cases for installing/uninstalling or configuring payment systems because a payment system is a type of an “external module” in SEMPER. There is a common generic module installation facility in SEMPER responsible for installing/uninstalling all types of modules.

There are four types of actors: the *GPSF* itself, the human *user*, an *application* which uses the GPSF on behalf of the human user via an API, and the payment *instrument* which provides the actual value transfer services. Users play different roles: for example, *payer*, *payee*, *issuer*, and *acquirer* are typical roles in a payment system. Sometimes, it is enough to talk of a user and his *peer*. The relationships between actors and use cases are shown in Figure 44. Note that the user will likely interact with the application acting on his behalf; we do not show it here since it is outside the GPSF.

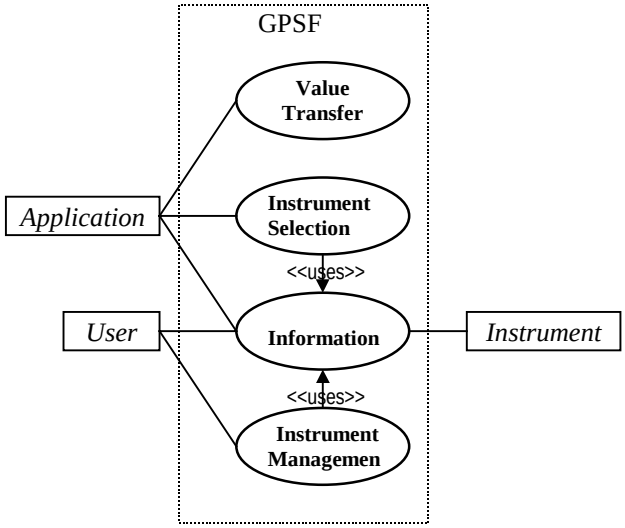

Figure 44: Relationships between actors and use cases

4.3.2.1 Instrument Management

The user is the actor involved in instrument management. There are several sub cases: insertion, configuration, activation, deactivation, deletion. *Insertion* of an instrument into the GPSF is a pre-condition before the instrument can be used within the GPSF. Further, an inserted instrument must be *activated* before it can be used in transactions. Changing the properties of an instrument is called *configuration*. An instrument may be *deactivated* within the GPSF, making it unavailable for transactions until it is activated again. A *deleted* instrument is removed from the GPSF. The GPSF may interact with the instrument when necessary.

- *Goal*: Facilitate the use of external instruments within the GPSF framework.
- *Conditions:* The conditions describe the effect of the use cases. In Table 1, the conditions for this use case are listed.
- *Main Success Scenario*: In each case, the success scenario is simple. The interaction consists of a single request from the user to the GPSF, and a single response in return.
- *Variations:*
- *Related Information:* This case will need to use the **information** case in order to extract necessary information. For example, when a user wants to insert an instrument, it is necessary to see which payment systems are locally available; when a user wants to configure, enable/disable, or delete an instrument, it is necessary to see which instruments are available for this operation.

Table 1: Conditions for the Instrument Management use case

	Pre-conditions	**Success condition**	**Failure conditions**
Insertion	payment system	new instrument available	no new instrument

	software/hardware is already installed		
Configuration	instrument already inserted	instrument available	desired configuration change not performed
Activation	instrument available but inactive	active (i.e., ready for use in transactions)	inactive
Deactivation	instrument active	inactive	instrument may still be used in transactions
Deletion	instrument available but inactive	instrument no longer available	instrument may still be available

4.3.2.2 Instrument Selection

The application is the actor involved in instrument selection. However, the GPSF may involve the user during instrument selection. Also, during instrument selection the GPSF will use information from itself, from the peer's GPSF, as well as from the instruments.

- *Goal*: Select a suitable instrument for a proposed transaction.
- *Conditions:*
  1. *Pre-conditions:* Any restrictions for the selection are known. These consist of:
  2. Direction of value transfer, identity of peer, amount, user-specified restrictions on types or instances of instruments that can be used, and user-specified security requirements for the value transfer.
  3. *Success condition:* A suitable instrument has been chosen. Any necessary addressing information, required to make the desired value transfer, is also known.
  4. *Failure condition:* No instrument with the specified requirements was found.
- *Main Success Scenario*: See Figure 46.
- *Variations:*
  - Sometimes, an application may want the GPSF to choose a suitable instrument and make a value transfer transaction. This supercase is shown in Figure 45.
  - Specification of restrictions mentioned in the pre-conditions is optional.
  - The user is prompted only when no automatic decision is possible.
  - Negotiation with peer is performed only when necessary.
  - *Related Information:* Selection of instrument depends on three factors: service requirements, expressed by the calling application at the time of request, user preferences expressed by the user ahead of time, and negotiations with the peer.

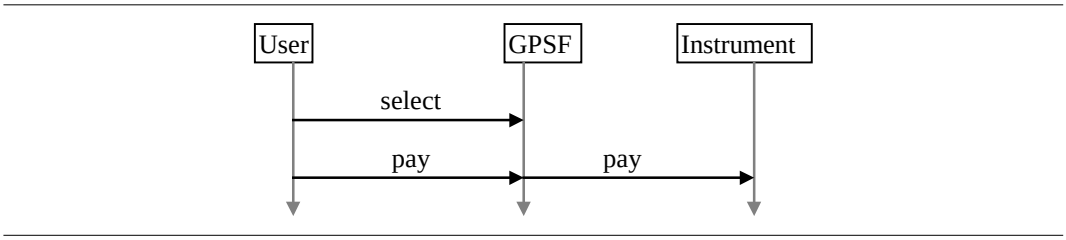

Figure 45: Combined instrument selection **and** value transfer

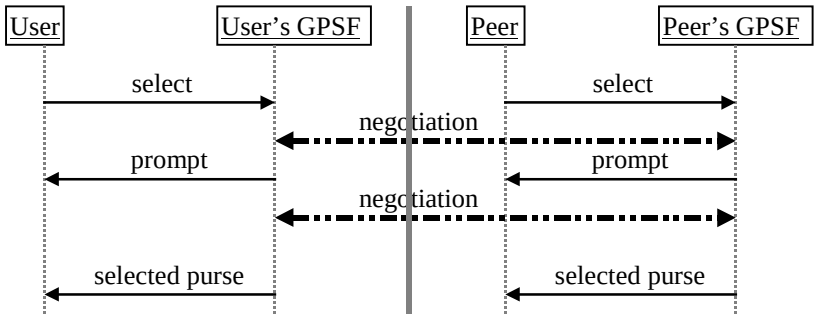

Figure 46: Instrument selection**: main success scenario**

4.3.2.3 Value Transfer

- *Goal*: Carry out a transfer of value according to specified requirements.
- *Conditions:*
  1. *Pre-conditions:* Requirements for the transfer are known. These consist of Type of value transfer, identity of peer, amount, user-specified security requirements for the transfer, and user-specified tag to identify the transaction.
  2. *Success condition:* Value transferred.
  3. *Failure condition:* No value has been transferred from the point-of-view of all players.
- *Main Success Scenario:* This is described in Figure 47. The value transfer is subject to an access control check. The entity that makes this authorisation decision could be the user himself (e.g., by simply approving each payment), or a special access control block within the system.
- *Variations:* none
- *Related Information:* none

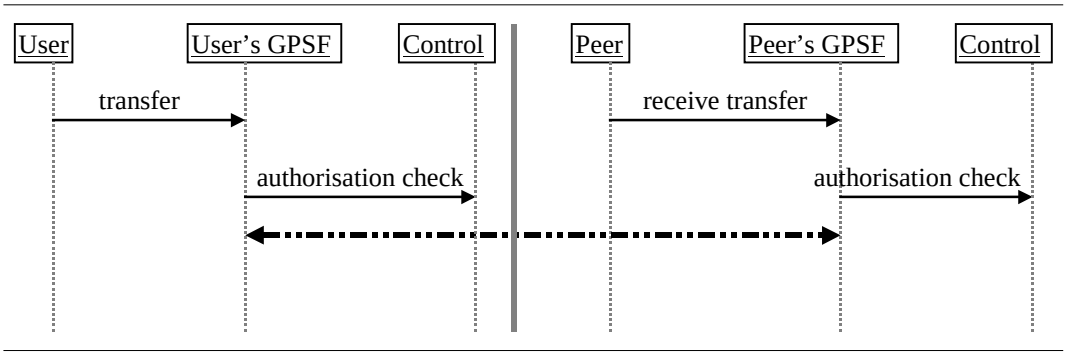

Figure 47: Value transfer**: main success scenario**

4.3.2.4 Information

- *Goal*: Provide information on the state of the GPSF
- *Conditions:*
  1. Pre-conditions: none.
  2. Success condition: requested information returned to caller.
  3. Failure condition: none.
- *Main Success Scenario:* The basic scenario is straightforward: the caller specifies the information requested, and the GPSF returns the information if available, possibly subject to an access check.
- *Variations:* The information request may be about a specific instrument; in which case the instrument itself may need to be queried.
- *Related Information:* none

4.3.3 Design Overview

Payment systems can be divided into various *models* [AASW98]. Two of the common models are *account-based* (also called *cheque-like*), and *cash-like*. All payment systems provide a certain set of common services (e.g., transferring value from payer to payee). Those belonging to a given payment model, may provide additional services specific to the model (e.g., the *withdrawal* operation in cash-like systems gives a payer the right to subsequently transfer value by electronic means). Our design is based on this observation. We define the common services in base classes and interfaces. Model-specific extensions are defined in sub-classes and interfaces.

We use the notion of a `Purse` to represent a payment instrument inside the GPSF. The value transfer services expected from an instrument are specified by the interface hierarchy `PurseServices` (also called).

The implementation of a payment system is called a *payment module*. It may contain software and/or hardware components. A payment module may correspond to a single payment instrument (e.g., in the case of a hardware purse) or it may contain common code that allows multiple instruments to be instantiated. In order to use a payment module within the GPSF, a module-specific *adapter* has to be implemented.

We describe the design in two ways. First, we present the **object view**, consisting of a static description of the objects and their behaviour. Then we present the **functional view**, describing the dynamic behaviour of the various objects for each of the use cases identified earlier.

4.3.3.1 Object View

We describe the object view by first presenting the object model, and then listing the interfaces offered by the objects.

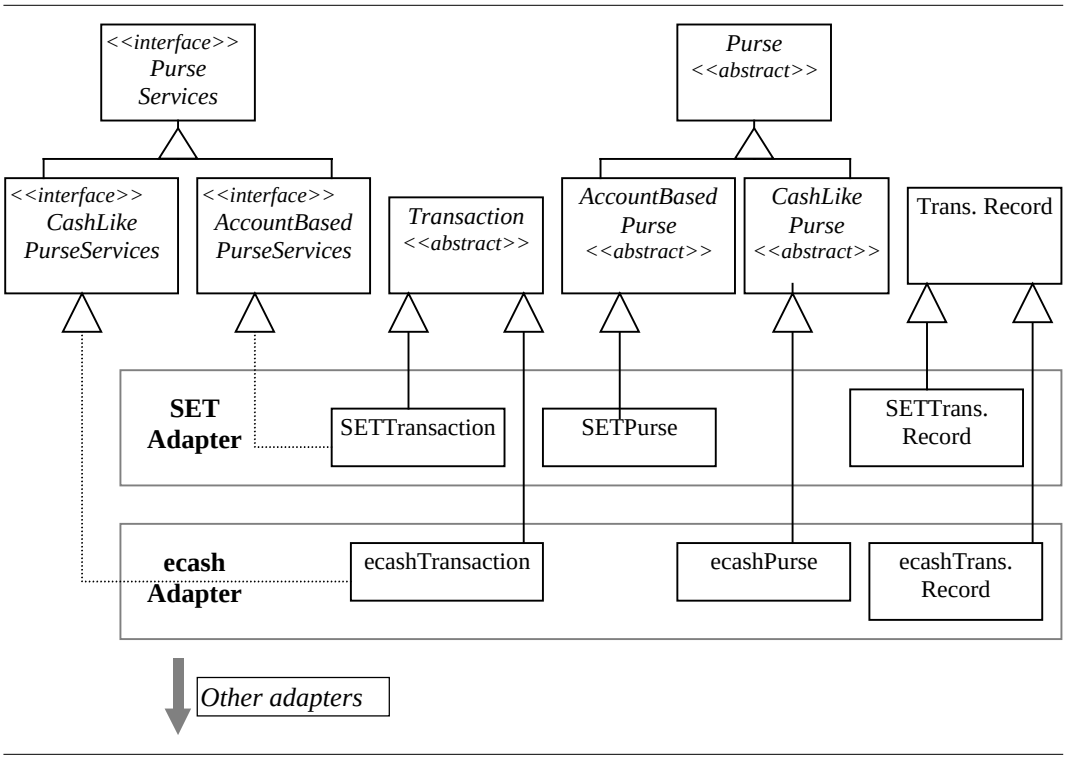

Figure 48: Primary data types in the GPSF

Figure 48 describes the primary data types in the GPSF.

- **Purse:** A purse is the representation of a payment instrument within the GPSF. The Purse class hierarchy consists of a common base class and model specific extensions. An adapter must provide a sub-class of an appropriate model-specific Purse class. A purse serves as an *abstract factory* for creating the correct Transaction and TransactionRecord objects.
- **PurseServices:** Value transfer services of payment systems are captured by the `PurseServices` interface hierarchy. As with the `Purse` class hierarchy, the `PurseServices` hierarchy consists of a base interface and model-specific extensions. `Transaction` classes in adapters are required to implement an appropriate model-specific `PurseServices` extension.
- **Transaction**: Each value transfer transaction is represented within the GPSF by a `Transaction` object. The `Transaction` base class defines generic services like aborting a transaction or querying its status. Each adapter must provide an

extension of the `Transaction` class which implements an appropriate `PurseServices` interface.

- **TransactionRecord:** The record of a transaction is kept in stable storage in the form of a `TransactionRecord` object. Unlike `Transaction` objects, `TransactionRecord` objects are long-lived. (`Transaction` objects do not persist across multiple SEMPER sessions).

In addition, there are various other classes.

- **Payment Manager:** The payment manager is the overall controller of the GPSF. It is responsible for initialising GPSF on start-up and for keeping track of existing purses.
- **PurseSelection:** The purse selection class provides services for choosing a purse for a transaction. When an application wants to select a purse, it creates a PurseSelection object first, and then invokes appropriate methods on it.
- **TransactionState:** The state of a transaction is described by a `TransactionState` object. `TransactionState` is a class hierarchy; the extension for each model may introduce minor states. Further system-specific extensions can introduce additional minor states.
- **ServiceType:** Each type of value transfer primitive (e.g., pay) is represented by a `ServiceType` object. This is a necessary input for purse selection, where the selection depends on the type of transaction.
- **PurseReference:** A purse reference is an abstract adapter-independent reference to a purse. Purses themselves maybe disabled, un-initialised, deleted, or otherwise unusable or even inaccessible. But a purse reference is always available once a purse has been created.
- **PaymentEntity:** Each service access point of the GPSF (e.g., payer, payee, etc.) is identified by a `PaymentEntity` object.
- **Amount, Currency:** These objects are used to represent the value to be transferred.
- **SecurityOption:** Security services required for a value transfer are specified in the form of SecurityOption objects. They can take pre-defined values like “confidentiality,” or “non-repudiation of origin.”
- **Special Applications:**
- Typically, users do not invoke GPSF services directly, but do so via applications acting on their behalf. However, there are some special applications within the GPSF that allow direct user interaction:
- **Purse Management Application**: The purse management application is used by the user to create and manage purses: i.e., to incorporate payment instruments into the GPSF and manage them. It also allows a user to examine the state of a purse, change its configuration, disable or delete it.
- **Transaction Browser:** The transaction browser application is used by the user to examine records of current or past transactions, and to export them to be used by other, external, applications.

The table below is an excerpt of the GPSF API. Java bindings of the complete API are also available. This table contains only the most significant interface methods (for example, it leaves out most accessor methods).

Table 2: Java bindings of important functions of the GPSF API

Class or Interface	Primitive	Input	Output	Description
PurseServices	pay	payee, amount, options, ref.		Send a payment
	receivePayment	[ref]	payer, amount, options, ref	Receive a payment
	reversePayment	trans. record		Ask/get a refund
	reverseReceivedPayment	trans. record		Make a refund
AccountBased PurseServices	receiveRawPayment	[ref]	payer, amount, options, ref	Receive a payment (defer authorisation)
	authorise			Authorise a previous raw payment
	capture			Capture a previous raw payment
CashLike PurseServices	withdraw	amount, options, ref.		Load money into purse
	deposit	amount, options, ref		Unload money from purses
Payment Manager	createPurse	purseclassname, pursename		create a new purse
	deletePurse	purse name		delete a purse
	registerPurseClassName	key, purseclassname		register a new adapter with the specified key and purse subclass
	getListOfEnabledPurse Classes			return the list of known, and enabled adapters
	getListOfPurses	[criteria][20]		return the list of all purses (possibly subject to some criteria)

[20] This method exists with various different signatures

Purse	startTransaction			start a new transaction for this purse
	setup	tinguin session		do any setup necessary to use this purse; use the given tinguin session for any user interaction
	init			initialise this purse for this session
	disable			disable this purse
	getPrintableStatusInfo		info	return the state information in printable form
	isCurrencySupported	currency	boolean	
	isSecurityServiceOffered			
PurseSelection	constructor	peer, self-alias, amount, servicetype, paymentsystem-list, purse-list, options		create a purse selection transaction with the specified attributes (which are recorded in the current state)
	selectCandidatePurses			select a first list of purses subject to the restrictions of the current state
	requestChoiceOfPS			propose a set of payment systems to peer and ask for an acceptable subset
	choosePS			receive a proposal from peer and choose a subset
	requestPurseConfirmation			pick a purse and ask the peer for confirmation
	confirmPurse			confirm peer's choice, pick a matching purse locally
Transaction Record	endTransaction			mark the transaction as completed
	suspendTransaction			mark the transaction as suspended
	resumeTransaction			resume a suspended transaction
	toString		state	return info about the transaction in printable form

4.3.3.2 Functional View

The entire operation of the GPSF depends on the pre-condition that the payment manager has been successfully initialised at start-up. Initialisation of payment manager is part of the SEMPER bootstrapping process. Figure 49 shows the steps in the initialisation activity. The SEMPER preferences block allows the user to set and manage configuration and preferences. The payment manager can be informed of a new adapter at the time of adapter installation. The payment manager interface contains a method (`registerPurseClassName()`) using which it can be informed of a new adapter during module installation. Creating and initialising purses is described below (in discussing the Instrument Management use case). There are currently two "special applications" within the GPSF framework: a *purse management application* and a *transaction browser*. The purse management application allows a user to interactively create, configure, and delete purses. The transaction browser allows a user to view transaction records of previous or current transactions, and to export them outside the GPSF.

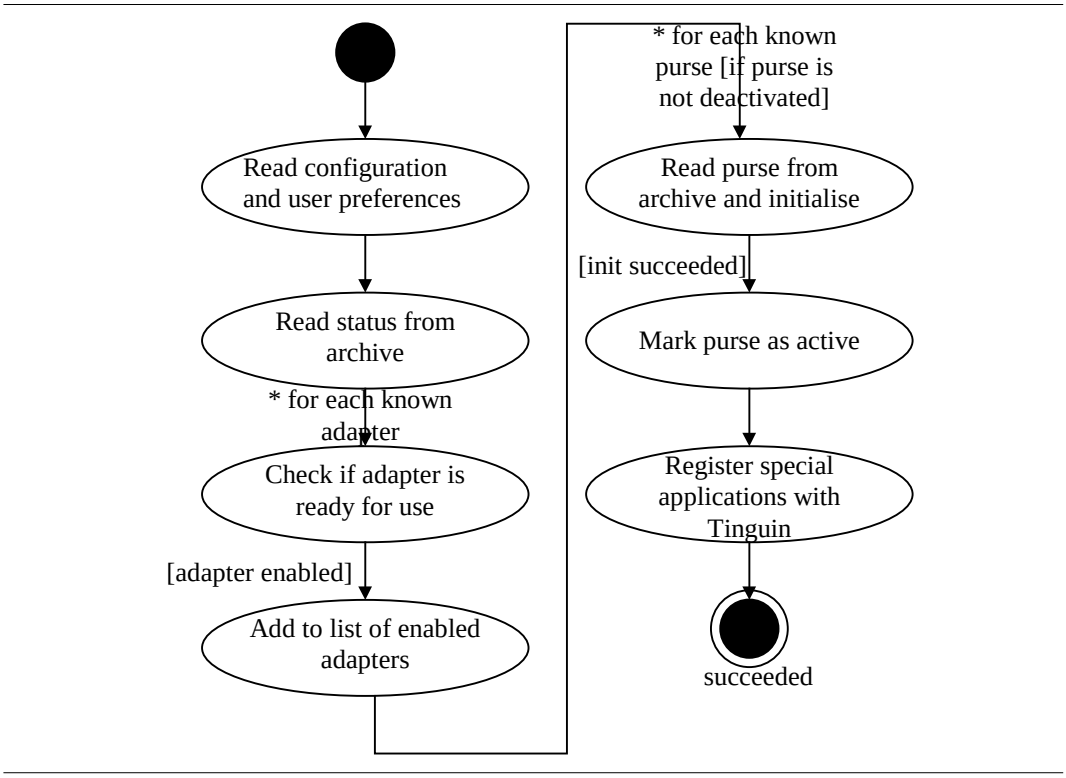

Figure 49: Initialisation of the payment manager

Now we describe how each of the use cases mentioned Section 4.3.2 in are handled.

Instrument Management

As mentioned, a payment instrument is represented within the GPSF by a purse object. Figure 50 shows the states of a purse. Only active purses are available for use in value transfer transactions. A user can explicitly deactivate a purse. A deactivated purse will be present in the GPSF, but is not available for value transfer transactions, until it is explicitly re-activated. Deletion is an unrecoverable operation: a deleted purse is not available anymore.

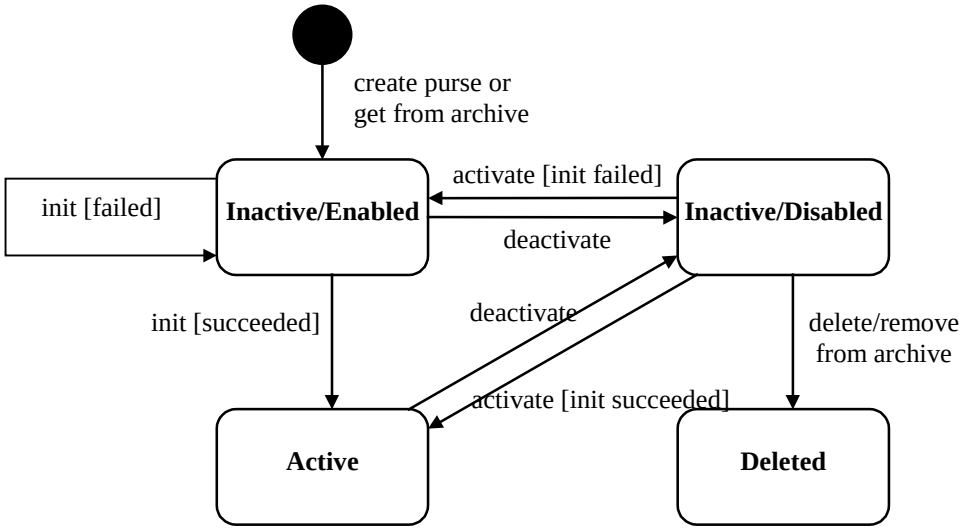

Figure 50: State diagram for the lifecycle of purses

A human user manages his purses via the *purse management application*. We illustrate the implementation of one use case, inserting an instrument into the GPSF, by means of an activity diagram in Figure 51.

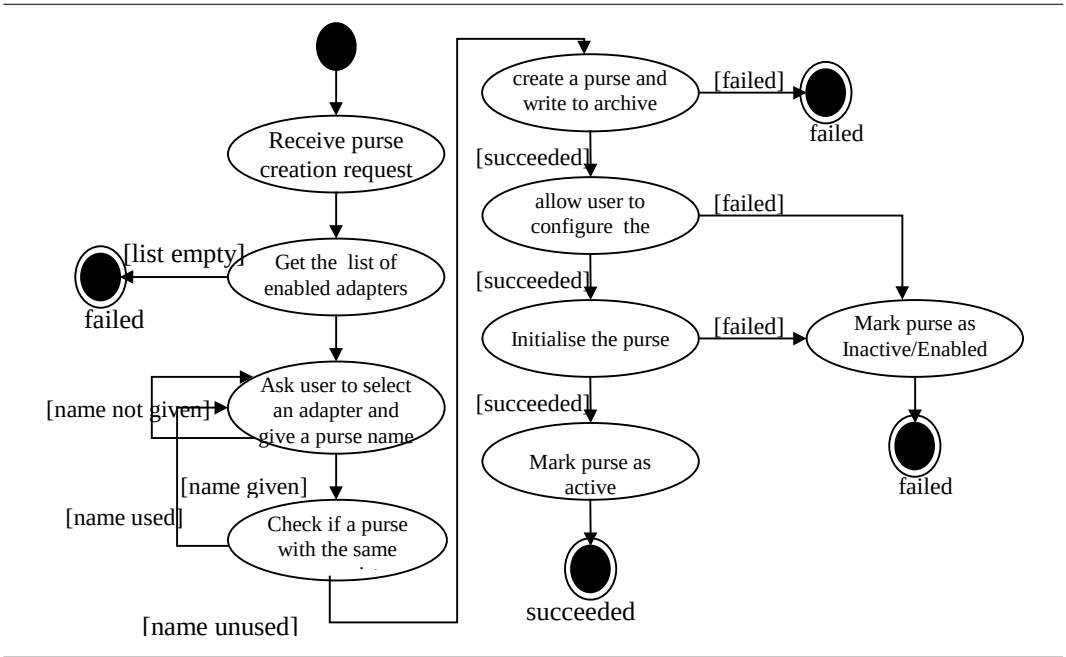

Figure 51: Activities during *purse creation*

Figure 51 does not identify the entities involved in the purse creation use case. These are described in detail in the interaction diagram (Figure 52). The other use cases start in the same way. The user is prompted with the choices:

- create a purse
- configure a purse
- view the status of the purse
- deactivate a purse

- delete a purse
- operate on a purse

The last operation allows the purse to present additional purse management functionality to the user.

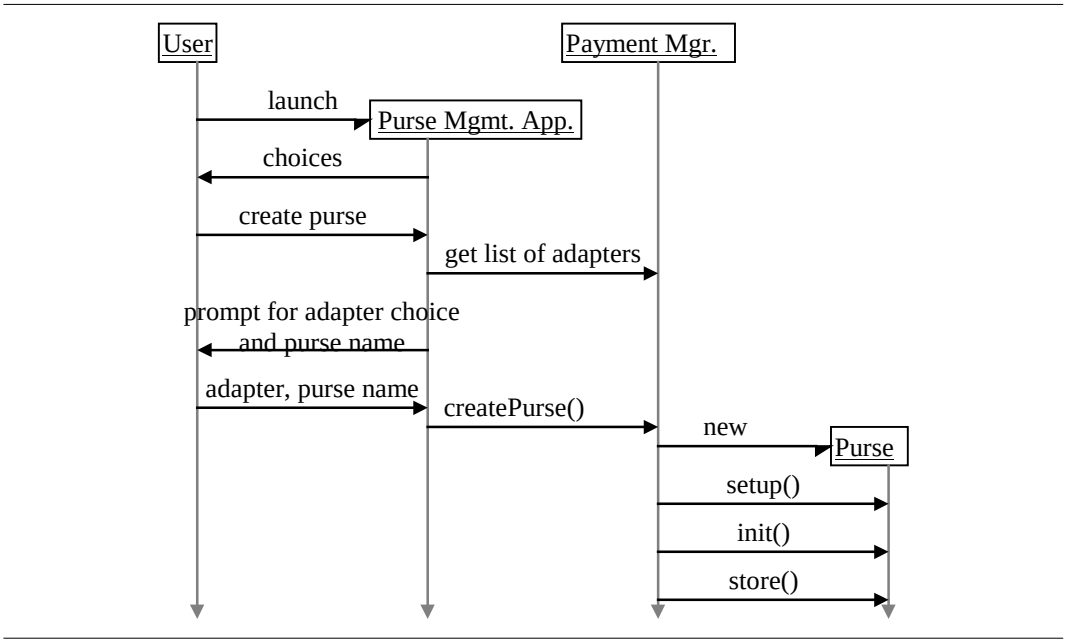

Figure 52: Interactions during purse creation

Instrument Selection

Instrument Selection services are provided by the `PurseSelection` class. The selection is constrained by three factors:

- *Requirements specified by the calling applications*: these include the amount, currency, and direction ("in" or "out") of the proposed transaction, and the security services requested. Additionally, the caller may restrict the choice by specifying a list of acceptable payment systems, or even specific payment instruments.
- *Preferences of the user*: Currently, the only preference a user can indicate is whether he wants to be prompted to break ties in the selection (the default behaviour) or whether the GPSF should break ties arbitrarily. The latter preference is useful for unattended servers. In theory, it is possible to support more elaborate user preferences.
- *Negotiations with the peer*: Finally, the chosen payment system must be acceptable to the peer entity in a transaction. A simple negotiation protocol is used for this purpose.

Figure 53 shows the interactions during instrument selection: the selection of a purse for a value transfer transaction. The application first creates a `PurseSelection` object with the necessary parameters characterising the proposed transaction. Then, the application can invoke various methods to arrive at a final choice of purse. Figure 53 shows a typical sequence, represented by the methods `selectPayingPurse()` and `selectReceivingPurse()` in `PurseSelection`. These constitute only one possible scenario. Other scenarios can be composed from the basic methods. Both

sides first construct a set of candidate purses. The activities during this local process are shown in Figure 54. Once the initiator has a set of candidate purses, it invokes the `requestChoiceOfPS()` method to present the responder with a set of possible payment systems, and ask for the responder's choice of a subset. After receiving a mutually acceptable subset from the responder, the initiator invokes `selectPurse()` to select a single purse from one of these payment systems, and then `requestPurseConfirmation()` to ask for the responder's confirmation. This latter handshake protocol will also exchange any addressing information necessary for the subsequent value transfer transaction. Once this step is completed, both sides would have chosen a single local purse respectively. The applications can retrieve the selected purse, and the peer value transfer address from the `PurseSelection` object and proceed with the value transfer.

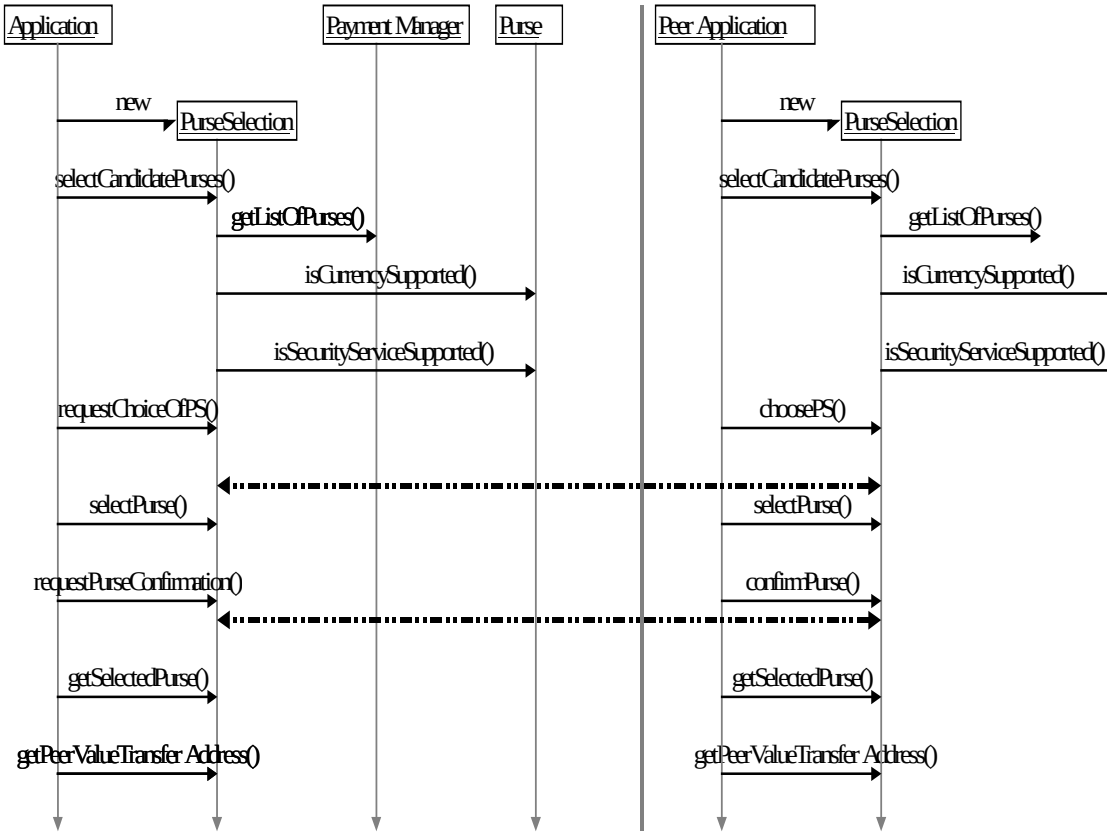

Figure 53: Interactions during instrument selection

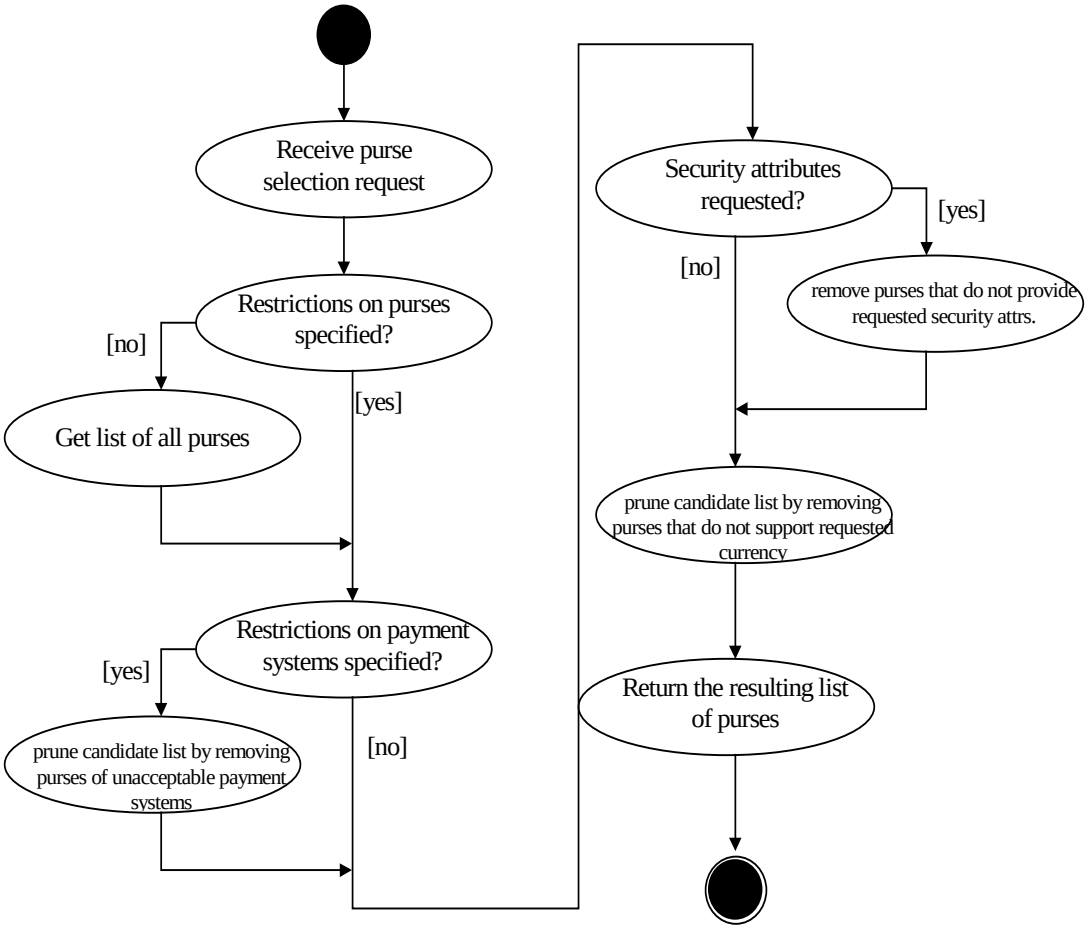

Figure 54: Activities during `selectCandidatePurses()`

Value Transfer

Figure 55 shows the interactions that occur during a typical value transfer scenario. The calling application has already selected a purse for the transaction. It begins the value transfer by instantiating a `Transaction` object. The *factory method* `startTransaction()` of the `Purse` object creates the Transaction and TransactionRecord objects of the correct types (e.g., `SETPurse` creates `SETTransaction` and `SETTransactionRecord`) and returns a handle to the `Transaction` object. Once the application has a handle to the `Transaction` object, it can invoke the method corresponding to the appropriate value transfer method (e.g., pay() on the payer side and `receivePayment()` on the payee side). Figure 55 depicts a slightly different approach: value transfer involves the invocation of a starter method (e.g. `startPay()`) and several subsequent invocations of a processor method (`processToken()`). Both these methods return a *token* to the calling application. The token is an encapsulated protocol message. The application is responsible for transferring the contents of the token to the peer application. The peer applicaiton receives the token and pushes it into the `Transaction` object by invoking the `processToken()` method, providing the received token as input parameter. We call this the **token-based** approach to defining service interfaces. The primary advantage of the token-based approach is that since all communication of protocol messages is done via an external communication channel, it is possible to *supplement* the security services provided by a payment instrument. For example, by

sending payment messages through an anonymous communication channel, it will be possible to hide the payer-payee relationship from eavesdroppers.

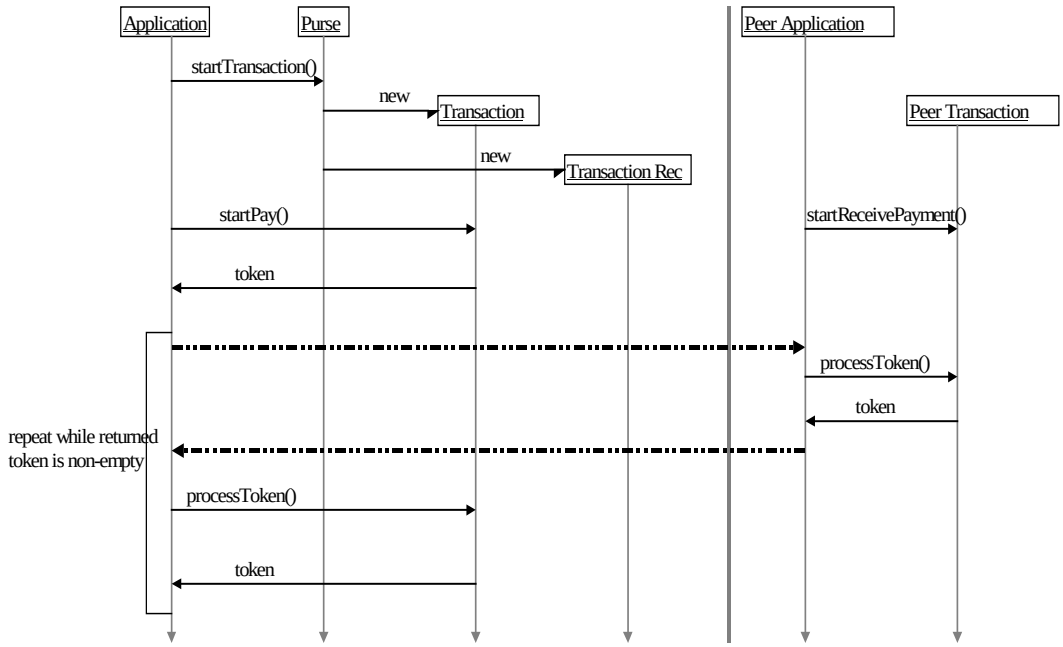

Figure 55: Interactions during value transfer

Information

Most of the information services are relatively straightforward. We already saw some examples, such as "get list of adapters" (implemented by the `PaymentManager` method `getListOfKnownPurseClasses()`). Information services are provided by a variety of different objects in the GPSF.

- The payment manager provides methods to retrieve the list of currently known adapters, and the list purses according to various criteria (e.g., list of all purses, list of active purses, list of purses belonging to a given set of payment systems etc.),
- The `Purse` class provides methods to retrieve the list of all transaction records,
- Each purse object provides methods to retrieve its own transaction records, its state information (e.g., amount of money associated with the purse, name and address of the associated financial institution etc.),
- The `TransactionRecord` object provides various accessor methods to retrieve the state of a transaction.

These services may be used by the calling applications. Two special applications within the GPSF are responsible for allowing the user to directly retrieve information from the GPSF. In addition to the instrument management services, the purse management application also allows a user to examine the state of a purse. A special application called the *transaction browser* allows users to scroll through transaction records in the archive.

4.3.4 Related Work

Two projects of similar scope to our GPSF work are the E-CO system project [Bahrem96] and the payment sub-project of the Stanford Digital Libraries project [KGPS96]. A mechanism for the negotiation of payment instruments in the E-CO project was described in [BahNar96]. Thereafter, the project has been discontinued. Their generic electronic payment services framework therefore has not been fully defined or implemented. A generic payment service called U-PAI (Universal Payment Application Interface), developed in the context of the Stanford Digital Libraries project was described in [KGPS96]. This system assumes the existence of a distributed object infrastructure like CORBA. It also does not appear to address the aspect of negotiating for suitable instruments (and parameters) before the payment transaction itself begins. On the other hand, the JEPI project, carried out by the W3 consortium, dealt *only* with the negotiation aspect. Our negotiation protocol is not as sophisticated as the JEPI protocol; however, unlike in JEPI, the GPSF includes an API which can be used to drive the protocol. IBM CommercePOINT e-till - similar to our work, the initial design of which was in fact based on our design.

Sun's Java Commerce Client (JCC) architecture (http://www.javasoft.com/commerce) is a major on-going effort in building an extensible framework for enabling electronic commerce. In scope, JCC is comparable to SEMPER as whole. Thus, some of the observations below are not limited to the GPSF alone, but are equally applicable to the other SEMPER blocks. The payment service portion of JCC differs from the GPSF in several respects. JCC is concerned only with the payer side. The payee (assuming it is acting as a server) is required only to know how to construct Java Commerce Messages, which are sent to the payer. The JCC model also distinguishes between a payment *instrument* and a payment *protocol*. In the GPSF, these correspond to the purse and transaction respectively. However, in the GPSF there is a tight binding between Purse and Transaction classes: a Purse object is an abstract factory for Transaction objects. In the JCC model, an instrument can, in principle, be used with multiple protocols. JCC also has a well-developed infrastructure for downloading and installation, as well as for access control.

None of these systems have a fine-grained API like in the GPSF. JCC and JCC-inspired systems allow protocols to be queried about what operations are supported by them. U-PAI provides a method to "start a value transfer."

4.3.5 Self Assessment

Protocol/Instrument separation *a la* JCC is arguably more elegant, even though it is not clear if it finally makes a difference in practice, given that most protocol-instrument relationships are 1-1 anyway. It also does not really buy any modularity for the following reasons:

- the instrument must keep any protocol-specific information (e.g., SET certificates in a credit-card instrument),
- the instrument class and its API may need to be extended each time support for a new protocol is added.

The JCC-style separation may be more intuitive to the user; for example, the user probably wants to select the instrument by name, but the protocol by security services (e.g., “pay with a credit-card, while getting a receipt fairly”).

What exactly is an instrument? Consider the case of a single account, with a single account balance, against which multiple debit cards have been issued: the access control is separate (each card has its own PIN), and the privileges are different (one card has a higher daily limit). In this case, what is an instrument? Is it the card or the account? The protocol/instrument separation does not help deal with this issue.

In SEMPER, we can re-introduce the notion of a “Purse Group” (which was present in an early version of the design) to address this problem: purses in a group will act as distinct entities in value transfers etc., but will appear as a single entity in some user interactions (e.g., checking account balance). There can be different types of purse groups, depending on which operations are grouped and which are treated individually.

The current implementation of the GPSF does not include generic mechanisms to deal with disputes. Our investigations into this issue have resulted in an article containing an overview of the issues involved and an approach towards a solution to the technical aspects of the problem [AVS98].

In our design, two class hierarchies (`Purse` and `TransactionRecord`) provide both default implementations of certain services as well as defining the interfaces for additional services which the subclasses are required to implement. A cleaner approach would have been to separate the two by defining the latter as interface hierarchies similar to the `PurseServices` hierarchy. Adapters can then access default implementations by means of object composition rather than via sub-classing.

4.3.6 Implementation notes and Recommendations

The current prototype implementation of the SEMPER generic payment service framework has the following differences from the design described in this document.

- The token-based interfaces are not implemented by all adapters. Consequently, the non-token-based interfaces are still being used. These are similar to the token-based interfaces except that any communication is done by the adapter itself. For example, instead of a `startPay()` method which returns a token, the non-token-based version has a pay() method which carries out the payment.

4.4 Payment Modules

4.4.1 The SEMPER Chipper Payment Adapter

(J. Swanenburg / KPN)

4.4.1.1 Introduction

The Chipper is a smartcard that is able to store the electronic equivalent of cash money. The Chipper is currently used in the physical world. In paragraph 4.4.1.2 a description of the use of the Chipper in the physical world is given and an analysis of the consequences of moving the payment scheme to the Internet is made. The Chipper implementation in Semper is called KPN Smartcard purse. The reason for this difference is that the commercial version of the Chipper on the Internet will be specified by Chipper NV. The current implementation in Semper is only pilot software. The design of the KPN Smartcard adapter is given in paragraph 4.4.1.3.

The conclusions on the KPN Smartcard adapter are given in paragraph 4.4.1.4.

4.4.1.2 Moving Chipper to the Internet

4.4.1.2.1 Chipper in the physical world

The Chipper is a multifunctional smart card with an electronic purse. It is a Dutch initiative of KPN and the Postbank. The Chipper is meant for the consumer market, a customer can use it for a number of electronic services. The consumer can use the smart card to make payments and when the purse runs out of money it can be reloaded via every public pay phone or at an Automated Teller Machine. Besides the electronic purse the Chipper offers possibilities to store tickets, support loyalty schemes and offer many other services.

The Chipper was designed according to the European ETSI standard. This implies that the Chipper can handle short pulse payments, like phone ticks.

All Dutch public phone boxes are made suitable for reloading the Chipper smart card. The user has to identify himself with his PIN code and chooses the amount with which to load the purse. If the user is valid the amount is put in the purse and the amount of uploaded money is withdrawn from the cardholder’s bank account. Loading a purse is an on-line transaction.

The purse can be used for off-line payments, without the user entering a PIN. Transaction counters are stored in a secured manner inside the secure module of the vending machine or pay phone. Periodically the counters are read and the turnover is paid by the Chipper consortium to the owner of the vending machine or pay phone. Counters can be read via phone lines, collecting cards or other means. During transport the counters are protected by cryptographic means. Also the communication between card and reader is protected by cryptographic means.

The high level interaction between the roles in given in Figure 56.

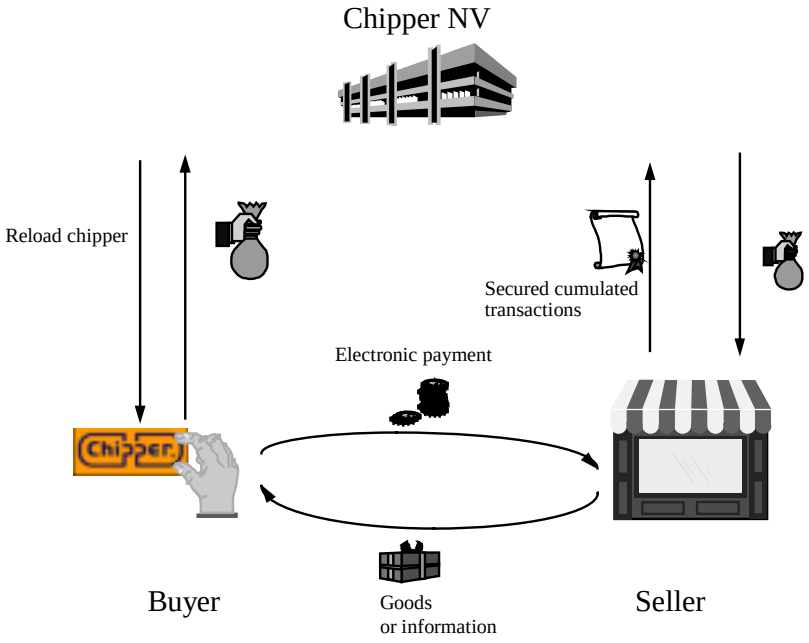

Figure 56: Interaction between roles

The system elements involved in a physical world payment are the user, the Chipper, the cardreader, the secure module and the merchant. The terminal is a cardreader with a display and keypad for the merchant and for the user. The secure module is a tamper resistant chip.

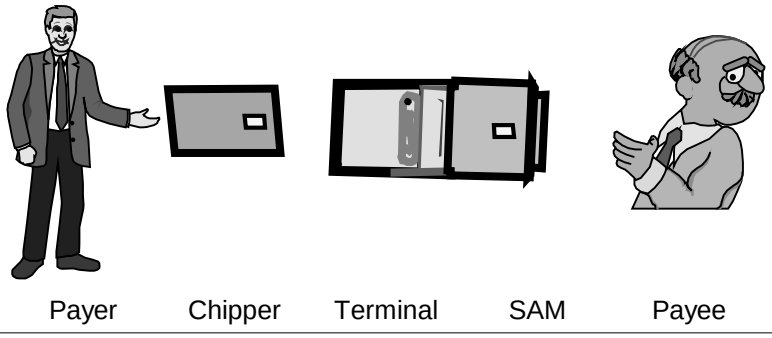

Figure 57: Chipper system elements

The process flows and the interactions between the elements are given in Figure 58.

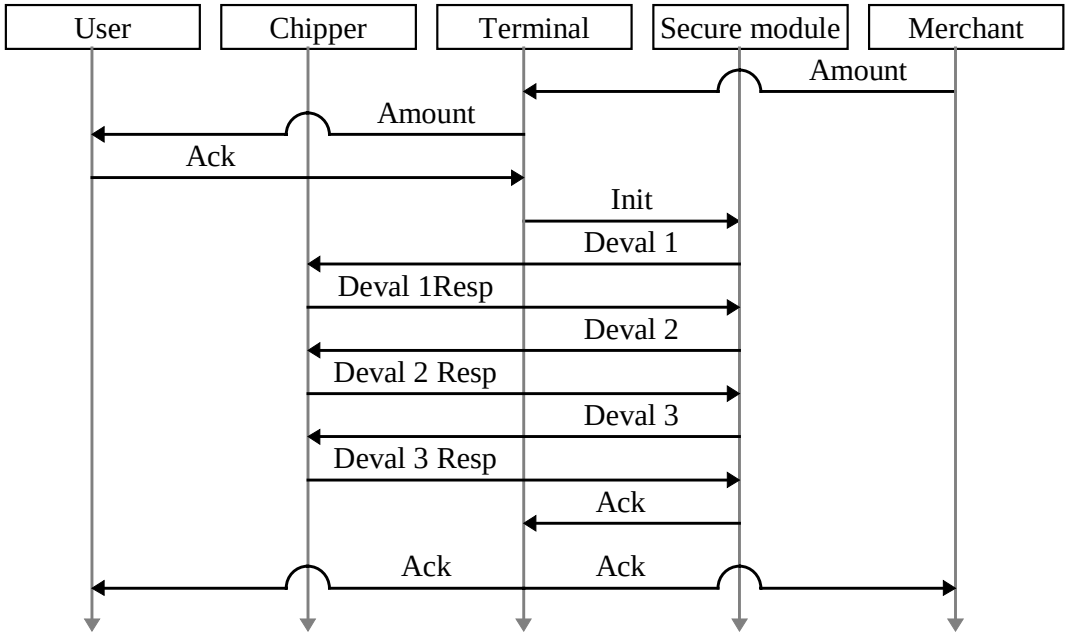

Figure 58: Chipper payment

The process flow can be described as follows:

1. The merchant enters the *Amount* to be paid on the keyboard of the terminal.
2. The terminal shows the *Amount* in the display of the terminal to the user
3. The user enters the Chipper in the terminal and *Acknowledges* the amount
4. The terminal *Initiates* the secure module
5. The secure module and the Chipper start the transaction by doing three challenge response messages
6. The secure module *Acknowledges* that the payment was successful to the terminal
7. The terminal shows this *Acknowledgement* to the user and the merchant in different displays.

The security mechanisms incorporated in the three challenge-response messages guarantee that the payment will not be successful in case:

- the Chipper is not devaluated,
- the counter in the secure module is not increased,
- the chipcard does not contain the right Chipper keyset,
- the secure module does not contain the right Chipper keyset and
- the protocol is not finalised.

Because the counter in the secure module is increased after the Chipper is devaluated, an error in between these phases may cause disputes. Keeping secure evidence solves this problem. This makes it possible to review all phases of a transaction.

4.4.1.2.2 Additional problems in the virtual world

At first sight it is very easy to adapt the physical world payment to a virtual world payment by simply separating the terminal functionality in two parts: One part at the payer and one part at the payee. This is not very difficult but endangers the security of the chipcard concept.

A payment in the physical world has certain security characteristics that are not present in the virtual world. The implicit security characteristics are:

- the payer has more trust in the payee because they are at the same physical location,
- the payer and payee use a certified terminal with trusted keypad and display and
- confidentiality, integrity and availability incidents are very unlikely on the messages from KPN Smartcard to the secure module. The messages flow is inside the terminal.

These implicit security characteristics do not hold for payments where payer and payee are not in the same physical location, like with Internet payments.

4.4.1.2.3 Solutions

The three implicit trust characteristics in the physical world mentioned in paragraph 4.4.1.2.2 have to be implemented for a KPN Smartcard Internet payment.

The trust in the payee in the physical world is based on being in the same physical location. The same kind of trust can be established over a network using authenticated communication channels. Through this mechanism, the payer has confidence in the identity of the payee.

The substitute of the shared certified terminal in the physical world is having two terminals: One terminal with the merchant and one with the user. The functionality at the user is implemented in the terminal and not in the computer for security reasons.

The confidentiality of the protocols is implemented by encrypting the communication between Semper client and Semper server. The symmetric encryption is based on a secret key established during the authentication of the payer and payee. The integrity of the protocol is checked by both the computers and the terminals, because both elements are able to inspect different elements of the protocol. The availability of the messages is very hard to assure through technical countermeasures. Unavailability of messages will result in failed payments. Some of these failures will result in a dispute that is handled in exactly the same way as disputes in a physical world payment.

4.4.1.3 KPN Smartcard adapter design

4.4.1.3.1 System elements

The system elements involved in network payment are the separated between the payer and the payee side of the network. At the location of the payer, the following elements are present: the user, the KPN Smartcard, the TeleChipper and the computer

running the Semper client. The TeleChipper is a card reader equipped with a display and keypad (Figure 59).

Figure 59: TeleChipper

The TeleChipper connects to the computer using a RS323 interface. The merchant has a webserver running the Semper server. A terminal with one or more secure modules is connected to this webserver. The elements are shown in Figure 60.

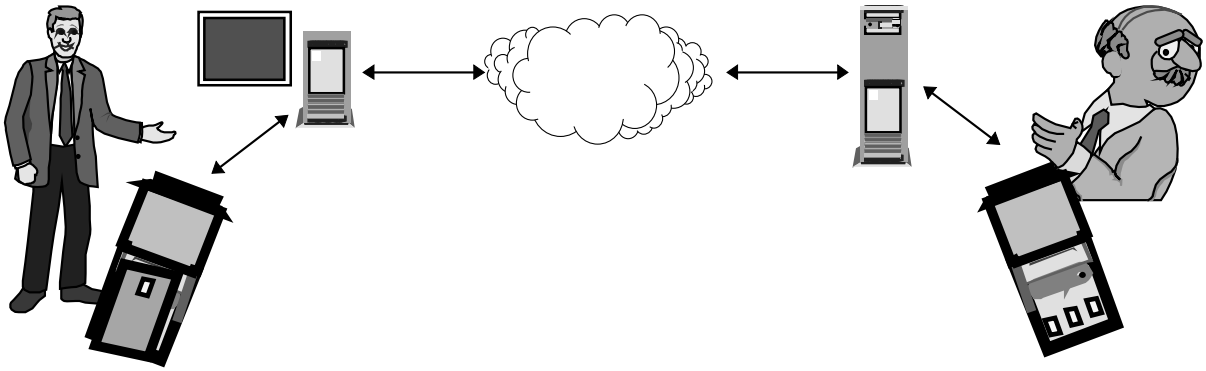

Figure 60: KPN Smartcard system elements

At the highest level of a KPN Smartcard payment in Semper is made up of the following events:

- First the payment manager tells the `chipperPurse` entity to start paying or to start receiving a payment.
- After receiving the message of the payment manager, a new `chipperTransaction` entity is created.
- The `chipperTransaction` has a `chipperTransactionRecord` associated with it. This record is used to store the result of the transaction to the archive.
- The `chipperTransaction` calls the `chipperModule` to start the payment or the receiving of a payment.
- The result of the payment is returned by the `chipperModule`. The result is in a `chipperEvidence` entity format.

- The evidence is stored in the archive and the payment manager is informed of the result of the payment.

The process flow and the interaction between the elements on a detailed level are given in Figure 61**.**

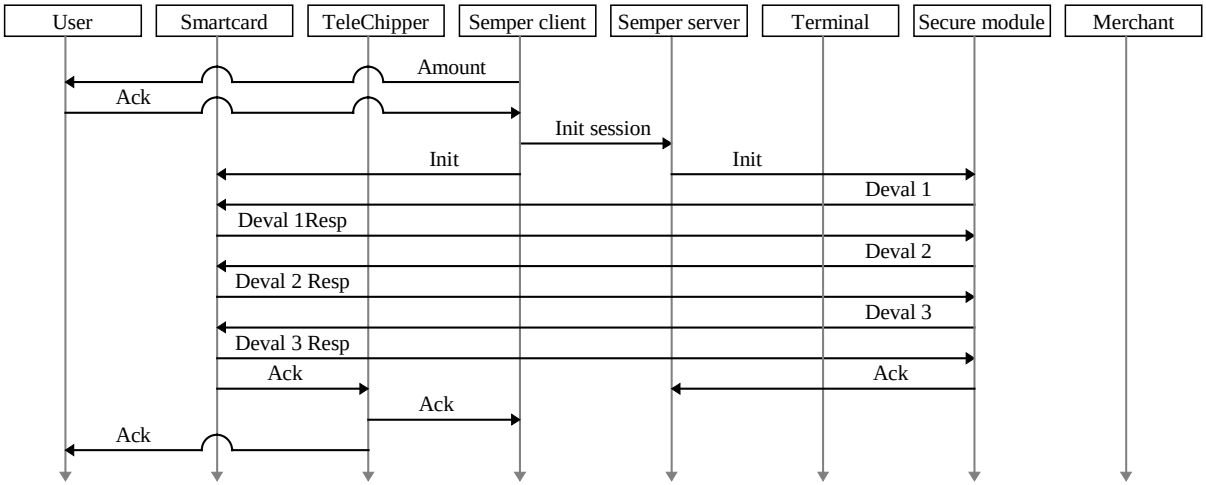

Figure 61: KPN Smartcard payment

The process flow can be described as follows:

1. The Semper Payment Manager initiates the KPN Smartcard transaction. The Tinguin shows the KPN Smartcard data and the *Amount* to the user.
2. The user acknowledges the *Amount* in the Tinguin.
3. The KPN Smartcard transaction object of the Semper client sends an *Init session* message to the KPN Smartcard transaction object of the Semper server. During the Init session message an authenticated and encrypted channel is established.
4. The Semper client *Initiates* the TeleChipper and the KPN Smartcard.
5. The Semper server *Initiates* the Terminal and the secure module.
6. The secure module responds with a *Devaluation 1* message. This message is send through the terminal to the Semper server.
7. The Semper server adds information and encrypts the new message and sends it to the Semper client.
8. The Semper client decrypts the message, checks the protocol information and forwards a part of the message, through the TeleChipper to the KPN Smartcard.
9. The Devaluation Response messages flow in the other direction.
10. The steps 6 to 9 are repeated 2 more times.
11. After receiving the correct *Deval 3 Response*, the secure module sends an *Acknowledgement* to the Semper server.
12. After sending *Deval 3 Respons*, the KPN Smartcard informs the TeleChipper of a successful payment.

13. The TeleChipper sends an *Acknowledgement* to the Semper client.
14. The TeleChipper shows this *Acknowledgement* to the user in the TeleChipper display.
15. The KPN Smartcard transaction object, both on the Semper server and on the Semper client, *Acknowledge* the successful payment to the payment manager.

Through this approach, the solutions described in paragraph 4.4.1.2.3 are implemented.

4.4.1.3.2 Software interfaces

The most important parts of Semper, the KPN Smartcard payment adapter interfaces with, are:

- the Payment Manager (see Section 4.3)
- the Secure Communication Block (see Section 4.7)
- the Tinguin (see Section 4.11)

The KPN Smartcard payment adapter implements the objects and methods expected by the Payment Manager. These objects are the `chipperPurse`, the `chipperTransaction` and the `chipperTransactionRecord`. The Secure Communication block is used for setting up the authenticated encrypted channel between payer and payee and for sending and receiving messages to and from the network. The Tinguin is used for interaction between the KPN Smartcard adapter and the payer.

4.4.1.3.3 Object model

In this paragraph a short description of the objects used in the KPN Smartcard payment adapter is given.

The adapter is implemented in the form of the classes:

- `chipperPurse`: This object encapsulates all the data elements and functions for a Semper purse entity. A `chipperPurse` entity interfaces with `chipperTransaction` and the Payment Manager. The `chipperPurse` is a subclass of `CashLikePurse`.
- `chipperTransaction`: A `chipperTransaction` entity interfaces with a `chipperTransactionRecord` , to write the transaction information to stable storage, and the `chipperModule`, which is the core of the transaction processing. The `chipperTransaction` is an implementation of `CashLikePurseServices`.
- `chipperTransactionRecord`: A `chipperTransactionRecord` entity is used to write a transaction to stable storage. The record encapsulates a handle to the `chipperEvidence` object associated with the record. A `chipperTransactionRecord` entity interfaces with a

chipperTransaction. The chipperTransactionRecord is a subclass of PaymentTransactionRecord.

- chipperEvidence: A chipperEvidence entity encapsulates all evidence accumulated during a transaction. The evidence collected during a KPN Smartcard transaction consists of all the messages read from or written to the network. A chipperTransaction entity interfaces with chipperTransactionRecord and the chipperModule that adds the messages to the evidence.

The payment system itself is implemented in the classes:

- chipperModule: A chipperModule entity encapsulates all methods needed for a transaction. A chipperModule entity interfaces with:
  - chipperTransaction, because it carries out the methods of the transaction,
  - chipperProcessor, because the chipperHardwareMessages are sent to a chipperProcessor entity,
  - chipperModuleException , because if an error occurs during transaction this exception is thrown,
  - chipperEvidence, the messages sent during a chipperModule method, the chipperNetMessages, are added to the evidence.
- chipperModuleException: Generic exception class for the chipperModule. This exception is thrown if an error occurs in the transaction protocol. This can either be the result of a chipperHardwareException thrown by the hardware or the result of an error during the communications over the network.
- chipperNetMessage: Message class for net protocol messages in the KPN Smartcard payment system. The protocol through which the chipperNetMessages are sent, is given in the chipperModule class.
- chipperHardwareMessage: Message class for hardware protocol messages in the KPN Smartcard payment system. The protocol through which the chipperNetMessages are sent, is given in the chipperProcessor class.
- chipperHardwareException: Generic exception class for the communication with the KPN Smartcard hardware.
- chipperProcessor: The class chipperProcessor is not really an object, but it is an implementation of the wrapper class and the protocol flow of the chipperHardwareMessages. A chipperProcessor entity interfaces with chipperModule, because it receives and passes through the hardware commands, and with chipperHardwareException, if hardware processing of the command fails.

The relation between the most important classes is given in Figure 62: Object model.

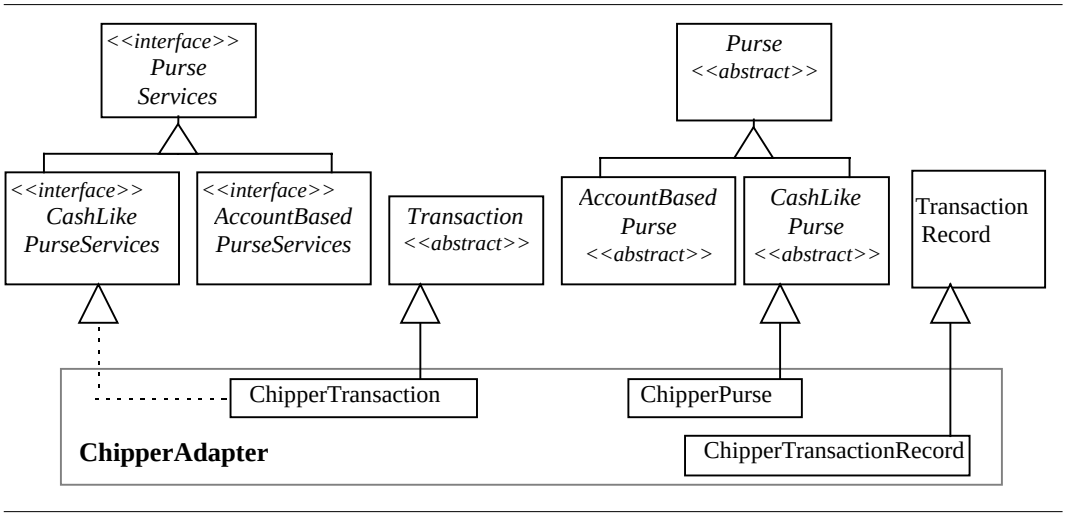

Figure 62: Object model

The relation between the software entity, chipperPurse, and the hardware entity, the smartcard is a one to one relation. As a consequence this means that there is a fixed relation between a purse and a card. This option is not chosen because of a software restriction, but it gives an easy way to explain both concepts to the user.

4.4.1.3.4 Protocols

The current implementation makes use of two stacked protocols. The separation of these two protocols is a result of changing an existing protocol to a new application. The two protocols used are:

- `chipperHardwareMessage`: The ChipperHardware messages are the same messages as used in a physical world payment. These messages guarantee the security characteristics given at the end of paragraph 4.4.1.2.1.
- `chipperNetMessage`: The ChipperNet messages encapsulate the ChipperHardware messages, handles encryption and error control. These messages guarantee that the additional problem mentioned in paragraph 4.4.1.2.2 do not occur.

The relation between the two protocols is given in Figure 63.

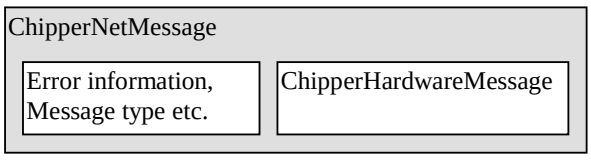

Figure 63: Protocols

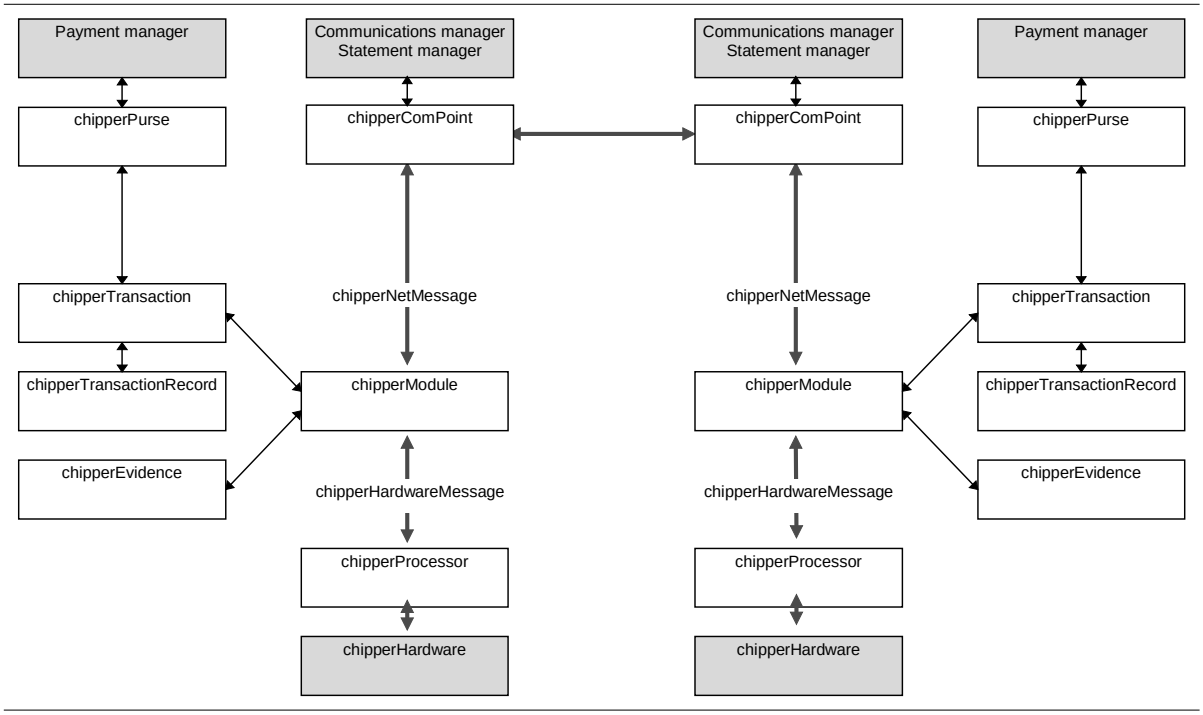

Figure 64: Protocol flow through the objects

4.4.1.3.5 Fault tolerance

Before getting into the fault tolerance of the KPN Smartcard payment system, it must be clear that the remarks made in this document are only valid for the current implementation of the KPN Smartcard purse. The fault tolerance and recovery strategies of the use of KPN Smartcard with a terminal in a shop depend on the pay terminal specifications. This means different characteristics in respect to fault tolerance. The characteristics of the real system for KPN Smartcard payments on the Internet will be different from the current Semper implementation.

A described in paragraph 4.4.1.2.1, a KPN Smartcard payment is a chip to chip payment. Between the chips there is a challenge-response protocol. Both the chips at the client side and at the server site are not able to reinstall a state before the crash. In general two kind of error can occur:

- a network error or
- a computer or chip error.

Network error

If a network error occurs the KPN Smartcard protocol message should be resent by the client or server before a hardware reset occurs.

Computer or chip error

After a computer error, the software or the computer has to be restarted. This will result in a reset of the chip. If a chip error occurs the chip has to be reset. Because it is not possible to reinstall the state in the chip, it is not possible to resume the transaction. In most cases this is not a problem. In case an error occurs in the grey box of Figure 65, there is a possible dispute.

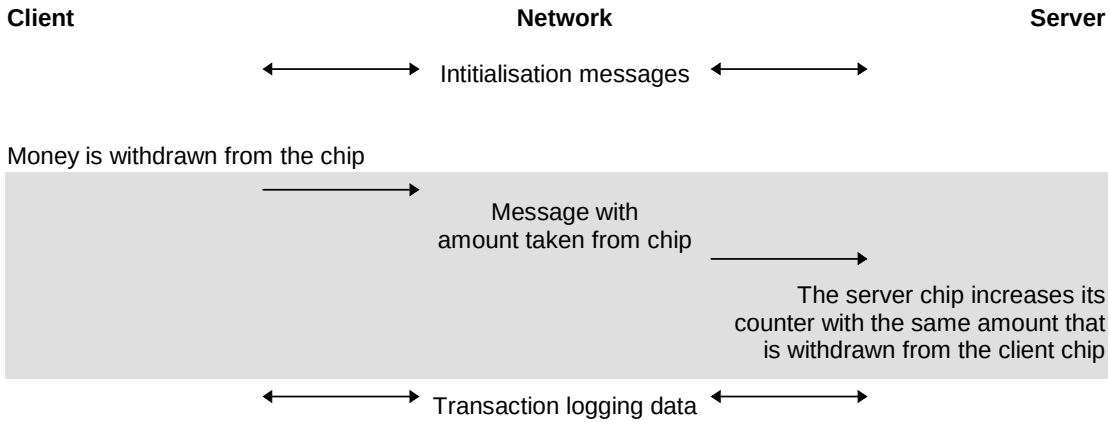

Figure 65: KPN Smartcard protocol flow

If something goes wrong at the client or at the server in the processes in the grey box, the evidence about the purse devaluation is stored within the client chip. This evidence can be presented to the money-issuing bank. The bank will take the appropriate actions.

4.4.1.3.6 User Interface

The interaction between the payer and the Semper client is minimised to an absolute minimum. During a payment the only user interface object that is shown to the user is given in Figure 66.

Figure 66: User interface

After the payment the display on the TeleChipper shows either the amount paid or an error message.

4.4.1.4 Conclusions

An existing physical world payment scheme can not be ported to a network environment without additional security countermeasures. The design of the Semper KPN Smartcard payment adapter solves the new security problems introduced by the migration of an existing physical world payment scheme to a network environment.

Because no software for network based KPN Smartcard payments existed before the design was made, it was possible to make a tight integration with the other Semper blocks. Examples of this tight integration are:

- the use of the Tinguin for the user interface,
- the use of the Semper cryptographic tools through the Secure Communication Block for establishing an authenticated and encrypted channel and for messaging, and
- the easy interface with the Payment Manager during purse setup and payment.

The current implementation in Semper is only pilot software; the commercial version of the Chipper on the Internet will be specified by Chipper NV.

4.4.2 The eCash Payment Adapter

(B. Schoenmakers / DIC)

4.4.2.1 Introduction

The *eCash* payment adapter forms the interface between the SEMPER payment manager on the one hand and an *eCash* engine on the other hand. In this way, the bulk of the electronic commerce transactions are performed within the SEMPER framework, while the actual payment may be deferred to the *eCash* engine. Also, due to the generality of the SEMPER framework, it will be possible to combine the use of several *eCash* accounts (each represented by an `eCashPurse`) from a single SEMPER client. This is a consequence of the fact that a SEMPER client, or rather its payment manager, may keep a plurality of so-called Purses, which need not even be of the same type. For instance, a user may have installed some SET purses, some Chipper purses, as well as some `eCashPurse`s.

4.4.2.2 eCash

eCash is a payment system designed and implemented for making purchases over open networks such as the Internet. The system finds its roots in the work by David Chaum, who invented the notion of electronic (or digital) coins as well as the basic protocols for electronic cash. Electronic coins possess similar properties as metal coins, among which is the unique feature that a payment transaction leaves no trace about the identity of the payer. Currently, *eCash* technology (as provided by DigiCash Inc., see http://www.digicash.com for more details) is used by a number of banks around the globe. These banks issue *eCash* to their customers, who can then spend it at affiliated merchants on the Internet.

eCash is a coin-based electronic payment system. Briefly, this means that monetary value is represented by digital coins, hence the phrase “numbers that are money.” Coins are created in an interactive withdrawal protocol between a customer and a bank. Later, these coins are spent at a shop, which means that the shop verifies on-line with the bank that the coin hasn’t been spent before. Conceptually, a payment consists of ordering a bank to transfer a particular amount of money to a shop by means of a set of unforgeable tokens.

Only valid coins will be accepted for payments, and it is assumed to be infeasible to create valid coins without a corresponding execution of the withdrawal protocol. The validity of a coin can be checked by anyone who disposes of the relevant public key; more specifically, the validity of a coin is checked by verifying its RSA signature (on a message of a particular form).

The *eCash* system is particularly suited for a networked environment in which all of the participants are online. The payment protocol itself is fast and relatively simple, and it requires a minimum of interaction only. Since every payment is checked online with the bank, it follows that no monetary value is kept at the shop. Hence, there is no

risk for a shop being robbed from its *eCash* coins, as the coins in a payment are all cleared with the bank as part of the payment protocol.

A special feature of the *eCash* system is also that the customer remains anonymous during the payment protocol. This is due to the fact that the coins used in a payment cannot be linked in anyway with the customer's account from which the coins have been withdrawn. This is accomplished by the use of Chaum's blind signature protocol.

We confine ourselves to the four basic *eCash* operations: open account, withdrawal, payment, and deposit.

4.4.2.2.1 Open account

The main inputs for this operation are:

- userID
- password
- accountID

This function allows any user to open an account after a negotiation stage with the bank. As a result of the negotiation with the bank, the user receives a password. The password may be sent to the user by a conventional (secure) channel, e.g., surface mail; it is required to open the account.

4.4.2.2.2 Withdrawal

The main inputs for this operation are:

- currency
- amount
- bankID
- accountID
- balance

This function allows any user to withdraw a sum of coins for a specified currency and amount. The account from which the amount should be debited and the bank at which this account runs are also specified.

4.4.2.2.3 Payment

The main inputs for this operation are:

- currency
- amount
- bankID /* of shop */
- accountID /* of shop */
- description

This function allows a customer to send a payment for a specified currency and amount. The account to which the amount should be credited and the bank at which this account runs are also specified. A description of the payment is included. (The accountID of the customer is not relevant for the payment protocol.)

As part of the transaction between a customer and a shop, the shop will await the incoming payment from the customer. If the received payment is of the correct form (e.g., the correct currency and amount are specified), a deposit transaction with the bank is initiated. This means that the payment is forwarded to the bank, after the payment description has been stripped from all superfluous information.

4.4.2.2.4 Deposit

The main inputs for this operation are:

- currency
- amount
- bankID
- accountID

This function allows any user to deposit an amount of a specified currency. The account to which the amount should be credited and the bank at which this account runs are also specified.

4.4.2.3 SEMPER/eCash adapter

The main building blocks of the implementation of the *eCash* payment adapter consists of the `ecashPurse` class and the `ecashTransaction` class. These classes are extensions of the corresponding abstract base classes from the payment block.

The `eCashPurse` takes care of the open account protocol as part of the set-up procedure for a purse. Furthermore, the purse allows the user to do withdrawals and deposits to control the amount of *eCash*. Hence, these operations are all initiated by the user. The payment operation, on the other hand, is initiated through the payment manager, as part of an electronic commerce transaction. For each financial operation (withdrawal, deposit, and payment) the `eCashPurse` generates an `eCashTransactionRecord`. The transaction records may be viewed later through the payment transaction browser. Access to the `eCashPurse` is possibly controlled by a password.

Depending on the way the *eCash* engine is implemented, two approaches have been used to communicate with the *eCash* engine. The preferred approach is to have the *eCash* engine available as a dynamically loadable library, which is then called from the *eCash* adapter (implemented in Java) using native method calls. However, for some OSs the dynamic library may not be available. The alternative approach is applicable when the *eCash* engine is available only as an executable with a command-line interface. In that case each *eCash* call is implemented by starting a process

executing the *eCash* executable; the parameters passed to the process are treated as command-line inputs. (See the `java.lang.Process` class.)

Since the bank that issues *eCash* need not be aware of the fact that a user is actually using its *eCash* client from within SEMPER, all communication with the *eCash* bank is done by the *eCash* engine itself. So, when a user performs a withdrawal, or a shop performs a deposits a payment, the *eCash* engine needs to connect to the *eCash* bank. Clearly, this requires that the *eCash* engine is allowed to connect through the firewall, if present. The communication between a customer and a shop, however, may be deferred to SEMPER.

4.4.3 Mandate

(T. Pedersen & B. Dahl Ebert / CRM)

MANDATE II[21] was a project under the EC DGXIII ETS (Electronic Trusted Services) programme developing an electronic cheque. It was the intention of the project to create an electronic negotiable instrument mirroring all the functionality of a paper cheque.

Central to the technical solution is a 'DOC-carrier' (DC), which is tamper-resistant and implemented in some special hardware such as an advanced smart card. The DC can be thought of as an electronic chequebook and is initialised by the issuing bank by entering account information etc. Furthermore two public key pairs are generated (one for signing and one for encryption) and certificates on the keys are issued. The corresponding private keys are generated and kept securely on the DC.

4.4.3.1 Background

The goal to achieve is that an electronic cheque - or rather, an electronic realisation of a cheque - at any particular time can be proved to be the (temporary) property of a particular user. With ordinary paper documents, giving the original of a document certain physical attributes that are difficult to reproduce solves the problem. With this precaution, it makes sense to speak of the original of a document, and define the owner simply as the person holding the original.

The important properties required are *uniqueness* and *ownership*. The problem is to find a suitable attribute on an electronic cheque, which somehow would give it the property of uniqueness, explicitly or implicitly in such a way that it makes sense to talk about being the rightful owner of the cheque.

Obviously, the concept of a paper-based original does not make sense for electronic documents, and other (e.g. cryptographic) methods must ensure that the owner of a particular electronic cheque can be identified.

Any document of some value must initially be generated by someone, who will guarantee its value. This requires the proof of, or non-repudiation of, origin service (one example is electronic cash), which is a well known art and realised using digital signatures.

An electronic cheque contains information on amount etc. and a digital signature by the issuer on this information making the cheque valid. In fact the (verifiable) signature is the piece of information that has the value. The question now is how to develop a protocol to cover the situation, where a cheque, once issued, is to change hands. The main problem is to ensure, that the new owner is uniquely identified, or, in other words, that the payee cannot circumvent the measures and sell the same cheque (the issuer's valuable signature) to two different entities.

[21] See *http://www.cryptomathic.dk/mandate/*

4.4.3.2 The Use of Tamper Resistance to Achieve Uniqueness

Realisation of freely negotiable electronic documents has only been analysed very recently in the literature. [TEDIS93] and [MANDATE]) give the general principles, but no specific solution. In the following, we give a brief overview of the requirements defined in these reports.

The only practical way to provide uniqueness is to physically prohibit free copying. This would involve tamper resistance to realise a protected communication with restricted functionality, if possible.

The basic principle for achieving ownership and uniqueness here is simple but fundamental:

Ownership: A message encrypted under a key known to only one entity belongs to that entity, as long as it is encrypted, and establishes indisputable ownership by the mere fact that it will only be useful to the owner of the key. Only the person in possession of the right key can make any use of the document, which in effect is the property of ownership. The rightful owner of a cheque is the person being able to present the issuer's (decrypted) verifiable signature.

Uniqueness: On the other hand, the only way the rightful owner can verify the encrypted signature of a cheque is by decrypting it. But this will give him access to the signature and he may subsequently be able to sell the cheque to two different persons by encrypting the signature with their respective keys. To achieve uniqueness this must be prevented by protecting the valuable signature using tamper-resistant hardware, perhaps a chipcard, or a hardware protected PC.

In MANDATE this tamper-resistant hardware is called the DOC-carrier. Its properties will be described in detail.

4.4.3.3 The MANDATE Cheque System

The MANDATE cheque system has the following properties:

Cheques are generated electronically on the tamper-resistant DC. This requires some care. It is essential that the signatures generated to provide the non-repudiation is never disclosed. It is of course sufficient to represent each cheque by a hash value and generating the digital signature inside the DC. The message itself does not need to be protected.

A cheque is transferred from one DC to another, through a public unprotected network (e.g. by e-mail), in such a way that the following properties hold:

- It can only be transferred as a meaningful document to one particular DC chosen from any number of DCs
- The protocol cannot be completed by any other device than an authorised DC.
- The system is completely open to communication between any two DCs, without bilateral agreements.

Each DC "possesses" two public key pairs; one for signing and one for encryption. The private keys are not known even to the owner of the DC. They are generated on the DC and never leave it, hence not even the system provider knows these private keys.

The cheque consists of the information (payee, amount, timestamp, expiration date etc.), and the corresponding digital signature, calculated by means of the private key of the payer and a hash value of the information. It is important to realise that the value of the cheque is represented by the digital signature of the issuer.

Now, this cheque shall only be released through a selling process to another DC. So the question is, how can one DC identify another?

In MANDATE the public key of the CA is installed on each DC, along with the before-mentioned key-pairs. When a document, or rather the corresponding protected signature, is entered on the DC, the DC software ensures that it can only be released again encrypted under a public key certified by the CA (and verified by means of the corresponding public key on the DC). This prevents the use of a non-authorised DC to get access to the vital signature that defines the cheque. In particular this encrypted message is useless except when imported into the DC holding the corresponding private key.

Furthermore, and this is an essential property, such an encryption of a particular cheque on an individual DC can take place only once, or rather, once a public key has been selected, it is impossible at any later stage to go through the same procedure with another public key.

4.4.3.4 The Pilot Project

A pilot project has been carried out with participation of three European Banks, viz. Nordvestbank, Denmark, Royal Bank of Scotland, and ING Bank, The Netherlands. In the pilot a software-only implementation was used to test the functionality of the electronic cheque concept.

4.4.3.5 Integration into SEMPER

To test the concept of electronic cheques, a MANDATE adapter has been developed for SEMPER. More precisely, a MANDATE Purse class has been written, which “plugs” into the SEMPER Purse Management and Payment Management system. MANDATE falls under the account-based model. The Java classes for the account-based model have been extended with an interface that is common to all electronic cheque systems. The MANDATE module is based on the DC used in the pilot project. Hence, it is software-only, and the DOC-carrier is replaced by a data file simulating the content of the DC, as well as a software component (a Windows DLL file) that simulates the behavior of the DC. The electronic cheques, i.e., the pieces of data that are transferred between different DC-devices, are similar to those that would be transferred between real tamper-resistant DOC-carriers.

Being a SEMPER module, some specific modes of operation has been forced upon the implementation. These modes are not necessarily the ones that would be used in other applications of MANDATE. Specifically, transactions occur online, which means that

bi-directional communication is used (e.g., as opposed to sending a cheque by email). More precisely, the paying purse initiates the protocol by requesting the encryption certificate of the receiving purse. After receiving and validating this certificate, the cheque is issued (i.e., signed by the signing key of the paying DC), encrypted for the receiving DC, and sent to the receiver, which sends back an acknowledge message. This completes the cheque delivery protocol for MANDATE in SEMPER.

4.4.3.6 Technical details of the integration

The integration of MANDATE into SEMPER consists of

- Writing a set of Java classes, most of which specialize classes belonging to the SEMPER Payment Manager.
- Writing a small program that issues a "DOC-carrier" (in a file) with specified parameters (e.g., Owner ID, Bank ID, account number).

The mentioned Java classes are

Cheque. Objects of this class represent the data transferred between buyer and seller in a cheque payment transaction.

mandateModule. Implementing low level MANDATE functionality, protocol, and access to functions exported by the DLL library, that implements the behaviour of a DC.

mandateModuleMessage. Low level message objects, wrapping e.g. Cheque objects.

mandateEvidence. For storing evidence from transactions. All messages in a transaction are stored in a mandateEvidence object.

mandateModuleException. Technical class for informing about exceptional conditions occurring during the execution of MANDATE methods.

mandatePurse. This is the adapter between the SEMPER payment system and a MANDATE DOC-carrier. It implements the functionality required by the payment manager, i.e. informing about capabilities of the system, starting transactions, letting the user configure the purse, etc. This class also implements the necessary interaction with the user for getting the password protecting the DC and presenting available and received cheques.

mandateTransaction. Objects of this class represent single transactions for a given purse. It implements the functionality of paying and receiving a payment by calling the relevant cheque issuing and reception methods.

mandateTransactionRecord. Objects of this class represent the information concerning a mandateTransaction that must be saved to stable storage (i.e., that persists after the SEMPER session). This is information about the remote party, the kind of transaction, the mandateEvidence of the protocol, whether the transaction succeded, etc.

Util. This class implements some technical utility functions needed by other parts of the MANDATE implementation.

One useful extension to the Semper payment negotiation protocol specific to MANDATE would be the inclusion of parameters for negotiating endorsement of cheques. However, no mandate-specific extensions have been made to the negotiation protocol, so endorsement of cheques is not possible with the current MANDATE payment adapter.

Also, every kind of interaction with the bank (i.e. reception of a bundle of blank cheques, cashing of cheques, etc) is left out of the SEMPER system, since these are actions that are performed independently of the Payment system of SEMPER.

The present integration has been successfully tested in the environment of the Payment Manager test application of SEMPER, as well as with the Fair Internet Trader application of SEMPER.

4.4.4 SET

(J.Abad-Peiro & T. Schweinberger / ZRL)

4.4.4.1 Introduction

This section describes the SET adapter for SEMPER. The SET Secure Electronic Transactions (SET) protocol, [SET1.0, SETSP1.0] is the standard protocol for credit card-like payments specified by Visa and MasterCard in cooperation with GTE, IBM, Microsoft, Netscape, SAIC, Terisa, and VeriSign.

SET is a pragmatic approach that paves the way for easy, fast, secure transactions over the Internet. It seeks to preserve the existing relationships between mer-chants and acquirers as well as between payers and their bank. SET concentrates on securely communicating credit card numbers between a payer and an acquirer gateway interfacing to the existing financial infrastructure.

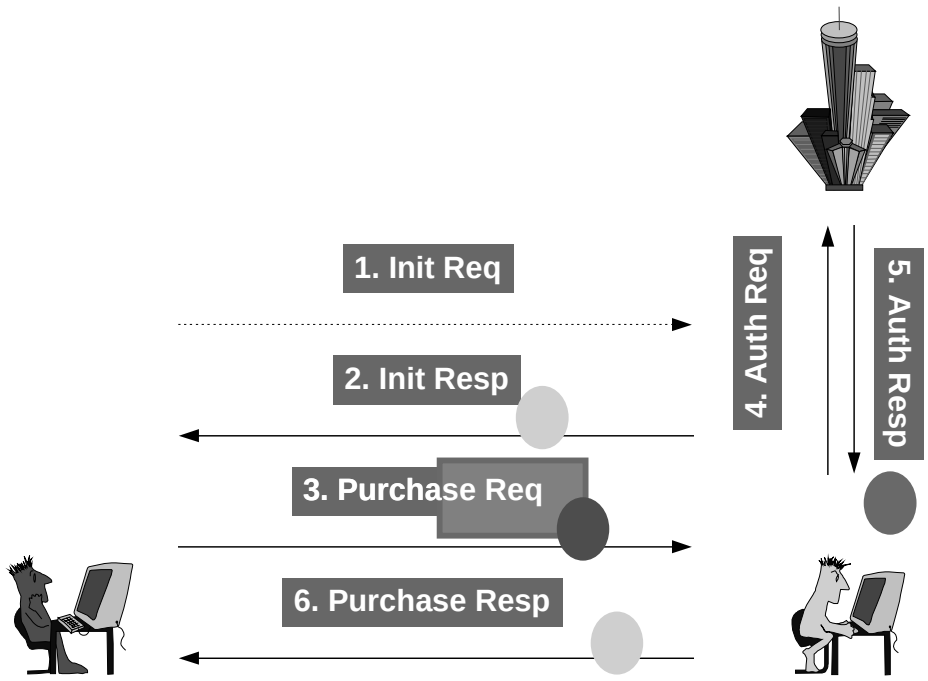

Figure 67: SET Protocol

In our classification, SET falls under the account-based model. The transaction is initiated with a handshake, with the merchant authenticating itself to the payer and fixing all payment data. The payer then uses a sophisticated encryption scheme to generate a payment slip. The goal of the encryption scheme is to protect sensitive payment information (such as the credit card number); limit encryption to selected fields (to ease export approval); cryptographically bind the order information to the payment message; and maximise user privacy. Next the payment slip is signed by the payer and is sent to the merchant. The merchant sends the slip to its acquirer gateway,

to authorise and capture the payment. The acquirer checks all signatures and the slip, verifies the creditability of the payer, and sends either a positive or negative signed acknowledgement back to merchant and payer.

4.4.4.2 Integration Environment

The SEMPER SET adapter builds on an existing toolkits and was validated in tests and a trial with CommerzBank and Otto Versand using IBM Payment Gateway and the IBM SET CA[22] described below.

4.4.4.2.1 IBM SET Development Toolkit

The SEMPER SET adapter builds on top of the IBM SET Development Kit (SDK) [IBMSET]. The SDK toolkit consists of a suite of libraries and APIs that can be used by electronic commerce applications to perform and receive SET payments.

The functionality of the SDK can be divided into two main branches: *First* to create (prepare, encrypt and store) PDUs (Protocol Data Unit) for buyer, merchant and acquirer applications; and *second* to retrieve (decrypt, process, check and store) financial information from the PDUs according to the SET protocol. The SET protocol implemented by the SDK grants confidentiality and integrity of the PDUs exchanged among players.

The SDK internally defines a component called IBM Payment Manager [IBMSET] as the kernel of the IBM's SET implementation. There is another layer called Generic Application Layer containing templates for electronic commerce applications.

The following table presents several modules required by the SDK.

Module	From	Function	Version
bsafe	RSA Data Security Inc.	Encryption/Decryption libraries	RC2_VERSION_PI_SUBST
oss	Open Systems Solutions	ASN1 libraries	OSS_VERSION_412
stl	Hewlett-Packard	To compile the same C++ sources in Win95/NT and UNIX systems	
Cms	IBM	Certificate Management System	

4.4.4.2.2 IBM SET Certification Authority

The SET Certification Authority (SET CA) [IBMCA] is an agent of one or more payment card brands that provides for the creation and distribution of electronic certificates for SET cardholders, SET merchants, and SET payment gateways [IBMSET].

The CA is a key player on any electronic commerce scenario. We assume that the reader is already familiar with certificate management concepts. The functionality of the SET CA can be divided into 3 main parts [SET1.0]:

[22] The certificates used in the trial were actually retrieved from a CA operated by GTE.

- Receive registration requests.
- Process and approve/decline requests.
- Issue certificates.

All three steps may be performed by the same organisation, e.g. companies that issue their own proprietary cards. Typically financial institutions receive, process and approve cardholders' certification requests forwarding the information to the appropriate card-brand certification issuer. Merchants and acquirers requests are usually received by an independent Registration Authority (RA) that processes card certificates for multiple payment brands and forwards the requests to the appropriate acquirer for processing.

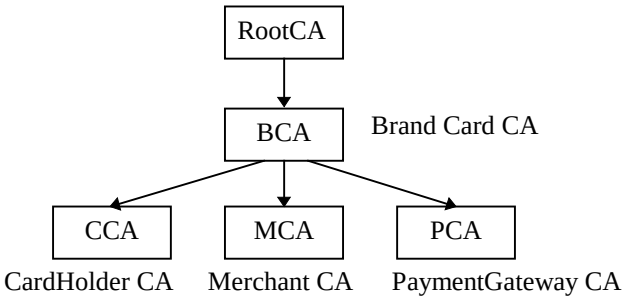

Figure 68: Hierarchy of SET CAs

4.4.4.2.3 IBM SET Payment Gateway

The Payment Gateway is the software operated by an acquirer or a third party that processes payment messages from a merchant (including payment instructions from cardholders) [IBMSET].

The Payment Gateway converts SET payment messages to specific protocol formats used in financial networks, for example to the banking industry standard protocol ISO8583. Customised translation services are provided by a simple user exit interface (see [IBMPG] for technical information).

The PG provides user exits to easily customise message formats to specific legacy systems via SNA LU 6.2, TCP/IP and X.25 interfaces. The Payment Gateway functions as a network gateway for SET payment messages. It redirects incoming and outgoing SET messages between an insecure Internet and financial networks or legacy credit card processing systems. Additionally it translates and checks the payment protocol formats.

The Payment Gateway, in Figure 69, has two main components: the Payment Gateway Transaction Manager (PGTM) and the Payment Gateway Application (PGA). The PGTM enables the development of generic applications independent of specific communication, encryption and translation services. The PGA is implemented in this way using PGTM services to perform all communication functions with merchant servers, legacy hosts and certificate authorities.

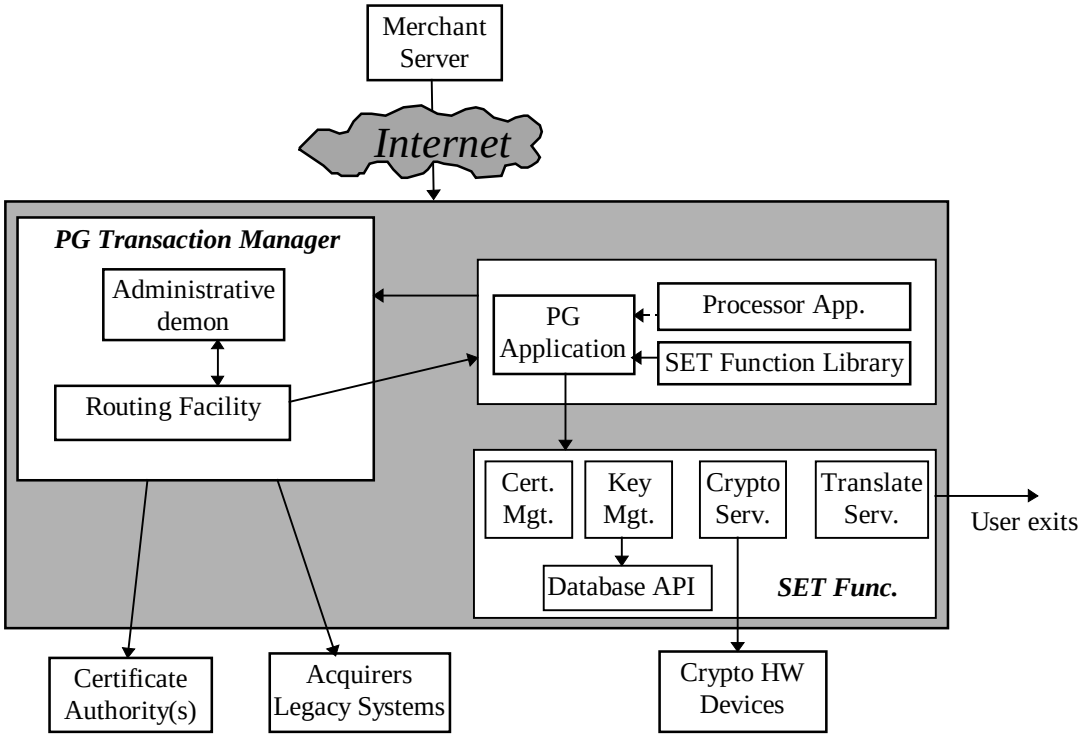

Figure 69: SET Payment Gateway

4.4.4.3 SEMPER/SET Adapter

The Payment Adapter for SET (SET Adapter) exists only for buyers and merchants as the SEMPER PaymentManager framework only focused on these two players. It would however be possible to build a SET Adapter for acquirers and the Certification Authority without too much additional effort.

4.4.4.3.1 Adapter Design

There is a single adapter module for cardholders as well as merchants with differences regarding some specific API calls mainly within the setup procedure. The design is based on the decomposition of the payment domain into instruments and protocols. In the context of the SET protocol an instrument logically represents a cardholder's credit card or a merchant's acceptance device by maintaining connections to certificate and transaction databases. SET registration protocols provide a means to register instruments whereas SET purchase protocols perform payments applying instruments.

The SET Adapter for SEMPER may be divided into two layers: a SEMPER related high level, and a SET related low level layer. The SEMPER related high level design of the adapter implements the SEMPER specifications for Payment Adapters. Figure 4 gives an overview of the design. These specifications are mainly defined in `setPurse` and `setTransaction` classes.

Purses represent instances of installed payment systems that can be referenced by the user. The `setTransaction` class will organise a payment call in order to execute

the SET flow in the right order (send *PInitReq*, get *PInitRes*, etc.). Error handling and temporary transaction storage takes place at this level.

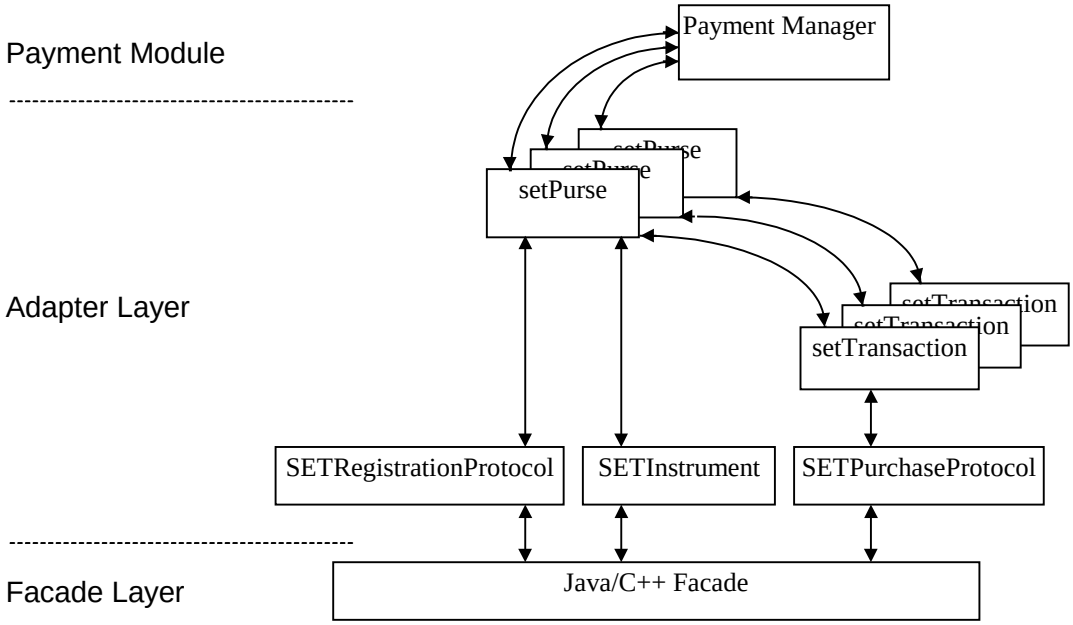

Figure 70: Design of the SEMPER/SET adapter

The SET related low level design of the adapter maps IBM SDK API into Java (see Figure 5). In this way, the SEMPER `setTransaction` can easily call the SDK methods. This part of the adapter is a dynamic library loaded during initialization of a Purse.

The process to build the Java SDK library consists of creating C++ objects to wrap the SDK API as well as the SDK data structures. These C++ objects are further referenced as a “Facade” to the SDK. The C++ Facade will declare each exported method from the SDK through the Java Native Code Interface [JNI1.1] to separate Java package, the `SETFacade`.

We follow Sun’s directives to build Java accessible dynamic libraries by linking all the following involved components: Java Facade, C++ Facade, SDK libraries, and several header files for the helper classes used in both facades. Header files for the helper classes are automatically obtained by using the *javah* application applied on each relevant Java file.

After the linking process, we obtain a dynamic library to be used in the adapter layer.

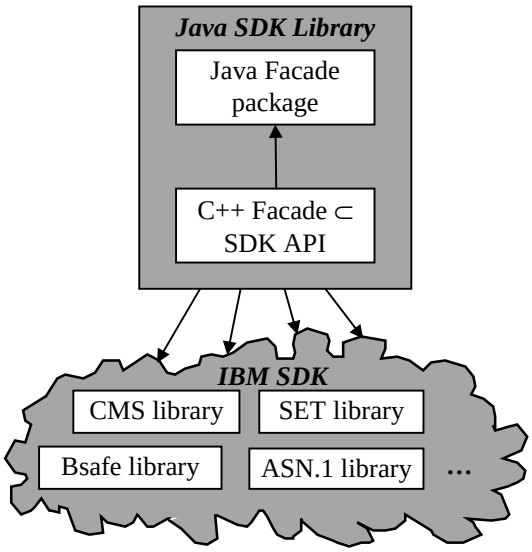

Figure 71: The native SDK within the adapter

4.4.4.4 Future work

4.4.4.4.1 Compatibility of SET Outside *SEMPER*

In general, one can think of two interesting situations:

- Analyse interaction between a SEMPER/SET client with non-SEMPER SET merchants.
- Analyse interaction between a SEMPER/SET merchant accepting non-SEMPER SET clients.

Internally the adapter already supports communication without relying on Java serialisation mechanisms. However, outside SEMPER a SET payment usually is kicked off by merchant servers [IBMSET]. The necessary initiation mechanism based on additionally MIME messages would yet have to be integrated.

4.4.4.4.2 Design Evolution

- Following functional extension are currently lacking:

  ⇒ SET reversal or batch processing

  ⇒ Certificate browsing, automatic renewal, etc.

- Refinement by decomposition of the adapter into cardholder and merchant specific parts. This specialisation would improve simplicity and performance. By this mechanism one could try to optimise the dynamic libraries in size. However, this seems to be hard due to the dependency on the SDK libraries.

4.5 The Certificate Block

(T. Pedersen / CRM)

4.5.1 Domain Description

The certificate block provides services needed for using and handling certificates. A certificate can be either a key certificate or an attribute certificate. A key certificate certifies that a public key belongs to the person described by the certificate (the actual description depends on the attributes of the certificate). An attribute certificate denotes that a given person has certain rights or obligations (described by the attributes in the certificate), e.g. a SECA certificate.

The certificate block supports operations on the user‘s side needed to deal with the following third parties, which are usually required for establishing a public key infrastructure:

Registration Authority (RA)

The RA is responsible for verifying the correctness of the user’s identity and the attributes a user wants to certify. The importance of the RA varies with the level of assurance needed. In the strongest case, the role of the RA will be filled in by a notary public and the applicant has to show up in person and manually sign a contract.

Certification Authority (CA)

The CA is responsible for issuing and maintaining certificates belonging to users. In the certification procedure a user will at some point receive a valid certificate based on a registration at the RA. In addition to issuing certificates, the CA is responsible for revoking certificates, and it may run an on-line service providing the status of given certificates.

Directory Authority (DA)

The DA is responsible for making information about certificates available. In particular it is possible to retrieve certificates of other uses from the directory.

The certificate block consists of a manager, which manages the certificates and a number of modules. Each module provides an implementation of certificates as well as the key management protocols required. Thus a module must implement:

- the certificate
- the key management protocols (registration, certificate update, queries to a directory, revocation, certificate renewal, etc.)

As part of SEMPER a module is implemented based on the crypto and statement blocks (obviously other certificate modules do not have to use these blocks). There is

however, a more intrinsic dependency between the certificate and the crypto block, as the crypto block must be able to use the keys that are certified in the certificate block. In particular the certificate and crypto block must understand common key formats. In SEMPER this is reflected by a special registration application, which uses the crypto block to generate a key. The key is certified using the key management protocols within a certificate module (through the manager). As mentioned in Section 4.9 the use of standardised formats and encoding for keys in certificate and crypto modules will reduce this dependency. Such translation must be provided by the modules (be it the crypto or certificate module).

This close dependence between the crypto and the certificate blocks is shown in Figure 72.

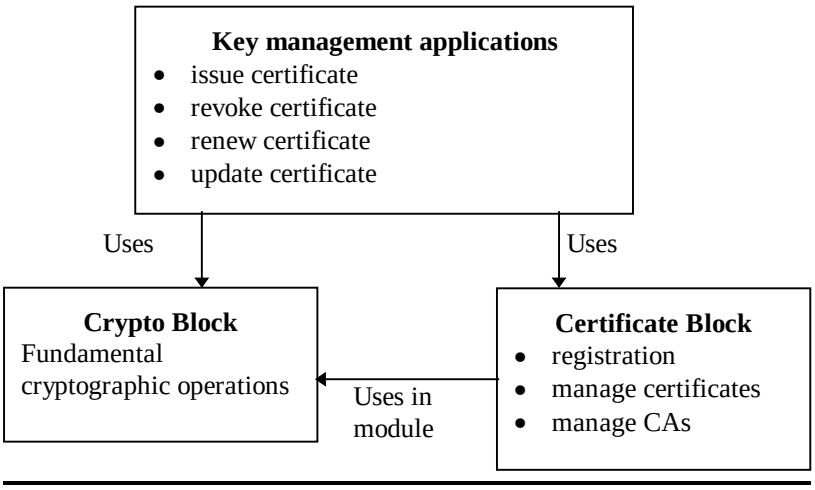

Figure 72: Dependence between certificate and crypto block

4.5.2 Requirements

The purpose of the certificate block is to provide access to certificates and operations on them within SEMPER. User access to key management protocols is supplied through special SEMPER applications dealing with the particular third parties involved. The actual implementation of certificates as well as these protocols is done in a module. Obviously, this puts some security requirements on the module, but these are outside the scope of SEMPER. The manager must fulfill the following requirements:

- store and provide access to the user's own certificates
- store and provide access to certificates received from other users (including CAs)
- provide access to key management protocols used by the special applications mentioned above
- provide methods for verifying received certificates (including on-line verification, if supported by the module)
- manage various modules and CAs
- provide a user interface allowing the user to specify policies for the use of certificates and the trust in CAs.

These requirements can be described by the following use cases:

Use Case 1: Module Management: Adding and removing modules.

Use Case 2: Module Information: Getting information about supported modules.

Use Case 3: Manage certificates: This comprises

- Storing certificates
- Removing certificates
- Assigning policy to certificates
- Viewing certificates

Use Case 4: Manage trust domains:

- Adding/removing CAs
- Add policy for use of CAs

Use Case 5: Select CA and certificate: this can either be done by negotiating with the peer or non-interactively. Here the level of trust in the CA as well as the policy of the certificate should be taken into consideration.

Use Case 6: Verify certificate: verify certificate (possibly on-line at directory or CA)

Use Case 7: Provide access to public key infrastructure:

- renew certificate
- update certificate (e.g. extend validity period)
- revoke certificate

In the following these use cases are described in more detail.

4.5.2.1 Module Management

The state of a module is either *available* or *active*:

- Available: the certificate block can use the module, meaning that all the operations are supported for this module (the module is registered with the Certificate Block Manager).
- Active: the module which at a given point in time is used by the manager.

Four actions are involved to manage the state of a module:

a) insertion (registering the module to be available for the manager)

b) deletion (making the module unavailable for the manager)

c) activation (making an available module the active one)

d) deactivation

The characteristic information of this use case is

- *Goal*: Facilitate the use of modules in the certificate block.
- *Conditions*: Table 3 lists the conditions for this use case.

The main success scenario consists of two steps:

1. The user makes a request to the manager
2. The manager performs the appropriate action, which would be one of the following:
   a) Verifies that that the module is available and inserts it in the list of available ones.
   b) Removes the module from the list of available modules
   c) Makes the requested module the active one (de-activating the currently active one)
   d) De-activates the (currently active) module.

	Pre-conditions	**Success end condition**	**Failure end conditions**
Insertion	software/hardware for the module is installed	new module available	new module is not available
Deletion	module available but inactive	module no longer available	module may still be available
Activation	module available and all registered modules inactive	active (i.e., ready for use)	Inactive
Deactivation	module active	inactive	module may still be active

Table 3: Conditions for the Instrument Management use case

4.5.2.2 Module Information

The user or application (in the following denoted the caller) will need to get information about all available modules and the active one, when activating a module.

The characteristic information of this use case is

- *Goal*: Show information about available and active modules.
- *Conditions:*
  - *Pre-condition*: none.
  - *Success end condition*: requested information returned to caller.
  - *Failure end condition*: requested information not returned.

The main success scenario consists of two steps:

1. The caller makes a request to the crypto block.
2. The manager returns the requested information. This information could include the name of the module, information about the implementation and the (trusted) CAs using the module.

4.5.2.3 Manage Certificates

This comprises management of the user's own as well as all received certificates.

- Storing certificates for later use.
- Removing certificates which the user no longer trusts or wants to use.
- Assigning policy to certificates: it should be possible to say that a given certificate can only be used in a given situation.
- Viewing the contents of certificates.

The characteristic information of this use case is

- *Goal*: Allowing the application to store and retrieve certificates and the user to view and assign policies to certificates.
- *Conditions*: Table 4 lists the conditions for this use case.

The main success scenario consists of two steps:

1. The user makes a request to the manager.
2. The manager performs the appropriate action.

	Pre-conditions	**Success end condition**	**Failure end conditions**
Store	certificate has been verified	certificate stored in archive	certificate not stored
Remove	certificate stored in archive	certificate deleted from archive	certificate still in archive
View	certificate available	details of certificate presented to user	details of certificate not presented to user
Assign policy	certificate stored in archive	the policy describing the use of the certificate is changed	policy is not changed

Table 4: Conditions for the Certificate Management use case

4.5.2.4 Manage Trust Domains

A trust domain is basically described by a CA and comprises all entities trusting certificates from that CA. Thus the user can add/remove himself to trust domains by adding CAs, and the user should further have the possibility to assign a policy to a CA specifying that a particular CA may only be trusted in certain situations.

The characteristic information of this use case is

- *Goal*: Allowing the user to manage trust domains.
- *Conditions*: Table 5 lists the conditions for this use case.

The main success scenario consists of two steps:

1. The user makes a request to the manager.
2. The manager performs the appropriate action, which would be one of the following:
    - Add the CA to the set of trusted CAs.
    - Removes the CA from the set of trusted CAs.
    - Assigning policy to CA: it should be possible to say that a given CA will only be used (i.e., trusted) in a given situation.
    - Viewing details of CA.

	Pre-conditions	**Success end condition**	**Failure end conditions**
Add CA	CA's module supported from the manager	CA added to list of available CAs	CA not added (e.g. the module is not supported)
Remove CA	CA added	CA will no longer be used	status of CA not changed
Assign policy	CA added	the policy describing the use of the CA is changed	policy is not changed
View CA	CA added	details of CA shown to user	details not shown

Table 5: Conditions for the Trust Domain Management use case

4.5.2.5 Select CA and Certificate

It must be possible to select a certificate for a particular application. The selection can either be done locally or by negotiating with the peer. In the former case there are already stored certificates of the peer (from a previous negotiation or by importing the certificate). The negotiation depends on the actual situation and the requirements defined for the certificates (e.g. preferred pseudonyms, SECA certificate). Moreover, the selected certificates should have been issued by CAs that both parties trust.

The characteristic information of this use case is

- *Goal*: Establish context describing the certificates to be used..
- *Conditions:* See Table 6.

	Pre-conditions	**Success end condition**	**Failure end conditions**
Local selection	User has certificates matching the given situation	The user has certificates to be used in the given situation	No certificates selected
Negotiation with peer	Both parties have certificates matching the given situation and issued by CAs trusted by the other party	User retrieves his own certificates and receives certificates from the peer, that can be used in the given situation. Both parties accept to use these certificates.	Required certificates not selected

Table 6: Conditions for selection of certificates

In case of local selection, the main success scenario is very simple:

1. The preferred certificates are retrieved corresponding to the given description of the situation.

In case of negotiation with the peer the negotiation must take the following into account:

1. For the particular situation the user should identify the CAs that are trusted them to the peer together with the requirements for the certificates.
2. Receive the peer's trusted CAs and the requirements for the certificates.
3. Select own certificates matching the description and satisfying all the requirements.
4. Send the selected certificates to the peer.

5. Receive certificates from the peer.

It is not a requirement that the steps are executed in this order, but the selection of certificates should take into account the user's own policy as well as the wishes from the peer.

4.5.2.6 Verify Certificate

When using a certificate it is essential to be able to verify it against the public key of a CAs that the user trusts. The verification may be done locally or on-line with the CA or a directory.

The characteristic information of this use case is

- *Goal*: Verify the validity of a certificate.
- *Conditions:*
  - *Pre-condition*: The user has a module available supporting the particular certificate. The user has (access to) the necessary public keys of the CAs in the certificate chain.
  - *Success end condition*: A correct answer about the validity given the user's trust domain.
  - *Failure end condition*: The certificate could not be validated

The main success scenario consists of one step:

1. Given a certificate the module corresponding to the certificate is identified.
2. The manager finds out if on-line verification is required
3. The manager identifies CA against which the certificate should be verified
4. The module verifies the certificate with the given parameters.

4.5.2.7 Provide Access to Public-Key Infrastructure

It must be possible through SEMPER to get access to the key management services supported by a given module. These services can for example be used to revoke certificates, renew certificate (to get a certificate on a new public key) and update certificates (to extend the validity period).

The characteristic information of this use case is

- *Goal*: Execute key management services.
- *Conditions:*

- *Pre-condition*: The user has a module available supported by the CA and certificate in question.
- *Success end condition*: Certificates changed according to service.
- *Failure end condition*: Certificates not changed

The main success scenario consists of one step:

1. Certificate manager identifies module and trusted third parties to contact
2. Certificate manager requests the given services from the module.

4.5.3 Design Overview

The central class in the Certificate Block is CerificateMan, which coordinates the modules, the trusted CAs, the user's own certificates and certificates received from the peer. In the following the datastructures supporting this are described in more detail.

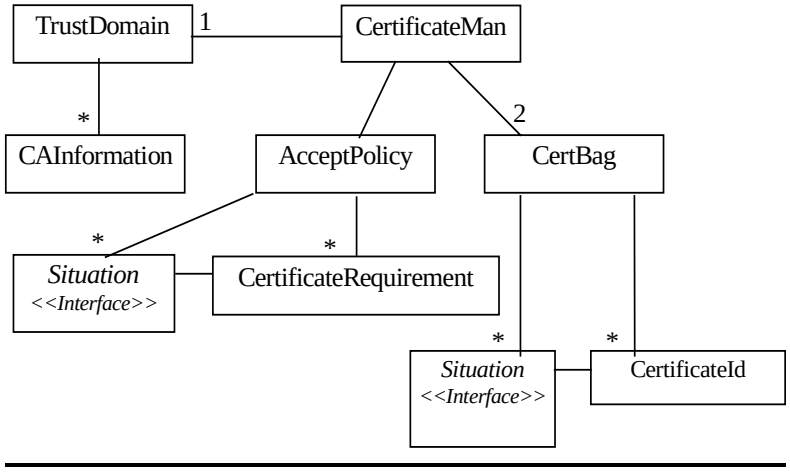

Figure 73: Information in CertificateMan about CAs and certificates

The certificate manager allows the user to assign policies to CAs and certificates. A policy describes when the CA/certificate may be used. The interface *Situation* describes the operations required by objects describing policies. In the current implementation two classes `GeneralPolicyId` and `PolicyId` implement this interface. See Figure 74.

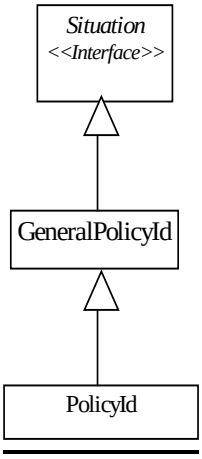

Figure 74: The Situation interface

Modules

In order to use a module the classes and libraries implementing the module must be installed. When someone tries to make a module available the Certificate Block first checks if the module is installed. If it is installed it becomes available by being registered and initialised at `CertificateMan`. Information about the module and a handle to each registered module are maintained in a `ModuleInformation` object. From this object it is possible to get, given the name of the module and the name of the CA that uses it, the handle to the actual, initialised module. Thus if several CAs use the same module, this module may have to be initialised several times.

Trusted CAs

The manager keeps a list of all trusted CAs in a `TrustDomain` object. A CA is simply classified as trusted if it has been added and can be found registered in the `TrustDomain` object. The `TrustDomain` object contains for each trusted CA (identified by the name of the CA) a vector of `CAInformation` objects. This vector keeps for each CA the information needed to use a particular module for this CA. Normally, one would expect that a CA only supports one module, but it may be possible that a CA supports several modules under the same name. For instance there could be two modules supported from different internet addresses. Thus it make sense to keep a vector of `CAInformation` objects for each trusted CA. The information in each `CAInformation` object includes:

- name of module (e.g., "dummy")
- the public key of the CA for this module
- the internet address of the CA.

Another class, `AcceptPolicy` maintains information about the policy under which a given CA may be used. In particular this affects the policy under which certificates issued by the CA can be used. The `AcceptPolicy` object maintains for each CA a list of `Situation` objects, each describing the applications in which the user will accept certificates from the given CA. For a given situation (defined by a

`Situation` object) `AcceptPolicy` contains a list of `CertificateRequirement` objects each describing requirements to

- Registration level : Requirements on the level of certainty with which the user is identified during the certification process
- Identification level: Requirements on the amount of information about the user in the certificates

These requirements on the name are described in a `NameRequirement` object. Possible levels will depend on the module and the registration procedures supported by the CA.

The assignment of policies to CAs is done through a `CAManagement` application, which can be used from the TINGUIN.

Information about the Certificates

Both the user's own certificates and the received ones are stored in his archive. These certificates are kept in two CertBag objects (one corresponding to the user's own certificates and the other to certificates received from other entities). Through the Certificate Browser Application, which can be accessed from the TINGUIN, the user can view the certificates and assign policies to them. It is anticipated that a user will have various certificates to be used in various situations. In the certificate block a situation is described by `GeneralPolicyId` and `PolicyId` objects that implement the `Situation` interface.

PolicyId may specify the name of an entity and a name of the business application in which the certificate can be used. `GeneralPolicyId` specifies the name of the business application in which a received certificate can be used (the name of the peer entity is in that case the pseudonym in the certificate). The name of the BA must be defined by the provider of the BA. Extensions to the `Situation` interface are conceivable such as saying that a certificate should only be used in certain legal settings (e.g., if certain contracts have been signed).

The class `CertBag` is basically a list of handles (represented by `CertificateId` objects) to all certificates. (The certificates themselves are in the archive.) A list of `Situation` objects is associated with each certificate, specifying that the certificate can only be used in the situations described by these objects. A special `Situation` object can be used to specify that there are no restrictions on the use of a certificate (i.e., such a certificate is a candidate for selection by default, but the manager will during the selection first look for certificates, which are intended to be used in the given situation).

Certificate Selection

Selected certificates are maintained in `CertificateContext` objects. Before certificate selection such an object can be used to indicate which kinds of certificates are needed. After certificate selection the object contains a signature and/or an encryption certificate according to this requirement. Thus in an interactive session two such objects are required - one containing certificates of the user and the other

containing the certificates of the peer. The two `CertificateContext` objects can be established either during the negotiation of certificates with the peer, or during the retrieval of certificates non-interactively (supposing that the user has already stored certificates received from a previous negotiation).

One part starts the negotiation using the requestChoice method and the peer must answer this request using chooseCertificates. Both of these methods take as arguments two `CertificateContext` objects, myContext and yourContext and a `SituationDescription` object, that describes the actual situation in which the certificates are required plus other possible requirements. Such an object contains information for the following (it is not necessary to set all requirements):

- preferred pseudonym of the user,
- preferred pseudonym of the peer
- name of the business application in which the certificates are to be used
- requirements on SECA (is SECA required, and if yes, which values of the SECA attributes are required)

Based on a `SituationDescription` object it is possible to construct `GeneralPolicyId` and `PolicyId` objects needed for the selection of certificates and CAs. The negotiation then works as follows:

Requester	Responder
Find policy for acceptable certificates from peer given `PolicyId` corresponding to the `SituationDescription` (peer name, BA). Send this policy to the peer	
	Find policy for my own acceptable certificates from the `PolicyId` corresponding to the `SituationDescription` (my name, BA). Ensure that the received policy does not contradict my own. Select my own certificates based on the resulting policy and the requirements from the requester (throw exception if not possible). Send the selected certificates to Requester together with my requirements on his name (peer name, BA).
Find policy for my own acceptable certificates from the `PolicyId`	

corresponding to the `SituationDescription` (my name, BA). Ensure that the received policy does not contradict my own. Select my own certificates based on the resulting policy and the requirements from the responder (throw exception if not possible). The CA that has issued the certificates should satisfy the policy received from the responder. Send the selected certificate to the responder.	

In case of local selection, the certificates are chosen as in the interactive case except that the requirements from the peer cannot be taken into account (and certificates of the peer are selected among those received in previous transactions).

Getting the Public Key of a Root CA

It is not necessary to have the public key of the CA when registering the CA. RCModuleInterface defines a service, getTrustedPK, which accesses the CA and returns the public key of the server. Depending on the status of the CA at the client side one of the following actions then takes place in the manager:

- If no key was previously installed for this CA, the manager shows a fingerprint of the key in the TINGUIN and asks the user if the key should be installed.
- If a key is installed and the installed key is the same as the new key nothing is done.
- If a key different from the new one is already installed the user must specify which one he wants to use.

4.5.3.1 Interfaces

The certificate block has defined two types of interfaces. One type, e.g. `Situation`, is defined to make the block flexible and make it easy to change some classes, while the other type corresponds to the interface that a module must satisfy in order to be used. The latter comprises four interfaces:

- `Template` defines the interface for registrations forms which the user must fill out during registration
- `Registration` defines an interface for registered information about a user. In previous versions of the certificate block this interface was needed on the client side, but this is not necessary in the latest version. Thus this interface is deprecated.

- `Certificate` implementing a certificate; and
- `RCModuleInterface` implementing services needed for registration and management of a certificate.

Certificate

It is up to the provider or the registration module to define the attributes of the certificates. The interface defines that it should be possible to ask for the following values:

1. Distinguished name of CA
2. Distinguished name of subject (user)
3. Serial number for certificate (e.g. using a local numbering within each CA)
4. The public key
5. Scope (or application) of the public key: Describes, what the key can be used for, e.g.:
   - signatures
   - identification
   - key exchange
   - encryption
6. Type of Certificate, which can be used to further describe the use of the certificate.
7. Time stamps (start and end of validity period, time of revocation if applicable)
8. Status of the certificate: Describes the status of the certificate. Examples could be the registration level or whether the certificate is revoked.
9. Reason for revocation (if applicable).
10. Signature of the CA
11. Signature and hash algorithm used
12. Extensions of the certificate.

If a certificate in some module does not contain the corresponding values, the implementation of this interface must supply a suitable value.

RCModuleInterface

This interface defines methods for

- filling out templates (possibly using the TINGUIN),
- make registrations,
- store/retrieve registrations

- issue certificates
- verify certificates
- change certificates (e.g. revoke certificate).

4.5.4 Related Work

In recent years much work has been invested in the development of public key infrastructures. Relevant work for the Certificate Block includes:

- X509 for further information [X509v3].
- PKCS for further information [PKCS10].
- PKIX for further information [PKIX].
- OpenGroup's CDSA for further information [CDSA97].
- Secude.

X509 defines a standard for public key certificates, which a module in the certificate block could support (in fact the SECUDE module in the SEMPER prototype uses X.509 certificates). This standard does not, however, deal with key management protocols. Such protocols are being standardised in the PKIX framework. An implementation of this standard could be used in a module in SEMPER supporting issuing and revocation of certificates. PKIX does, however, specify much more scenarios than are currently envisaged for handling certificates in SEMPER.

PKCS#10 defines a method for requesting certificates, currently used by many Internet browsers. An implementation of this could be used when getting certificates in SEMPER, but revocation of certificates is outside the scope of PKCS#10.

4.5.5 Self-Assessment

In the first implementation of the certificate block RA, CA and DA are unified. Furthermore, we only consider the case where a user wants to get a certificate on a key (thus attribute and hybrid certificates are not supported in the first version).

As a result of the development so far the following changes should be considered in a future design of SEMPER:

- The certificate block should only handle key certificates and their extensions
- Another block should be added dealing with attribute certificates. See Section 5.3

These two blocks may have some commonalities that can be singled out to another service block, e.g. a block handling third parties that the user wish to register as trusted. This block could manage

- CAs

- Third parties issuing attribute certificates
- Third parties making time stamps

The certificate block should only manage the users (certified keys). Furthermore, just as getting purses is outside the payment block it is worth considering public key management outside SEMPER. The certificate block should just be able to import certificates obtained from accepted CAs.

It would make sense to split into two modules the basic certificate mechanism from the key management protocols giving , a core module and different key management modules (e.g. a X509v3 as the core module and PKIX as a module for key management protocols). In the current design this split is reflected by different interfaces in the certificate block.

4.5.6 Implementation Notes

The implementation has focused on handling policies and providing visualisation of the policies through the certificate browser. This means that the current prototype does not include support for key management protocols with third parties, except what is needed for registration. Revocation of certificates was also considered in the design but it is not provided in the prototype. Furthermore, the implementation does not support certificates on public keys of Cas (be it self-certificates or certificates issued by other CAs in a hierarchy of CAs).

4.6 Certificate Modules

4.6.1 Generic

(T. Pedersen / CRM)

As part of the certificate block a generic certificate module is supplied. The design goal of this module is to provide minimal functionality allowing the use of certificates in SEMPER demonstrations, so that these can be carried out without relying on external certificate authorities. In addition the module must support properties that are considered essential parts of SEMPER. In particular this means that the module must support SECA certificates. Thus the goal of this block is not to provide a complete public key infrastructure as this is outside the scope of the SEMPER core.

4.6.1.1 A Minimal Module

As mentioned in Section 4.5 a module in the certificate block must implement the following four interfaces

- `Template`, which defines the interface for registrations forms which the user must fill out during registration
- `Registration` defines an interface for registered information about a user.
- `Certificate` implementing a certificate; and
- `RCModuleInterface` implementing services needed for registration and management of a certificate.

Thus the generic certificate module consists of classes implementing these four interfaces. Of these, the classes implementing a certificate and `RCModuleInterface` are the two most interesting.

A generic certificate contains the following fields:

- (distinguished) name of the CA
- (distinguished) name of the user
- serial number, which is unique for the CA
- public key
- scope (or usage) of the key (e.g., for encryption, for digital signature,…)
- type of the certificate;
- validity period
- status (e.g., validity of the certificate);

- reason for revocation in case the certificate is revoked
- signature of the CA plus information about the algorithms used for signing
- a field containing possible extensions (in SEMPER the SECA attribute is an extension, see below)

In order to sign the certificates, the fields must be encoded as an array of bytes. In the generic module this is simply done using the Serialization feature of JAVA. This is convenient short cut, but it should be avoided in general, as the serialization of an object is not unique (one is only guaranteed that the original serialised object is recovered during deserialisation).

The implementation of `RCModuleInterface` supports methods for

- filling out a template (registration form),
- sending the form to the CA
- requesting a certificate from the CA
- verifying a certificate against the public key of the CA (locally)

Using these protocols it is possible for a user to get a certificate (based on a pre-registration) and to verify received certificates.

4.6.1.2 Handling SECA in the Generic Module

As mentioned above, SECA is supported as an extension in the generic module. In principle SECA could also have been implemented as a “normal” attribute certificate but as SECA was included rather late in the project, it was more conveniently added using an extension.

The extensions to generic certificates are contained in a single class implementing an interface called CertificateExtensions of the Certificate block. This class has a single attribute describing the SECA property (implemented by the class `SECA`). This class describes SECA as follows:

- whether the user acting as a private person or as a business
- whether the user acts as seller or buyer
- whether the user is “adult” at the time when the certificate is issued
- a number which, in case the private key is compromised, limits the amount that the user can be held responsible for during the period between the compromise of a certificate and the actual revocation
- description of security level of the equipment of the user.

These fields reflect some options in the SECA used in SEMPER. The reasons for these options are outside the scope of this document. Please see related documents on SECA for a discussion of this.

4.6.2 Secude

(K. Zangeneh / GMD)

4.6.2.1 Introduction

Secude (Security Development Environment) is a security toolkit for open networks which incorporates a variety of well-known and intensely studied crypto algorithms as security basis for higher-level applications, such as digital signatures and encryption. Furthermore, Secude provides two possible solutions of a Personal Security Environment (PSE) for the secure storage of private keys: an encrypted file (software-PSE) and interfaces to a variety of smartcards (smartcard-PSE), e.g. TCOS, CardOS, PKOS and GemPlus. Access to the secure processing and secure storage module is provided by the SECURE API. Another important API is needed to achieve authentic communication. The functions for the management of public keys are accessible through the Authentication Framework API. On top of these basic modules there are higher level APIs, e.g. SME (Secure Message Envelope, TeleSec standard from Telekom Germany), PEM (Privacy Enhanced Mail) and MTT (MailTrust Extension of PEM), PKCS (Public Key Cryptographic Standard), GSS (Generic Security Services) and S/MIME (Secure Multipurpose Internet Mail Extensions).

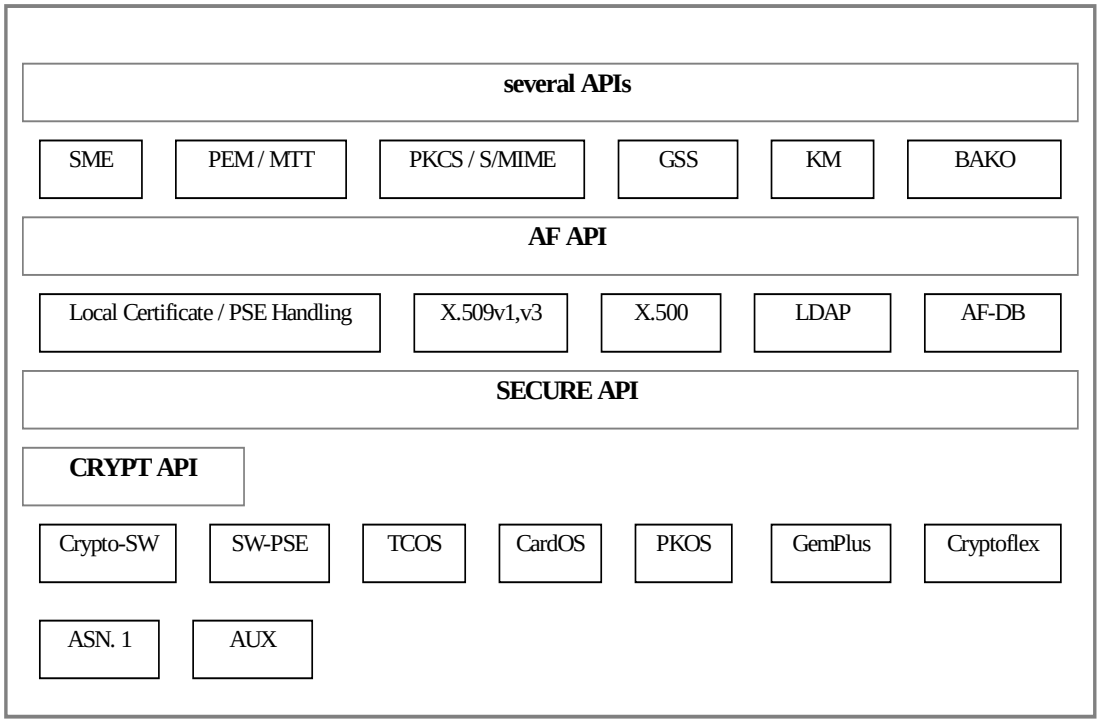

Figure 75: Secude Development Toolkit

Secude provides a full Key Management (KM) functionality. Some secure business application, like BAKO (Basic Cooperation) are developed which use the whole security framework of Secude.

Secude provides a wide range of APIs for supporting various directory services, e.g. X.500, LDAP (Light Directory Access Protocol) and AF-DB (Authentication Framework Database; a local file-based directory service).

For coding/decoding within Secude the ASN.1 notation is an overall used concept.

Finally, these functions are utilized in a number of ready-to-use utilities and in security plug-ins for existing software products, e.g. SAP/R3 or Microsoft-Exchange.

4.6.2.1.1 Secude Key Management

Secude provides functions for the generation and management of keys. All well-known symmetric, asymmetric key generation algorithms, hash functions, key exchange algorithms are implemented in the Secude key management module. Further algorithms see section 4.6.2.

4.6.2.1.2 Secude CA Management

The use of public key cryptography for authentic and confidential communications in open networks assumes the development and operation of a certification infrastructure. All participants possess a private and a public key; the public key is digitally signed (certified) by a certification authority. This procedure is comparable to issuing an ID card. Secude certification authority (CA) management offers the know-how for certification authorities, to develop and operate such a certification infrastructure.

Secude CA management fulfills all requirements that are to be fulfilled by a certification authority such as issuing certificates, maintaining revocation lists and administering users. Secude CA management enables various key generation and certification models. The users can generate their own keys, which then need only be certified by the certification authority. The certification authority can, however, also generate and certify keys for the users. If required, the CA management can be linked to a WWW server for online certification. All user data is filed in a database which is addressed via a manufacturer independent interface.

4.6.2.1.3 Secude Secure Archive

All security relevant information of a user is stored in a so-called Personal Security Environment (PSE). For example, a PSE typically contains the users private and public key (the latter contained in an X.509 certificate), the public root key which the user trusts, and the forward certification path to the users root key. In addition, the PSE allows to securely store other objects, e.g. other public keys after their validation (allowing henceforth to trust them like the root key without verifying them again), and certificate revocation lists (CRLs). Secude provides two different PSE realizations:

- a DES-encrypted file or directory (software PSE), protected by PIN
- a Smartcard environment (smartcard PSE), protected by PIN

4.6.2.1.4 Secude SmartCard Technology

Where a greater degree of security is required, Secude offers solutions to achieve a higher standard of security. The technology required for this is provided by SmartCards. SmartCards have their own processor, their own memory and an operating system which allows limited access to information on the SmartCard - e.g. read only, but not change - or prevent access from the outside entirely. With the version realized by Secude the whole PSE is stored on the SmartCard. Thus two factors are indispensable to obtain the security-sensitive information - the card and the corresponding PIN (two-factor authentication).

When a SmartCard is used for the security sensitive information such as the private key or certificate of the user, the SmartCard processor deals with the signing and decrypting procedures where the private key of the user is required. The private key is stored on the SmartCard and can only be accessed by its operating system. This ensures that the private key never leaves the SmartCard and can therefore not be compromised.

4.6.2.2 Security Features in Secude

In the following, the functionality of the different Secude APIs is addressed with the focus on the latest developments and new features. More information may be found in the Secude 5.1 Hyperlink documentation [RiShAd98]

4.6.2.2.1 CRYPT API

The CRYPT-API provides multi-precision integer arithmetic, modulo arithmetic, random number generation and implementations of symmetric and asymmetric crypto algorithms.

The available symmetric algorithms are DES [DES88,DES80], Triple DES and IDEA [Lai92]. The public key algorithms supported in the CRYPT API are the Diffie Hellman Key Agreement [DifHel76], the NIST DSS [DSS94] and the RSA algorithm [RiShAd78].

Furthermore, there is a need for cryptographically strong hash functions for signature generation. Secude provides the hash-functions MD2, MD4 [Rivest92], MD5 [Rivest92a], SHA-0, SHA-1 [SHS] and RIPEMD-160 [DoBoPr96]. MD2 and MD4 are unsuitable for signature generation, MD5 is suspected to be broken soon (c.f. [Dobber96], [Dobber97]) and SHA-0 bears some weaknesses in the expand-function. Therefore, we recommend SHA-1 and RIPE-MD160 as the most secure hash functions for signature generation. The other hash functions are still supported in Secude, to keep compatibility with standards and old signatures.

4.6.2.2.2 SECURE - API

As mentioned in the introduction, Secude provides a technology-independent SECURE-API, which connects the CRYPT API and the PSE-handling with the higher-level APIs. Secude supports two PSE-types: an encrypted file or directory (SW-PSE) and interfaces to different smartcards (smartcard-PSEs). Both PSE-types

are PIN-protected. The PSE- and smartcard-type(s) used may be selected during the configuration process. Only the desired SC-interface is linked dynamically.

Secude 5.1 supports the STARCOS and the TCOS smartcard systems. Interfaces to the GEMPLUS smartcard GPK2000, the G&D PKOS smartcard and the German Telekom Security Module TSM95 are under development.

4.6.2.2.3 Authentication Framework API

The AF (Authentication Framework) module adds X.509 certification functionality to Secude. Former Secude versions supported the X.509 version 1 certificates, but since Secude 5.1 all releases contain the X.509 version 3 certificates. The version 3 format adds optional extension fields to X.509 certificates. Some extension fields are specified in standards, others may be defined and registered by any organization or community.

Both local (i.e. PSE-located) certificates and Directory-located certificates can be addressed. Obtaining public security information, like public keys, certificates, cross-certificates and certificate revocation lists used to be done using the X.500 Directory Access Protocol (DAP). Secude 5.1 now uses the Lightweight Directory Access Protocol (LDAP) instead of DAP to retrieve security information from Directory servers.

4.6.2.2.4 X.509 version 3 certificates

The certificate fields of a version 3 certificate are basically the same as of a version 2 certificate, except for the extension field. The extension field allows addition of new fields to the structure without modification of the ASN.1 (Abstract Syntax Notation One) definition. An extension field consists of an extension identifier, a critical flag and a canonical encoding of a data value of an ASN.1 type associated with the identified extension.

The extensions defined for X.509 version 3 certificates provide methods for associating additional attributes with users or public keys, for managing the certification hierarchy, and for managing certification revocation list distribution. The X.509 version 3 certificate format also allows communities to define private extensions in order to carry information unique to those communities. Each extension in a certificate may be designated as *critical* or *non-critical*. Use of each extension is optional to the certification authority issuing a certificate. A certificate using system (an application validating a certificate) must reject the certificate if it encounters a critical extension it does not recognize. A non-critical extension may be ignored if it is not recognized. Extension definitions indicate whether they are always critical, always non-critical, or whether criticality can be decided by the issuer of the certificate. For all extensions, there ought to be no more than one instance of each extension type in any certificate.

The following eight extensions are supported in the current version (Version 5.1) of Secude:

- Basic Constraints

- Subject Alternative Name
- Issuer Alternative Name
- Key Usage
- Certificate Policies
- CRL (certification revocation list) Distribution Points
- Authority Key Identifier
- Subject Key Identifier

4.6.2.2.5 LDAP integration in Secude

LDAP is supported in Secudes' various Unix ports using the LDAP reference implementation library of the University of Michigan [LDAPa]. In the Secude for Windows NT/95 version the Dynamic Link Library (DLL) of the same package is used [LDAP].

Secude allows the user to add or replace the certificate in the Directory server with a certificate stored in the user PSE. The user may also delete the certificate from the user Directory entry. Certification authority PSE administrators may add, replace or delete the CA's certificate. In addition, the CA's certificate revocation list can be added to the Directory or an old certification revocation list may be replaced by a new one. Deletion of certification revocation lists is currently not supported as it would violate policies. In addition to the user-oriented functions, Secude now uses LDAP to dynamically access the Directory service in order to retrieve missing certificates and certification revocation lists when verifying digital signatures.

4.6.2.2.6 Privacy Enhanced Mail API

The PEM (Privacy Enhanced Mail) [PEM93] module is a full implementation of RFC 1421 (Message Encryption and Authentication Procedures), RFC 1422 (Certificate-Based Key Management), RFC 1423 (Algorithms, Modes, and Identifiers), and RFC 1424 (Key Certification and Related Services) [Bauspi96] plus SECUDE-specific add-ons according to the German MailTrusT PEM Specification [Bauspi96] which includes correct processing of raw binary data.

Furthermore, the PEM routines and commands have been extended to provide stream-based reading, processing, and writing of files in order to be able to handle large files.

4.6.2.2.7 Generic Security Services API

The *Generic Security Services* (GSS) API is a set of functions and data structures to incorporate security into a program independent of the underlying security and communication protocols [Linn93]. It enables application-level security on an end-to-end basis by letting the application call services from the underlying security system via the provided GSS-API functions. The level of abstraction that the GSS-API provides is, that all security systems behave in the same manner towards the application, independent of the systems' architectures. However, the GSS-API does

not generally provide interoperability between different security systems. The focus is on using existing mature third-party security systems that offer strong user authentication and administrative tools.

Secude provides two different GSS-API mechanisms: the Simple Public-Key GSS-API Mechanism (SPKM) [Adams96] and a second mechanism optimized for Secude. The first one is not fully implemented yet, but is under development.

SPKM is a flexible mechanism for a public key based GSS-API, that is, the algorithms for authentication and encryption are negotiated between the GSS-API callers. Secudes' tailor-made implementation is more restricted. Secude supports uni-lateral and mutual authentication. Only DES and IDEA can be used for encryption. Pure Secude-to-Secude communication leads to better performance.

4.6.2.2.8 PKCS API

While the family of PKCS-standards [PKCS11] comprises standards for RSA encryption, DH Key Agreement and other cryptographic primitives that are already available in earlier versions of Secude, Secude 5.1 now includes several programs and functions dealing with PKCS#7 and PKCS#10 formats.

The PKCS #7 standard describes a general syntax for data that may have cryptography applied to it, such as digital signatures and digital envelopes. The PKCS #10 standard describes a syntax for certification requests.

4.6.2.3 Secude Module in SEMPER

The certification application block is an example of a special application using SEMPER APIs for registering users and also for issuing certificates for them. The certification application block takes advantage of the services of the crypto, archive and certificate block.

One implementation of the underlying module in the certificate block is a generic module implemented for the SEMPER software especially; this module is implemented in the Java programming language.

Secude is another module which is used in the certificate block/certification application for issuing and maintaining the users' key certificates. Secude implements necessary interfaces from the certificate block which are essential for deployment in the SEMPER architecture as certification management module. The usage of Secude as another module in SEMPER architecture is an evidence for the open and generic architecture of SEMPER.

As mentioned above, Secude consists of three important management modules, namely key management, certificate management and archive management. Therefore, Secude as a whole package replaces the functionality of a crypto, certificate and archive block in SEMPER. However, the SEMPER possesses its own key management module, i.e. cryptographic manager, and its own storage system, i.e. archive manager. For generic purposes, the certificate management module of Secude is used within SEMPER only. For this reason, it was not possible to deploy Secude as it is. A cooperation and interoperability between the Secude certificate management system and the SEMPER archive/cryptographic/certificate manager had to take place.

Integration of native Secude code in the SEMPER architecture caused a few difficulties which had to be solved.

- *Splitting the Secude package*: As mentioned before Secude covered multiple blocks of SEMPER. This means that we had to split the Secude package and use only the parts that were necessary. A new specification was made regarding the use of Secude in SEMPER. Design and implementation of a C library was necessary as well. Only the certificate management module of Secude was deployed in SEMPER. The integration of the key management unit of Secude into the corresponding SEMPER ArchiveMan (see Section 4.13) and CryptoMan (see Section 4.9) was left open due to lack of resources.
- *Working with JNI (Java Native Interface)*: By deployment of JNI, simple C data structures had to be mapped onto more complex Java objects and vice versa.
- *Key conversion*: The key pairs were generated by the crypto module of CRYPTOMATHIC which has a different structure and encoding as Secude keys. The CRYPTOMATHIC keys, which are in an unknown format for the Secude module had been converted into the format operational for Secude. After the key conversion process, the converted key is put into the Secude certificate and is sent back to the user. The solution to above problem clearly would be to interfaces to the crypto block via standard encodings (see Section 4.9.5).
- *Secude certificate extensions*; Secude supports X.509 version 3 certificates. The Secude X.509 version 3 extensions, like Basic Constrains, Subject Alternative Name, Issuer Alternative Name, Key Usage, etc. had to be inserted and interpreted in SEMPER certificates. New methods had been added in Certificate interface for supporting X.509 version 3 extensions in SEMPER certificates.
- *Redundant storage*; because of the internal functionality of Secude, it was not possible to do without the PSE, the archive management of Secude. This causes redundant storage of some sensitive data as soon as Secude is deployed. Besides of the SEMPER archive, the Secude secure archive should co-exist. This redundancy could have been prevented by providing the cryptography functions and some of the key management part of Secude as a module of the Cryptographic Block of SEMPER. But as mentioned above this was left open due to lack of resources.

Secude offers the Directory services for SEMPER as well. A Directory server was up and running for SEMPER based on LDAP (Lightweight Directory Access Protocol).

In the current implementation, the Secude module is integrated in SEMPER and operates as follows:

The certification authority deploys Secude as underlying crypto module for generating its key pair and possesses the Secude archive system. A SEMPER user generates a user key pair using the CRYPTOMATHIC security toolkit. After the key generation process is completed, a request for registration is sent to the registration/certification authority. As a result of this a template is sent back to the SEMPER user. The template is filled out by the user and the public key is attached to the template. The whole template data structure is sent to the registration/certification authority which operates on the basis of Secude module. Since the user's key pair was generated by the CRYPTOMATHIC security toolkit, the Secude package on the registration/certification authority side is not able to handle with the incoming key.

The incoming key has to be converted to a "Secude-like" key which will be certified by the registration/certification authority after registering the user.

The issued certificate which contains the user's public key from the CRYPTOMATHIC is sent back to the user. A copy of the user's newly issued certificate is put on the LDAP Directory server which is running on a well-known host, currently at the GMD promises.

The incoming certificate at the user side is verified using Certificate Manager which operates on the basis of Secude module. The Secude module at the user side performs the certificate verification via the public key of the certification authority which has been received and stored in a previous step. The verified certificate is stored in the SEMPER archive at the user's side; because of internal functionality of Secude security toolkit the incoming certificate has to be stored in Secude PSE as well so that a redundancy of certificate storage is not preventable.

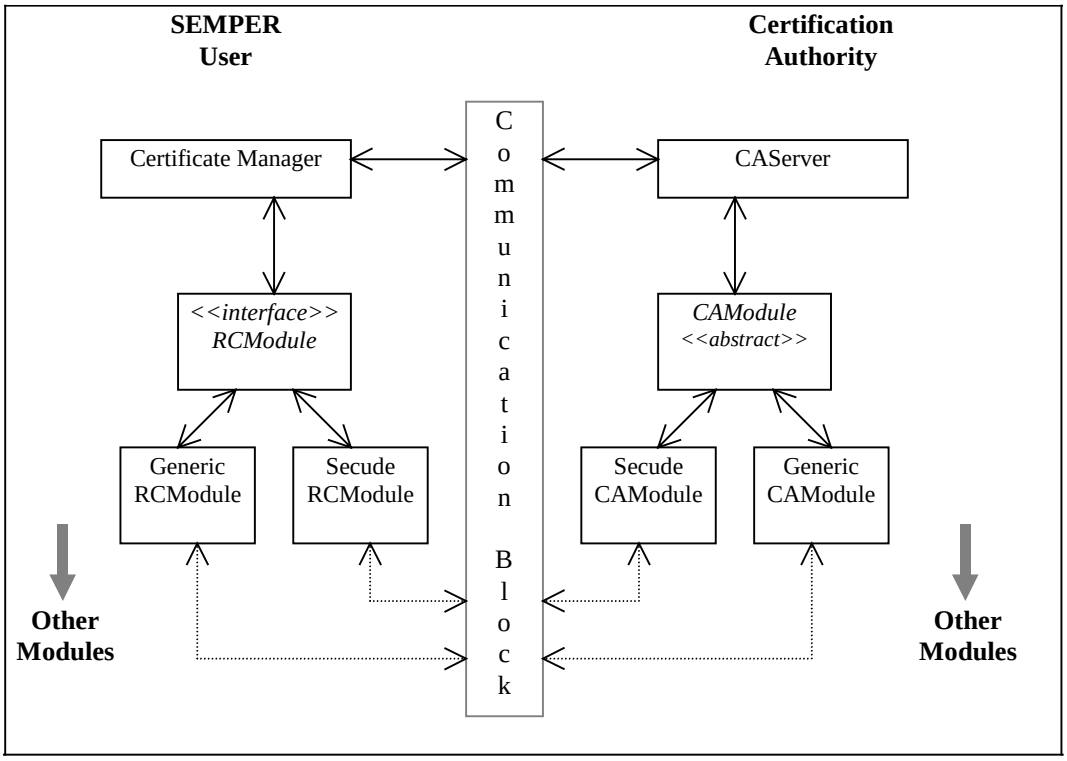

Figure 76: Structure of Client and Certification Authority units in SEMPER

4.7 Secure Communication

(J. Brauckmann / SRB)

4.7.1 Domain Description

The purpose of the secure communication block is to provide facilities to **transport** arbitrary **data** between two communication partners in a way that certain security goals are met. The communication takes place between **hosts** connected to the internet. Communication **connections** have to be explicitly **opened** before data can be transported. Two roles are involved in connection establishment: The **responder** waits for an incoming connection, while the **initiator** opens a connection to a responder.

Important security goals which should be implemented by a Secure Communication Channel are **authenticity**, **confidentiality** and **anonymity**. Note that we consider non-repudiation a security attribute not of the channel, but of the individual transfer, therefore it is implemented in the TX layer. This is necessary for implementing signatures that legally bind a person.

4.7.2 Requirements

4.7.2.1 Introduction

The requirement on the secure communication block is the following:

> *Provide a generalized interface for a variety of communication blocks which provide different security services. The secure communication block should*
>
> - *enable bidirectional communication channels between two internet hosts,*
> - *support the security attributes authenticity, confidentiality and anonymity,*
> - *allow for easy integration of modules that enforce security attributes,*
> - *have a very similar interface as the existing SEMPER communication block to allow for easier integration of the new secure communication into old communicating SEMPER blocks.*

The secure communication block is used by applications via its API.

4.7.2.2 Use Cases

4.7.2.2.1 Connection Establishment

Before any data can be transmitted, a connection must explicitly be opened. We differentiate between two roles, the initiator and the responder. The initiator opens a connection to another host. The responder waits for incoming connections from other hosts.

- Goal: Establish a connection between two peers, meeting certain security attributes.
- Conditions:
  - Preconditions: The initiator knows the responders address. The responder knows the local address to listen to.
  - Success conditions: Established connection.
  - Failure conditions: No established connection.
- Main Success Scenario: The initiating user asks the SecComManager to open a connection and passes the required attributes to it. The SecComManager chooses a module and opens a connection using it. (For discussions on more sophisticated module choice; see the self assessment below.) The responding user asks the SecComManager to start a server. The SecComManager chooses a module, which is used to actually start the server. The responding user can then open a responder connection using the server. Figure 77 shows the interaction sequence diagram of a successful secure channel establishment.
- Variations: Depending on the set of security attributes a module is chosen.

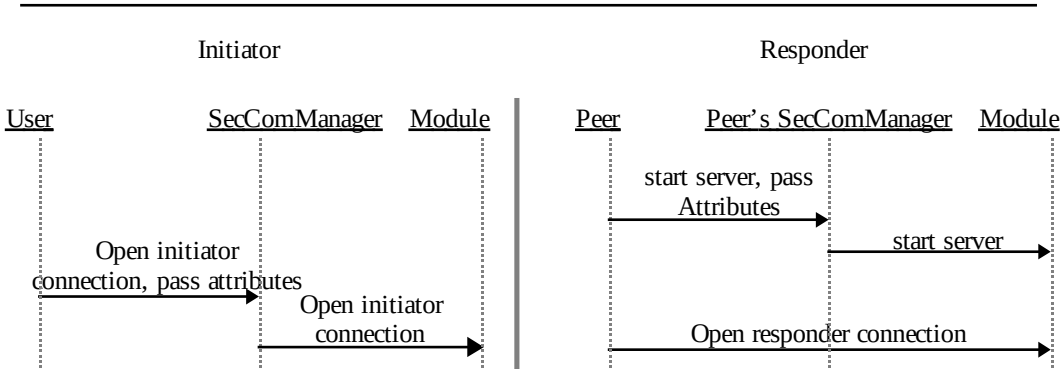

Figure 77 *Connection Establishment* main success scenario

4.7.2.2.2 Object Transfer

- Goal: Transfer an object between two peers, meeting the security attributes.
- Conditions:
  - Preconditions: Established connection; object and all its content is of a class that implements the JAVA `Serializable` interface.
  - Success conditions: Transfer of object enforces established security attributes and the receiving peer gets the object.
  - Failure conditions: A waiting peer does not receive object.
- Main Success Scenario: The scenario is straight-forward: One peer writes the object to its connection, the other one reads it from there.

4.7.3 Design Overview

4.7.3.1 Introduction

A main requirement on the secure communication block was to provide an interface very close to that provided by the SEMPER communication block (see Section 4.10). Thus many concepts are identical.

We use the notion of a SecComPoint to represent a connection endpoint that can be used to write to and read single objects from. A SecComPoint is the result of the connection establishment process. A SecComAddress contains data needed for connection establishment.

The basic SecComPoint service is refined by the SecChannel services, which allow multiplexing of several connections over the same communication link. For this, both peers specify a correlator that distinguishes this particular connection.

Connections are established by the secure communication manager SecComManager or by the secure channel manager SecChannel. On opening of a connection the SecComManager or SecChannel manager examines the set of security attributes and chooses a module that can apply all attributes. This module is then used to open the connection.

4.7.3.2 Object View

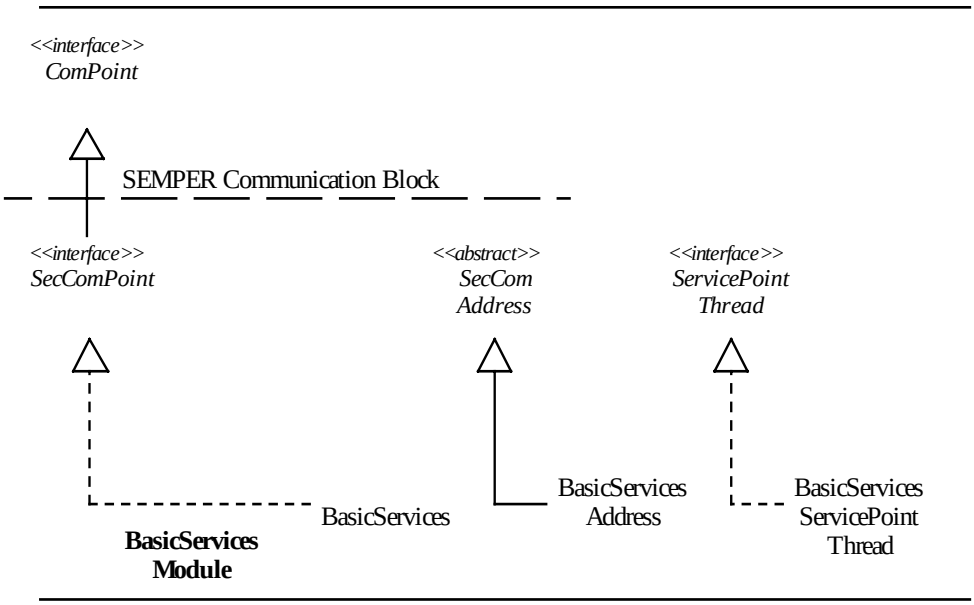

Figure 78 Primary data types

Figure 78 describes the primary data types in the secure communication block.

- Interface `SecComPoint`: A connection is represented by a `SecComPoint`. The main service of `SecComPoint` is object reading and writing. A module must provide concrete implementations.

- `SecComAddress`: An address that identifies a peer or specifies local resources to be used (e.g., ports) is bundled in a `SecComAddress` object. Modules must provide subclasses which contain all module-specific information. The main reason for having specific SecComAddresses is that anonymous communication will not work with the usual http or email addresses. Anyway, it is observed in the self-assessment that the ComBlock would also be neater with an abstract address class; in this case the top-level class could be the same.
- `ServicePointThread`: A special object (informally called server so far) that waits for incoming channels. This is only needed for opening secure responder channels.

There are two managers in the secure communication block: The `SecComManager`, which is responsible for opening normal SecComPoint connections, and the `SecChannel` manager, which provides methods for opening initiator connections with correlators and starting service point threads.

All modules have to provide classes that implement `SecComPoint` and `ServicePointThread` and extend `SecComAddress`.

The secure communication block uses the attribute classes defined in the Transfer-and-Exchange Layer, thus there is no own notion of a secure communication attribute.

4.7.3.3 Functional View

Here we describe how each use case from chapter 4.7.2.2 is met.

4.7.3.3.1 Connection Establishment

A peer can have one of two roles in connection establishment, initiator or responder. Connection establishment for the responder needs as a pre-condition that a local SecComAddress is available that is suitable for opening responder SecComPoints. In general this means that it has to contain a port number. When the channel services are to be used, the responder has to start a ServicePointThread, and afterwards he can open so-called responder channels on this ServicePointThread with a given correlator. For normal SecComPoint operation, the responder first creates a server SecComPoint with the SecComManager and then opens the responder SecComPoint on this server.

Connection establishment for the initiator needs as a pre-condition that there is a SecComAddress available that contains data about how to contact the responder. This address might either be generated by the user itself, or the user might have received it in earlier communication sessions. Another pre-condition is that the responder already started its server SecComPoint or its ServicePointThread to wait for a connection from the initiator. Then the initiator can open either an initiator SecComPoint with the SecComManager, or a responder channel with the SecChannel manager with a given correlator.

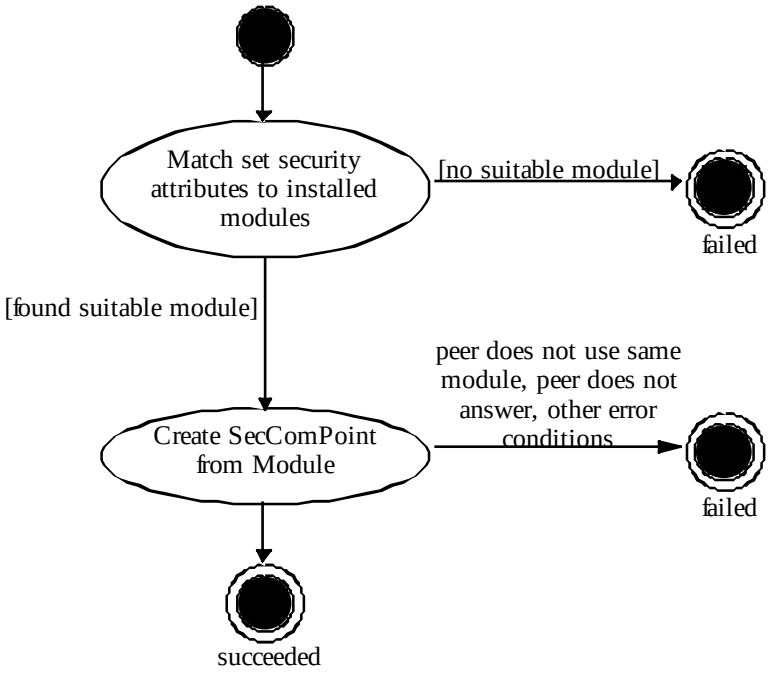

Figure 79 Activities during module selection

4.7.3.3.2 Object Transfer

When there is an established connection, there is no difference between the initiator and responder side. Both can use the connection to read and write arbitrary objects that implement the JAVA Serializable interface. (Transferring more complicated objects, or with additional per-object security attributes, is done by the Transfer-and-Exchange Layer.)

Writing an object operates in non-blocking mode because the underlying operation system buffers all written data. Attempts to read an object always block when there is no data to be read, and the user has to wait until something is received from the peer.

The secure communication services apply any security attributes to the objects transparently to the user. No interaction is needed by the user once the connection is established.

4.7.4 Self Assessment

Currently the SecComManager does no negotiation with the peer about the modules to be used. This leads to the situation that both peers must use the same module by some unspecified predetermination, otherwise communication will fail. Note, however, that we cannot simply use the manager-to-manager negotiation as in the Payment Block because here we do not have a channel yet at the beginning.

Therefore e suggest to change the procedures for opening responder connections. When a user requests the secure communication block to open a server or ServicePointThread, the SecComManager should not need to ask a service module to do this. Instead the manager itself should open a conventional communication block

server and wait for a peer trying to connect. This should be possible in most cases, and also works for anonymous connections. After a connect, the module at the initiator´s side must send a message to the manager of the peer, revealing the module and the security attributes used by the initiator. The responding peer can then check its own security requirements, and examine whether the module chosen by the initiator satisfies them. The drawback of this approach is that it will be difficult to provide functionality for modules which do not wish to use the SEMPER communication block.

The interconnections between the modules and the SecComManager/SecChannel are a bit mixed up. A redesign of the secure communication block should lead to less interactions between manager and modules and a clearer structure. See also in Section 4.10.5 for some issues in the Communication Block.

The block currently does not contain any module management functions (e.g., information about available modules and the security attributes they support). However, those could easily be added similar to the Payment Block.

4.7.5 Related Work

4.7.5.1 IPSEC

IPSEC is a proposal how data traffic can be secured[23] in networks that use the IP family of protocols [IPSEC]. The intention is to have security at the lowest possible layer in the network architecture, so all upper layers (including arbitrary applications) are secured automatically. Key features are:

- Integration into the standard protocol on the internet (IP).
- Support for a variety of external key distribution and agreement protocols.

It is hardly possible to compare the SEMPER secure communication block to IPSEC directly, since IPSEC defines a protocol rather than a framework or API. When IPSEC becomes widely available as a standard communication stack in operating systems, however, an IPSEC implementation might be a module in the SEMPER secure communication block. This would have the following advantages:

- Compatibility with other computers running IPSEC.
- Usage of already defined systems resources.
- This implementation could be used not only for SEMPER, but for all other TCP/IP applications.

The following disadvantages are to be expected:

- The management of keys or certificates used for the connection lies outside of SEMPER.
- It is unclear how it can be controlled which cryptographic algorithms are to be used.

[23] IPSec dos not support the security goal „anonymity“ up to the moment.

- Naturally IPSEC does not support other protocols, in particular not e-mail. Also anonymity is currently not provided by IPSEC.

To avoid the disadvantages while getting the advantages of IPSEC it would be an interesting approach to integrate a module that does communication with IPSEC into the secure communication block.

4.7.5.2 SSLeay BIO Library

SSLeay is a library implementing the SSL protocol and message types [SSLeay, SSL]. SSL specifies certificate and session key negotiations and agreements about algorithms between two communication partners. SSLeay contains a communication module called the BIO library. Key features are:

- I/O abstraction for files and communication endpoints (sockets).
- Enabling codings and cryptographic operations through filters. It is possible to build a stack of filters to apply different operations in sequence.
- C interface.

The SEMPER secure communication block has the following advantages over the SSLeay BIO library:

- Far easier to use.
- Transparent application of cryptographic operations.
- Extensibility (e.g., it should easily be possible to add a module enabling anonymity).

4.7.6 Implementation Notes and Recommendations

Currently there is only one module ("BasicServices"), supporting authenticity and confidentiality. This module uses services of other SEMPER blocks to perform its task. Used blocks are: The SEMPER communication block, which carries out all basic insecure communication with the protocols TCP, HTTP and internet mail, and the SEMPER statement and certificate blocks (see Section 4.8 and 4.6), which are responsible for negotiating certificates, cryptographic algorithms and further parameters such as session keys that are to be used on the connection.

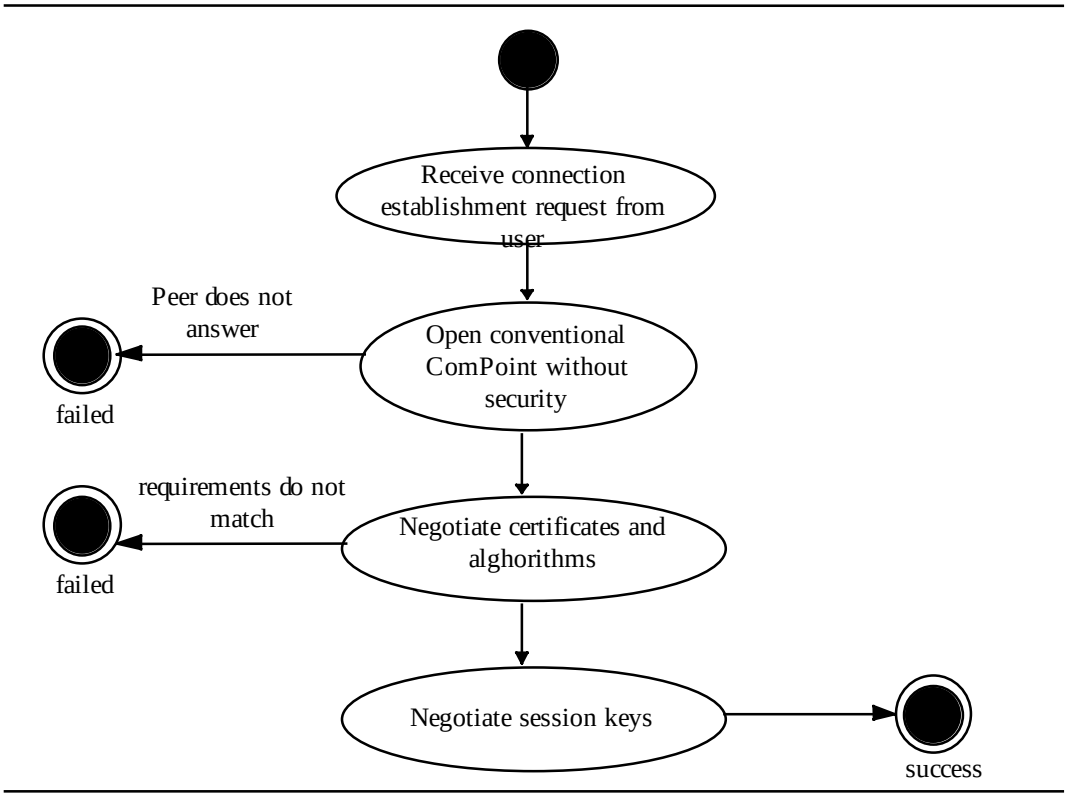

Figure 80 Activities during connection establishment in basic services module

The module performs the following operations in order to secure the communication link: On establishment of a connection (in the sense of a conventional ComPoint), negotiations of the certificate and statement block are started to agree with the peer on certificates and algorithms and to confidentially exchange session keys. When sending data, first a sequence number is attached to every object that is to be send. The recipient checks the sequence number, and if it does not match its own counters, an exception is thrown. This prevents most replay attacks where an attacker sends old messages again that he got from the network earlier. When the authenticity attribute is set, a message authentication code (MAC) is attached to every object that is to be transferred. The recipient checks the MAC; thus an attacker cannot change the contents of the object unrecognized. If the confidentiality attribute is set, data is encrypted under the session key of the connection with the encryption system that the user selected in her preferences. If both authenticity and confidentiality attributes are set, the MAC is attached before encryption.

A drawback of the basic service module is that it destroys the independence between openInitiator and openResponder calls as defined in the communication block, so an openInitiator call blocks until the peer invokes openResponder to do statement and certificate block negotiations.

A description of an anonymity module suitable as extension to this Secure Communication Block can be found in Section 6.

4.8 The Statement Block

(T. Pedersen / CRM)

4.8.1 Domain Description

The Statement Block provides easy access to cryptographic operations on messages so that the entity (application) using the Statement Block does not have to worry about cryptographic details (such as keys). The four standard cryptographic functions MAC, digital signatures, conventional encryption and public-key encryption are required.

The use of keys depends on the recipient(s) of the messages. Therefore, the Statement Block has a manager which negotiates keys with the peer, and uses these keys to establish a context in which the cryptographic operations take place. The block also allows for non-interactive local selection of the key material without negotiation with the peer. In both cases the context is represented by a StatementTransaction object.

The Statement Block does not provide communication of protected messages and it does not in any way interpret the contents of the messages – it only provides the cryptographic operations within the given cryptographic context. It is up to the entity requesting the protection to do any further processing (e.g., sending the protected message to a peer or storing a signed message for non-repudiation purposes).

A StatementTransaction can be used to establish a secure channel (a la SSL). Such a mechanism using encryption and MACs under symmetric session keys is provided by the Secure Communication block (SecCom) (see Section 4.7). The actual protection in the SecCom block is provided by a module. In the current implementation of SEMPER, a module is implemented based on the Statement Block. The relation between the three blocks (crypto, statement and SecCom) is depicted in Figure 81.

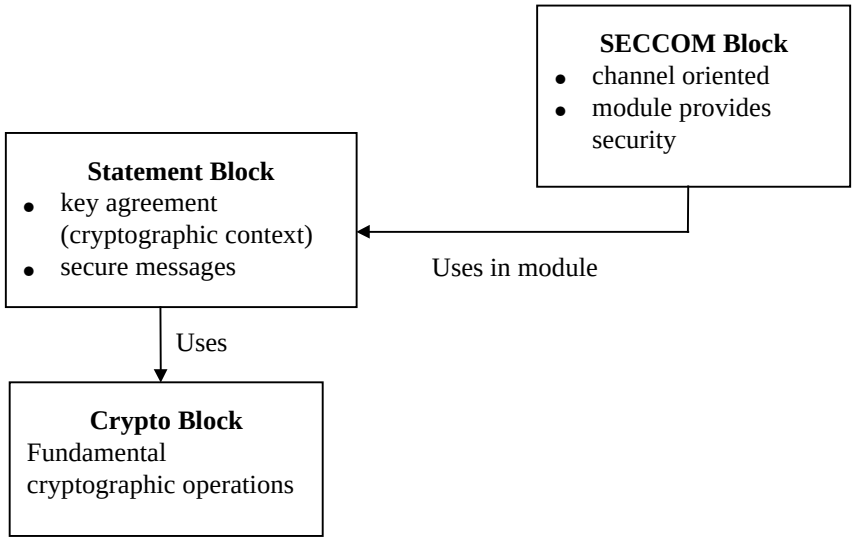

Figure 81: Relation between crypto, statement and SECCOM blocks.

4.8.2 Requirements

The Statement Block must provide a high level interface to the cryptographic functions provided by the Crypto Block. This interface must hide cryptographic details from the user of the Statement Block. This leads to the following more specific requirements:

- allow cryptographic operations on small messages (in RAM), files and more generally streams of data
- allow repeated cryptographic operations on the same message (e.g., several signatures, encryption under various keys, etc.)
- provide methods to interactively set up context with peer enabling the four cryptographic operations (including agreement of session key material)
- provide methods to locally set up context enabling securing messages to a peer
- maintain the security offered by the Crypto Block
- enable the user to select preferred cryptographic algorithms

The Statement Block is not required to deal with certificates. These will be handled by the Certificate Block. The Statement Block will only use certificates as necessary to transport public keys. The idea is that the user of the Statement Block will get the necessary certificates from the Certificate Block and supply these as input to the Statement Block. The Statement Block will carry the certificates, and use the Certificate Block for verification of certificates. Figure 82 depicts this relationship.

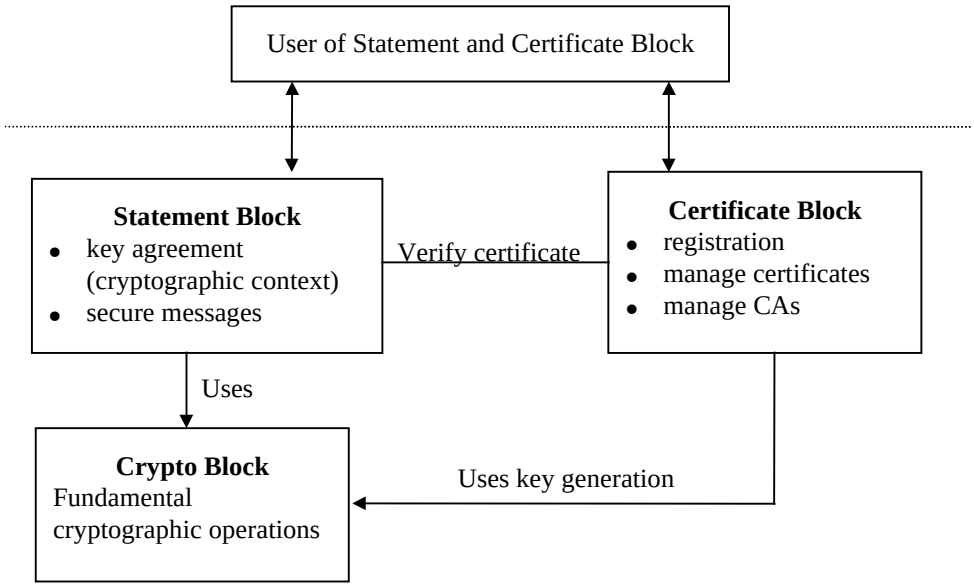

Figure 82: Certificate, Crypto and Statement Blocks

This gives the following use cases for the Statement Block:

Use Case 1: Select cryptographic algorithms: Show possible choices to the user and allow the user to select preferred choices.

Use Case 2: Establish context: Set up context for cryptographic operations.

Use Case 3: Protect document: Perform cryptographic operation (MAC, signature, conventional encryption, public-key encryption)

4.8.2.1 Select Cryptographic Algorithms

The user will need to get information about available cryptographic algorithms and to identify the preferred ones:

The characteristic information of this use case is

- *Goal*: Show information about available cryptographic algorithms and let the user select the preferred algorithms.
- *Conditions:*
- *Pre-condition*: Crypto Block is initialised, so that a list of available algorithms can be retrieved.
- *Success end condition*: The Statement Block knows the user's preferred algorithms
- *Failure end condition*: Default algorithms will be used independently of the user's preferences.

The main success scenario consists of two steps:

1. The user gets information about available algorithms
2. The user selects preferred algorithms.

4.8.2.2 Establish Context

In order to ease using the cryptographic algorithms a context is established describing which

- public-private key pairs to use for encryption and signatures,
- algorithms to use for conventional encryption,
- algorithm to use for MAC computation
- hash function to use when signing a message

If the context is established by interactive negotiation between the two peers it may in addition contain session keys that can be used for conventional encryption and MAC computation.

The characteristic information of this use case is

- *Goal*: Establish context as described above.
- *Conditions:*
- *Pre-condition*: Crypto Block is initialised, so that a list of available algorithms can be retrieved, a certificate context is established (in case of interactive negotiation, a communication channel must be available)
- *Success end condition*: A context is established allowing the operations described above.
- *Failure end condition*: The context is not established.

The main success scenario consists of two steps:

1. The preferred algorithms for the user are identified (possibly using default choices).
2. The user's context is opened based on the selected algorithms

Extensions of Step 2:

If negotiation with the peer is required, the selected algorithms will be refined in comparison with the requirements of the peer. It must be ensured that there is no contradiction between the requirements of the two parties. If explicitly requested, session keys may also be exchanged in this case.

4.8.2.3 Protect Document

This covers both doing the actual protection (encrypting, authenticating) and validating a received document (decryption, verifying MAC or signature).

The characteristic information of this use case is

- *Goal*: Perform requested operation within the given context.
- *Conditions:*
- *Pre-condition*: Context supporting the required operation has been established.
- *Success end condition*: The requested operation is performed
- *Failure end condition*: The requested operation could not be performed.

The main success scenario consists of two steps:

1. Given the message, the algorithms and keys to be used are identified from the context.
2. The actual operation is performed using the services from the Crypto Block.

4.8.3 Design Overview

The statement manager provides services for setting up a session. In case session keys are to be negotiated, this will be handled by the `StatementModule` class (there is not really a module in the Statement Block, but the class is called a module for historical reasons and as it is quite easy to be replaced in order to provide different types of key exchange).

The two essential classes are `Statement` and `StatementTransaction`. Basically the Statement Block offers services on instances of `Statement`. A `Statement` initially contains a message. Any `Statement` can be changed by encryption (symmetric or asymmetric encryption) or authentication (signature or MAC). Any of these operations requires key material as input.

A `StatementTransaction` object offers the same services on `Statement` objects, but the interface is much simpler as this class contains a context describing the parameters for the corresponding `Statement` operation. The actual cryptographic operations on statements are performed by the Crypto Block.

In the following the manager, the key exchange methods and the two main classes `Statement` and `StatementTransaction` are described in more detail.

The Manager

The class `StatementMan` provides services opening a `StatementTransaction` non-interactively (i.e., a party sets up a context allowing him to send signed and encrypted messages to a peer) and interactively (i.e., through netotiation with the peer `StatementMan`).

In both cases the manager takes as input `CertificateContext` objects describing the certificates to be used. It is possible to refine these contexts (in the Certificate Block) to have only one certificate for signature or only one for encryption. The user can then respectively open a `StatementTransaction` only for making / verifying signatures or only for encryption / decryption.

Thus if a (new) `StatementModule` does key exchange via a third party, it is up to the module to handle the certificates of such a third party. The current design of the statement manager does not provide support for such third parties (but on the other hand, it does not prevent the use of them as part of the module).

In the non-interactive case, the manager will construct a context for `StatementTransaction` consisting of the keys defined by the certificate contexts as well algorithms for conventional encryption and hashing.

In the interactive case, the context additionally describes algorithm for MACs, and if requested it may contain a session key object. A session key object basically consists of the following four conventional cryptographic keys:

- Conventional encryption key
- Conventional decryption key
- Key for MAC generation

- Key for MAC verification

Interactive negotiation of algorithms is done in the following natural way (see also [SSL]):

1. Both parties make a list of accepted algorithms within each category (conventional encryption, MAC and hash—functions).
2. One party, referred to as the initiator send his list to the peer.
3. The peer finds the common elements in the two lists picks one of these and returns to the initiator a list of the chosen algorithms.

If requested, this negotiation may also comprise negotiation of methods for exchanging session keys. This is done in the class StatementModule. Two simple protocols are implemented here for demonstration purposes. In real applications the security of these protocols should be analysed and possibly improved. The two possibilities supported (the party initiating the exchange is considered a client, and the responder a server) can be characterised as follows:

- the client chooses a key and sends it to the server signed and encrypted
- the two parties select a key mutually at random

Given a random value agreed between the two parties a SessionKey object can be created. This object will compute session keys for encryption and MAC based on this shared value.

Figure 83 and Figure 84 illustrate the principles of the two simple protocols used for test purposes in the prototype (e.g., names that should be included in signed messages are not shown).

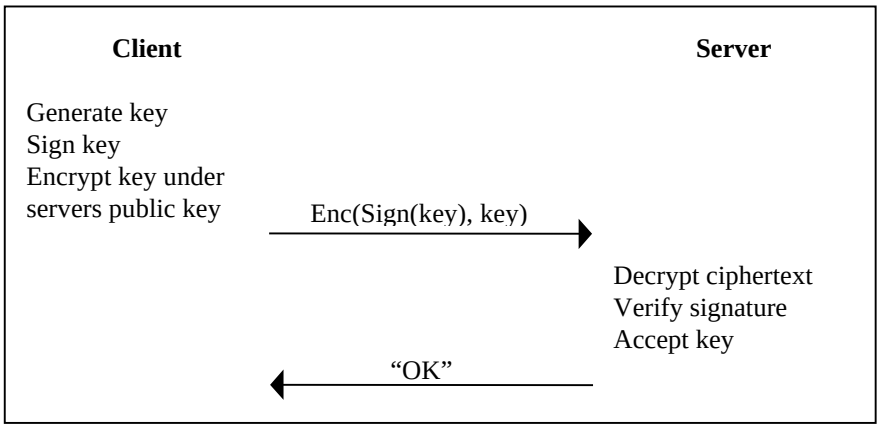

Figure 83: Simple key exchange used in Statement Module

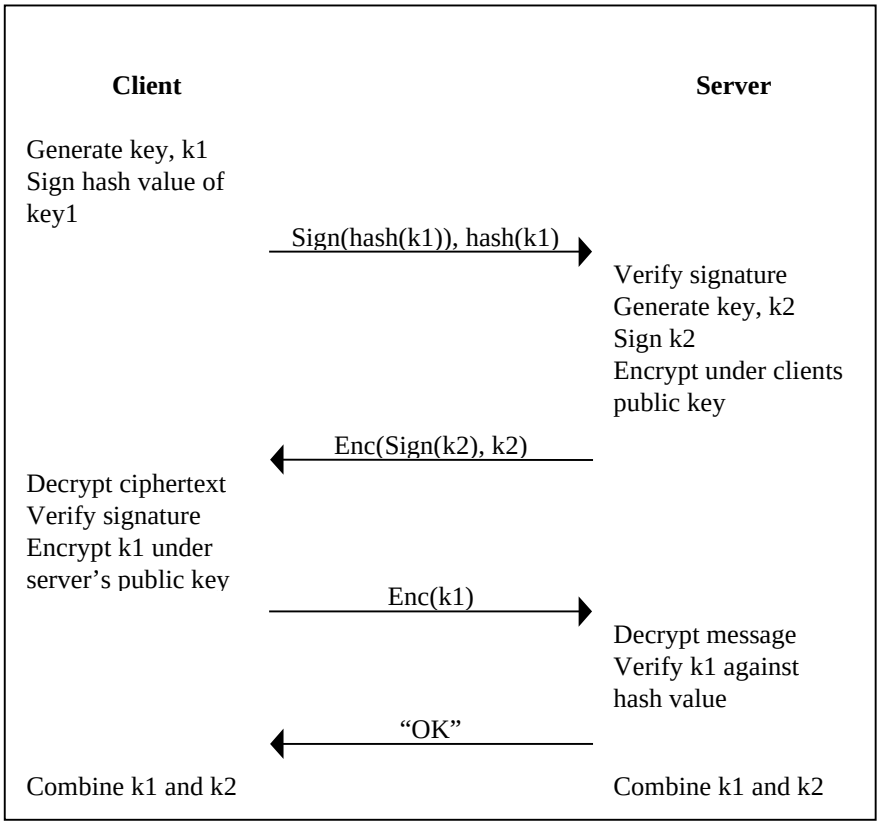

Figure 84: Selecting a key mutually at random.

Statement

A statement contains information processed by the Statement Block. Thus messages to be signed or encrypted should be inserted in a `Statement` object, and the cryptographic operation will be performed on this object. The `Statement` class is abstract, but it has two subclasses corresponding to statements containing files and statements containing information in RAM. The latter is actually subclassed further, by statements containing strings and statements containing any serializable object (in RAM). This makes it easier to operate on `String` objects, which is a very common situation. The design is depicted in Figure 85.

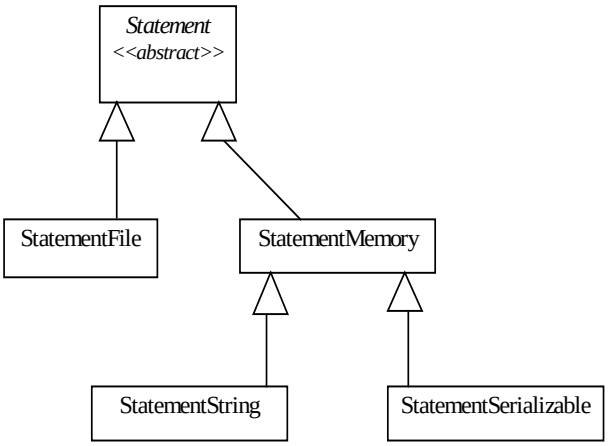

Figure 85: Statement classes

The following operations are available on a `Statement`.

- Generate/verify signature
- Encrypt and decrypt using a symmetric crypto system
- Encrypt and decrypt using an asymmetric crypto system (this means that a symmetric key is generated, the Statement is encrypted under this key and finally the symmetric key is public key encrypted).
- Generate/verify MAC.

StatementTransaction

All the above operations on a `Statement` can also be performed by calling the corresponding operation on a StatementTransaction object giving only the Statement as parameter. The cryptographic context defined by the `StatementTransaction` ensures that the right parameters are passed to the corresponding `Statement` method. This context consists of the following:

- an `AlgorithmContext` object describing the algorithms to be used for hashing, MAC and conventional encryption
- a `SessionKey` object containing (secret, shared) conventional keys for MACS and encryption as described above
- `CertificateContext` object describing the certificate context of the user self (i.e., certificates for encryption and signatures) plus the (two) corresponding private keys
- `CertificateContext` object describing the certificate context of the user self (i.e., certificates for encryption and signatures).

This is depicted in Figure 86.

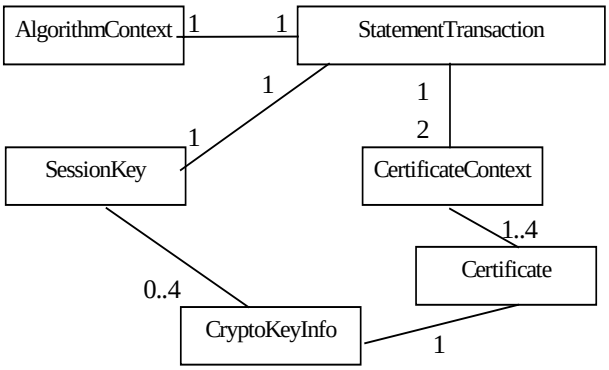

Figure 86: Information about cryptographic context in StatementTransaction

4.8.4 Related Work

Providing cryptographic services on a more abstract level than the Crypto Block, the services of the Statement Block have some resemblance with the GSS API and in particular with the IDUP-GSS for the non-interactive case.

4.8.5 Self-Assessment

The Statement Block currently supports message based mechanisms as well as connection oriented mechanisms, the latter feature being used in a SECCOM module. On one hand, this exploits the common properties of these two situations but, on the other hand, it may conceptually be more clear to divide these in two different blocks. In the message oriented scenario it would be natural to support standardised formats, such as S/MIME, in the Statement Block.

In both scenarios, the negotiation should be token based, so that properties of the communication line (e.g., anonymity) can be used when negotiating a cryptographic context.

4.8.6 Implementation Notes

The implementation has focused on `StatementMemory`. Files are also supported whereas the prototype does not support streams.

4.9 Crypto Block

(T. Pedersen/ CRM)

4.9.1 Domain Description

The crypto block must provide basic cryptographic functions for

- Message authentication
  - MAC values
  - Digital signatures
- Confidentiality
  - symmetric encryption
  - public key encryption
- Key generation
  - symmetric keys
  - public key-pairs
- Generating (pseudo-) random bits and numbers
- Computing message digests (cryptographic hash values)

While the actual operations are done by a **cryptographic module**, a **cryptographic manager** will provide access to these modules through a module-independent interface to the operations.

The crypto block must provide for secure storage of key material. This may be done either by the module (e.g., in case of hardware based module) or by the manager, based on the Archive, in case the module is not able to store keys.

4.9.2 Requirements

The overall requirements for the crypto block is to provide module-independent access to the cryptographic operations listed in Section 4.9.1. As the provision of cryptographic modules is outside *SEMPER* (a module is only provided within *SEMPER* for test purposes), the requirements on the block are in reality requirements on the manager.

The manager must

- Provide access to cryptographic operations provided by a module
- Provide access to key generation (provided by module).
- Provide access to random number generation (provided by module)
- Support secure key storage in case the module does not provide it
- Preserve the security provided by the module (e.g., if the module provides secure storage of keys, the manager should preserve the given level of security)

- Provide module handling allowing the user to install and use a particular module (hardware as well as software modules)
- Provide module negotiation

The manager is not supposed to provide interoperability between different modules. Thus if one entity uses a module, that does not use the same formats for cryptographic tokens as the modules used by the peer, then these two entities may not be able to communicate. This problem must in practice be solved by using standardised formats for these tokens (including digital envelops, digital signatures, cryptographic keys). In recent years a few standardised formats (most notably those proposed by RSA Laboratories, in the PKCS suite of industrial standards) seem to be used by most cryptograhic service providers. As a result the problem of interoperability between cryptographic modules is now smaller than at the time *SEMPER* started.

This gives the following use cases for the crypto block:

Use Case 1: Module Management: Adding and removing crypto modules.

Use Case 2: Module Information: Getting information about possible modules

Use Case 3: Key generation: Generating cryptographic key

Use Case 4: Pseudo-random generator: Generating random bits and numbers

Use Case 5: Cryptographic operation: Doing a cryptographic operation (this could be refined to simpler use cases corresponding to the four types of operation mentioned above).

Use Case 6: Key storage: Storing and retrieving keys.

Use Case 7: Inform about algorithms: Returns information about supported cryptographic algorithms.

These use cases are described in more detail below.

4.9.2.1 Module Management

This is done by the user. The state of a module will be described by the words *available* and *active*:

- Available: the crypto block can use the module.
- Active: the module which is used by the crypto block

Four actions are involved to manage the state of a module:

- insertion (making the module available for the manager)
- deletion (making the module unavailable for the manager)
- activation (making an available module the active one)
- deactivation

The characteristic information of this use case is

- *Goal*: Facilitate the use of external modules in the crypto block.
- *Conditions*: Table 3 lists the conditions for the this use case.

The main success scenario consists of two steps:

1. The user makes a request to the crypto block
2. The crypto block makes the requested action and returns a response to the user.

	Pre-conditions	**Success end condition**	**Failure end conditions**
Insertion	Software/hardware for the module is installed	new module available	new module is not available
Activation	module available	active (i.e., ready for use). Other modules that were active are now inactive	inactive
Deactivation	module active	inactive.	module may still be active
Deletion	module available and inactive	module no longer available	module may still be available

Table 7: Conditions for the Instrument Management use case

Making a module active corresponds to saying that this module should be used by default until another module is made the default one. In case a particular module should be used by default and other modules only should be used occasionally, the default module will be activated during start up and it will only be inactive when other modules are explicitly requested to be active.

The crypto module must be able to supply random bits be it pseudo-random bits or „real“ randomness (e.g., obtained by measuring random noise). As part of activating a module the generator of random bits must be initialised, and this may require some input seed (but a module may store a state, in which case a seed will not be required in every activation).

It is up to the module to ensure that this generator is initialised in a secure way, and in particular if and how to use the supplied seed. The seed will be supplied through the manager, who initially gets random inputs from the user, and manages it securely so that each module can get a unique (pseudo-)random seed as needed for activation.

4.9.2.2 Module Information

The user or application (in the following denoted the caller) will need to get information about available modules, when activating a module.

The characteristic information of this use case is

- *Goal*: Show information about available and active modules.

- *Conditions:*
  - *Pre-condition*: none
  - *Success end condition*: requested information returned to caller
  - *Failure end condition*: requested information not returned

The main success scenario consists of two steps:

1. The caller makes a request to the crypto block
2. The crypto block returns the requested information

4.9.2.3 Key Generation

The crypto block must be able to generate symmetric keys as well as public key pairs. The caller gets a handle to the key.

The characteristic information of this use case is

- *Goal*: Generate requested key for a given cryptographic algorithm.
- *Conditions:*
  - *Pre-condition*: a module supporting the cryptographic algorithm is active
  - *Success end condition*: requested key is generated and a handle returned to the caller
  - *Failure end condition*: requested key not returned

The main success scenario consists of four steps:

1. The caller makes a request to the crypto manager
2. The manager identifies which module to use and asks it to generate the key.
3. The manager stores information about the key (the actual key may already be stored by the module).
4. A handle to the key is returned.

The module may either use its already initialised generator of (pseudo-)random bits, but it should also be possible that the caller as part of the request includes a random seed.

4.9.2.4 Pseudo-random Generator

The crypto block must be able to generate (pseudo-)random bits.

The characteristic information of this use case is

- *Goal*: Generate (pseudo)-random bits.
- *Conditions:*

- *Pre-condition*: a module is active
- *Success end condition*: requested number of bits are generated and returned
- *Failure end condition*: requested bits not returned

The main success scenario consists of two steps:

1. The caller makes a request to the crypto block
2. The crypto block generates the requested number of bits and returns these

The module may either use its already initialised generator of (pseudo-)random bits, but it should also be possible that the caller as part of the request includes a random seed.

4.9.2.5 Cryptographic Operation

The crypto block must provide the following operations:

- Message authentication
  - MAC values
  - Digital signatures
- Confidentiality
  - symmetric encryption
  - public key encryption
- Computation of hash values (e.g., used to compute fingerprints of public key)

These operations are performed on binary data. These basic operations can be used by other blocks to provide cryptographic operations on other data structures (see Section 4.8 for an example).

The characteristic information of this use case is

- *Goal*: Perform required operation
- *Conditions:*
  - *Pre-condition*: a module is active supporting the required operation and the key for the required operation is available
  - *Success end condition*: a given message is protected as described
  - *Failure end condition*: protection of document not changed

The main success scenario consists of three steps:

1. The caller makes a request to the crypto block

2. The crypto manager identifies the keys and verifies that they may be used for the given operation
3. The active module performs the operation given a handle to the key from the manager.

A „handle to the key“ contains information which allows the module to use the key (see Section 4.9.3 for information on how this is done).

4.9.2.6 Key Storage

It must be possible to store keys securely in the crypto block. The manager is required to be able to securely store the information about keys that it maintains. Depending on the module this information may contain the actual key or just a reference to the key allowing the module to identify it. In the latter case, the module must store the actual key securely (e.g., using smart cards).

The characteristic information of this use case is

- *Goal*: Store key
- *Conditions:*
  - *Pre-condition*: access to archive
  - *Success end condition*: key is stored (private keys stored securely) and a handle to the key is defined.
  - *Failure end condition*: key not stored

The main success scenario consists of two steps:

1. The caller makes a request to the crypto block
2. If required the (information about the) key is encrypted and then stored.

4.9.2.7 Inform about Algorithms

The crypto block must be able to inform other blocks (in particular the statement block) about the available cryptographic algorithms.

The characteristic information of this use case is

- *Goal*: Retrieve information about the available cryptographic algorithms.
- *Conditions:*
  - *Pre-condition*: crypto module active
  - *Success end condition*: information about available algorithms returned
  - *Failure end condition*: no information returned.

The main success scenario consists of two steps:

1. The caller makes a request to the crypto block
2. The crypto block returns information about the available algorithms for the given application (e.g., signatures, conventional encryption, etc.)

4.9.3 Design Overview

The Crypto Block provides cryptographic services, which are needed for secure services in *SEMPER* (note that a module in another block may use its own cryptographic tool: e.g., a payment module may sign a payment without using the crypto block). The crypto block is entirely used as a supporting service and does not provide a transaction class allowing the creation of crypto sessions.

The crypto manager offers cryptographic operations through a cryptographic module. These operations are called by using the static methods of the manager, CryptoMan, which then uses the cryptographic module to actually perform the operations.

The two essential classes are `CryptoMan` and `CryptoKeyInfo`. `CryptoMan` is the manager, providing access to the cryptographic services, while CryptoKeyInfo is used to contain module-dependent information about key material. In the following the manager, the prototype module and `CryptoKeyInfo` are described in more detail. In addition the class implementing the cryptographic module is essential for performing the operations.

The Manager

The crypto manager has a state consisting of the choice of the crypto module and the user-specific master key (DES is used in the present prototype, although it is clear that this is not sufficient beyond prototyping - 112 bits triple-DES should for example be used instead). This key, which is implemented by the class `CryptoMasterKey`, is used to encrypt confidential information, which must be stored by the manager (in the actual implementation this is (information about) the private keys of the user and the state of the cryptographic module which is mainly the state of its pseudo random generator). During initialisation of the manager two things take place:

- The manager gets a password from the user, which protects the master key.
- The master key is retrieved and the required module is loaded.

The following operations are available from the crypto manager

1. Cryptographic operations:
   - Key generation (public/private key pairs and conventional keys)
   - Random bit generator (which is also used indirectly for key generation)
   - Making and verifying signatures
   - Making and verifying MAC values

- Encryption (conventional/public-key)
- Computation of message digests using a hash function

2. Provide information about available cryptographic algorithms
3. Provide methods for defining/changing password protecting the master key

While the operations mentioned in item 2 and 3 are provided by the help of the module the last one is provided using the `CryptoMasterKey` class.

Cryptographic Keys

All keys used in SEMPER are represented by the `CryptoKeyInfo` class, which contains the module-dependent information about the key as well as information about the algorithm for which it can be used. Before requesting an operation in the module, it is verified that the key may actually be used for the required operation. The module-dependent part of `CryptoKeyInfo` is represented as a sequence of bytes, which is only interpreted by the module. Thus in principle this can just be a reference to a key on a smart card or it can be the actual key.

It is possible to store a `CryptoKeyInfo` object in the archive (possibly encrypted under a conventional key with the master key as default). In this case, the `CryptoKeyInfo` object contains the `AccessName` needed to retrieve the key from storage (and the actual key value in the object may be cleared).

To handle public-key pairs, a class `CryptoKeyPair` is available. This class simply contains two `CryptoKeyInfo` objects corresponding to the two keys in the pair.

Modules

The cryptographic module provides an API implementing the cryptographic functionality mentioned above. When executing a cryptographic method, the manager will transfer the module-dependent key information received in a `CryptoKeyInfo` object to the module, which will then interpret the key and do the actual operation.

4.9.4 Related Work

A number of standards for Crypto APIs have been proposed.:

- CDSA [CDSA98].
- GCS, see [GCS96].
- MS CAPI, see [CAPI96].
- PKCS#11, see [PKCS11].
- Java Cryptography Architecture (JCA) by JavaSoft. E.g., see [Knudse98].

It is out of the scope of this document to review these APIs. As all these standards provide the required cryptographic functionality, one could in principle use either of them in SEMPER. Here we mention that PKCS#11 (or Cryptoki) from RSA Inc. is a widely used C-API, and it could have been used to define the internal interface that

modules of the crypto block would have to implement. This would have made many cryptographic packages available in SEMPER.

However, since SEMPER is implemented in JAVA there is no doubt that had JAVA's cryptographic architecture been available at the beginning of SEMPER, it would have been natural to use the API's and SPI's proposed by JAVA. JCA consists of the java.security package, which supports digital signatures, and the cryptographic extensions (JCE), which supports confidentiality and message authentication. Unfortunately, JCA has not been available until JDK12, which at the time of writing is still only in beta release. As JCE cannot be exported from US, it would have been necessary to implement as part of SEMPER functionality corresponding to JCE.

4.9.5 Self Assessment

The implementation of the crypto block reflects the fact that no single agreed standard cryptographic API exists, and that there has been no intention to develop as part of SEMPER yet another cryptographic API. This means that the primary goal of the implementation of the crypto block has been to provide the cryptographic services, which are required for electronic commerce. However, still several issues could be added and improved in the design such as:

- Module management: A proper interface to the modules should be defined and module management needs to be supported.
- Currently the manager is the central class of the block. It could be an advantage to delegate more functionality to the CryptoKeyInfo class. In particular, instead of calling cryptographic operations through the manager, the operation should be performed on the object representing the key. This object would then use the manager to be informed about which module to use. This will require a hierarchy of classes representing different types of keys.
- Import and export functionality should be added to the `CryptoKeyInfo` class providing methods allowing to get the keys encoded in various formats. This would have made it more convenient to use the cryptographic keys outside the cryptographic module (in particular in the certificate block, as the public key is explicitly encoded in certificates).

4.9.6 Implementation Notes and Recommendations

In the current implementation, there is no access control. The crypto block protects keys and other confidential information using a master key, which is again protected under the user's password.

Access to the keys can be controlled by the users of the crypto block by requesting a password, and using the crypto block to verify it, but nothing in the crypto block ensures that such verification has taken place.

However, a secure implementation of the crypto block would require that access to operations operating on keys is restricted to entities allowed to use the keys.

4.10 Communication Block

(M. Nassehi / ZRL)

4.10.1 Domain Description

In order to discuss the requirements, we need to introduce some terminology. The purpose of the communication block is to deliver messages between entities. A sequence of message exchanges between two entities starts by one entity sending a message to another. We refer to the entity that sends the first message as the initiator and the other as the responder. After receiving the message the responder may send a response to the initiator. Note that this response may be the first message in the exchange with a semantic content; in this case, the purpose of the initiator's message would be only to allow a response.

4.10.2 Requirements

The overall requirements for the communication block are

- to make application development independent of the underlying communication protocols, and
- to multiplex channels using the same address.

There are no security requirements on the communication block other than that the it transfer exactly the information the sender wants to send; in SEMPER security is provided by the layers above this block. The functional requirements on the communication block are illustrated by the following use cases. We use the Universal Modelling Language [FowSco97] to describe the these use cases as well as the design.

4.10.2.1 Description of Use Cases

There are three types of actors: the communication block itself, the communicating entity and the communication protocol, which provides the actual message delivery. The communicating entities play different roles, namely, initiator and responder. The relationships between actors and use cases are shown in Figure 87.

4.10.2.1.1 Basic Communications

In basic-communications use case, there are two communicating entities involved: an initiator and a responder.

- ♦ Goal: To deliver messages and responses between the initiator and responder.
- ♦ Conditions:
  - Pre-conditions: None.
  - Success conditions: Messages and responses have been delivered.

- Failure condition: a message or a response has not been delivered.

♦ Main Success Scenario: This is described in Figure 88. The responder opens a server-socket entity. The initiator opens a client-socket entity using the responder's address. The initiator writes a message while the responder accepts a client. The responder reads a message and sends a respond.

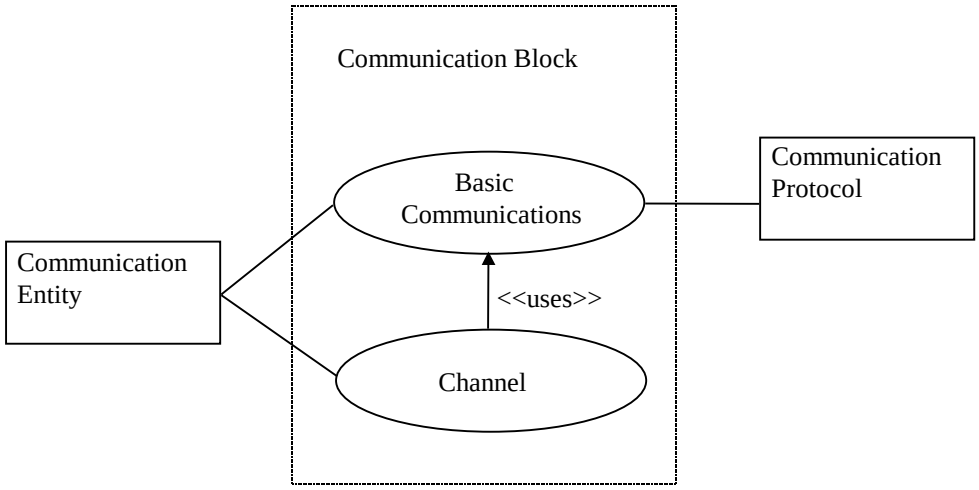

Figure 87: Relationships between actors and use cases.

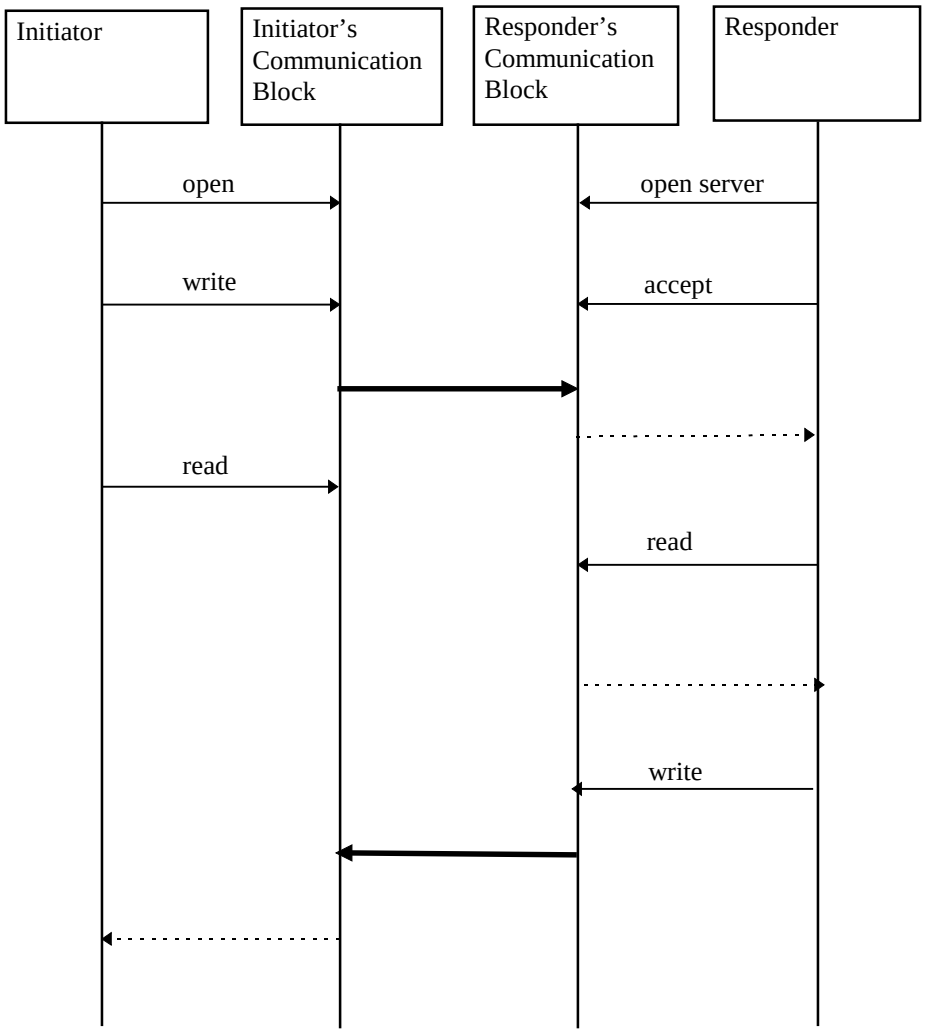

Figure 88: Main success scenario for basic-communication and channel use cases.

4.10.2.1.2 Channel Multiplexing

In the channel-multiplexing use case, there are again two communicating entities involved: an initiator and a responder.

- Goal: To deliver messages and responses between the initiator and responder using an address and a specific correlator. A different correlator and the same address can be used to establish communication between a different initiator/responder pair.
- Conditions:
  - Pre-conditions: None.
  - Success conditions: Messages and responses have been delivered.
  - Failure condition: a message or a response has not been delivered.
- Main Success Scenario: This scenario is similar to that of the previous use case, which was shown in Figure 88. The only difference is that in the open functions a correlator string is specified.

4.10.3 Design Overview

4.10.3.1 Introduction

The design of the communication block is based on an object referred to as `ComPoint`. Like a socket, a `ComPoint` provides access to the underlying communication protocols. For each protocol supported an adapter is required. Currently, the adapters for TCP, HTTP and mail have been implemented. The selection of the protocol is done only in the addresses, i.e. `ComPoint` provides a unified interface across all protocols. Additional classes provide the channel service. Note that in the design presented here the `ComPoint` and Channel APIs are based on the synchronous, i.e. blocking, model.

4.10.3.2 Object View

4.10.3.2.1 Object Model

Figure 89 and Figure 90 represent the primary classes for basic communications and channel multiplexing, respectively. We now briefly refer to these classes.

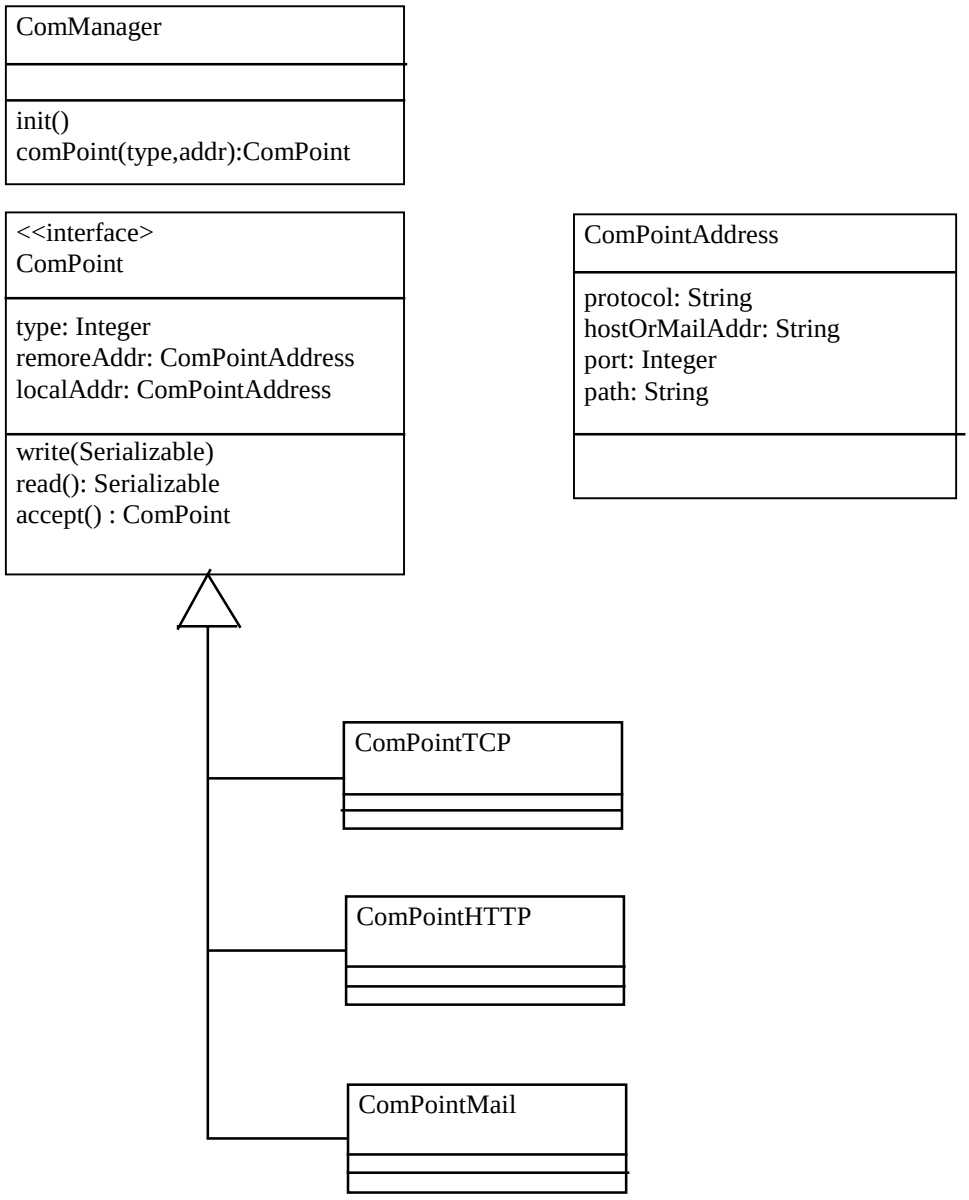

Figure 89: Classes for basic communications.

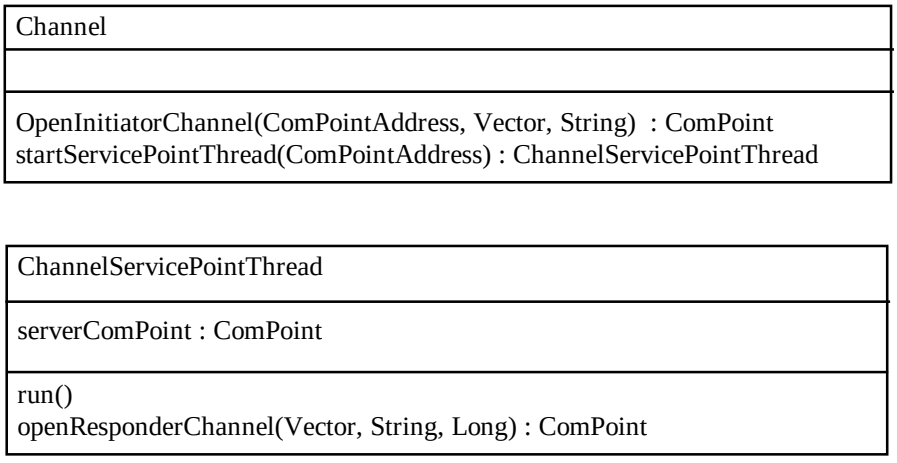

Figure 90: Classes for channel multiplexing.

`ComManager`: is a class which provides the mapping between the communication field of an address and the communication protocol. This mapping is established based on the configuration file when the `init` method is invoked. Whenever the `comPoint` method is invoked, a `ComPoint` object with the requested type, i.e., initiator or server, and the requested address is created and returned.

`ComPoint`: is an object which allows an application program to access the underlying communication protocol. For each supported protocol, there is class that implements it. In order to simplify the task of the application programs, these implementation classes are not a part of API.

`ComPointAddress`: is similar to the `java.net.URL` object, but allows the protocol field to be expanded and accommodates the mail protocol.

`Channel`: provides a class method for opening a server-point thread and one for opening a channel on the initiator side.

`ChannelServicePointThread`: is the server-point on the responder side and provides the method for opening a channel..

4.10.3.3 Functional View

The Communication Block must be initialized through the `init` method of the `ComManager` class. This `init` method invokes the corresponding methods of the classes implementing the communication protocols, currently `ComPointTCP`, `ComPointHTTP` and `ComPointMail`. The invocation of the `init` method of

`ComPointMail` will result in an actual initialization, only if the three parameters which are used by `ComPointMail` (see the section on the configuration parameters below) are set in the configuration file. Otherwise, the mail protocol may not be used.

4.10.3.3.1 Basic Communications

The system on which the initiator runs must have initialized the communication block by invoking the `init` method of `ComManager`. Similarly, the communication block on the responder side must be intialized. Examples of code segments best illustrate how the basic-communications service is implemented. Here is a sample code for the initiator side:

```
// Create the address of the responder.
ComPointAddress address = new ComPointAddress(protocol, host, port, path);

// Open an initiator ComPoint.
ComPoint comPoint =
ComManager.comPoint(ComPointConstants.COMPOINT_INITIATOR, address);

// Send a message.
Serializable message = ...;
comPoint.write(message);

// Receive a response.
Serializable response = comPoint.read();

// Exchange of more messages and responses.
...

// Close the initiator ComPoint.
comPoint.close();
```

Here is the corresponding code for the responder side:

```
// Create the local address.
ComPointAddress address = new ComPointAddress(protocol, port);

// Open the server ComPoint.
serverComPoint = ComManager.comPoint(COMPOINT_SERVER, address);

// Accept the connection from the initiator, creating the responder ComPoint.
ComPoint comPoint = serverComPoint.accept();

// Receive the message.
Serializable message = comPoint.read();

// Send the response.
Serializable response = ...;
comPoint.write(response);

// Exchange of more messages and responses.

...

// Close the responder ComPoint.
comPoint.close();

// Eventually close the server ComPoint.
serverComPoint.close();
```

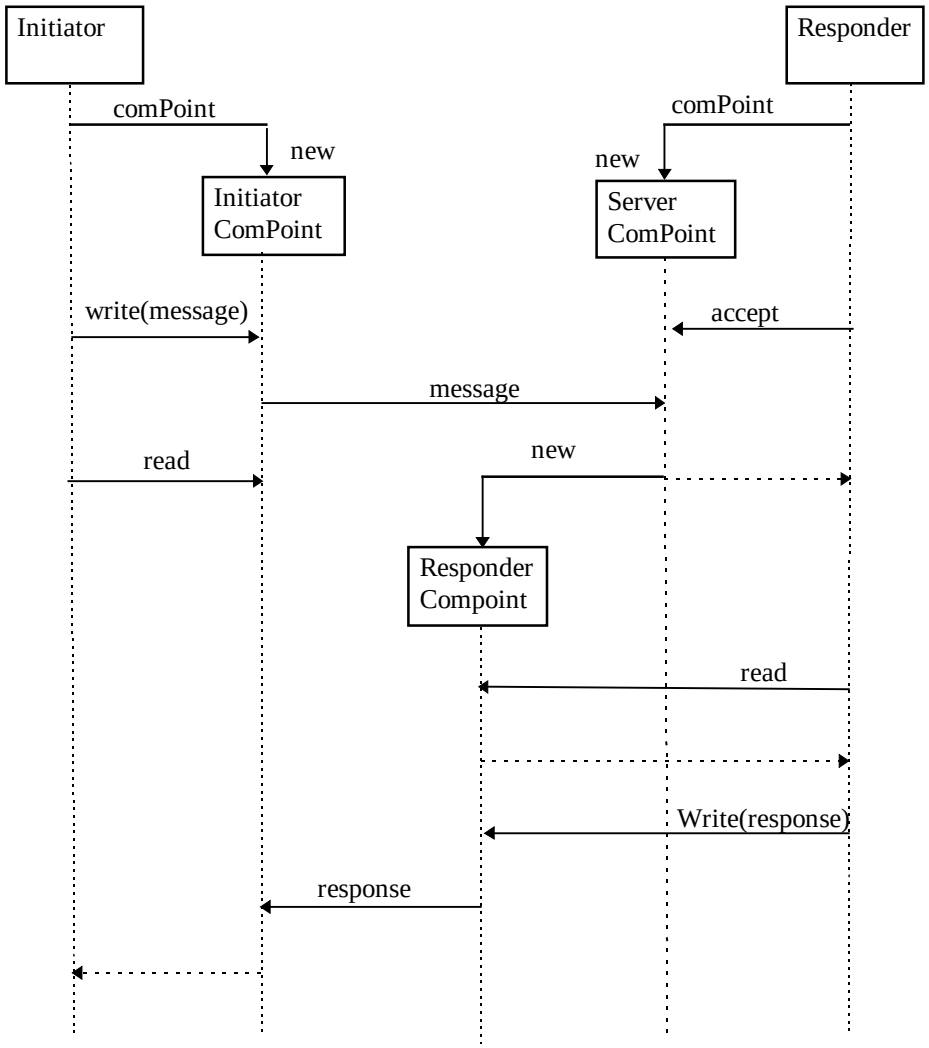

Figure 91: Interactions during basic communications.

These two code segments show the sequences of method invocations on the initiator and responder sides when they exchanges massages and responses using the basic communication service. The resulting interaction is shown by the sequence diagram in Figure 91. Before the initiator takes any action, the responder opens a server `ComPoint` by invoking the `comPoint` method of the `ComManager` for a specific port. Any time after that, the initiator may open a `ComPoint` by invoking the `comPoint` method for the host address and the port number of the server `ComPoint`. The initiator can then send a message by invoking *write* on its `ComPoint`. The communication block uses the specified network protocol to deliver the message to the responder. If *accept* is already invoked on the responder `ComPoint`, a new responder `ComPoint` is returned. Otherwise, a `ComPoint` is returned when `accept` is invoked. When the responder invokes *read* on this `ComPoint` the message is returned. It can then send a response to the initiator. Using the same pair of initiator and responder `ComPoints` an unlimited number of message/response pairs may be exchanged.

4.10.3.3.2 Channel Multiplexing

Channel refines the basic-communications service of `ComPoint`. It provides two features. First, each address in the address space of a higher-level protocol using the communication block can be used for communication between more than one pairs of entities. Secondly, it provides a synchronization feature in that the responder need not indicate to its server that it is expecting a message from an initiator before that initiator can send a message; the message waits at the server until the responder indicates that it is expecting a message.

A channel between two hosts is identified by a port number together with a correlator string. The initiator module requests a channel to a specific host with a specific port and correlator by invoking `openInitiatorChannel`, which returns an initiator `ComPoint`. If the responder module has already invoked `openResponderChannel` to indicate that it expects such a channel, then a responder `ComPoint` is returned to the responder module. Otherwise, the channel is queued at the host where the responder resides until the responder invokes the `OpenResponderChannel` method. Therefore, Channel not only provides multiplexing over individual port numbers, it also lets the responder module to act after the initiator module. Here is a sample code for the initiator side:

```
// Create the address of the responder.
ComPointAddress address = new ComPointAddress(protocol, host, port, path);

// Open an initiator ComPoint and initiate a channel.
ComPoint comPoint =
Channel.openInitiatorChannel(address, options, correlator);

// Send a message.
Serializable message = ...;
comPoint.write(message);

// Receive a response.
Serializable response = comPoint.read();

// Exchange of more messages and responses.

...

// Close the ComPoint and channel.
comPoint.close();
```

Here is the corresponding code for the responder:

```
// Create the local address.
ComPointAddress address = new ComPointAddress(protocol, port);

// Start a service-point thread.
ChannelServicePointThread servicePoint =
Channel.startServicePointThread(address);
```

```
// Accept a channel with a specific correlator and open a responder ComPoint.
ComPoint comPoint =
servicePoint.openResponderChannel(options, correlator);

// Receive the message.
Serializable message = comPoint.read();

// Send the response.
Serializable response = ...;
comPoint.write(response);

// Exchange of more messages and responses.

...

// Close the responder ComPoint.
comPoint.close();

// Eventually stop the service-point thread.
servicePoint.stop();
```

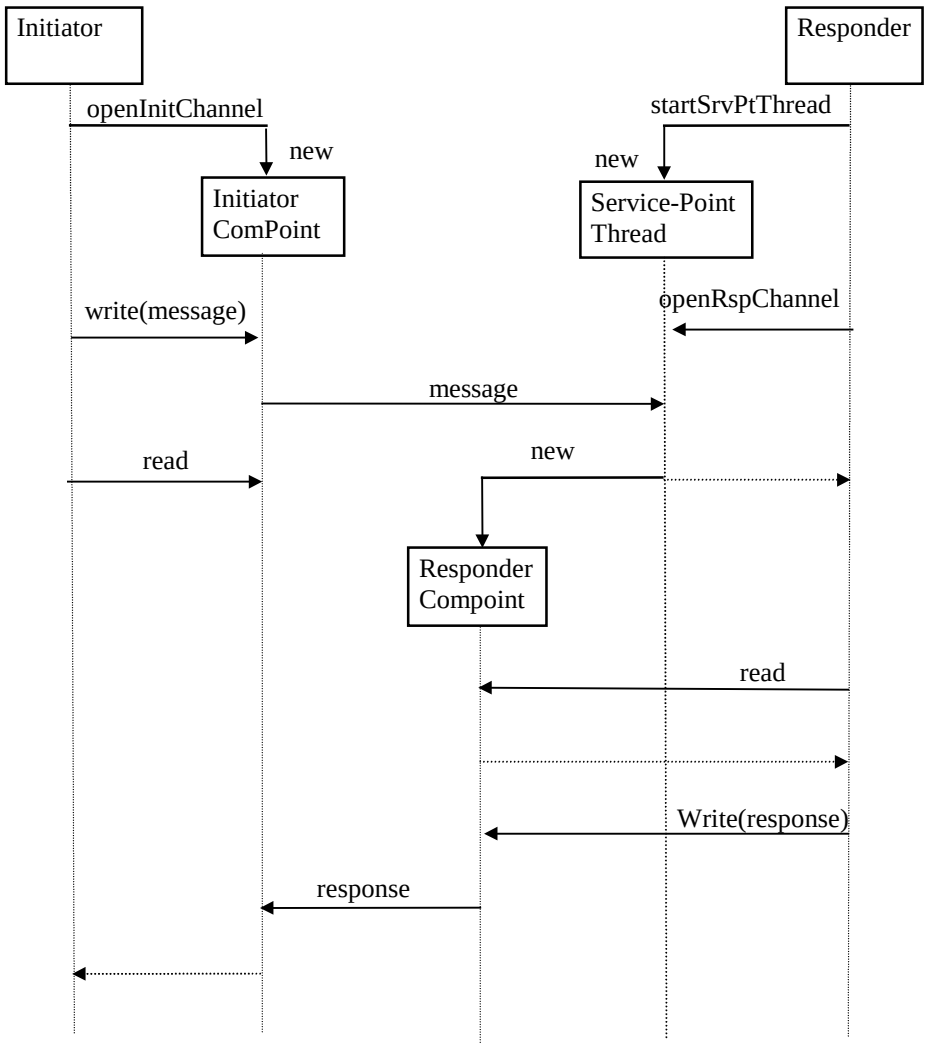

Figure 92: Interactions during channel multiplexing.

These two code segments show the sequences of method invocations on the initiator and responder sides when they exchange messages and responses using the channel-multiplexing service. The resulting interaction is shown by the sequence diagram in Figure 92. Before the initiator takes any action, the responder must have opened a service-point thread by invoking `startServicePointThread` of Channel for a specific port. Any time after that, the initiator may open a ComPoint by invoking `openInitiatorChannel` for the host address and the port number of the service-point thread and a specific correlator. The initiator can then send a message by invoking *write* on its `ComPoint`. If `openResponderChannel` with the same correlator has already invoked on the service-point thread, a new responder `ComPoint` is returned. Otherwise, a `ComPoint` is returned when `openResponderChannel` is invoked. The exchange of messages and responses proceeds as in the case of basic-communications service.

4.10.3.4 External Connections

4.10.3.4.1 ComPointMail

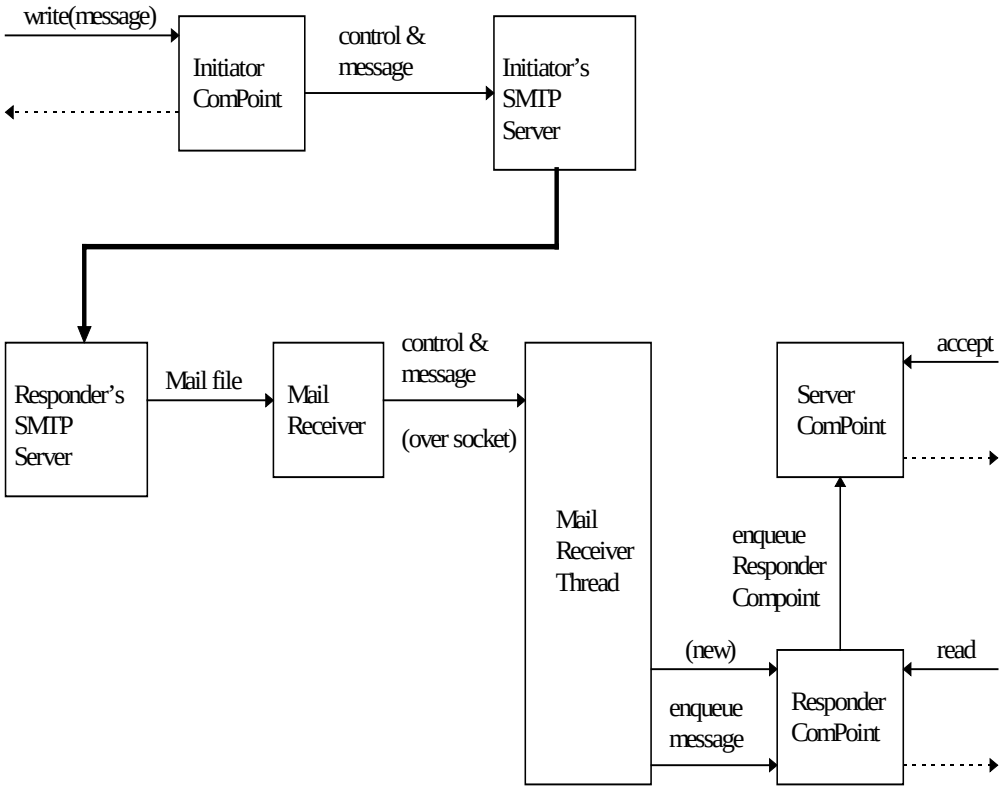

Figure 93: Initiator sending a message through mail.

`ComPointMail` requires interaction with the local SMTP servers and a Mail-Receiver program, as illustrated in Figure 93. This figure corresponds to the case when an initiator sends a message to a responder; a response is handled similarly. As shown in the figure, the initiator opens a `ComPoint` and writes a message to it. The communication block sends the message together with some control information to the local SMTP server, which forwards the message to the responder's SMTP server. The Mail-Receiver class is a main program that is invoked on the content of the mail

in order to send the message and control information to the communication block. A Mail-Receiver-Thread class examines the control information and looks for the responder `ComPoint` to which the message is directed. If the `ComPoint` exists the message is enqueued into it. Otherwise first a `ComPoint` is created and enqueued in the server `ComPoint`. The `ComPoint` (message) is returned immediately if accept (read) is already invoked. Otherwise, it is returned when there is the corresponding invocation.

4.10.4 Related Work

The `java` Socket provides the main TCP service provided by the communication block. This service is provided by the `ComPointTCP` class. An additional feature is provided by `ComPointTCP` which allows using telnet, as would be required in the scenarios involving a firewall. Such a scenario was also the main motivation for implementing `ComPointHTTP`, which would allow using a proxy server. The motivation behind implementing `ComPointMail` was that for some high-level protocol interactions it is not feasible to use sockets, as would be the case, for example, when the remote machine is off line.

4.10.5 Self-assessment

There are at least two aspects of the design that can be improved. First, the `ComPoint` interface is message oriented, where as an stream-oriented alternative would be more inline with Java stream classes and more flexible. Secondly, the addresses for different communication methods, such as TCP, HTTP, and mail are different; for example, TCP does not require a field for path. Therefore, it would conform more closely to the object-oriented paradigm to provide one address class for each communication method. For the same reason a separate class for server `ComPoints` is more desirable. This would require a corresponding `serverComPoint` method for the `ComManager`.

4.10.6 Implementation Notes

The `semper.comm` package implements the design presented here. In the design, only the primary methods were described. Additional methods are provided in the implementation such as those for accessing the fields of an object and closing an object.

4.11 TINGUIN block

(B.Patil/GMD)

4.11.1 Domain Description

The security systems require a secure, trusted graphical user interface. We call this interface TINGUIN - Trusted Interactive Graphical User Interface. TINGUIN is a basic building block within the architecture of SEMPER. It represents and realises a trusted graphical user interface to different SEMPER managers and services. The user is assured that all input and output through TINGUIN is secure and can therefor, be trusted. Trust and security functions are specific features of TINGUIN. TINGUIN is used to present security relevant data such as passwords, PINs, digital signatures, business offers and orders, certificates and payments.

The main purpose of TINGUIN is to provide facilities for the management of TINGUIN sessions and to provide a set of GUI components. A TINGUIN session is a class that manages the internal window session with necessary synchronisation and session access control.

4.11.2 Requirements

The basic requirement of the TINGUIN block is twofold:

1. *Provide an environment for the effective management of TINGUIN sessions.*

2. Provide a generalised, simple and semantically correct, graphical user interface components to different SEMPER managers and services.

The TINGUIN block should

- *Support a simple and semantically correct GUI*
- *Manage TINGUIN sessions, multiple sessions and access control*
- *Enable text-only and split-tinguin displays*
- *Give a consistent and uniform user interface to business applications*
- *Support GUI components such as buttons, forms, menus, text and password fields, plain and HTML text, hyperlinks etc.*

The TINGUIN block is used by applications via its API.

4.11.3 Use Cases

The first actor is an *application* (e.g., Deal browser, Certificate browser) of the SEMPER services that uses the TINGUIN management session and TINGUIN GUI components on behalf of the human user via an API. The second actor is the end user. The end *user* is the actual person using SEMPER system either as a service consumer or provider. The user must have full control of security critical operations. All SEMPER managers use TINGUIN to realise their GUI and provide an effective user

interface to SEMPER services. The relationships between actors and use cases are shown in Figure 94.

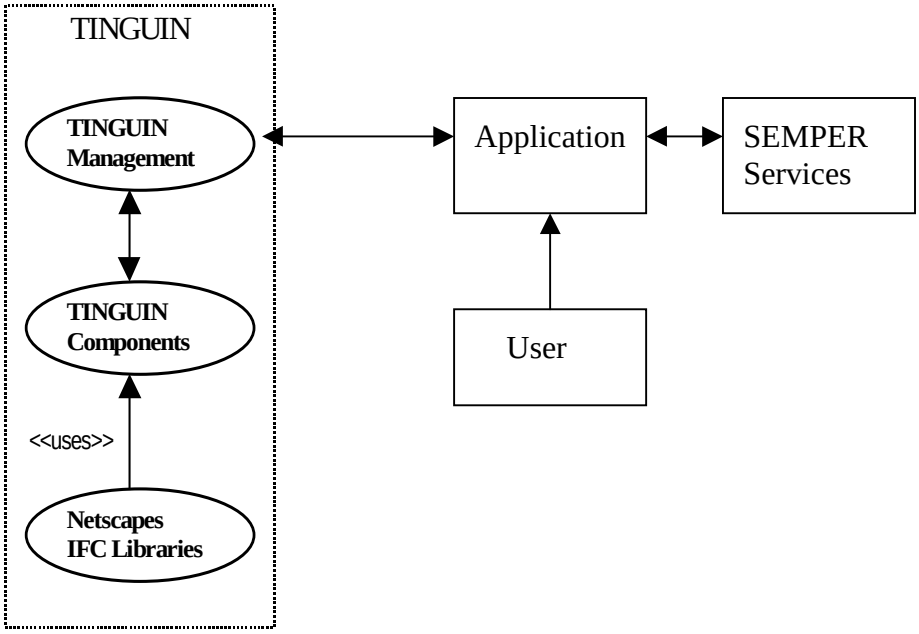

Figure 94: Relationships between actors and use cases

Use Case 1: TINGUIN management is the main use case of the TINGUIN management services. It represents and realises a TINGUIN session with necessary services. A TINGUIN manager manages a set of TINGUIN sessions. TINGUIN manager is an entry point for the programmer. It manages the multiple sessions, allocate and deallocate a TINGUIN session. It supports the integration of TINGUIN components. TINGUIN display is a shared resource. Therefore, resource conflicts are managed by using synchronised access methods.

Use Case 2: TINGUIN components are derived from the Netscape's IFC libraries. It allows easy integration of standard GUI components such as buttons, forms, menus, text and password fields, ASCII & HTML text, and hyperlinks.

4.12 Design Overview

4.12.1 Introduction

As introduced by the use cases above, the TINGUIN block provides a unified GUI interface to SEMPER services, and it provides an environment for TINGUIN management services.

TINGUIN components are derived from Netscape's IFC libraries. Most applications must respond to events of some kind, whether they are user interactions with the interface, such as button clicks or key presses or hyperlinks or a program controlled actions. The event processing is based on Netscape's IFC event model. TINGUN is designed and implemented in different classes which are integrated into a package with a Java API. The important classes and their relationships are described in Figure 95. We describe selected public classes of TINGUIN. Table 8 shows an excerpt of the Java API of TINGUIN.

4.12.2 Object View

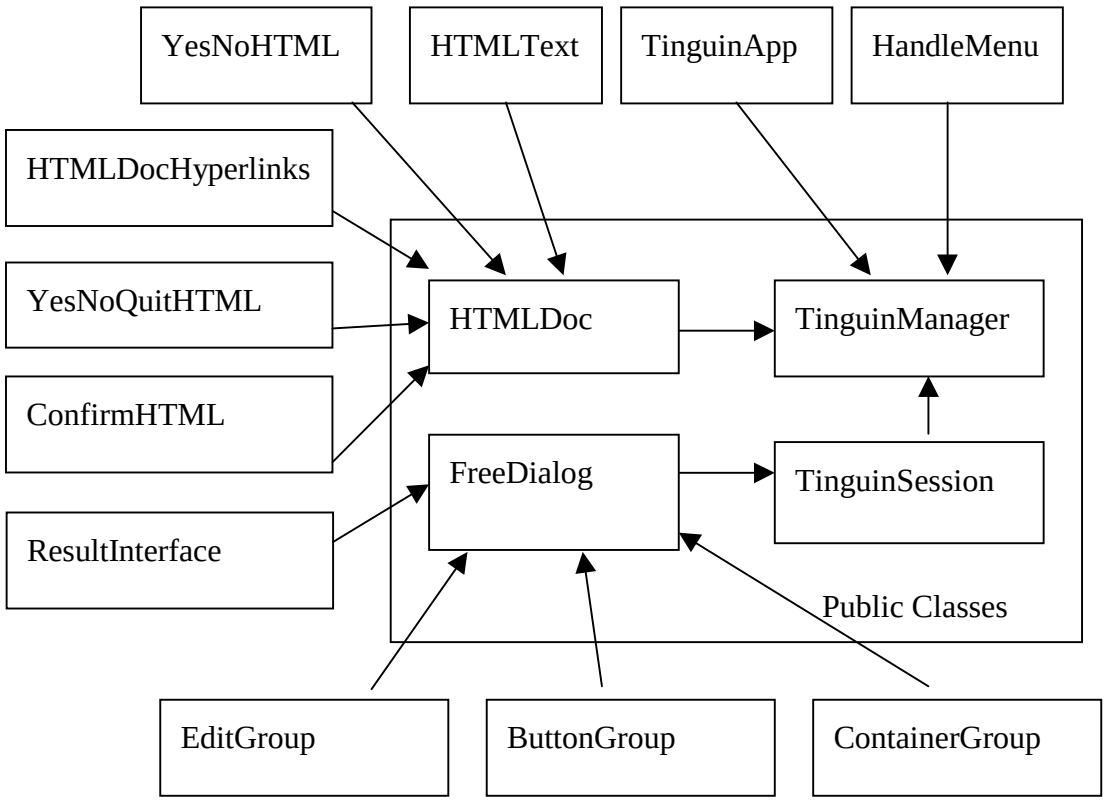

Figure 95: Object View for TINGUIN Services.

TinguinManager is a static class. It does not need an instantiation although an initialisation will be necessary in order to build a menu for the user preferences and to register various SEMPER menu-launchable applications. TINGUIN is initalised by *init().* The *init()* method loads the user preferences, foreground and background colors and TINGUIN data paths. The TinguinManager is used to create a new TINGUIN session. Each created session appears in the "Sessions" menu in TINGUIN with associated buttons and status window. The *newSession()* method returns a reference to TinguinSession. It is possible to integrate menus, add or set status texts used for short status information.

T**inguinSession** is an internal window with necessary GUI components. Since the TINGUIN display is a shared resource, only one session is allowed at a time to display its output or wait for its inputs. TINGUIN exploits multi-threaded Java environment to synchronise threads around a condition variable (viz, Boolean acquired) through the use of monitors. A monitor is associated with a specific data item and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked and cannot access or modify that data. The critical sections of the code are implemented as synchronised methods.

TinguinSession supports acquiring a session, releasing a session, adding multiple sessions, adding multiple windows and removing windows. When a session is closed, all related windows of that session are closed automatically.

HTMLDocuments are HTML strings, which are used to display the business offers and orders, certificates and payments. The *HTMLDoc()* class displays the HTML strings in an internal window. The user can confirm HTML documents by selecting one of the options. TINGUIN offers hyperlink support to browse HTML documents. The access to external URL is not permitted because of security reasons.

FreeDialog Services are special classes, which can hold several types of views in one window. These are *ContainerGroup, EditField, EditGroup, PopupGroup, RadioGroup, ButtonGroup and others.* The *FreeDialog c*lass contains a large set of classes/methods to support effective GUI components. A *FreeDialog* has two buttons at the bottom of the TINGUIN screen. The *FreeDialog* supports the integration of buttons, popup groups, radio buttons, text and password fields. The results of the *FreeDialog* services can be obtained by *getResult () method.*

Syntax	Description
Void TinguinManager::init()	*Initalise TINGUIN.*
Void TinguinManager::newSession(String name)	*Start a TINGUIN Session*
Void TinguinManager::endSession(TinguinSession session)	*End a TINGUIN Session*
MenuItem TinguinManager::addMenu(String title)	*Add a menu.*
Void TinguinManager::setStatusText(String text)	*Add status text.*
Void TinguinSession::acquire()	*Acquire a TINGUIN session.*
Void TinguinSession::release()	*Release a TINGUIN session.*
Void TinguinSession::addWindow(InternalWindow win)	*Add a window.*
Void TinguinSession::removeWindow(InternalWindow win)	*Remove a window.*
Void TinguinSession::removeAllWindows(	*Remove all windows.*
HTMLDocHyperLinks(String title)	*Add a HTML hyperlink document.*
oHTML(String html)	*Display for yes/no option.*
YesNoQuitHTML(String title)	*Display for yes/no/quit option.*
FreeDialog(String title)	*Start a free dialogue.*
Void FreeDialog::addText(String text)	*Add a text to the free dialogue.*
Void FreeDialog::addButtonGroup(Vector elements, int alignment)	*Add button group*
Void FreeDialog::addRadioGroup(String title, Vector ele, int align)	*Add radio group.*
Void FreeDialog::addPopupGroup(Vector elements)	*Add a pop group.*
Void FreeDialog::addHTMLText(String html)	*Add HTML Text.*
Void FreeDialog::addEditField(String text, String edittext)	*Add edit field.*
Void FreeDialog::addPasswordField(String text, String pass)	*Add password field.*
Void FreeDialog::addEditGroup(Vector editFields)	*Add an edit group.*
Void FreeDialog::addGroup(Vector views)	*Add a group.*
Void FreeDialog::addSpace(int pixel)	*Add Space.*
Void FreeDialog::endDialog()	*End a dialogue.*
Vashtable FreeDialog::getResult()	*Get the results.*
Void PrograssBar(String Tittle)	*Start a progress bar.*

Table 8: Public TINGUIN Java API.

4.12.3 TINGUIN Display and User Interactions

In principle, there are only a few possibilities, how the user may interact with the system. The dialogues are used as mechanisms to model the user interactions.

- No Reaction: A text/document is only displayed. No response from the user is necessary, for example display of messages or progress bar.
- Confirmation: A text/document is displayed and it has to be confirmed by the user by selecting the options.
- Selecting alternatives: A text/document is displayed and the user has to answer a "Yes/No" or "Yes/No/Quit" question by clicking corresponding buttons. For more alternatives, user can interact with the help of list boxes or radio buttons.
- Entering text: A text/field is displayed, in which the user enters a text, for example a business form. In some applications, the user has to enter secure data such as passwords or PINs, which should be invisible to an observer.
- Hyperlink Support: User can click hyperlinks and browse the HTML text.

The TINGUIN screen and its GUI components are shown in Figure 96.

Menu Bar

Currently the menu bar offers only a selection of a few menu items. The preferences menu contains the possible preferences the user may select and edit; for example, the user may select security algorithms. In the session area, different TINGUIN sessions are listed. It is possible to switch between them by clicking on an associated session button.

Document Area

This part of the window displays documents, which are potentially security relevant or confidential. For example, a business offer could be displayed, and if the user would like to order, the user has to fill out a business form.

Interaction Part

In this part of the window, all interactions with the user will be managed. The user interactions are modelled using standard GUI components such as buttons, labels, menus, lists and scroll controls.

Status Window

The status window of the TINGUIN screen will show the status of different TINGUIN and SEMPER events.

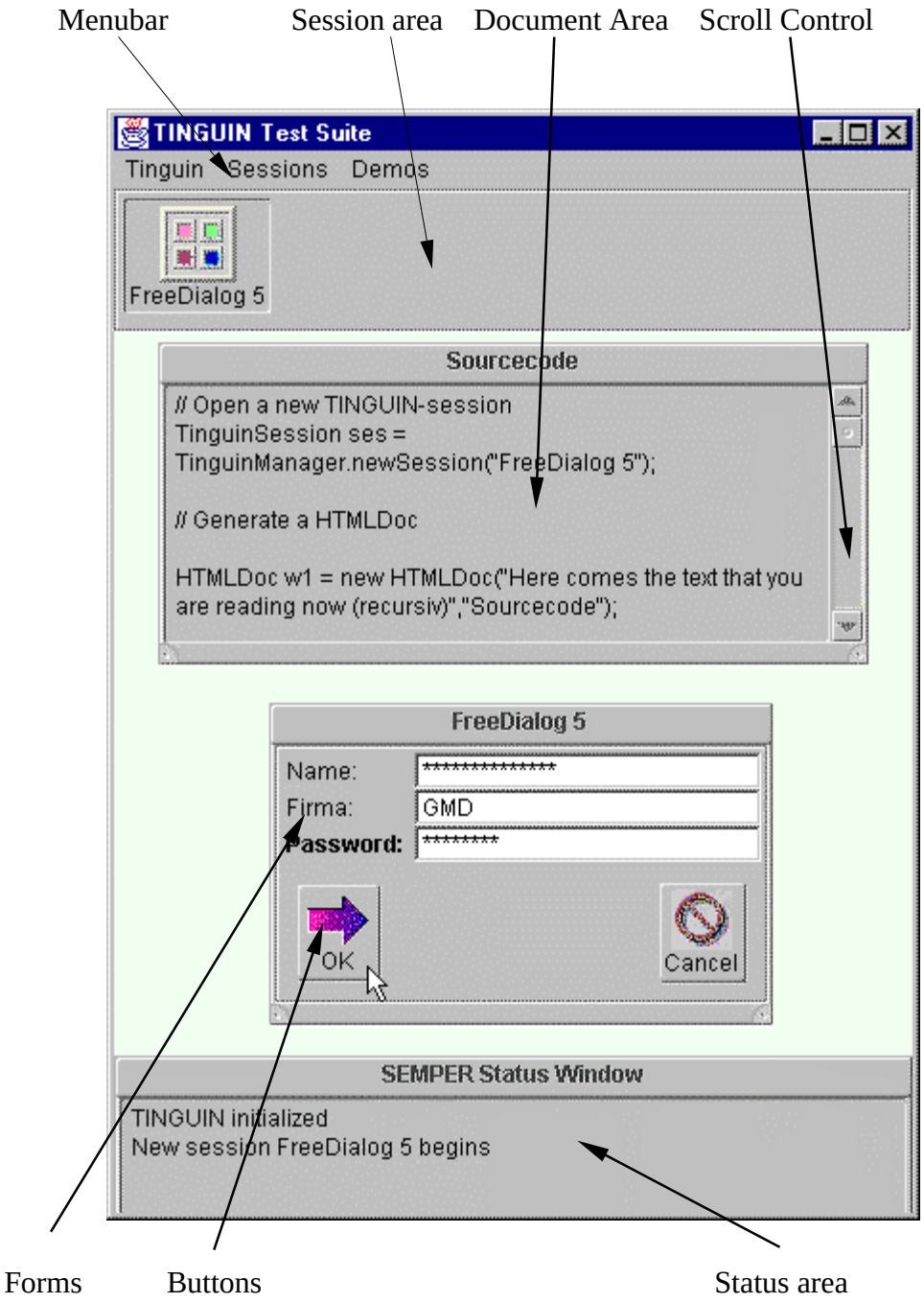

Figure 96: TINGUIN windows and GUI Components.

4.12.4 Related Work

Here we mention the use of standard Java and Netscape's IFC libraries. Basic TINGUIN components are derived from the IFC libraries. TINGUIN is an effective GUI for SEMPER services. Trust and security are unique features.

4.12.5 Self Assessment

Java and Netscape's IFC libraries allow an easy integration of GUI components in an application. It clearly distinguishes untrusted data and trusted data. TINGUIN offers most of the functions of a standard browser. It also supports effective management of multiple sessions.

4.12.6 Implementation Notes and Recommendations

In the current implementation, the important trust feature is that only SEMPER managers can access TINGUIN API for the realisation of their GUIs. Another trust feature is the customization of backgrounds screen.

We propose the concept of "split-tinguin" for future work. In split-tinguin, TINGUIN is divided into two parts: the main part shows a graphics or icons or texts on the main display and minimal part displays few line characters implemented in a temper resistant device. The main idea of this split-tinguin is that all security information is displayed/entered on the secure device. The current version does not support graphics, error recovery and "undo" operations.

4.13 Preferences block

(M. Mazoué / SPT & Louis Salvail / CWI)

4.13.1 Domain Description

The preferences block provides preference management functionality to the other *SEMPER* blocks. It works on so-called **Preference Groups,** which are hierarchical. Each preference group contains some **Preference Fields** and its values. Fields and preference groups are to be defined by the other *SEMPER* block managers.

The Preference Services define what choices a user has and what default values are given. The part that defines how the user is shown the available preferences is closely related to the TINGUIN.

4.13.2 Requirements

The following requirements must be covered by the preferences services:

- user front-end for accessing preferences groups
- storage and retrieval of preferences groups
- management for preferences groups (creation, deletion, put, get, ...)
- management of preferences fields (get value, put value, show me, ...)
- initialisation of preferences based on a user-configuration file

In addition, the preference block stores the actually installed configuration, i.e., all information specific to a particular installation like paths, the target system, the memory size, the screen type or any other hardware devices currently supported.

The *functional requirements* of the preferences service can be modelised by use cases that describe typical interactions between the archive and its users as shown below.

4.13.2.1 Use Cases

The first actor is an *application* (practically, a service of the *SEMPER* application) which uses the preferences on behalf of the human user via an API. The second actor is the human *user*. The Preferences service uses the Archive services. The relationships between actors and use cases are shown in Figure 99.

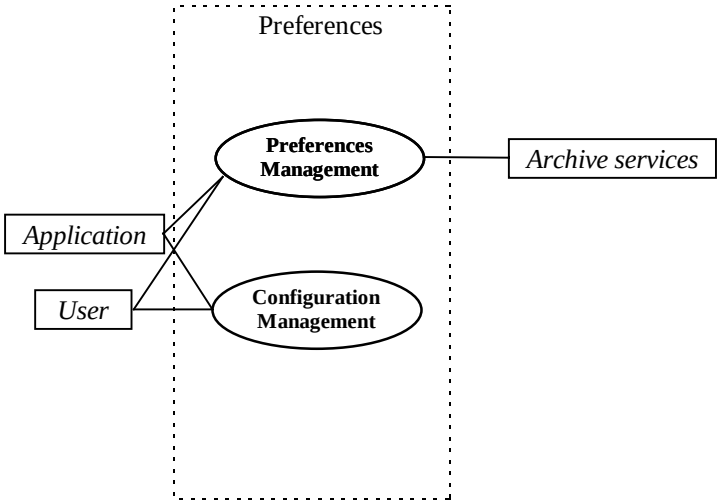

Figure 97: Relationships between actors and use cases

Use Case 1: Preferences management: the core use case of the Preferences services. Preferences represent dynamic data that can be observed and changed by the user at any time. These data are stored via the archive services in a secure way. An application such as every *SEMPER* service can process preferences data through the Preferences Manager.

Use Case 2: Configuration management: inside the Preferences Block, configuration information represents an almost static data structure *SEMPER* needs at start-up. This includes all information specific to a particular installation like paths, the target system, the memory size, the screen type or any other hardware devices currently supported. Usually the configuration information is changed at *SEMPER* installation or when a new component is added to the system. Although configuration data are quite static, the Preferences Block provides services to change them.

4.13.3 Design Overview

4.13.3.1 Introduction

As introduced by the use cases above, the preferences block provides a unified interface to *SEMPER* preferences and configuration information. Configuration is usually retrieved from a standard file that can be found on the user storage device whereas preferences are stored encrypted in the Archive.

Each module/manager defines it's own set of preferences by grouping together several fields. The data structure allowing grouping of field definition and manipulation is called `PrefGroup`. Fields are defined by the `PrefFields` data structure. In addition to its actual value and label, each field also defines its graphical appearance, which is used for interactive editing.

4.13.3.2 Object View

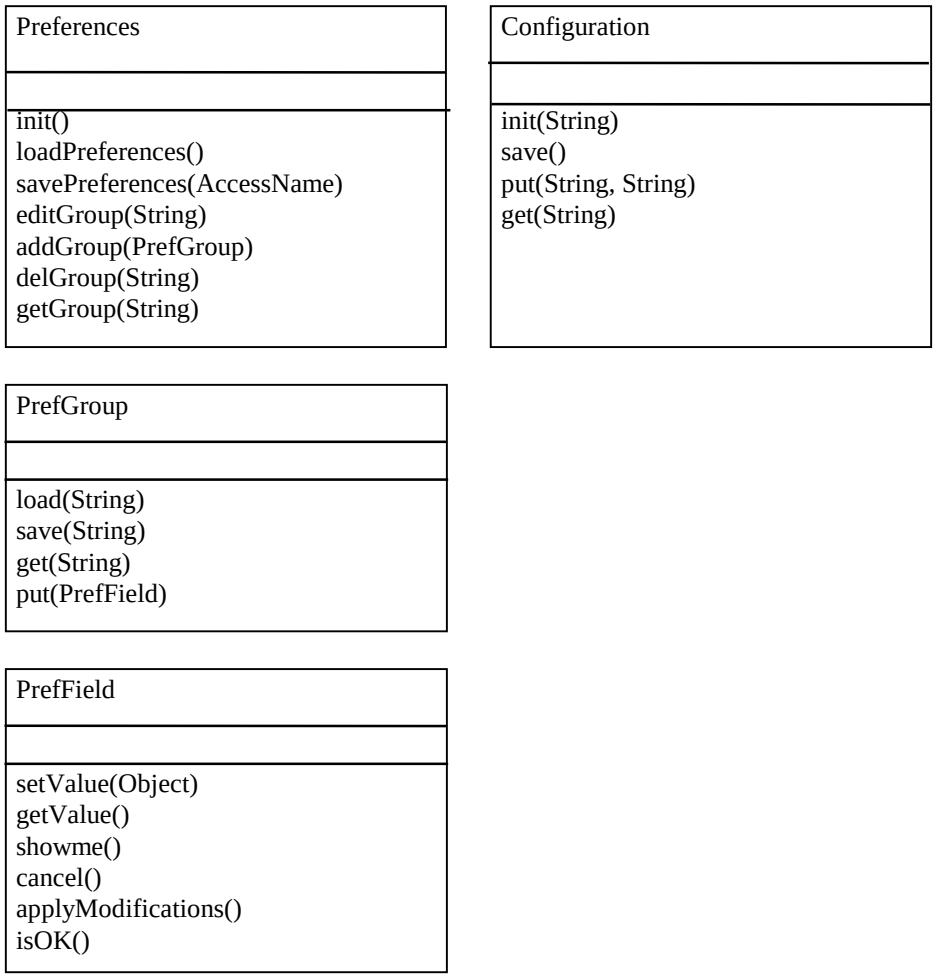

Figure 98: Classes for Preferences Service

4.13.3.2.1 Configuration

Configuration data have very simple structure defined in the `Configuration` data class. This is a collection of configuration fields each having a name described by a string. Associated to each field, its actual value is represented by another string. All the methods use the local file system because configuration data have to be available before the Archive manager is initialised. Below are described the main methods of the `Configuration` class:

init

Before using any of the Configuration services, initialisation must done. A call to init loads into internal representation the default configuration file "_sempconfig". It is also possible to ask the init method to load user configuration from another file.

save

This is used to save in the default configuration file the current user configuration set-up. It is also possible to save in a different file than the default one. This function is

also used to build a new configuration file from which the Configuration can be initialised.

put, get

These methods are used to put a new field and get the value of a field in the configuration database.

4.13.3.2.2 Preferences (manager)

The Preferences Manager is implemented as the class `Preferences`. It provides services for initialisation, input-output and groups manipulation as described in the following methods

init

Configuration must be initialised before Preferences; otherwise the Preferences Manager refuses to be initialised. Preferences initialisation loads the preferences environment from the default file stored in the Archive. If no user preferences are found, the Preferences Manager is initialised with an empty set of preferences. This initialisation process only requires the availability of the Access Control in order to be completed but needs also Archive Manager if a non-empty preferences environment is expected.

loadPreferences, savePreferences

Once initialisation is done, the preferences manager can load a preferences environment previously saved in the Archive. The loadPreferences method can be used when initialisation has not succeeded to retrieve a preferences environment.

editGroup

This allows interactive editing of a preferences group. This service is closely related to TINGUIN. Calls to editGroup are usually made by TINGUIN who gives access to the editing of each group in the current environment. The preferences manager allows simultaneous editing of different groups. This allows the user to look at different groups of preferences simultaneously.

addGroup, delGroup, getGroup

These methods give the basic tools to manipulate groups of preferences. The first two services are also used when a module/manager needs to build its group. When the business application does not succeed to retrieve an environment, each module/manager should be able to construct its group with default values by adding it to an initially empty environment. To make preferences available, the new environment must be saved and the Preferences Manager and then initialised with the new preferences.

The method getGroup is used to retrieve all data contained in a group. This is used in particular when managers are initialised.

4.13.3.2.3 PrefGroup

The preferences manager processes groups of preferences fields. Each group is implemented as `PrefGroup`. A `PrefGroup` processes preferences fields in the same way as the preferences manager does. The following methods are used:

load, save

This allows to load and save individual groups in the Archive. This can be used to build a preferences environment out of several one.

get, put

These are used to get or put a particular field from a group. Fields are indexed by their name. The order in which fields are put is important because of the way formatting is processed (see below).

4.13.3.2.4 PrefField

A preferences field is the atomic data structure for preferences. Each field is implemented by a `PrefField` object. These objects have a current value and an appearance. Different fields appearance are currently supported; they are implemented by subclasses of PrefField:

- `PrefFieldLabel`: a string displayed in the preferences group,
- `PrefFieldString`: an input string with a current value,
- `PrefFieldCheckBox`: a check box with a description of its purpose,
- `PrefFieldChoice`: a list of items with one of them selected,
- `PrefFieldNegotiable`: a set of items from which some are unused and some are part of a priority list,
- `PrefFieldPassword`: password modification facility.

The field `PrefFieldNegotiable` allows to define fields to represent priority lists. The field looks like 2 lists of items; the first one contains a list of unused items while the other is the priority list itself. By selecting one element from the unused list and one from the priority list the user indicate that the selected unused item will be added in the priority list at the selected position. If no position is selected in the priority list, the new item is added in last position. It is also possible to select an item in the priority list and put it back in the unused list.

The `PrefFieldPassWord` class implements preferences fields dealing with passwords. It is possible to associate to a password a preferences field allowing to modify it. A `PrefFieldPassWord` appears like a button on a preferences group. When the user press the button, the control is given to the TINGUIN for asking the user to enter the password to change. If the user succeeds then (s)he is asked to select a new password by giving it twice. The change is processed if at the end the user press the `"use"` button of the preferences group.

Below are some methods provided by every field:

getValue, setValue

Allow to get and set the value of the field. The type of value depends on the field type.

showme, cancel, applyModifications, isOK

These methods provide the management tools for interactive editing. The first method shows the field in a frame that can be displayed by the TINGUIN. The user may then decide to apply the modification or to cancel changes. Changes are applied only if the new value set by the user is allowed.

4.13.3.3 Functional View

4.13.3.3.1 Initialisation

The Preferences Manager is one of the first managers which has to provide its services during start-up. For that reason, the initialisation process of the preferences manager does not rely on any other manager except the Access Control. Therefore, we assume that in order to be completed, the Access Control initialisation does not rely on configuration information. However, the retrieval of a preferences environment needs services from the Archive Manager and the Crypto Manager. It follows that Archive Manager and Crypto Manager cannot use a preferences environment to provide the subset of services required for preferences environment retrieval. The general *SEMPER* initialisation process should then follow the following steps:

1. Configuration is initialised,
2. Access Control Manager, Crypto Manager and Archive Manager use user configuration in order to make available the subset of services needed by the Preferences Manager for initialisation,
3. Preferences Manager is initialised and user preferences are loaded,
4. The other managers are now completely initialised according to user configuration and preferences.

4.13.3.3.2 Formatting fields within a group

Each field contains formatting information in order to create a user-friendly display of each group. When a group is displayed, fields are put one after the other according to formatting constraints. These constraints are relative to each other and are applied in the same order fields were put in the group. When the preferences manager puts a new field for display, the constraint for that field is added to the previous one. Since a constraint A followed by a constraint B is not the same as the constraint B followed by the constraint A, the order in which they are applied is important. The preferences manager applies constraints from the first field of a group to the last one in the same order these fields were added when the current group has been defined. These formatting constraints allow flexible interaction between the user and the group appearance by supporting automatic frame resizing. Each field type comes with default formatting constraints which usually give good results. For a more complex group, field constraints can be redefined allowing to design the desired group appearance. For more information about constraints see the `GridBagConstraints` class in the standard Java JDK.

4.13.3.3.3 Preferences and TINGUIN

The default TINGUIN window contains a preferences menu by which groups can be selected by the user. When a group is selected, the preferences manager starts to run interactive editing of that group. Each field appears in the window giving the user the ability to select new values. Two options are then possible:

- *Cancel* of every modification performed so far in that group editing,
- *Use* new values save them before closing the current group editing. If some values have not been set correctly, the user is prompted to correct the setting.

In addition, the preferences menu may provide choices for loading and saving a preferences environment.

4.13.4 Related Work

We can mention here the standard Java properties. The **java.util.Properties** class represents a persistent set of properties which can also contain a default properties list. The differentiating factor is that *SEMPER* Preferences are more structured, GUI editable and are stored in a secure way.

4.13.5 Self assessment

As mentioned above, Java properties allow easy manipulation of persistent information (Strings). Thus it should have been probably a good choice for the *Configuration* information management design.

4.13.6 Implementation notes and Recommendations

Some functions described in this document have not been implemented or fully implemented in the current version:

- loading/saving individual preferences groups in the archive,
- loading/saving a preferences environment through the TINGUIN preferences menu.

Independently, an interesting point is that the preferences manager can be extended in order to manage **Profile** information. This kind of information contains results of previous negotiations between two entities. When the user performs the same service frequently, it is natural to re-use the result of previous negotiations in order to make the negotiation process free of user intervention when it is possible. In general, Profiles define a detailed environment for a specific business session. This information can be stored in the Preferences Manager like for user configuration and preferences. This can be done with a *PrefFieldNegotiable* class.

4.14 Archive block

(M. Mazoué / SPT)

4.14.1 Domain Description

The archive service is located in the lowest layer of the *SEMPER* architecture together with the other supporting services. It should be used by the other *SEMPER* entities when they need a **local persistent storage** for their data. This concerns public and secret keys, certificates, transaction containers and all data required for services, audits or arbitration in dispute handling. Persistency is obviously essential for the conservation of user's personal data and traces of transactions. It is also necessary for fixing the recovery points a fault-tolerant service needs.

The physical location of the archived data is the user's computer hard disk (there is no centralised remote archive in the *SEMPER* architecture). A complete framework might include the storage of sensible data in a tamper-resistant memory like a smart card or other external secure devices.

4.14.2 Requirements

For portability's sake, the whole archive must be written in Java. Though some of them are quite largely used, the use of native commercial relational databases accessible via the JDBC gateway is not foreseen here. The first reason for that is the portability's point. Another reason is that the *SEMPER* architecture does in principle not need a complex (and therefore costly, and resource consuming) database management system for its archive service.

The first requirement for the archive is to provide the functions of a basic database system:

- **Basic persistent objects management (storage**, **retrieval**, **update, deletion,...)**: these functions are only intended to the *SEMPER* application programs; the archive does not need to provide a human user interface since human interactions must be exceptional. When necessary, the *SEMPER* TINGUIN should be used. The database management does not need to be of the relational type. A unique key is enough to handle each object in the base.

The *SEMPER* application requires a certain level of security from its archive service. This level of security can be accorded to the kind of data to be stored. The following security requirements can be identified:

- **Integrity**: the retrieved object must equal the stored object (no external interference), and objects should only be modified or deleted by authorised users (usually only the owner of the object who stored it in the archive).

- **Confidentiality**: the objects can only be read by authorised users (usually only the owner of the object who stored it in the archive). Data stored in the archive should not be exposed by reading the external files storing the archive and its backups: sensible data should be encrypted.
- **Availability**: objects stored in the archive should always be available. An object stored in the archive cannot get loss unless if explicitly deleted or it has reached its expiration date. The risk of losing data in the archive due to hardware failures should be limited (fault-tolerance).

Additionally the archive should provide multi-user and mobility facilities:

- **Multi-user support**: different users using the same computer should be able to consecutively execute the same *SEMPER* program and thus share the same archive.
- **Concurrent access:** two *SEMPER* applications simultaneously running on one computer should not cause problems regarding the archive.
- **Mobility/exportability**: a user should be able to move or duplicate its *SEMPER* archive from one computer to another, eventually running under different operating systems.

The *functional requirements* of the archive can be modelised by use cases that describe typical interactions between the archive and its users as shown below.

4.14.2.1 Use Cases

The first actor is an *application* (practically, a service of the *SEMPER* application) which uses the archive on behalf of the human user via an API. The second actor is the human *user*. The relationships between actors and use cases are shown in Figure 99.

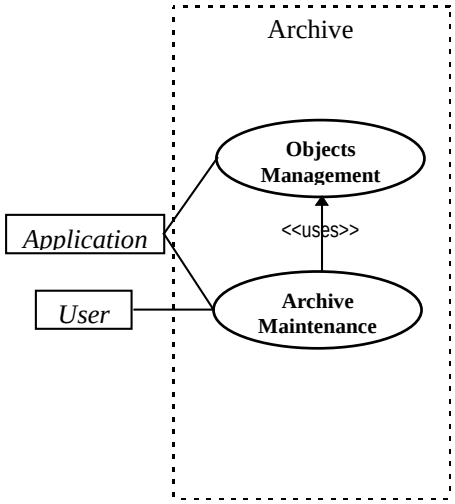

Figure 99: Relationships between actors and use cases

Use Case 1: Objects management: this is the core use case of the archive. The archive gets the objects to store from the other *SEMPER* entities and makes them persistent. When explicitly required, objects are encrypted before being stored. Once stored, the archive manages every operation on the persistent objects: retrieve, update, delete, etc. with applying security settings (security attributes of each object).

Use Case 2: Archive Maintenance: this use case covers some helpful services some of which are related to security and some other more to user's comfort. The first category includes database integrity checks and backups, the second addresses services like garbage collection on stored objects and database exportation. In contrast to use case 1 some of the services in use case 2 need user interaction. These interactions can be done via the TINGUIN by preferences settings (periodicity of automatic backups,...) or by a specific archive menu (backup demand, exportation demand,...). The archive may also eventually use the TINGUIN for displaying dynamic information (« action on progress », warnings,...).

4.14.3 Design Overview

4.14.3.1 Introduction

The archive manager object is the input point for all the archive services. Some helper classes are around it. An underlying database module carries out the object management basic functions.

4.14.3.2 Object View

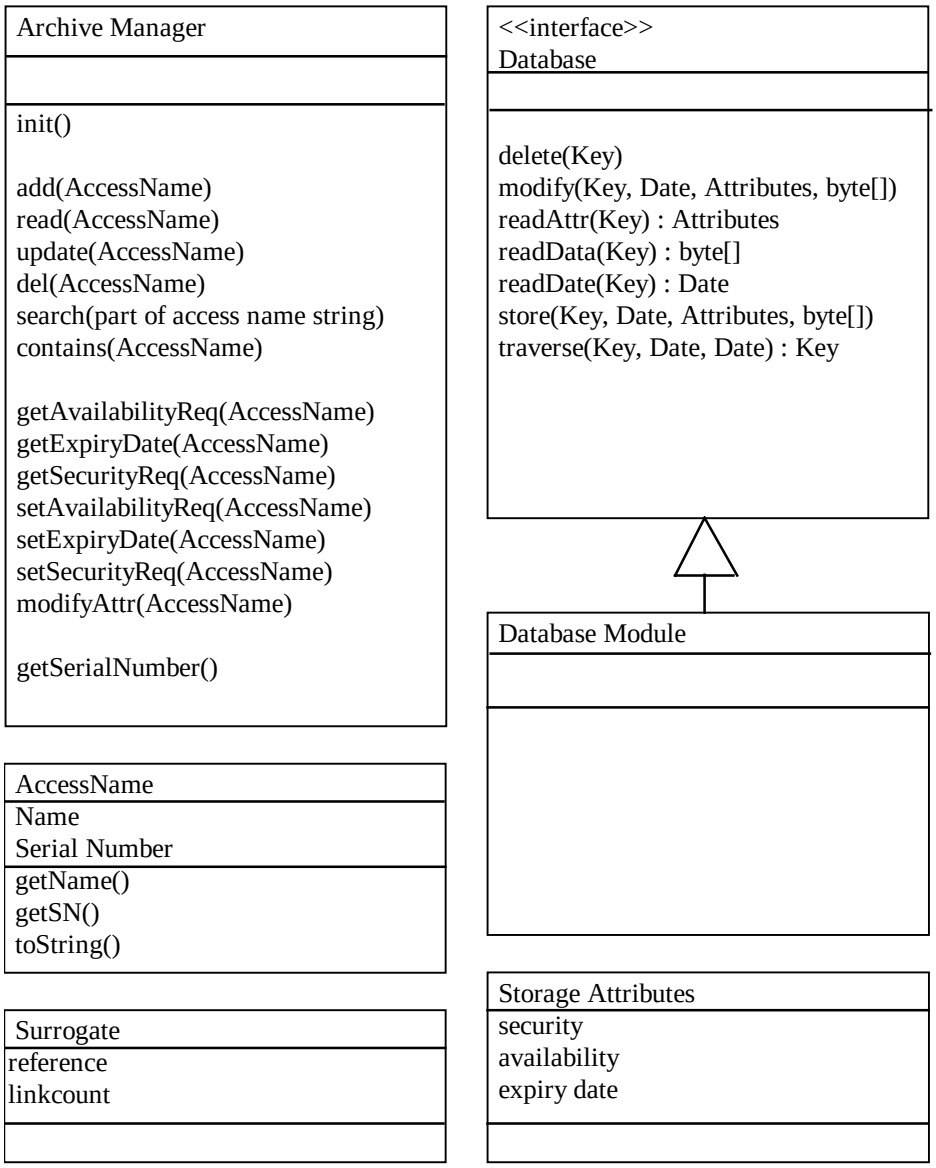

Figure 100: Classes for Archive service

4.14.3.2.1 ArchiveMan

This is the entry point of the archive service. Objects from all classes can be archived via the archive manager provided they implement the `Serializable` interface or they are standard Java classes; they are stored into the database under a serialized form (bytes stream). Handling of archived objects is done trough a unique key which is an instantiation of the class AccessName (see below).

The Archive manager uses a capability based access control with the help of the Access Control block. The capability required for the use of the archive services is fixed according to the security requirement associated with each stored object. Possible capabilities in the frame of the archive are passphrase, encryption/decryption

key, or none. All the services are performed according to the current role's capability. Hereunder the public methods:

Init

Initialises the manager to allow the archive services to be processed. At the very first init, it creates all the files needed by the archive.

add

Adds the given object identified by its access name to the archive. Sets the date of storage attribute. The object is stored according to the security requirement given as input parameter.

read

Searches in the archive for an object identified by its access name and returns the handle of this object.

update

Searches in the database for an object identified by its access name, removes it, and replaces it by the new object with the same access name. The time of storage attribute is updated according the current date.

del

Removes from the database an object characterised by its access name.

search

Searches for all objects identifiable by the given part of access name, and returns a corresponding set of access names.

The search may optionally be done according a time window, by reference to the time of storage attribute associated with the stored object.

contains

Indicates whether an object identified by its acces name is already stored in the archive or not.

getAvailabilityReq, getSecurityReq, getExpiryDate

Gets the value of the respective storage attributes associated with a stored object: security requirement, availability requirement, and expiry date.

setAvailabilityReq, setSecurityReq, setExpiryDate

Sets the value of the respective storage attributes associated with a stored object: security requirement, availability requirement, and expiry date.

modifyAttr

Updates the values of the attributes attached to the stored object identified by its access name in accordance with the given parameters. The object itself is not modified neither the time of storage attribute. If the security requirement attribute is modified, the way to store the data must be updated in accordance (encrypted or not).

getSerialNumber

A serial numbering service is provided for the use of the other blocks. This service gives on demand a number which is unique and can be used for purposes like naming of objects.

4.14.3.2.2 AccessName

This class provides handling to archived objects. Every archived object must be referenced by a meaningful name which is an access key from the point of view of the database. AcccessName currently includes two attributes:

name: a string implementing the meaningful name of the object. The syntax of the name is chosen by the users of the archive services.

serial_number: long integer optionally joined to the name.

The accessName objects are used as input parameters of the Archive manager services. An entity which submits an object for archival can create an access name with a serial number or without.

4.14.3.2.3 Attributes

For each archived object some associated storage attributes must be set at the storage time. Except the time of storage attribute, these attributes are input parameters of the add and modifyAttr Archive manager methods.

Security requirement indicates the requested level of security for the object. This information is used to perform access control using the capabilities and provide the security features.

Availability requirement indicates the requested level of availability for the object. This information is to be used to provide the availability features. Used by the back-up function.

Expiry Date gives the date on which the archive manager is allowed to remove the archived object by garbage collection.

Time of storage is associated with each stored data by the archive manager in order to allow further searches according to the date.

4.14.3.2.4 Surrogate

A surrogate table keeps the reference to memory instantiated stored objects and also the number of references. This avoids useless accesses to the database files and improves the response time. As there is exactly a single live persistence object it can be used as lock for atomic operation based on the Java synchronized methods.

4.14.3.2.5 Database interface

The database provides the basic functions needed by the archive services. It must be accessed only via the Archive manager.

The interface between the archive manager and the database module is given by the interface class Database which has the following methods: *store, delete, readDate, readAttr, readData, modify, traverse* (search). The parameters passed to these methods are:

key: the string shape of the access name.

partial key: a string representing the searched part of the access names.

date: the date of storage.

attr: the string shape of the storage attributes.

data: the string shape of the object to be stored, after serialisation.

startDate, endDate: dates used for definition of the time window of the search service.

4.14.3.2.6 Database module

In the current implementation DatabaseNoIndex is the class that implements the database module. This implementation is a light database manager with mechanisms as simple as possible. Data are stored into a single file, using Java.io.RandomAccessFile.

The storage model uses the following data structure:

Entry: (entry-size, record-size, **record**, gap)

Record: (key-size, **key**, date, attr-size, attr, data-size, **and data**) where key is the string form of the AccessName and data is the serialized object.

4.14.3.3 Functional View

Security

The secure storage provided by the archive manager covers the needs of confidentiality and integrity of the sensitive data. Depending on the kind of data to be stored it uses data encryption and, optionally, storage in a tamper-resistant device. Every entity submitting an object for archival must indicate the security level required for storage of that object. This security requirement is then fulfilled by the use of encryption algorithms or hash functions, using the services provided by the Crypto and Access Control blocks.

The integrity control consists in checking the content of the stored data in order to be sure that they have not been modified by an unauthorised entity. To do this, the archive uses the MAC services provided by the Crypto block.

Back-up

In order to increase the availability, selective back-ups can be performed within the archive. These back-ups are more selective than general back up the user could do on his own initiative using the operating system of his computer.

The archive back-ups are done on user's request via the TINGUIN. The user can ask for a selective back up: a list of items is proposed for him to choose (e.g. waste files). A back-up automatic proposal can also be done to inform (or remind) the user that some important data should be saved apart.

The peripherals on which back-ups can be done are specified within the archive user's preferences and can be selected interactively by the user when asking for the back up.

Garbage collection

This provides a cleaning of the archive database as a function of the expiry date associated with every stored data. The expiry date of an object within the archive is set by the entity that submits that object for archival.

Out-of-date data are not definitely removed from the database but copied in waste files.

Three cases are foreseen for the garbage collection. They must be selected within the archive user's preferences:

- automatic garbage collection,
- automatic garbage collection with confirmation by the user,
- garbage collection on demand, only on user's request.

4.14.4 Related Work

When the design of the *SEMPER* archive was started, no object oriented database all written in Java was available, including in the public domain. Some are now available such as Jeevan (W3apps).

We intentionally do not mention JDBC here since it is only a gateway to relational databases and is not inheritantly object oriented.

4.14.5 Self assessment

Several aspects of the design can be improved. Generally the design of the Archive block should be more object oriented, particularly concerning the object handling (see ArchiveMan and AccessName).

A known weakness of the archive is its slowness. This appears especially when handling complex big objects; in this situation the Java Serialization becomes very time-costly. Some possible improvements might also be done about the object indexing into the base.

4.14.6 Implementation notes and Recommendations

The current implementation of the archive block has the following differences from the design described in this document.

- Access control is not integrated,
- Integrity control is not fully implemented,
- Garbage collection is not implemented,
- Multi-user support : as access control is not implemented this is not really supported,
- Archive storage in tamper-resistant devices has not been implemented.

It should also be noted that confidentiality is currently brought by straightforward encryption under the master key, what could be improved by the use of secondary keys.

Another point is the archive protection against hardware failures; this is currently only based on time window reduction and high priority setting for every file input/output. For a full control of accidental base corruption one might think to a mirrored base implementation.

5 Protocols

5.1 Fair exchange

(M. Schunter / UDO)

5.1.1 Introduction and Overview

Everywhere in electronic commerce the issue arises that two parties need to exchange two items fairly, like on a physical counter.

Imagine Bob has requested consultancy services from Alice, e.g., a piece of software, a translation or a legal expertise. Alice wants to deliver to Bob a file containing the report. The file represents work of several person-months, so Alice wants a receipt if Bob receives the file. Bob, on the other hand, only wants to issue a receipt if it received the file.

Our solution to this problem are so-called fair exchange protocols: For a fair exchange, each inputs what it wants to give and what it expects in exchange. I.e., Alice wants the report for a receipt whereas Bob wants a receipt for the report. A simple solution to this problem would be if Alice sends the report and Bob sends the receipt. This solution, however, does not guarantee fairness: If Bob does not send the receipt, Alice sent the report while not obtaining the expected receipt. Therefore, in order to avoid such situations, we will describe protocols which *guarantee* fairness in any case, i.e., either both receive what they expect or else, no one gets even part of the expected item.

Besides the delivery of valuable information for a receipt (called *certified mail*) or for a payment (called *fair purchase*), there are many more types of fair exchanges. On an abstract level, any two items (i.e., signatures, data, or payments) can be exchanged for each other. This requires nine different well-known fair exchange protocols.

So why developing *generic* fair exchange?

The reason is that this abstract view does not hold in practice: In practice, differentiating between payments, signatures, and data is not sufficient to guarantee fairness. A fair exchange protocol for payment for receipt may work with one payment scheme but not with another. So instead of having at most nine different protocols, each new implementation of a good may require new fair exchange protocols. Furthermore, exchanging parcels of goods would require specific fair exchange protocols for and fixed combination of goods to be exchanged. Therefore, for a given number of n different modules with electronic goods this leads to n^2 different fair exchange protocols if one only wants to exchange one item for another. Furthermore, adding a new item to be exchanged (such as a new payment module) means adding another $n+1$ fair exchange protocols.

The solution to this problem is to provide exchanges, which are independent of the goods. We do this by implementing fair exchanges using so-called transfers of the goods (Figure 101).

A *transfer* sends an item from a sender to a recipient. Examples of transfers are payments (transfers value), signatures (transfers a signed document), or messages (transfers data). Furthermore, different items can be bundled into parcels of electronic items that we call containers.

A *fair exchange*, on the other hand, is then implemented as two virtually parallel transfers of two items satisfying the fixed expectations of both exchanging parties. Compared to just two transfers, the focus lies on the ‘‘atomic parallelism” which two subsequent transfers cannot guarantee: One party (the first one to receive a complete transfer) always has an advantage if one cannot reverse the sending of valuable information or signatures.

In order to guarantee this ‘‘atomic parallelism”, we require that these transfers are required to guarantee certain so-called exchange-enabling properties. Furthermore, in case of faults, we involve an additional third party into our protocols, i.e., if both players are correct, the third party is not actively involved.

As we will show, there are three exchange-enabling properties of transfers that are sufficient to guarantee fairness. For the generic fair exchange protocols described in this chapter, the underlying transfers of the two items to be exchanged must enable certain combinations of the following properties:

External Verifiability: A third party is enabled to observe a transfer, i.e., find out whether it succeeded or not.

A simple example for providing verifiability is sending a signature to the third party that verifies it and forwards it.

Generateability: The parties first fix the item in a preparation protocol. If the transfer fails, the third party is then enabled to ‘‘redo” the transfer using the information stored during the preparation.

A simple example for providing generateability for signatures is to authorise the third party to sign one one’s behalf.

Revocability: After the item has been fixed, revocability enables the third party to revoke a transfer.

A simple example for providing revocability is to enable the third party to revoke a credit-card payment.

Based on these properties, we are able to exchange any two items with only six different generic fair exchange protocols. With this approach exchanging parcels of goods is also simplified: The transfer-based exchanges do no longer differentiate between goods and parcels of goods as long as their transfers provide the exchange-enabling properties. Furthermore, adding new modules is considerably easier: Now, the provider of the new module is only required to provide one or more of the exchange-enabled properties in order to enable fair exchange of the items.

In principle, the protocols for fair exchange all follow the same pattern: First, both exchangees sign an agreement what items will be exchanged. Then, they just send their items using the underlying transfer protocols. After successfully receiving the transfers, the fair exchange protocol ends. However, if something goes wrong, the exchangees may ask the third party to restore fairness. How this is done then depends on the properties of the items. Some examples are:

- If one of the items is generateable and was not sent, this item is replaced by the third party.
- If one of the items is revocable and the other item was not sent, this item is revoked.

Thus, the third party is able to restore fairness if the items supported the assumed combination of properties. In the contract signing example mentioned above, this means that both digitally sign that they want to sign a contract. If these letters of understanding have been exchanged, both send their signatures under the contract. However, if one of the signatories receives no signature from the peer, it asks the third party to sign an affidavit on behalf of the party not sending its signature.

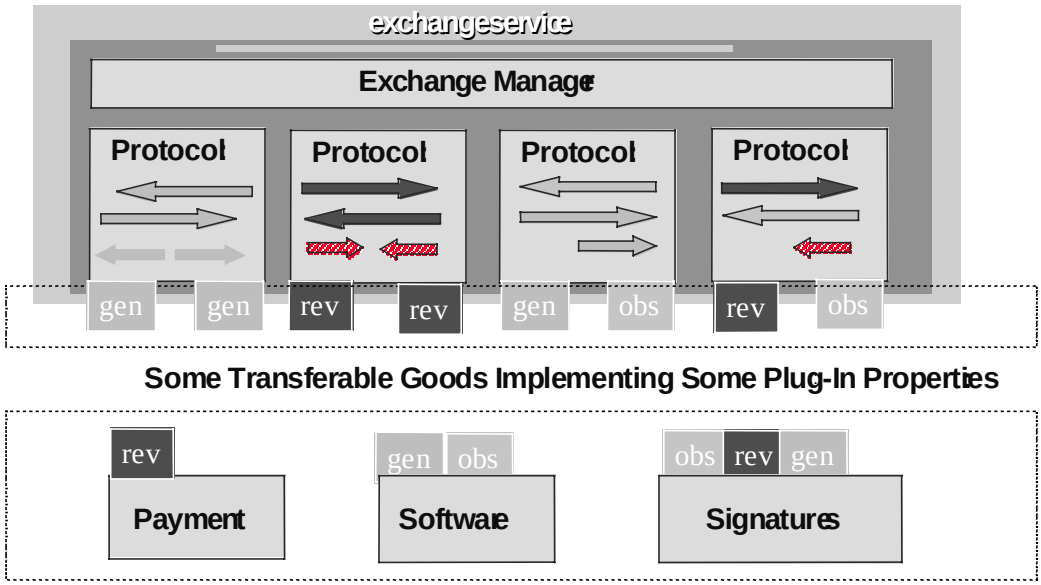

Figure 101: Selecting Exchange Protocols by Properties of the Goods[24].

5.1.2 Notations and Assumptions

This chapter deals with protocols independent from the actual implementation. A protocol takes some input and produces some outputs. In an implementation, this will be modelled as parameters and return values.

The exchange protocols described later assume that the underlying transfer protocols take an input *transfer(description)* and produce either an output *transferred(description)* at the recipient or else *aborted* if the transfer failed. For fairness, we assume that a transfer occurred if an output *transferred(description)* was made to a correct recipient and that no transfer takes place if a correct sender did not input *transferred(description)* to the underlying transfer protocol.

24 In practice, each type of good may provide any of the properties.

5.1.3 Transfer-Properties enabling Fair Exchange

Before describing the actual exchanges, we describe the properties of the underlying transfers, which are required to guarantee fairness of the exchanges.

Note that each good to be transferred is represented by its description. I.e., descriptions of the goods are input and output at the interface instead of the actual goods. The reason for this model is that most goods (i.e., payments, rights, and signatures) are transferred by protocols rather than data and it should be clear that a user cannot understand these protocols but rather only describes them.

The same holds for exchanges: In principle, both parties input a good to be sent (described by a container) as well as a description of the container to be received.

5.1.3.1 External Verifiability

External verifiability allows the third party to observe the transfer, i.e., listen into the transfer and verify whether it was successful or not. One possibility to achieve external verifiability on synchronous networks are receipts: By sending a receipt, the sender can convince the third party that a transfer was successful. Another possibility is if an honest recipient is able to prove to the third party if it was not able to use a good.

We now describe the notion of external verifiability in more detail. The scenario underlying this definition is depicted in Figure 102: The sender first prepares the container (a parcel of electronic goods) to be sent whereas the recipient checks that the description is correct. Then, the sender sends the container. Finally, the third party is able to verify whether a transfer was successful and, if this is the case, what item has been transferred.

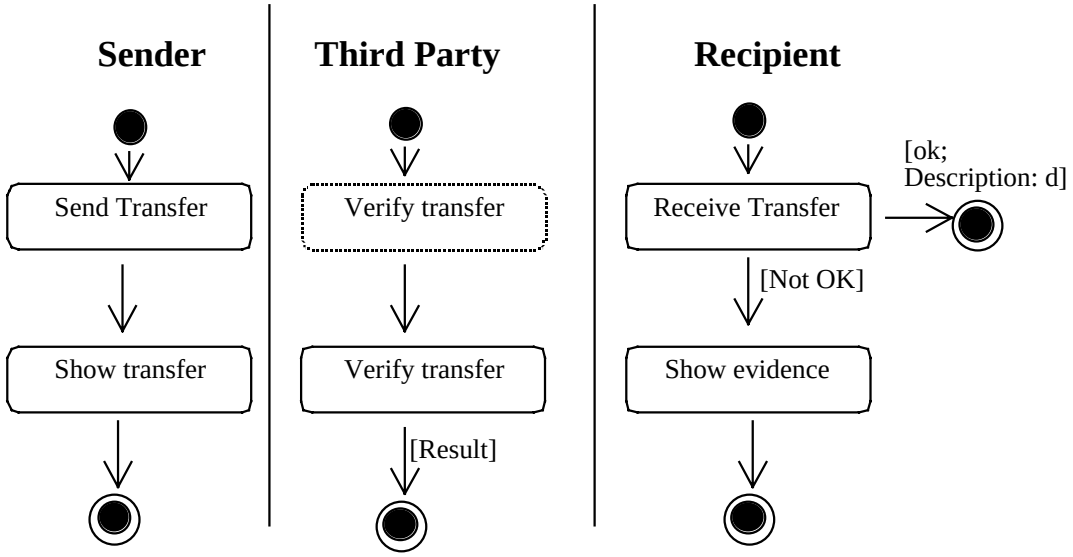

Figure 102: Activities for External Verifiability

More formally, the transfer enables one interactive protocol “verify” in addition to the pure transfer:

> ***Verify***: This protocol is started by the third party on input *verify(Description desc)* and enables the third party to verify if a transfer happened. If a transfer happened, it outputs *verified()* to the third party and else, it outputs *aborted.*

Note that if the third party does not participate in the transfer, the information needed for verification is stored at the sender, the recipient, or at additional participants such as a bank. Where the verify protocol fulfils the following requirements after a successful execution of prepare:

1. *Correct Verification:* If a correct sender sends an item with description *desc*, then the verification protocol outputs *success* after a verification request by the third-party with the same description.
2. *Correct Abort:* If a correct recipient did not send an item with description *desc*, then the verification protocol outputs *aborted* to the third party.
3. *Termination:* The verification protocol eventually outputs a result to the third party.

Note that most goods can be adopted to provide this external verifiability:

- Public-key payment systems can be made externally verifiable since the third party is able to verify the signatures and ask the bank whether the coins have been deposited.
- Idempotent goods can be sent via the third party,
- Ordinary signatures as well as fail-stop signatures are externally verifiable [Pfitzm96].

Another way to add external verifiability is to map existing weaker properties like idempotency or receipts onto external verifiability.

5.1.3.2 Generateability

An electronic good provides generateability if the third party is able to produce an equivalent replacement without co-operation of the sender. A common example is a so-called affidavit, i.e., a replacement signature from a notary who signs on behalf of an unwilling or absent signer.

Formally, generateability of a transferable good is provided by two additional protocols: A *prepare*–protocol is used to fix and agree on the good without transferring it. The usual *transfer*–protocol may then be executed to transfer the good. If something goes wrong, the third party may execute a *generate*–protocol to ensure that the recipient really received the good. The activities are depicted in Figure 103. Note that only activities with in- and outputs are depicted even though, e.g., the third party may participate in the transfer and prepare protocols, too.

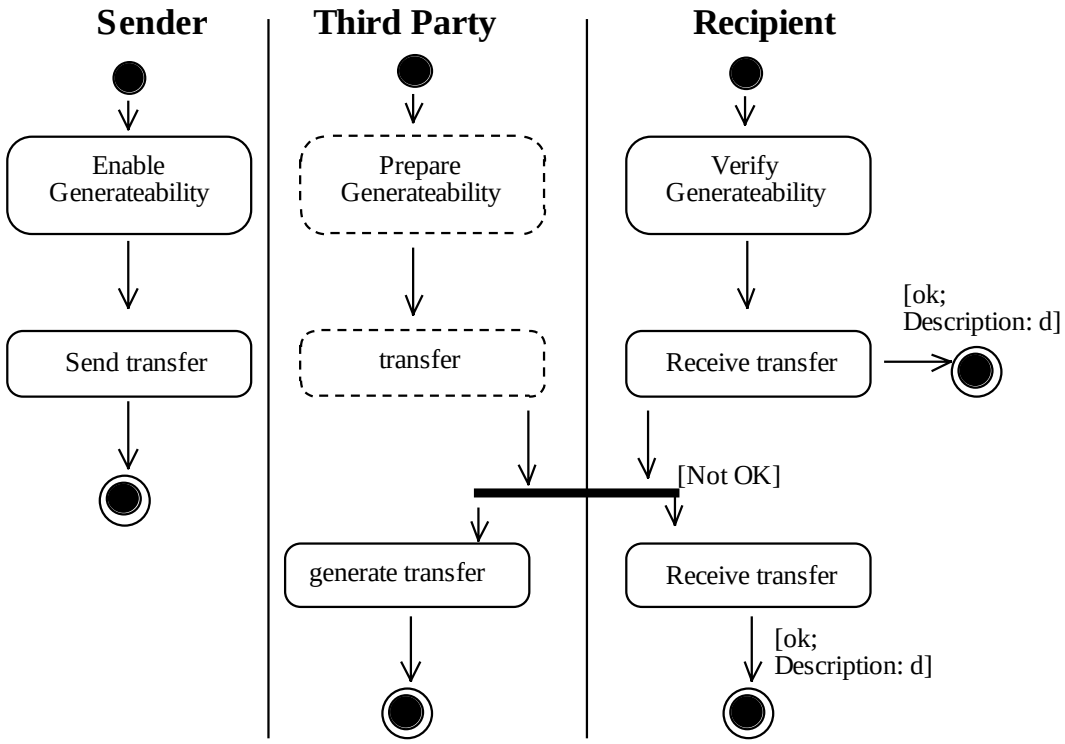

Figure 103: Activities for Generateability by Roles.

Formally, the additional protocols for generateability are:

Prepare: In the *prepare*-protocol, the sender calls a method *prepare(Description desc, Properties prop)*, the recipient calls *verify(Description desc, Property Generateable)* where *desc* is the description of the good to be transferred and *Generateable* is the property to be verified.

Generate: This protocol is executed between the third party and the recipient. It is started by the third party on input *generate(Description desc)* and enables the recipient to receive the good as usual. Note that the description input by the third party is used for verification only: If the information shown by the recipient contains a different description, the protocol aborts.

The protocols must fulfil the following additional requirements:

1. *Correct Execution*: If all players are honest the prepare protocol outputs *verified* if and only if the input descriptions are identical and the good provides the property which was verified.
2. *Termination*: The protocols eventually terminate with a correct output as defined above.
3. *Generateability:* If the third-party and the recipient are honest and the recipient received an output *verified* on input *verify(desc, Generateable)*, a subsequent input *generate(desc)* by the third-party will eventually lead to the recipient receiving the good described by *desc*.

5.1.3.3 Revocability

Revocable goods can be made unusable by a third party after a recipient has received them, i.e., the information from the *prepare*–protocol enables the third-party to render a good unusable after a successful transfer. Examples are blacklisting of spent coins or revocation of a credit-card payments. The scenario is depicted in Figure 104.

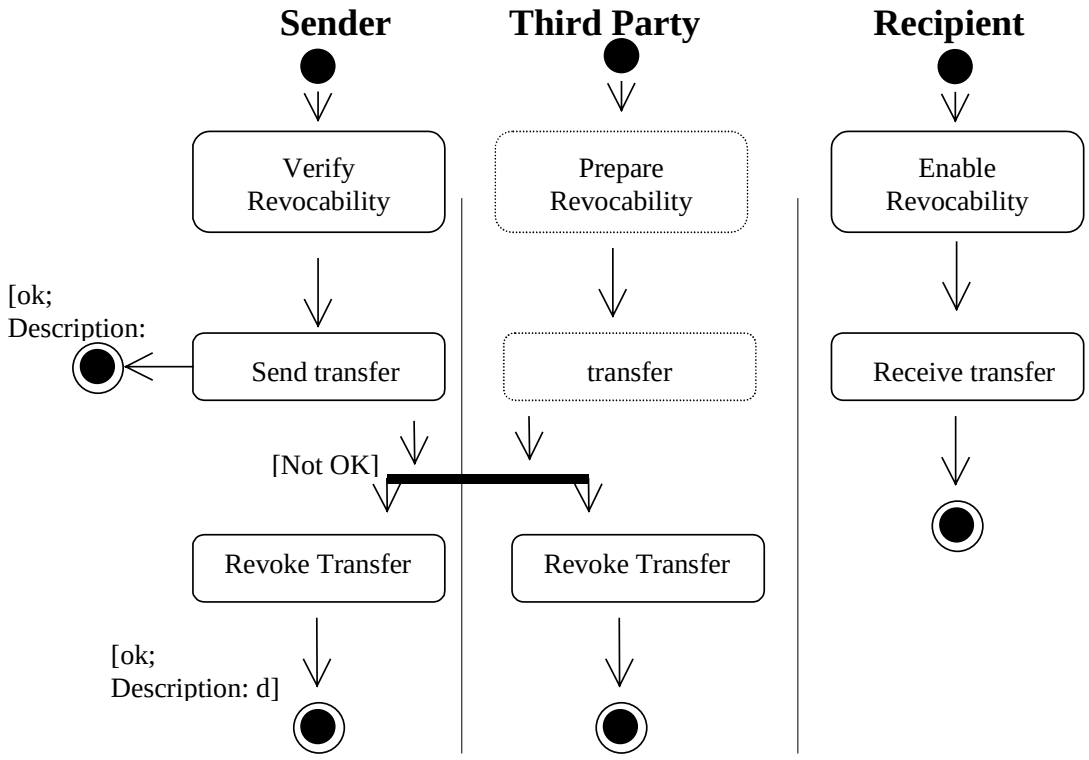

Figure 104: Activities for Revocability.

Formally, the additional protocols for revocability are:

Prepare: In the *prepare*-protocol, the sender inputs *prepare(Description desc, Properties prop)*, the recipient inputs *verify(Description desc, Properties Revocability)* where *desc* is the description of the good to be transferred and *Revocability* is the property to be verified.

Revoke: This protocol is started by the sender and the third party on input *revoke(Description desc)* and prevents the recipient from using the good.

The protocols must fulfil the following additional requirements:

1. *Correct Execution*: If all players are honest the prepare protocol outputs *verified* if and only if the input descriptions are identical and the good provides the property which was verified.
2. **Revocation***:* If *verified* was output to a correct sender on input *requestRevoke()* and a correct third-party inputs *revoke(Description desc)*, then no recipient is able to obtain the good use during prepare.

5.1.3.4 Deriving Properties of Containers

If one transfer transfers multiple goods, the transfer is generateable if all goods are generateable. The transfer is revocable if all goods are revocable. Furthermore, it is externally verifiable if all goods are externally verifiable. The container transfer should eventually implement this.

5.1.3.5 Optimistic Protocols and Optimistic Properties

In [AsScWa97,AsScWa98], so-called optimistic exchange protocols have been described. An optimistic exchange protocol only needs a third party in case of exceptions to restore fairness.

A natural question is, how our properties relate to these optimistic exchange protocols. The answer to this question is that they are orthogonal: For each property there exists an optimistic version, too. The basic distinction between “optimistic” and “basic” versions of the properties is that a property is optimistic, iff the third party does not participate in the *prepare* and *transfer* protocols. Else, if it participates in all protocols, it is not optimistic.

For the exchange protocols described later, however, this makes no difference: The generic exchanges guarantee fairness. If the properties are implemented in an optimistic way, the resulting fair exchange protocol is optimistic. If not, the resulting exchange protocol will not be optimistic.

5.1.4 Exchange Protocols based on Transfers

We now sketch three simple exchange protocols based on transfers with the properties described above. Each of these protocols follows the same pattern: The parties exchange their goods by means of two subsequent transfers and, if something goes wrong, the third-party restores fairness.

5.1.4.1 Externally Verifiable and Generateable Goods

We now describe an exchange protocol similar to the protocol described in [AsScWa97]. It assumes that one of the goods offers optimistic external verifiability whereas the other good offers optimistic generateability. The basic idea is that the participants first prepare the generateable good and then the externally verifiable good is sent. Finally, the generateable good is sent. If the generateable good is not sent, the third party is invoked to generate it after verifying that the externally verifiable good was really transferred. The behavior of the scheme in the fault-less case is sketched in Figure 105. The machines are depicted in Figure 106, Figure 107, and Figure 108.

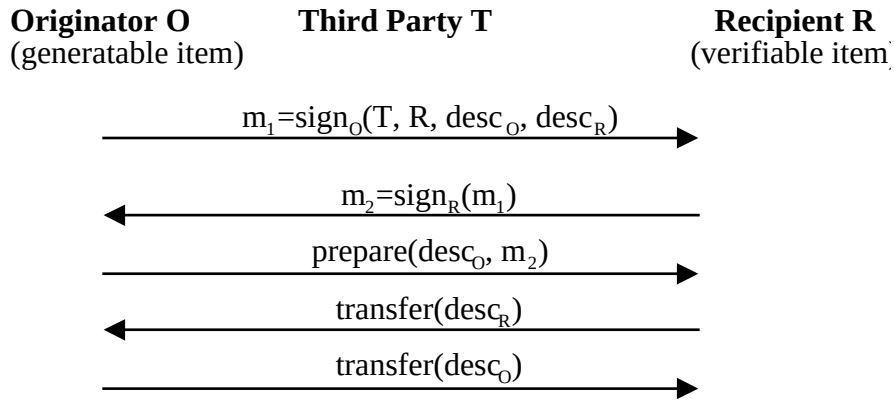

Figure 105: Exchanging Externally Verifiable and Generateable Goods[25].

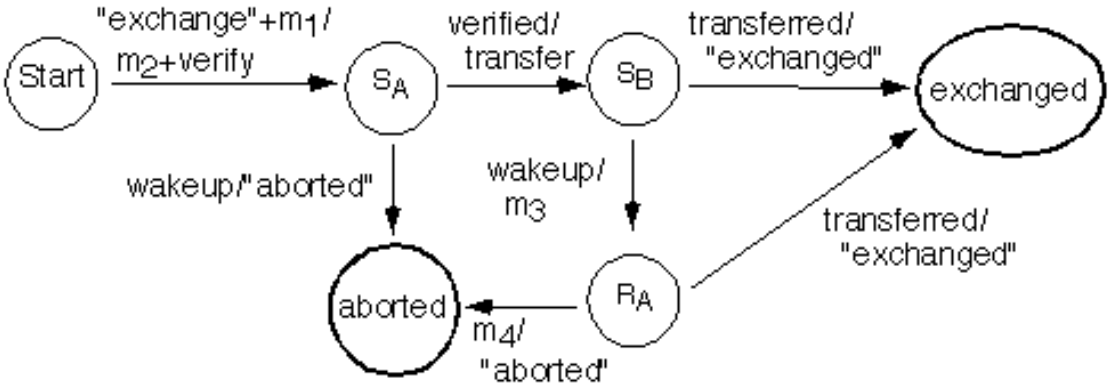

Figure 106: Optimistic Exchange Protocol; Player R

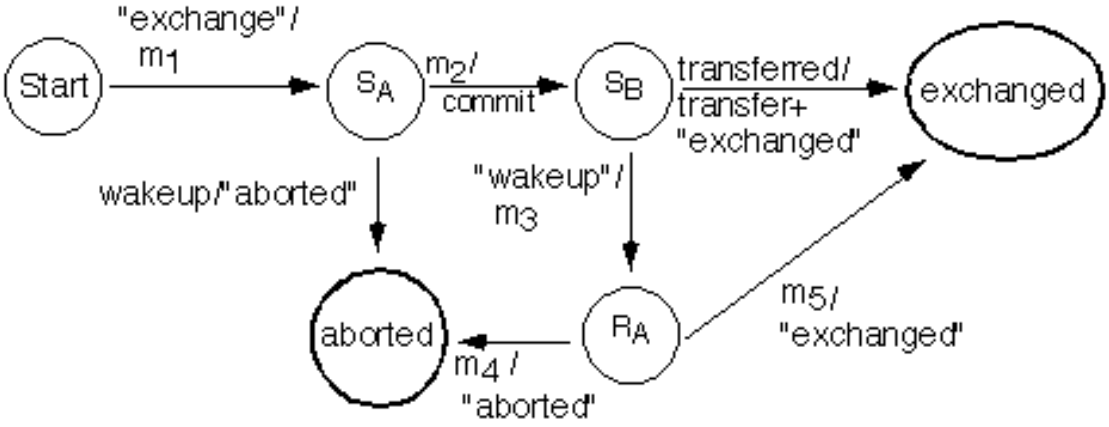

Figure 107: Optimistic Exchange Protocol; Player O

25 Only the fault-less case is depicted. Recovery including messages m_3 and m_4 is described by the automatas.

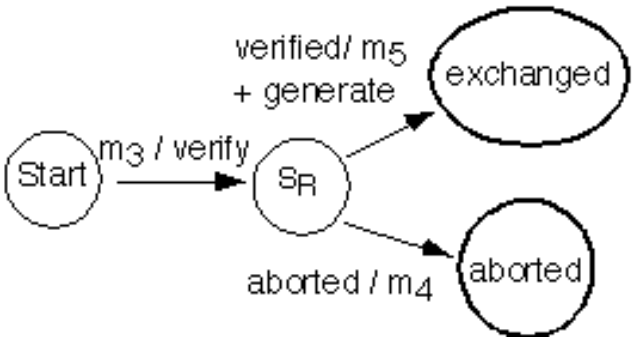

Figure 108: Optimistic Exchange Protocol; Third Party T

We now describe these protocols in more detail:

Exchange: On input *exchange(desc$_O$, desc$_R$)* to the originator *O* and *exchange(desc$_R$, desc$_O$)* to the responder R, the originator sends a signed message m_1=sign$_O$(T, R, desc$_O$, desc$_R$) with the parameters of the exchange to the responder. The responder then sends a message m_2=sign$_R$(m_1) to the originator and checks whether the expectations match. If not, the responder outputs *aborted* after sending m_2. If both agree, the originator starts the preparation protocol by inputting *prepare(*desc$_R$) to the underlying transfer. Else, it outputs *aborted*. The preparation is then verified by the responder by inputting *verify*. If the verification was successful, i.e., *verified(*desc$_O$) was output, the responder transfers its good by inputting *transfer(desc$_R$).* When the originator receives the output *transferred*(desc$_R$), it starts its transfer by inputting *transfer(desc$_O$).* and outputs *exchanged().* On output *transferred*(desc$_O$), the responder outputs *exchanged*(desc$_O$). If player O does not get an output *transferred*, it starts recovery with the third party.

Recovery by the Responder: This recovery protocol is executed to generate a good if the responder sent its good but did not receive the expected transfer. The responder sends a signed request *m_3=sign$_R$(generate, m_1)* to the third party in order to prove the deal the exchangees made. If the third party is not in the *start*–state, it aborts. Else, the third-party checks whether the responder sent its good to the recipient relying on the optimistic external verifiability, i.e., the third-party inputs *verify(desc$_O$)*. If the verification outputs *verified* to the third-party, the third-party generates the missing item by inputting *generate(desc$_R$)* and sends a message *m_5=sign$_T$(continue, m_3)* to O. If the responder receives an output *transferred*, it outputs *exchanged*. If the verification outputs *aborted* to the third party, the third party sends a message *m_5=sign$_T$(aborted, m_3)* to the recipient and the recipient outputs aborted, too.

Recovery by the Originator: This recovery protocol is executed by the originator to abort the exchange if it received an abort request before receiving a transfer. After an abort request, the originator sends a message *m_3=sign$_R$(abort, m_2)* to the third party. The third party then inputs *verify(desc$_R$)* to the external verification protocol of the transferred goods received by *O*. If the verification outputs *verified*, it sends *m_5=sign$_T$(continue, m_3)* to the originator and tries to generate the good and changes to the *exchanged*–state. Note that this should only be the case if the preparation protocol and the transfer were successful and *O* was not yet notified. Else, if the verification outputs *aborted*, the third party sends *m_4=sign$_T$(aborted, m_3)* and changes to the aborted state. On receiving *m_4*, *O* outputs *aborted* and changes to the aborted state, too.

5.1.4.2 Exchanging Other Goods

The exchange protocol described above was able to exchange externally verifiable for generateable goods (Figure 105).

The same pattern can easily be transferred to exchanges where one good is revocable and the other is externally verifiable. For this kind of exchange (Figure 109), recovery is done by revoking the good which was sent first. For exchanges where both goods are either revocable (Figure 110), recovery is done by revoking both goods if someone complains, and if both are (Figure 111), recovery is done by generating both if someone complains.

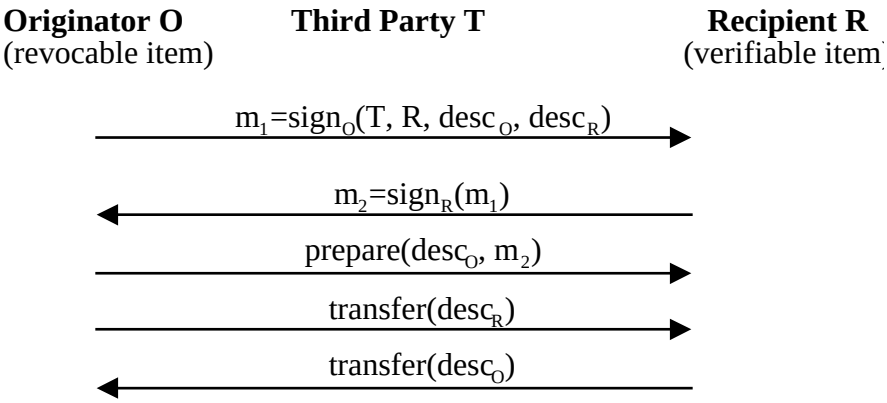

Figure 109: Exchanging Externally Verifiable and Revocable Goods

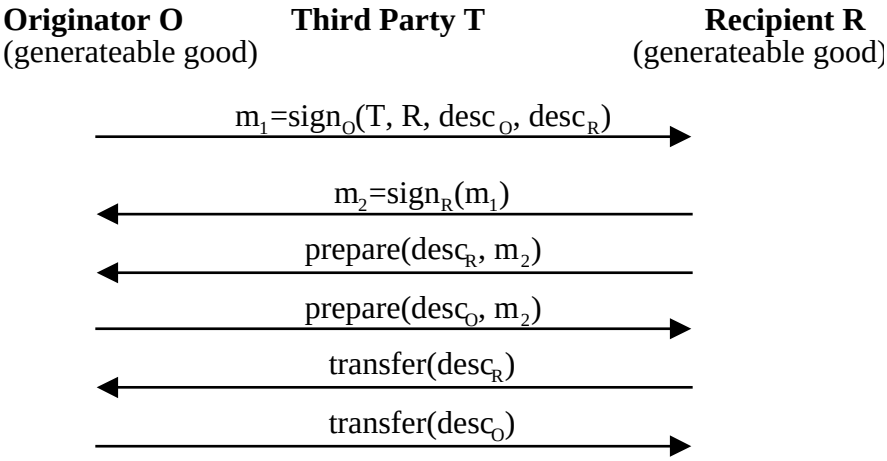

Figure 110: Exchanging Generateable Goods

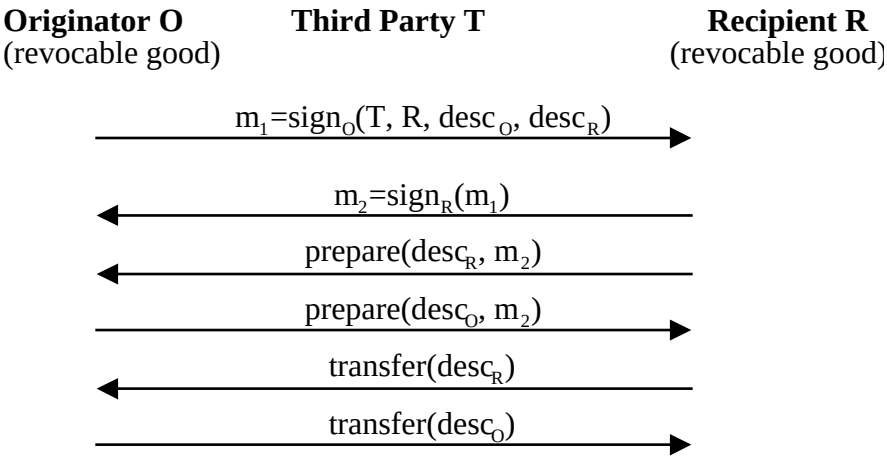

Figure 111: Exchanging Revocable Goods

5.2 Registration

(K. Zangeneh / GMD)

5.2.1 Introduction

An essential problem of committed telecooperation over an open and insecure network like the Internet is authentication of individual users. Each communication partner needs trustworthy information about the authenticity of his/her remote partner. An electronic marketplace depicts a typical form of committed telecooperation in which buyers and sellers are, on the one hand connected to each other through a network, and on the other hand separated through the applications of the network and also because of the geographical distance between them. The remote communication partners neither see nor hear each other, although they are motivated to be involved in an electronic business transaction and play their role telecooperatively. They communicate by means of digital data which is prone to forging, loss or manipulation. Consequently, they need some techniques and methods that provide the opportunity to authenticate a remote communication partner.

User registration and certification is required in order to link electronic actions to physical organizations and individuals, which is the basis of all kinds of electronic commerce.

5.2.2 Purpose of Registration and Certification

A public key cryptography scheme offers the means for providing security services like authentication of communicating partners, integrity of communication data, non-repudiation of the origin/receipt of digital information, privacy, anonymity, fair exchange, confidentiality, provability, etc. Public key mechanisms in open environments create the new problem of the reliable binding of a public key to its authentic owner in a widely distributed network of users. Certificates are the best technology available to perform secure authentication across public networks like the Internet. They combine the ease of use of password-based access control with the sophistication and security of public key cryptography.

Another problem caused through deployment of open network environments is how to bind a physical individual, to his/her name and make this binding digitally provable.

We distinguish between the binding of a person to his/her name and the binding of a person to his/her cryptographic public key and/or his/her capabilities. Consequently, secure communication on the basis of cryptography relies on an authentic binding of following entities: a social person, his/her name, his/her cryptographic public key, the capabilities (what may a person do with his/her authenticated public key).

The goal of *User Registration* is to bind a person to his/her name. A user is registered on the basis of his/her credentials, e.g. physical appearance, name, passport, postal address, public key, phone number, email, bank account, etc.

The goal of *User Certification* is to associate a user name to his/her public key. A certificate is a piece of digital data which includes a user name, the name of a

certification authority (CA), the public key of the user, etc. and which is signed digitally by the CA. A certificate is evidence for personal ownership of a public key to a certain person. The responsibility for issuing a certificate lies with the certification authority which confirms the relationship between the public key and the person with a digitally signed certificate.

To each public key, there is a corresponding secret key. In every committed electronic business transaction, clients and merchants have to commit their offers and orders for instance, by signing them digitally. Digital signatures are made using secret keys. Verification of digital signatures uses the corresponding user's public key. A certificate guarantees that a public key belongs to a certain person. In this way, a digitally signed document could be verified using a user's public key which is certified in a certificate that could be send along with the document or could be requested from public (trustworthy) Directory servers.

The usage of certificates is a fundamental need in an electronic marketplace in which remote partners have possibly never met each other.

5.2.3 Trusted Third Parties

Concerning the business transactions, the role of trusted third parties is essential and significant. Two remote partners who possibly never met before attempt to settle a business relationship without knowing and necessarily trusting each other. Trusted third parties support creating indispensable trust among all parties involved in a digital business transaction over open networks like the Internet. Trusted third parties themselves are highly trusted, of course. They are comparable to trusted public organizations like a police department.

There are various categories of trusted third parties; the most important ones are described briefly here.

- *Registration Authority,* a registration authority (RA) is a trusted third party which is responsible for the registration process, that is for linking a physical person to his/her name. In particular, the registration authority is responsible for defining a distinguished name (unique identifier) for a user.
- *Certification Authority,* a certification authority (CA) is a trusted third party and is responsible for issuing certificates, i.e. associate a person with his/her public key. A certification authority receives the registration already issued by the registration authority and issues a certificate based on the information of registration and information from the user. A certification authority is responsible for extending, changing, renewing and revoking certificates too.
- *Directory Authority,* a directory authority (DA) is an another trusted third party which offers a database of certificates. In particular, there are statements whether a certificate is valid or revoked. A Directory authority is responsible for maintenance of the database and response of queries to the database.

5.2.4 Types of Registration

According to the type of business transaction various levels of trust among parties are required. For high secure data transfer or payments of large amounts of money, a

higher level of trust is necessary. Generally one can say, the more sensitive the transaction is the more trust is required. The level of registration corresponds to the trust level it reflects. In the following, the categories of registration types are briefly outlined:

- *No Registration*, no registration procedure should take place at all.
- *Automatic server process with policy (e.g. check table entries)*, a server process runs permanently at a registration authority site. The registration authority and CA have implemented a registration and certification policy. Issuance of registrations and certificates depends on this policy. For instance, a table with entries of privileged customers is an implementation of a very simple policy.
- *Non automatic server process with registration authority's interactivity*, in this case, the registration authority is interactive and checks some information out-of-line. An example could be checking the fingerprint of a customer's public key. Another example is checking of a shared secure information between a customer and the registration authority, viz. a password.
- *Letter Registration*, a customer obtains a letter from the registration authority which has to be filled out and signed. On this basis an electronic registration could be performed later.
- *Notary Registration*, notary registration is proper and necessary for sensitive business transactions in an electronic marketplace. In this case a customer obtains his/her registration after s/he appears at the registration authority, identifies him/herself, fills out the registration form and signs it.

To limit the risk of the customers as well as service providers, all issued certificates should be accompanied by an attribute stating the method of registration and capabilities stating what a particular customer is permitted to do with his/her certificate or public key.

5.2.5 Types of Certificates

In SEMPER, we distinguish four types of certificates:

- *Public key certificates*, here, the statement is only that a particular public key belongs to a particular identity (distinguished name). For instance, the well-known X.509 certificates are of this type.
- *Attributes certificates*, these certify attributes of either the person or of a certain key (identified by a certificate identifier) of this person, e.g., that the person is willing to accept legal responsibility for signatures made with this public key in certain transactions. These certificates may also be used to give a proxy to a user.
- *Hybrid certificates*, these certificates contain both a public key and attributes, typically a description of what this public key can be used for. They are mainly intended for service providers issuing certificates particularly for their own service.
- *SECA certificates*, based on the SEMPER Electronic Commerce Agreement (SECA), SECA certificates could be requested from an appropriate certification authority. A SECA certificate contains statements to prove buyer's and merchant's liability, to prove credit guarantee of buyers and to limit the buyer's liability per

transaction. With the help of a SECA certificate for a certain business transaction, a guarantee is given to the merchant that the buyer's liability has not been exceeded, it is a certificate which guarantees the buyer's credit-worthiness up to a limit according to the current transaction.

5.2.6 User Registration/Certification in SEMPER

SEMPER as a framework for an open secure electronic marketplace provides services for registration and certification. Directory services are supported as well.

The steps for registering and certifying in SEMPER are described as follow. The registration and certification should be performed when the proper registration and certification authorities do exist and a global sophisticated infrastructure has been established across nations and continents.

Within a two-step procedure, a potential customer obtains his/her registration and certificate. After a user has generated a pair of keys, the user registers his public keys and credentials (e.g., name, surname, organization, city, id-card no., etc.) at the registration authority. The registration authority verifies all information supplied by the user with the help of his/her personal id-card, bank card, etc. In particular, the registration authority assigns a unique identifier, a so-called *distinguished name*, to the user. In the case the information is correct, the registration authority issues a registration and hands it over to the customer. One copy of the issued registration is sent to the certification authority.

In the second step, the certification authority issues a certificate corresponding to the registered information at the registration authority and sends it to the user. In this certificate, the certification authority confirms the association of the customer and his/her public key.

The registration authority is completely trusted by the certification authority. The communication between registration and certification authorities has to be secure and authentic; authenticity is needed because the origin of the message is essential for the certification authority. On the other hand the message should be sent from registration authority to the certification authority in a secure way so that the data integrity is provided, for example, by using digital signatures. In an advanced registration infrastructure, registration authority and certification authority branches or representatives are responsible for a certain geographical area.

After the two-step procedure, there is a certified relationship between a customer, his/her name and a public key. In an advanced version of certificates, X.509v3, some other credentials could be put in the certificate. For instance a statement regarding the usage of the certified public key could be contained. Every communication partner involved in an electronic marketplace or even in the whole network would be able to check and verify that a certain public key belongs to its owner.

For the time being there is no support of any registration authority in SEMPER therefore, registration and certification are done by one authority. The GMD, the German National Research Institute for Information Technology, plays the roles of both registration and certification authorities. The registration/certification application supports the "automatic server process with policy" version. We apply a pre-registration policy based on passwords as registration policy. Our general approach for

registration/certification in SEMPER is that the customer and registration/certification authority should be interactive at least in one step of the process, which is logically the most significant and meaningful step.

In an electronic registration procedure an appropriate mechanism should be integrated in the application which enables both client and server to authenticate themselves against the other communication partner. In other words, the authentic communication and identification of two parties must be guaranteed. The client and the server might make use of this mechanism to authenticate themselves to the other remote party. For this purpose, SEMPER pursues the idea of "password authentication".

Each user who intends to deploy SEMPER software should contact the service providers involving in trials in order to receive a so-called *registration letter*. Every registration letter contains among other things a *registration key* for the user. Another essential information in the registration letter is the *fingerprint* of the registration/certification authority.

The registration keys are collected by the service providers and are sent to the registration authority using a secure channel so that a table with all users and their registration keys is at the registration authority's disposal. With the help of this table the registration authority is able to check the users' authentication on-line.

On the other hand, it is extremely important for users to know that the registration authority is authentic. In other words, the registration authority should authenticate itself against users. We deploy a fingerprint-check mechanism for this purpose. As mentioned before, users receive a hard copy of the fingerprint (hash value) of RA's public key in a registration letter. Another possibility is receiving the fingerprint as a file from the service providers. During the registration procedure, a user receives the public key of the registration/certification authority. Its fingerprint is computed and shown on TINGUIN. The user should compare two fingerprints. Equality of two fingerprints assures the user that the registration/certification authority is authentic. In case the fingerprint is handed over as a file, another file which contains the fingerprint of a RA's public key is produced locally. The two files are compared and in case of equality the contacting RA is authenticated.

The service providers FOGRA, EUROCOM and OTTO Versand, and all companies involved in the trials, will provide GMD, as a unique RA/CA, with a set of customer credentials. Besides social statements, there is also a registration key assigned to individual customers. The customers receive their individual registration key with the SEMPER software. In the registration request, a customer has to send his/her confidential registration key to the registration authority (in this case, the GMD) along with some personal information. On the basis of this information, the GMD will be able to register customers and issue certificates for them afterwards.

Note that one owns two key pair in SEMPER, one key pair for signature and one for encryption purposes. Each key pair consists of one secret and one public key. Both public keys must be registered. Thus the whole registration procedure should be performed twice. In the first run, the signature-key is registered and in the second run, the encryption key is registered. In the first run, the certification application needs some information about the user, however in the second run no additional information is needed about the user, because the registration process can be performed based on the previous information.

5.2.7 Certification Authorities for SEMPER

The GMD, the German National Research Center for Information Technology, has taken the responsibility for running and maintaining the various certification authorities for SEMPER.

- *Certification Authority for SME-Trials*, this registration/certification authority supports registration of SME, small and medium enterprises, field-trial users and issues certificates for them. The list of pre-registered users is at this CA's disposal. The entries of pre-registered users in this list have been delivered from service providers to the CA.
- *Certification Authority for Supervised-trials*, this authority supports registration and certification of the users in supervised-trials. Similar to the above certification authority, this CA maintains a pre-registration table.
- *Certification Authority for supporting SECA certificates*, this is a certification authority for issuing key and SECA certificates. It demonstrates the usage of SECA certificates within SEMPER. Besides key certificates, this certification authority can be contacted for receiving SECA certificates.

5.2.8 Design of Registration/Certification Application block

Registration/Certification Application Block "certappl" is designed as a client-server-model.

At the client site, the certificate manager offers and manages all necessary functions for requesting and receiving registrations and certificates. An interface is defined between the certificate manager and the implementation of the underlying module. This is necessary for keeping the whole architecture as generic as possible. All client enquiries are passed through the RCModule Interface (registration certification module interface) to the real implementation of the module. The module implementation block deploys necessary functions from other SEMPER blocks, e.g. key generation and signature verification from the crypto block, etc.

The server site, i.e. the certification authority, is designed very similarly to the client site of the registration/certification application block. At this part of the block, the CAServer manages all activities related to client requests for receiving registrations and issuing/sending certificates to the SEMPER clients.

Figure 112 Shows the design model of the registration/certification application block.

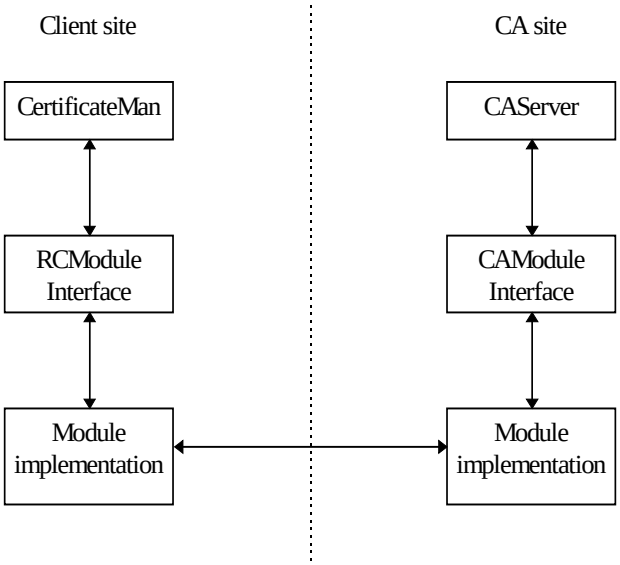

Figure 112 Model of the Registration/Certification Application

5.2.9 Short Overview of the Registration/Certification Protocol

Registration and certification has its own protocol. Through the registration and certification protocol, a SEMPER user is able to request and receive his/her certificates. For this purpose, there are a number of CA servers employed that are based on a similar protocol.

Note that, as mentioned before, for the time being there is no separate registration authority, so that the registration and certification authorities are unified in the same location/host.

In the following the registration and certification protocol is described.

SEMPER Client	Certification Authority
SEMPER software generates public and private key pairs for signature and encryption. Communication address of peer (certification authority) is established. Via the peer address, the public key of the certification authority (the so-called trusted public key) is retrieved. Client requests a registration template	
	Certification authority sends the

	requested template to the client including the RA's distinguished name and a unique serial number
SEMPER client gets the template, fills it in, and sends it back to the RA/CA.	
	The certification authority checks the authentication of the client by invoking a pre-registration check process using a list of all pre-registered clients. If the check authentication process is passed, the certification authority continues its task of issuing clients' registrations. Otherwise an error message is sent to the client. Once the client's registration is issued, it is stored locally. The registration is now sent to the client.
The client gets the registration and stores it at his/her SEMPER archive. After that the client sends a request for getting a certificate to the certification authority.	
	Based on the previous information, the certification authority issues a client certificate. After storing the certificate locally, the certificate is sent to the client.
The client gets the certificate, verifies it, and when the verification succeeds the newly incoming certificate is stored in the client's SEMPER archive.	

5.2.10 Steps for registration and certification

The registration procedure is divided into four phases. During the first phase some default values, like the login name, name of supported module(s), host name/address, etc. are set. In the second phase user's keys are generated and stored in the SEMPER archive. During the third phase, a user's signature key is processed. For that purpose, a user first registers at a registration authority by filling out a template, after sending a request for getting it. Once the registration is complete, the user's signature key is certified by the certification authority, which in our case is the same as the registration authority. In the fourth and last phase, the user's encryption key is processed.

Phase 1

At the beginning, the user is asked through the TINGUIN display to enter his *login name*. The login name is used to have a suffix for a user's sensitive data in the SEMPER's archive. It facilitates the software to keep this sensitive data, when more than one user uses the same machine, under distinguished names. If a user starts the application for the first time, his login name is retrieved from his environment and is set in the entry field. The user should fill out the field, if necessary, and press the OK bottom. A progress bar shows the progress of the work at each stage.

Now the user is asked to enter a string (of at least 16 bit) as a *random seed*. A random seed is used for the key generation process. It guarantees the randomness of data used for generating keys. The user does not need to remember the random seed, but he should be aware that the more characters he enters (randomly), the securer are the keys.

In the TINGUIN display, the user is then asked to enter a password for security protection purposes. All user's sensitive data will be stored securely with the usage of this password. If the user runs the registration/certification application for the first time, he will be asked to enter his password twice for avoiding typing mistakes. That way, he has chosen his password for protection his sensitive data. The user should be aware that he has to enter this password every time he runs the SEMPER application from now on. Therefore, he should take a note of it and keep it in his mind. The password he enters at this stage should be kept secure. The user MUST NOT make it public for security reasons.

In SEMPER, one might make use of the logging facility. The logging facility enables users to create a logging file on their disk for keeping track of what is happening in detail in the SEMPER software. For setting a logging file the "logging" item from the pop down menu "Preferences" in the TINGUIN menu bar is chosen. The SEMPER logging facility is configurable. Many tracking levels are defined. Users might configure their logging file, as they like, by choosing various values for configurable fields. Entering a name for logging file should not be forgotten.

Phase 2

To run the registration, the user has to choose the item "Registration" from the pop down menu "Applications" in the TINGUIN menu bar.

The key generation process is invoked now and the user is asked to choose the algorithm for generating his signature key pair. Two algorithms are supported currently, RSA and DSA. By pressing the radio button and clicking on OK button the user can choose the algorithm he wants to apply.

Now the user is asked to choose the length of the keys which are to be generated. The user has the choice between 512, 768 and 1024 bit as the length of his signature and encryption key pairs. The user should be aware of the fact that the length of a key corresponds to the security level this key possesses. Beside the security level of a key, the length of a key has an impact on the performance of the whole system. Encryption/decryption and signing/verification processes takes more time using a key of 1024 bit length, than using a key of 512 bit length. The user will be asked to choose the security level for his encryption key which is going to be generated after his signature key.

After this step the user communicates with the SEMPER RA/CA server automatically in order to obtain the RA/CA's public key, the so-called *trusted public key*. On the TINGUIN display, the fingerprint (hash value) of the CA's public key is shown, if he runs the registration/certification application for the first time.

With the SEMPER software, the user gets the fingerprint of a CA's public key as a file or on paper. He is asked to check the two fingerprints. If the two fingerprints match, the user should press the "YES" button; otherwise, he has the opportunity to reject. In this case, the registration process is stopped. After the fingerprint of a RA's public key is stored, the user might attempt to run the registration procedure once again. In this case, a message is displayed informing the user that the public key of the registration authority, he is contacting at the moment, is already installed on his machine. He presses OK to continue.

Phase 3

Now the user is ready to obtain his registration/certificate. By clicking the Quit button the registration process will abort.

A TINGUIN display with some fields appears. This is the template the user received from the RA/CA automatically via the SEMPER software.

All fields should be filled in: user's name and surname should be entered in the first and second fields respectively. The name of the company, organization or institution in which the user works, should be entered in the third entry field. The city and the country of the user's residence in the fourth and fifth fields. In the registration letter (see above) a registration key is assigned to the user which should be entered in the last field.

In first attempt to become registered, texts at the left side of all fields will appear as bold. In the next attempt, the previous entries will appear in the entry fields as default values. The text left of the field for entering the registration key appears bold always, that means, the user has to enter his registration key every time he is intending to obtain a registration/certificate. As mentioned before, the registration key serves for the user's authentication against a RA/CA server.

An authentication check is processed at the CA server which is not visible for the user. As soon as the authentication check procedure at the server side has succeeded, a certificate is issued for the user's signature key which is sent to him afterwards. The TINGUIN displays this information. After receiving the issued certificate, a verification process is going to be invoked at the user's side which verifies the CA's signature in the incoming certificate.

The SEMPER software has already installed the RA/CA's public key which is necessary for performing the verification. The result of the verification procedure is displayed on TINGUIN.

If the authentication check fails, an error message is displayed on TINGUIN saying that the user is not pre-registered. One reason for this could be that the user is not pre-registered at all. In this case he should contact his service provider or the registration/certification authority. Another reason could be that at least one of the entries in the template was wrong. In this case the user might try the registration

process again to get the opportunity to set his entries correctly. Time out or network error could cause errors as well.

Phase 4

After the first registration/certification process is completed a TINGUIN message is displayed, mentioning the status of the registration. The registration process for the user's encryption key could be started now. The next process, i.e. registering the user's encryption key, is performed based on the information entered in the first step.

During the registration and certification of the user's encryption key, he will encounter the same TINGUIN displays than when his signature key was processed. After the certification application is completed, the user will own two registrations and two certificates which are stored in the SEMPER archive.

The registration/certification process is completed now. By pressing the "Finish Registration" button, the session finishes.

5.3 Credentials

(T. Pedersen / CRM)

A credential denotes a token defining some rights or properties. The owner of a credential can show it to other parties in order to obtain or execute certain rights. In the following we shall consider credentials that are represented and used electronically. Electronic cash is a special kind of credential, since money can be used to get access to certain services. The distinction between credentials that are money and those that are not is not always clear: a telephone card could, for example, either be considered a credential giving access to telephone services, or a special payment system in which the phone company plays the roles of issuer, acquirer and seller.

Next a model for credentials including security properties is presented and then two types of credentials, static and dynamic, are described in more detail. Finally, it is discussed how credentials fit into the SEMPER architecture.

5.3.1 Model for Credentials

The model for credential mechanisms must capture possible applications including the following:

- Tickets:
  - one-way ticket
  - return tickets
  - “strip” cards
  - time limited tickets (month cards)
- Driving License
- Passport
- Keys
  - Physical access (e.g., to a building)
  - Logical access (e.g., to information in a database)
- Membership cards
- Insurance card
- Business cards
- Loyalty schemes
- Attribute certificate (e.g., used to give power to act on behalf of someone else)
- Diploma
- Lottery ticket
- Negotiable document (e.g., bill of lading)

- Acts of property (e.g., statement that you own a house)
- Prescriptions

While more applications are conceivable we believe that the above list covers most of the different types of credentials that are met in practice. The list may even be too extensive as some of these examples may not be used electronically for quite some time.

5.3.1.1 Information in Credentials

For all applications mentioned above the following information seems to be obligatory for a credential:

- *issuer* describing the organisation which has issued the credential (i.e., granted these rights to the subject)
- *subject information* describing the user or users which are granted the rights
- *scope* describing the rights associated with the credential
- *validity period* (start and end dates)
- *proof of authenticity*, usually this is a digital signature on the above four attributes.

We will distinguish two types of credentials, *static* and *dynamic*. A credential is said to be *dynamic* if it changes when used. A phone card is an example of a dynamic credential: after it has been used it can be used for fewer future phone calls. Electronic cash is another example as it can only be spent once. In most cases dynamic credentials will resemble some form of electronic money.

A credential, which is not changed during its entire validity period, is called *static*.

Dynamic credentials will normally be more difficult to support than static ones as there must be some means preventing reuse of (previous versions of) credentials. This problem is similar to that of preventing that pre-paid electronic cash can be copied and spent several times. Such reuse is typically prevented using tamper resistant hardware and/or on-line verification. Static and dynamic credentials are discussed in more detail in Section 5.3.3.

5.3.1.2 Pseudonyms and Credentials

A pseudonym is a (digital) name, which a user can operate under in a session. It should in particular be possible to have credentials issued and shown under a pseudonym. In case of non-anonymous credentials the pseudonym may be the real life name of the user or for example the distinguished name as defined by one of the user's certificates. In the case of anonymous transactions a pseudonym is just an electronic token unique for the user but not explicitly linked to that user.

Just as a person must be registered for non-anonymous transactions is it necessary that an anonymity protecting pseudonym is validated by a third party, a *pseudonym issuer*. Otherwise it may possible to fraud the credential scheme (e.g., see [MAS7304]). Depending on how much the user trusts the pseudonym issuer privacy protecting

pseudonyms may or may not be shown to this party. In the latter case the pseudonym will be issued using a blind signature protocol.

The processes of validating a pseudonym and issuing a credential can in principle be separated, and in many applications they are really separate: first the user gets a pseudonym, and next a credential which gives certain rights to that pseudonym. However, it should be noted that in some privacy protecting credential mechanisms, these two processes are closely related (see [MAS7304] for a concrete example).

5.3.1.3 Roles

A *user* is any party in the scheme having or requesting a credential. Sometimes a user may not participate directly but only through a *representative*. This will allow the user to hide (or remain anonymous) behind the representative.

An *organisation* is any party in the scheme to whom a credential is shown. A party playing this role may also be an issuer (see below) since the presentation of a credential may lead to the organisation issuing another credential.

A *Registration authority* is needed in case a user is not registered beforehand by a pseudonym issuer or as part of getting a certified key.

A *Pseudonym issuer* issues pseudonym as discussed in Section 5.3.1.2.

A *Issuer of credentials* is responsible for issuing a credential, for revoking credentials (in case of key compromise, misuse, …) and for renewing expired or expiring credentials

A *Directory Authority* is responsible for giving information about issued credentials. This authority can be local to a particular issuer or serve many issuers.

In addition there may be some third parties involved. Two examples are

- A third party providing secure time-stamps
- A third party clearing credentials between organisations

Some of these roles may not be necessary in some situations. E.g., in many cases a user may be registered beforehand and just get a credential on the registered public keys. In other situations the user will want to register a new pseudonym for a particular organisation. The example below shows how some of these roles can come into action.

5.3.1.4 Example

Assume a user A wants an electronic membership card to a loyalty scheme of shop S to be used in all branches of S. It should be anonymous so that only the main office, but not the branches, can trace use of the scheme to A:

1. A registers and gets a key certificate at a CA.
2. A goes to S. Here A generates a new public key, and asks S to be member of the loyalty scheme under this public key (pseudonym). S verifies that A knows the corresponding secret key and issues a credential stating that the person takes part in

the loyalty scheme (thus here S registers the user, issues a pseudonym and a credential).

3. The user can then get the discount by identifying herself against the pseudonym (public key) validated by S when shopping at a branch of S.
4. When being presented the credential, the branch can ask S working as a directory whether the credential is still valid.
5. The branch can keep a record of the transaction and later get a special bonus for serving many members of the loyalty scheme (hence some clearing takes place).

The branches cannot identify A. A's transactions can of course be linked. Hence, the electronic credential should be updated quite frequently in order to obtain good privacy (see Section 5.3.2).

5.3.2 Security Properties

See [MAS7304] for details on security properties. Most notably, it should be infeasible to forge pseudonyms and credentials.

- It should be infeasible to create a valid credential under a pseudonym unless the issuer has accepted this.
- Showing a credential requires knowledge of the secret corresponding to the pseudonym of the credential.
- It should be infeasible to show a credential belonging to pseudonym P1 under a different pseudonym P2.

Only the credential issuer must be able to revoke credentials, and this should be possible as soon as misuse is detected.

In some applications the user may require to be anonymous. We consider two levels of privacy protection

- The scheme is called *anonymous* if it is not feasible to identify the user given all showings of the credential (assuming no additional information).
- The scheme is called *unlinkable* if it is anonymous and it is not feasible to recognise that several uses of a given credential are by the same person.

An anonymous but not unlinkable scheme (as the one in the example of Section 5.3.1.3) may not protect privacy very well, since all the showings can be linked and therefore be used to make (unique) user profile which given additional information can be used to identify the user.

When making a privacy protecting credential scheme it is necessary to consider the trust model. A user may want to reveal more or less personal information to the pseudonym issuer, credential issuer and organisation.

If the user does not want her privacy to depend on the pseudonym and credential issuers blind signatures will be necessary for issuing pseudonyms and credentials. However, if the user only wants to be protected against the organisation to which a credential is shown and is willing to trust the issuers it may be possible to obtain a

simpler credential scheme. In other words, putting more trust on some third parties may make it easier to construct a credential scheme.

Special types for credential schemes may require additional security properties. E.g., dynamic credentials must, as mentioned above, prevent reuse of already used rights.

5.3.3 Static and Dynamic Credentials

Above we made the distinction between static and dynamic credentials. While our model allows for both types of credentials, it will usually be much simpler to support static credentials (as dynamic credentials require precautions against the aforementioned attempts to reuse already used rights). In the following we investigate these two types of credentials.

5.3.3.1 Static Credentials

A static credential is a credential that keeps the same value until it expires. It typically makes a statement about the owner of the credential, that provides access to certain services as described by the scope of the credential. The validity period is given by an expiration date after which, the credential can no longer be used. Note however, that credentials can be revoked before they expire (either on request of the user or on request of the issuer), and in general it should also be possible to extend the validity period.

Passports and ID cards are credentials that keep value for a longer period of time. They can be showed to prove one's identity over and over again. A passport is valid for, say five or ten years, and after expiration the old passport shall be destroyed (revocation) and a new one (renewal) must be created. Membership cards, driving licenses, insurance cards, company cards and access cards or tokens are — technically seen — equal to a passports or ID cards. Usually the expiration time for a membership or insurance card is somewhat shorter, typically those cards are valid for one year, while a driving license has a much longer validity period (although it may be revoked if, say, the user drives under influence of alcohol).

Credit cards give access to a payment method. They are used to authorise the credit card company to pay money from the payer's account into the payee's account.

Tickets, like tickets for theatre shows and flight tickets (for particular departures) can also be described as static credentials, as they have an explicit expiration date corresponding to the time of the show or flight, respectively. However, the Dutch *"strippenkaart"* (stripes card), which gives the owner right to several bus rides, is not a static credential as its value decreases after each use (at each ride the card is stamped and it devaluates).

Other examples of static credentials include:

- attribute certificate, e.g. describing the right to sign for your company
- access tokens to Internet sites
- access tokens to games

- cards for getting discounts (like *BahnCard* in Germany for getting 50% discount for all your travels in a year.)

A business card is meant to give away to business acquaintances. Its primary purpose is to pass information, and it is therefore not seen as a credential here.

5.3.3.2 Dynamic Credentials

Unlike static credentials the value or (or rights associated with) dynamic credentials will change over time or as the credential is used. This is described in the scope attribute of the certificate. As mentioned above this change of value makes dynamic credentials resemble electronic cash quite a lot and the security concerns are much the same. In particular it is important to prevent that the user can copy the state of a credential in order to later (after the credential has been used and, hence, lost value) restore this state. The same mechanisms that are used to secure electronic cash, can prevent reuse of credentials:

- On-line verification that the credential has not been used before
- Tamper-resistant hardware combined with cryptographic protection when the credential is shown to an organisation ensures that the credential cannot be copied and reused.
- For anonymous credentials it is possible to add a mechanism identifying the user if it is detected afterwards that a credential has been reused. This is most notably used for so-called *k*-showable (for some positive integer, *k*) credentials which can be used *k*-times anonymously (these can be linked), but if it is used more often the user will be identified (se [CFN88]).

The first measure is expensive in terms of communication and may be impractical in certain applications, but provides very good security. The security of the second depends on the security of the tamper-resistant module, but this solution will often be quite practical, and more appealing as smart cards become more common. The third solution does not prevent misuse and should either be considered as a back-up solution to the two proceeding solutions for anonymous credentials or be used for low value credentials.

Some examples of using these precautions:

On-line verification

A user could get (buy) a credential allowing to visit a server 10 times. The server may as well keep account of the number of visits and disallow the credential after too many visits

Tamper resistance + identification after the fact

In the above example it could be an advantage to have the credential on a smart card if the credential can be used to access many different servers. In that case an on-line look-up to a central credential server may be too costly and time consuming, not to mention the annoyance if the central server cannot be reached. In this case tamper resistance prevents misuse of the credential. In order to recover

from users breaking tamper resistance, it could be useful to add identification of cheaters.

The choice of protection obviously affects the design of the credential system. A system based on tamper resistant hardware protection is probably easiest to design as the security mechanism is implemented locally in the hardware device. In its simplest form this can just be like a static credential, as the tamper resistance prevents misuse of the credential.

A system based on on-line verification requires that the organisation when being showed a credential can ask a credential database whether the credential is still valid. The design may support local as well as remote databases.

The ability to identify cheaters in otherwise anonymous schemes often requires more complicated protocols for issuing credentials (here the mechanism for such identification — usually based on a secret sharing scheme — must be set up) and possibly also for showing credentials. Furthermore, methods for detecting and identifying cheaters must be available (requires a central database where all shown credentials are registered).

5.3.4 Credentials in SEMPER

Apart from electronic money, which is handled by the payment block in SEMPER the certificate block was initially supposed to support credential mechanisms, as the model for credentials resemble that of public key certificates. In particular, pseudonym handling is quite similar to the management of public key certificates. However, showing a credential differs substantially from public key certificates, and resembles a payment more than certificate handling. More precisely, while public key certificates provide services *enabling* public key cryptography (in particular binding signatures), showing a credential is (just as like a payment) an integral part of the business process. Thus credential mechanisms provide on one hand functionality similar to the certificate block with respect to management of pseudonyms, while on the other they provide functionality similar to the payment block with respect to using credentials.

Thus credentials seem to be most easily handled using a new service block. This block must provide functionality for getting and showing credentials. Furthermore, this block must provide methods for managing the credentials of a user (assigning policy to the use of these, as well as for selecting credentials).

Pseudonyms are most easily handled by extending the certificate block (as a public key certificate is a pseudonym). In order to be able to use pseudonyms, each pseudonym is associated with private information needed to use it (e.g., to get or show a credential under that pseudonym). For public key certificates the private information is simply the private key, but in general the private information may consists of further information such as random information used when generating the pseudonym.

6 Anonymity in SEMPER

(L. Fritsch/SRB, S. Prins/KPN, M. Schunter/UDO)

6.1 Introduction

6.1.1 Anonymity in the Electronic Marketplace

Today's electronic marketplaces usually provide little or no privacy for the players. From consumers' navigation over a commerce server to credit card billing and consumer profile trade to information about who establishes data communication with whom, all kinds of data can be collected and used without the users being able to know or to prevent this invasion into their privacy.

Anonymous, unobservable commerce in SEMPER aims at giving back control about this kind of information to users. The foundation for anonymity in SEMPER are certified pseudonyms and anonymous communication. Anonymous certified pseudonyms help protecting information about a player's actions in the marketplace by hiding its identity. Anonymous communication channels provide the unobservability of communication relationships and originating hosts. Based on these foundations, other SEMPER blocks then need to provide similar "anonymous mode of operation", such as untraceable payments.

6.1.2 Scenarios of Anonymous Transactions

In order to determine what should be provided by the new Semper components when anonymous actions are processed, we first look at three scenarios. The next section will go deeper into architecture details.

6.1.2.1 Simple Anonymous Requests (scenario #1)

Suppose a provider offers to answer anonymous simple requests. For example, a car dealer may provide an up-to-date price list for all models in stock. A potential buyer may be interested in this offer, but (s)he does not want to receive advertisements each time (s)he seems to be interested to buy something. If the request is anonymous, the seller cannot trace the customer back in order to add his name in a database preventing companies to exchange information on customers for their advertisement campaigns.

This is the simplest scenario: first an anonymous request is sent from the customer to the seller. The seller sends back an answer to the anonymous customer. The only required component needed to perform such operations is the availability of an anonymous channel.

6.1.2.2 Anonymous Offer Requests (scenario #2)

Suppose that car dealer has special offers for some customers. It offers special prices if the customer is a regular client of this particular seller. The client may have advantage to submit its request anonymously to ensure the dealer does not make the offer according to the customer identity. The only thing the customer is willing to disclose is the fact that (s)he is a customer of this particular seller. This kind of request combines credential and anonymous communication.

This credential information must not leak more information about the customer than its ability to make special requests. The anonymous message contains as part of the message some credential information issued to a pseudonym of the client.

6.1.2.3 Anonymous Partially-Linkable Requests (scenario #3)

We can imagine an even more general scenario when in addition to credentials, the request is accepted if the pseudonym has some privileges. Let us imagine that a travelling agency gives reduction according to the number of travels a particular customer has bought so far from this agency. The customer may be interested to get an offer for his next trip in Easter Island. Like the above two cases the customer wants to keep its identity secret until the decision to buy or not has been made. The main difference between this case and the previous one is that here the seller requires to know something about the pseudonym. The pseudonym is not necessarily fresh anymore but can be already known by the seller.

6.1.3 Types of Anonymous Communications and Transactions

In this section different aspects and types of anonymity are described.

6.1.3.1 Secure Identities

To be able to define what anonymity means for SEMPER, the components of identity need to be considered.

Every user in SEMPER has a certificate that constitutes her identity. But besides the certificate, the computing and networking context also define elements of identity: network addresses, data rates, communication time profiles, network addresses frequently connected to. With repeated use of the same service in the network even with the use of a pseudonym, behaviour patterns can be used for identifying a user, i.e., if a user does not change ist pseudonym on a regular basis, deanonymization becomes likely.

6.1.3.2 Physical and Content Information

In general an anonymous electronic transaction means that the initiator identity is not revealed by the interaction with the server. Over the Internet full anonymity cannot be achieved by cryptographic techniques alone. Even if the transmitted messages (sent using Internet protocols as TCP, HTTP, e-mail, etc.) reveal nothing about the sender, the receiver might be able to get information about its identity because the underlying

protocol needs physical information (i.e. IP address) about the user in order to operate. The identity can be revealed in two orthogonal ways: by the message's content and by the internal protocol used to transmit messages over the Internet. The former can be solved by cryptographic techniques while the latter needs physical solution. In order to achieve perfect anonymity, it is necessary that the interaction between the initiator and the server satisfies the following two informal properties:

a) The content of the messages sent by the initiator to the server during a transaction cannot help the server to find the initiator identity.

b) The protocol used by the initiator and the server does not reveal the initiator's identity.

If one wants perfect anonymity one must make sure that conditions a) and b) are satisfied. For instance, no anonymous transaction is possible if a message from the initiator and the server contains the signature with the true name of the initiator. Similarly, even if the messages sent by the initiator are anonymous according to a), if the communication protocol unveils the initiator identity then no anonymous communication is possible. Two different techniques can be used in order for both conditions a) and b) to be satisfied.

Condition a) states that each message content cannot be used to find the originator identity. If no security attribute is applied to the message to be sent, the originator's identity is not revealed unless the message itself contains a direct reference to it. However, even if a communication is anonymous it may be required to contain some credential information in order for the server to determine if the request is accepted or rejected. For example, the initiator may be required to give a proof that (s)he is at least 21 years old for the request to be processed. This requirement can be satisfied while keeping the communication anonymous by signing the initiator message with a pseudonym for which a guaranteed 21 years old credential is also sent. In general condition a) can be satisfied with cryptographic techniques like blind signatures [Chau82].

Condition b) states that not only the message content must be anonymous but also the channel by which the message is sent. This condition is not met in general by Internet communication protocols because they include the return address and/or the sender identity. For instance, it is not possible to achieve anonymity according to b) with ordinary e-mails. Even if a message contains no reference to the sender identity the sender's return address, is part of the message. One may consider to use e-mail with an anonymous account in order to hide its own identity. This solution may provide anonymity if the sender identity cannot be linked to the computer IP address which is not always the case especially when personal computers are used. The general requirement b) cannot be solved only by cryptographic techniques. In most situations, a third party is needed to ensure full anonymity; this is what so called MIX (see the next section) channels achieve. This requirement is about the physical means by which anonymous messages are sent.

It is then clear that including full anonymity in SEMPER transactions requires to integrate at least two different and new components. One component will be responsible to deal with credentials and pseudonyms, another will be responsible to provide anonymous channels. In addition, security attributes corresponding to an anonymous communication should be processed somewhere in the transfer layer. The

transfer layer becomes where the integration of requirements a) and b) will be mostly done.

6.1.3.3 Different Flavours of Anonymity

Given that secure anonymous channels are available, this section discusses informally how in principle a user's identity can be securely concealed during the execution. In some case however, perfect anonymity might not be possible to achieve or simply unsuitable. As an example, it might be impossible because some server requires user's privileges in order to process requests made through an anonymous channel. It might also be non-suitable (for efficiency reasons) if the request processing requires several interactions. For example, a MIX channel can be used without *batches* allowing real interactive anonymous sessions between the user and the server to take place. The drawback of this approach is that anonymity is no more guaranteed against *traffic analysis* (it is then possible to find the user's IP address by looking at what comes in the MIX and what comes out towards the server).

Regarding the IP numbers it is worth mentioning that most users actually dial using dynamically assigned IP addresses which, assuming the Internet Service Provider is large enough and trusted, gives already anonymity to some extent. Though one would have to be very careful not to do any network-operation in parallel to anonymous communications during that connection, which might reveal your identity. Additionally one also would have to make sure that the user's PC wouldn't leak information when probed (e.g. on the mail, ftp or telnet port).

Even when the user is concerned about not revealing its identity, it might be enough to hide its identity to the server unless traffic analysis takes place. Therefore, anonymous actions appear in different forms not only defined as opposition to non-anonymous ones. From non-anonymous to completely anonymous actions there are different flavours taking into account efficiency, privileges and different possibilities in the choice of anonymous communication means. The integration of anonymous actions in SEMPER architecture should allow all different flavours to be provided by the same mechanisms and structures. It must be natural to select what kind of anonymity is needed according to the context in which anonymous requests are made.

6.1.3.4 Degrees of Security in Anonymous Communications

As discussed in the previous section, security in anonymous communications depends in particular on the adopted channel, the power of potential opponents and the efficiency/security trade-off. Because the nature of the Internet and electronic commerce, unconditional security cannot be achieved in general. However, it is possible to define an almost continuous spectrum of anonymity from the stronger to the weaker form where some intermediary points can be defined. Using [ReRu97] terminology degrees of security and different type of attackers can be identified. Some types of opponents are weaker than others and therefore easier to be protected against. Situations occurring in electronic commerce do not require security against all powerful opponents but usually against a subset of them with no real risk for the security of the transaction. The types of opponents defined below are independent on

the underlying anonymous channel. However, some anonymous channels may or may not guarantee a particular degree of anonymity against some kind of opponents.

1. *Local eavesdropping*: A local eavesdropper can observe what comes in and comes out a particular recipient or user. Its view is local and cannot be extended to several machines.
2. *Traffic analysis*: That is the strongest opponent against anonymity. Such an opponent is able to follow the communication along the path form the originator to the end server. Its view is global like a network big brother.
3. *Third parties coalitions*: That is when the third parties involved in the implementation of the anonymous channel are allowed to behave maliciously by forming coalitions. It is implicit that the view of the coalition is limited to what is submitted to them and what is sent out to them.
4. *Malicious end server*: That is the main security thread that anonymous communication should protect the user from. The opponent's view is limited to what the server receives.

6.2 General Framework for Anonymity

We now sketch how anonymity can be provided by SEMPER.

6.2.1 Anonymity means Complete Anonymity

In SEMPER, services are provided by so-called blocks. Each top-level commerce service uses a variety of services from other blocks on lower levels. Sending an order, for example, will include the transfer block who will use the statement block for signing and the secure communication block for sending the message. The secure communication block then uses the communication block as well as cryptographic services.

Therefore, any level of anonymity for commerce transactions can only be provided if all blocks involved in it provide at least this required level of anonymity. If any of the involved blocks leaks some information which destroys the anonymity, the anonymity of the commerce transaction is lost since all involved (sub-) transactions are linked with it.

Therefore, all blocks which may leak identifying information need to know that a given commerce transaction requires a given level of anonymity.

6.2.2 Anonymous Contexts and Services

Anonymity in SEMPER has two major building blocks: Anonymous contexts and anonymous services executed in these contexts.

Initially, when an anonymous commerce transaction is started, all lower layers are notified about the level of anonymity requested. This request for anonymity is done by so-called security attributes included in the contexts of the different layers: When a

commerce transaction requires anonymity, it establishes an anonymous deal[26] while including an anonymity security attribute which specifies the level of anonymity to be guaranteed. The commerce layer then opens a secure txlayer context while again including the anonymity attribute. This secure transfer context then establishes an anonymous communication link with its peer using the secure communication manager which is again notified by passing the security attribute on.

After this set-up phase, all involved layers know that the current commerce transaction is anonymous while the level of anonymity required is specified by the given attributes. Furthermore, all layers are able to communicate anonymously with the peer by using the secure transfer contexts.

When now services (i.e., so called transactions) are executed in this anonymous environment, each manager of each block providing these services is required to negotiate with its peer so that the selected module provides the required level of anonymity under the assumption that all blocks of lower layers provide this level of anonymity, too.

If we consider an anonymous payment, for example, the payment managers may negotiate and agree on the ecash module. This module, however only provides anonymity, if the communication services provide anonymity, too.

6.2.3 Overview over Anonymity Services

As mentioned before, not only the communication channel has to be anonymous, also the SEMPER architecture has to be adjusted to support anonymous transactions. For each service, there should also be an anonymous version of it. This is illustrated in Figure 113: The white areas of the figure describe components of the SEMPER framework responsible for the anonymous actions to take place.

- The commerce layer integrates the anonymous versions of what is already provided.
- Anonymous containers are implemented in the transfer layer
- Credentials/pseudonyms are closely related to the certificate manager.
- The statement block should provide a way to create statement sessions according to a pseudonym (it seems to require only minor changes to the actual `StatementTransaction` class).
- The payment manager provides anonymous payment systems (e.g. *e-cash*) which will be accessed over the token-based interface (see Section 4.3) and carried in the anonymous containers of the TX block.

26 Security Contexts of the commerce layers are called “deals”.

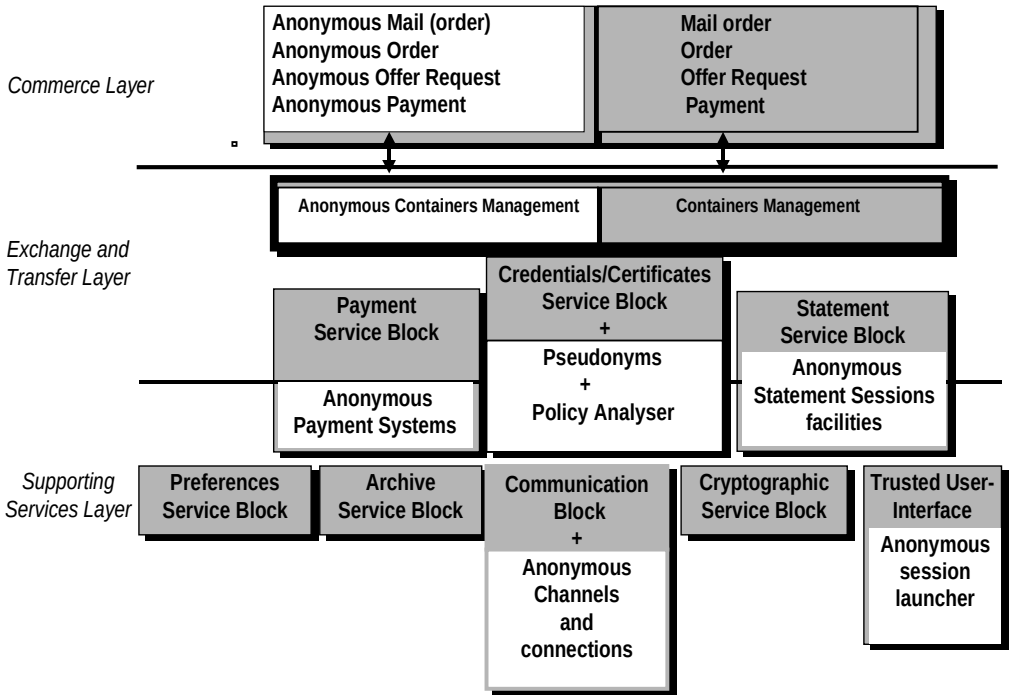

Figure 113: Anonymous services in the SEMPER architecture.

6.3 Design of Anonymous Channels

This chapter describes the anonymous communication module of the SEMPER SecComm manager which implements anonymity be means of connection-oriented MIXes [PfPW85].

We first give an overview over the basic MIX principle and then we describe the different parts of our object-oriented design.

6.3.1 MIX Terminology

A message is **anonymous** if the sender cannot be identified. A communication channel is **unlinkable** if an eavesdropper can identify that a message was sent, but not to whom it was sent. A **MIX** is a service that collects, reorders and re-sends data from and to many data connections. Its purpose is the unlinkability between the incoming and the outgoing data traffic of a MIX. When several MIXes in a row are used they form a **MIX chain**. In analogy to the SEMPER communication block, the **initiator** of a data connection is the party that starts a communication while the party accepting a connection is called the **responder**. In SEMPER, data connections in client-server scenarios are called **channels**, and the channels in this paper are **SecChannels** provided by the secure communications block. A **MIX** server is a service that accepts and forwards many anonymous secure channels, whereas the word **client** is used for the two end-points of such a channel.

When discussing the internals of a MIX server, we need to distinguish the two directions data can enter a MIX from a channel. As we always draw the initiator’s side of a channel in a diagram's left side, we call the channel that links the MIX server with

the initiator's client **l-channel**. The channel connecting the MIX server with the responder is called **r-channel**.

An extended finite state machine is a basic building block of the MIX servers. We sometimes just write "state machine" or "channel machine" as there exists such a machine for each anonymous channel operated by a MIX server.

Finally, the expression **dummy traffic** describes the method to send random data in order to cover the routes of genuine traffic.

6.3.2 The MIX principle

The SEMPER anonymous communication follows the MIX idea of D. Chaum in [Chau81] and the results of Pfitzmann, Pfitzmann and Waidner in [PfPW86, PfWa87, PfPW91, Pfit90] for MIX channels.

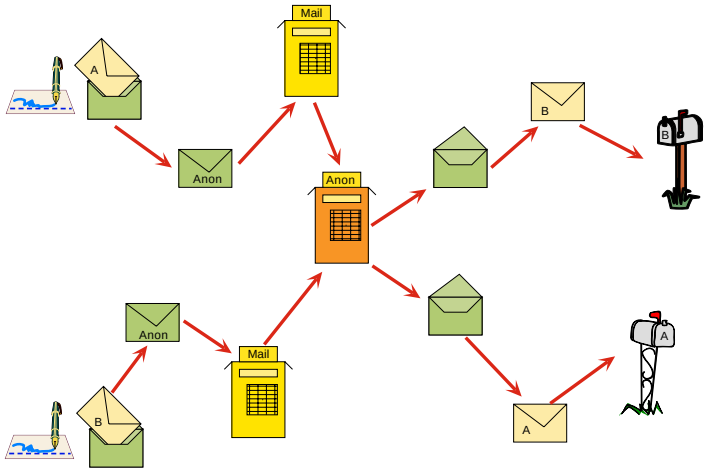

Figure 114: The MIX mailbox example.

6.3.2.1 Message-oriented MIXes

The main idea of a Chaum MIX [Chau81] is to hide individual communication relationships in a large set of communication relationships. A MIX is a node where all the communication of the set of communicating partners is routed through in a standardized form that prevents identification of sender, recipient, content and message identification.

A good example for the function of a MIX is the *anonymous mailbox*. Imagine a mail service for delivering anonymous romance letters. Nobody - not even the mailman or the yellow press - is supposed to be able to link the sender of a letter to the recipient. Figure 114 illustrates the anonymous mailbox.

First, two writers write their messages and put them into the bright envelopes. The recipient's address is written on the bright envelopes. Then, the bright envelopes are put into dark envelopes that address the anonymous mailbox. The dark envelopes then are delivered, and sorted into the anonymous mailbox. Finally, the content of the anonymous mailbox is mixed, and then the dark envelopes each are opened and the contained bright envelopes are delivered.

6.3.2.2 Connection-oriented MIXes

Chaum's idea of a MIX for e-mail was extended to data connections with continuous data flow in [PfPW91, JMPP97]. A problem in Chaum's e-mail MIX is the delay it adds to messages for buffer filling. Connections with continuous data flow usually have some time constraints on the data transfer. A long delay of the data sent cannot be accepted.

The solution to the problems of traffic analysis and message delay in MIX nodes is the construction of time-sliced MIX channels. Such a channel has a clock initiating data transmission at the end of constant intervals. If an interval finishes without data ready to be sent, some random dummy data is sent instead. Thus, in an outside observer's view, there is always traffic at the same bandwidth on the data connection. All MIX nodes on the connection follow the same policy for their output: clock synchronicity and dummy data.

6.3.2.3 Existing Anonymity Systems

Several implementations of anonymity systems are available. The following list presents some of them:

- Mixmaster e-mail MIX: a non-commercial implementation of Chaum's MIX. Uses strong cryptography. Source code available.
- Babel e-mail MIX [GuTs96]: an anonymous remailer system implemented at IBM Zurich research lab. Features pooling, nested encryption, return anonymous addresses.
- Onion Router [GoRS96]: a project implementing a connection-oriented MIX for standard protocols (e.g., http, rlogin). Uses strong cryptography. Source code and program are export-restricted and thus only available in the USA. A demonstration system is available at http://www.onion-router.net/ for testing.
- Crowds [ReRu97]: anonymity system to hide a user's WWW-reading behaviour. Protects data links with symmetric cryptography and forwards WWW requests to other participating Crowd hosts.

6.3.3 The SecComm Module

The SecComm module provides `ComPoints` which enable the users to send and receive anonymous messages. We distinguish an initiator and a responder `ComPoint`. For opening an anonymous connection to a responder, the originator who wants to stay anonymous needs to know the address where the originator is listening for incoming communication requests (i.e., the address of the so-called responder ComPoint). With this address, the initiator then initializes a so-called initiator `ComPoint`. During this initialization, the machine retrieves the list of MIXes trusted by the originator from the preferences and then establishes a connection via those MIX servers as described below. Note that the responder need not be aware that the originator is anonymous since the connection seems to come from the last MIX.

6.3.4 The Design of the MIX

The MIX for SEMPER consists of three main entities that take care of the tasks in the MIX. These elements - called *server*, *coordinator*, and *automaton* - are shown in Figure 115 and will be discussed in detail in the following paragraphs.

- The *server* entity listens on a fixed ComPoint and accepts incoming requests for anonymous connections. For each incoming connection request, it creates a new MIX *automaton* which is responsible for looking after this particular connection.
- The *automaton* then observes the data traffic on its anonymous connection. The detailed behavior of these automata is presented later in this text.

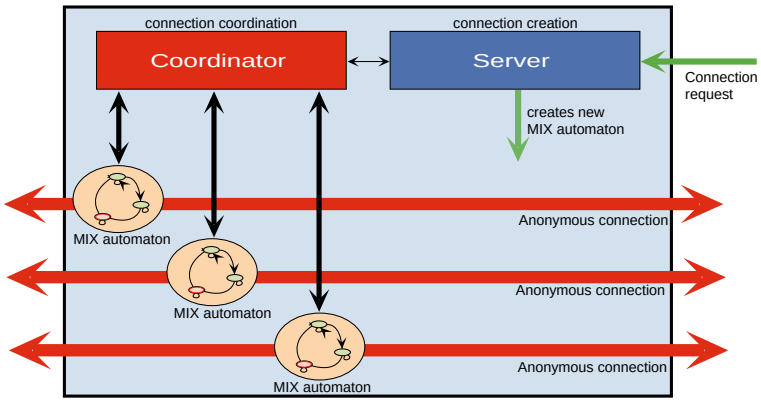

Figure 115: Scheme of the MIX server.

- A *coordinator* element finally is responsible for coordinating all active automata inside the server. It interacts with all automata over a so called *coordinator interface* and takes care of dummy traffic and establishment and shutdown of channels. The coordinator is also notified by the server when a new automata is created. An overview of MIX coordination is presented later in the “policy control” section.

These three important components of the MIX for SEMPER are introduced later in this section.

6.3.4.1 The MIX Server

The MIX server is a Java thread listening for incoming Secure Channel requests on a defined port with the default correlator “MIX”. Requests are accepted up to a limit for the number of simultaneously open connections. For each accepted request, a MIX automaton is created and initialized that is responsible for processing messages of the MIX protocol. The automaton communicates with a `Coordinator` object which is part of the server.

Elements of the MIX server that actively participate in established anonymous connections are the MIX automata and the `Coordinator` object.

The following sections describe the automaton and the `coordinator`.

6.3.4.2 The Automaton for each MIX Channels

Each automaton is responsible for looking after one anonymous connection inside the MIX. We define its behavior by defining its states, its messages, and its state transitions. Given this information, the automaton has been implemented using the object-oriented design pattern for extended finite state machines.

6.3.4.2.1 Automaton states

The state classes are the JAVA classes inherited from `MixState`:

- `MixStateConnected` (initial state): In this state, the automata is connected to the originator of the connection. It waits for incoming requests (`MixForwardMessage`-messages) to connect to the next MIX.
- `MixStateForwarded`: In this state, the automaton is connected to the originator as well as to the next MIX automaton or to a responder. Most incoming messages are just forwarded to next machine.
- `MixStateTerminated` (final state): An automaton is in this state during the shutdown of a connection.

6.3.4.2.2 Automaton messages

Messages inherit the `MixMessage` class. Each message contains padding data to enable fixed message lengths. Messages implemented are:

- `MixForwardMessage`: instructs a MIX automaton to open a bridge to the contained address. It contains a `ComPointAddress.`
- `MixDataMessage`: responsible for data transport in anonymous channels. It contains a serialized object,
- `MixDummyMessage`: is a dummy message which is empty besides its padding,
- `MixCloseMessage`: contains information about channel shutdown to a MIX automaton,
- `MixStatusreqMessage`: requests MIX status information,
- `MixStatusMessage`: transports MIX status information to the initiator.

By definition, only messages of the following subclass types can be output to other nodes: `MixDataMessage`, MixDummyMessage and `MixStatusMessage`. The other messages contain instructions for the automaton and are never forwarded.

Short of unified message lengths, at this stage of the implementation, all messages contain a random-length field of random bytes. Depending on the purpose of the message, more data members (called parameters in the protocol automaton design pattern) can be defined.

6.3.4.2.3 Automaton I/O and concurrency

The anonymous channel that is managed by the automaton is a bi-directional communication channel. Messages can arrive from two directions: from a so-called l-channel and a so-called r-channel. The same holds for the automaton's output: outgoing messages can be sent into two distinct output channels.

Figure 116: Output queue operation of an Anonymous Channel

Therefore, we extended the design pattern by adding an additional process()-method for processing messages in both directions. The `lprocess(message)` processes l-channel messages and `rprocess(message)` processes r-channel messages. Messages received on the l-channel can be forwarded into the r-channel only and vice versa. For concurrency, two input and two output threads are installed for each automaton to deliver input messages to the transition function and send output messages. The automaton's processing method queues outgoing messages into its r-channel and l-channel outgoing queues. Then, the coordinator object is notified of the queued messages by calling its `send_data()` method. Depending upon the coordinator's policy, the queued messages are eventually sent. This is depicted in Figure 117.

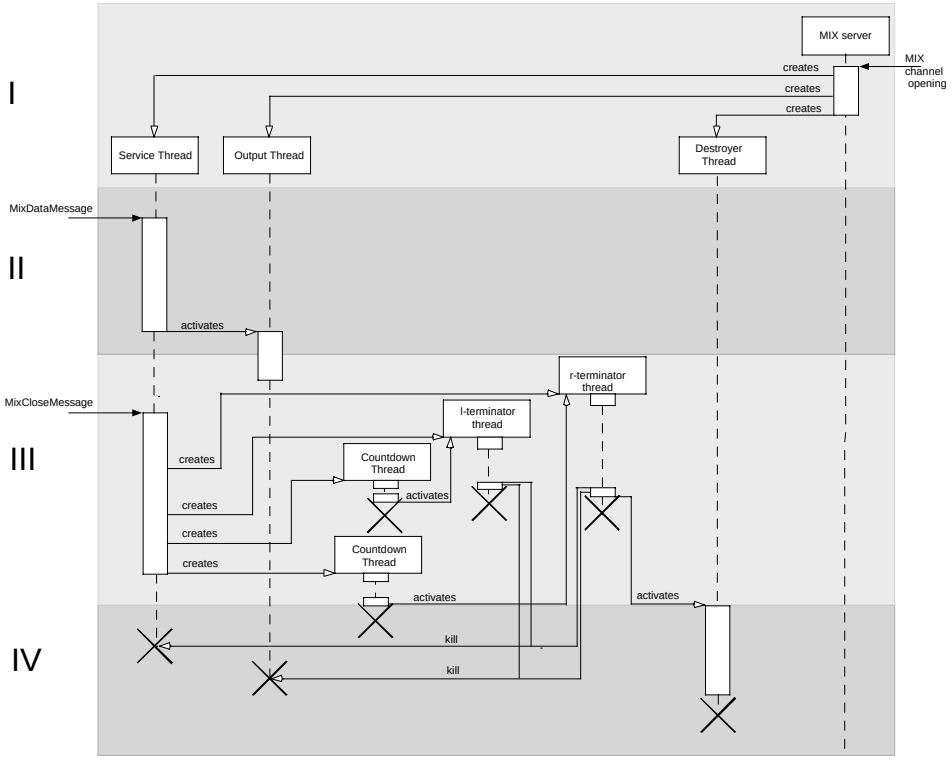

Figure 117: UML interaction diagram for MIX server threads.

Figure 117 shows the UML interaction diagram for the MIX server threads. For better understanding, only one input thread and output thread and their interaction are presented. A MIX in operation has two input and two output threads.

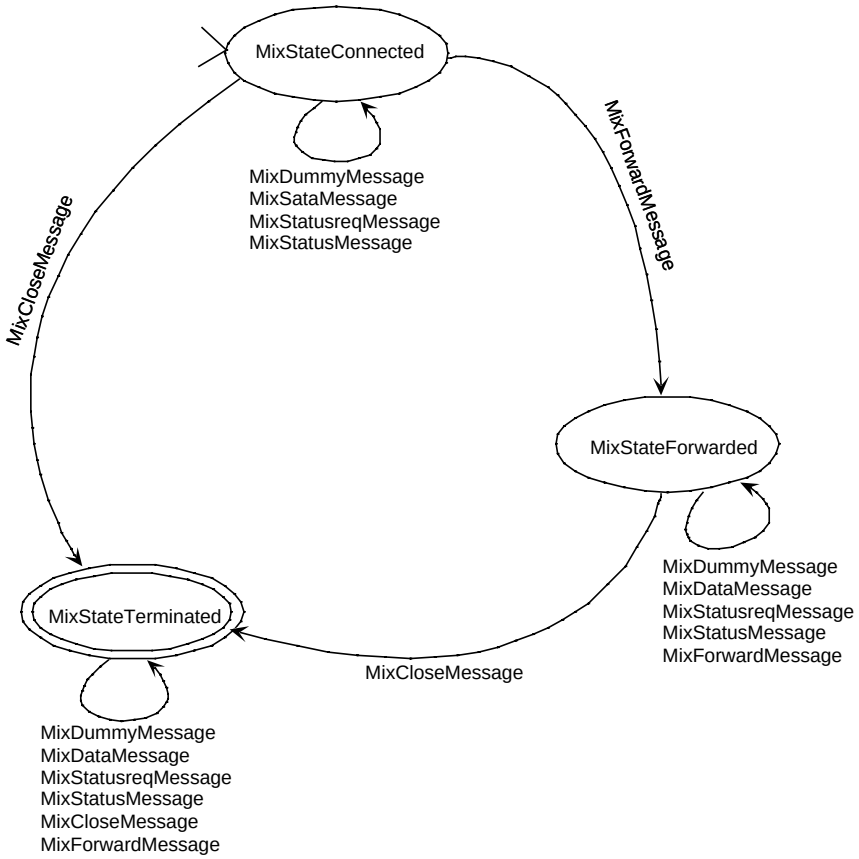

Figure 118: Automaton state transitions.

Once created, the automaton inspects all incoming messages. Depending on the state, it then either processes data or dummy messages being sent through it or executes command messages that tell it to open a forwarding channel, close the channel or do other operations. Figure 118 shows the automaton transition diagram with all possible messages.

As the data transported through the anonymous connection is stored in `MixDataMessage`s, every data object received by the automaton that does not belong to the `MixMessage` class hierarchy is a protocol violation.

Unless an error condition occurs, the automaton processes the incoming messages on both input directions (initiator, responder). Error conditions are reported to the `Coordinator` object.

6.3.4.3 The Coordinator

Within a MIX server, many anonymous connections are managed concurrently. To ensure defenses against some of the known attacks against anonymity systems, cooperation among the connections is necessary. For example, reordering of outgoing messages has to be coordinated to avoid correlation of the MIX servers' input with its output.

Several critical operations exist on anonymous connections: connection establishment, data transmission, connection closure (either on error or on shutdown of the connection).

To inform the MIX server about these events and to coordinate the MIX servers' behavior, a coordinator is informed when those events of interest happen. The coordinator may then decide to wait until shutdown, to open dummy connections or to increase/decrease dummy traffic on specific connections.

Coordination between several channels is done with the `Coordinator` interface and classes that implement it. An interface is used beers coordinator objects can implement many different policies for a MIX server's behaviour. Thus, an object that implement policies only have to consider the coordinator interface as a common, unified access point. The coordinator policies are part of other work, thus the interfaces is used to provide modularity of the MIX policies.

All automaton states call methods of a class implementing `Coordinator` that is written into the machine on initialization time. The coordinator object belongs to the MIX server. Some of the coordinator methods return directions to the automaton about how to react on certain events.

6.3.4.4 A Flow-Oriented View of the Design

Figure 119 describes the functions of the SEMPER anonymous connections following the notation for MIX functions in [FrJP97]. Table 1 shows the options for MIX functions and the abbreviations used in Figure 6.

The SEMPER MIX first decrypts incoming messages at the responder `ComPoint` of its Secure Channel the message came from. Replay prevention is done in the next step by storing packet sequence numbers. The buffering phase follows, and finally, sending data from the buffers in a new order along with dummy message generation is done in the reorder phase.

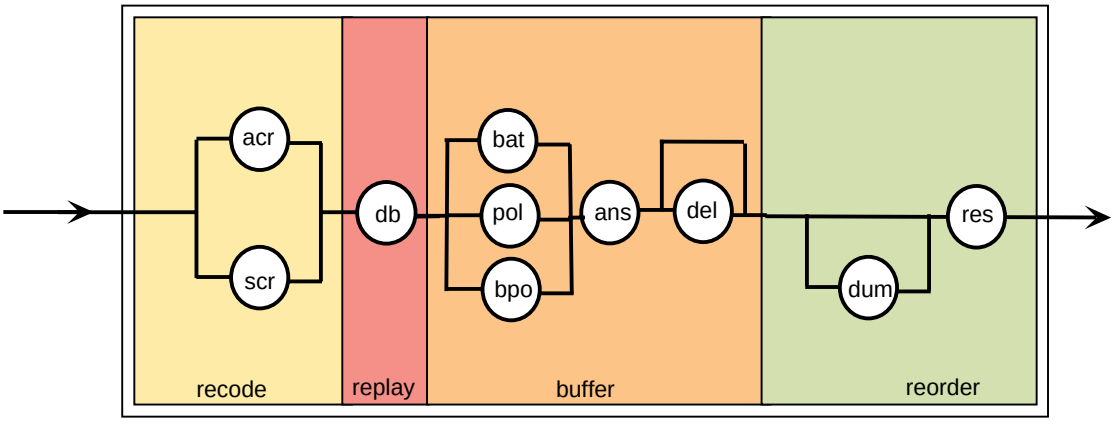

Figure 119: Function of the SEMPER MIX.

Function	Variants	Label
Replay prevention	message storage in database	db
	time stamping of messages	ts

	synchronous stream cipher	ssc
Buffering	Test of sufficient anonymity set	ans
	delay messages	del
	message batch	bat
	message pool	pol
	batch and pool	bpo
Recode messages	store state information	sta
	asymmetric encryption	acr
	symmetric encryption	scr
Reorder output	reordering of message sequence	res
	dummy traffic	dum

Table 9: MIX components in the flow oriented model (as in [FrJP97]).

6.3.5 The Behavior of the MIXed Connections

We now describe the full life-cycle of an anonymous connection in detail. This illustrates the dynamic interaction of the MIXes with the SecComm module for anonymous communication.

The next sections deals with the establishment of anonymous connections. Then, the operation of a channel automaton is described. Finally, the shutdown protocol for anonymous connections are presented.

6.3.5.1 Goal: nested, encrypted channels

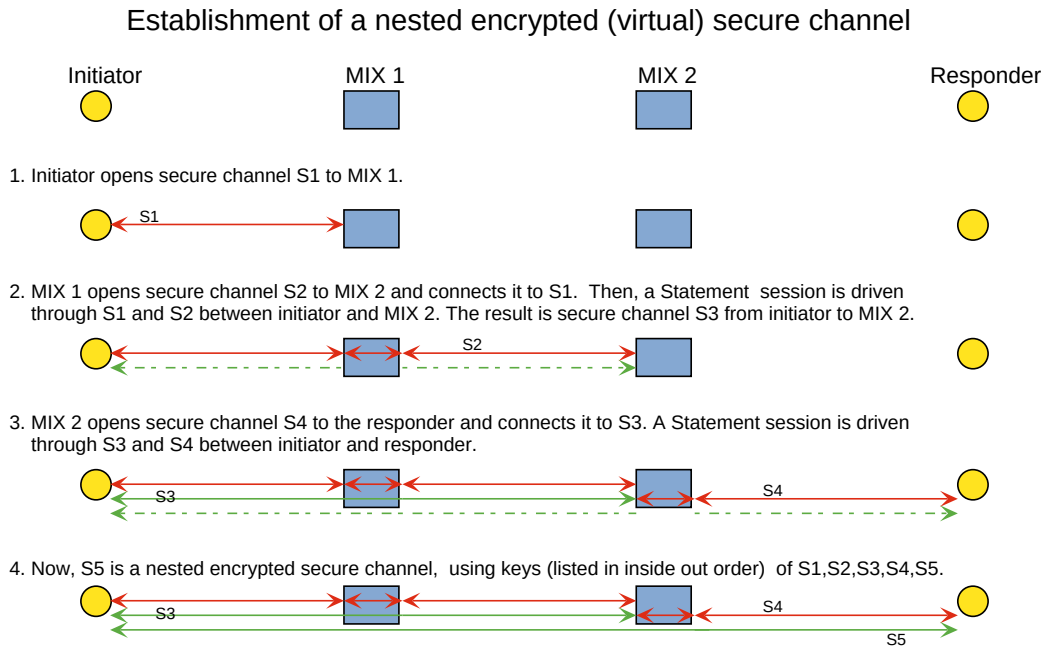

Figure 120: A virtual secure channel using two MIX servers.

Nested encryption is implemented by multiple application of secure channels. As shown in Figure 120, establishing the nested encrypted secure channel is a process that builds bridges of secure channels that are connected to each other inside a MIX. Every time a new bridge adds to the existing channel, a new key negotiation is driven between the initiator and the new host connected by the bridge - but contrary to standard secure channels, no new connection is opened, but the old connection is used instead. This way, a new virtual secure channel between the initiator and the new host is constructed that is using all other secure channels nested into the secure channels established formerly. The created channel from initiator to responder is named “virtual secure channel”.

6.3.5.2 Step 1: Establishment

A user-selected sequence of MIX servers is used to open an anonymous connection. As the basic building block, Secure Channel bridges are used. The initiator first connects to the first MIX of the user’s MIX sequence. Then, after the MIX accepted the connection, a `MixForwardMessage` is sent that contains the address of the next MIX server. The first MIX server now opens a Secure Channel bridge to the address that is contained in the `MixForwardMessage`. More MIXes are addressed in the same way. Finally, the last channel bridge addresses the responder instead of a MIX.

Figure 121 shows MIX protocol messages that open the anonymous connection.

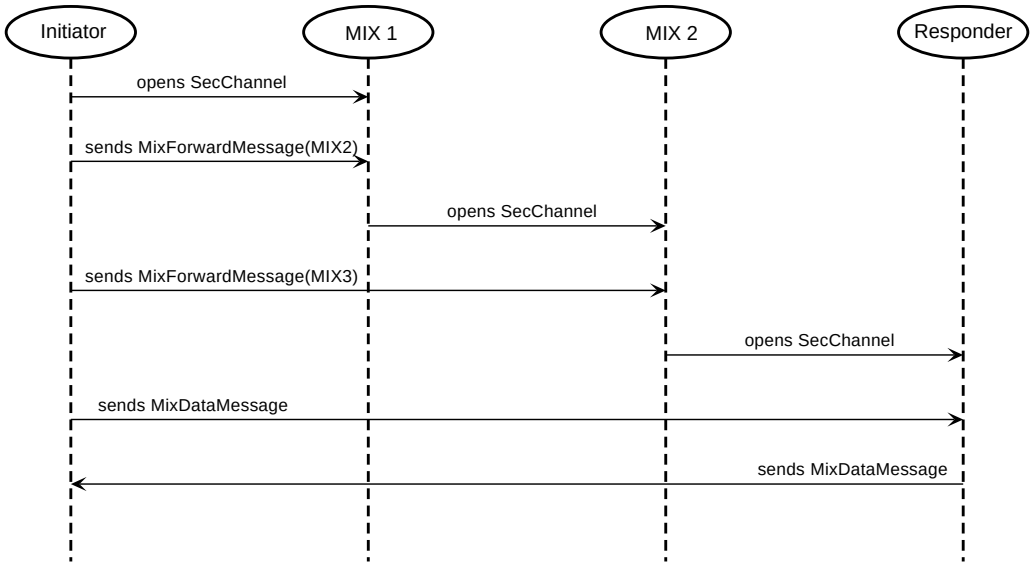

Figure 121: Establishment of anonymous connections.

6.3.5.3 Step 2: Message Processing

Messages are created by either the initiator or the responder of the anonymous connection. Data sent from one connection endpoint to the other has to be packaged into a `MixDataMessage` by the sending endpoint. Control messages addressed to MIX servers can only be sent by the initiator. Due to the layered encryption scheme,

all messages are packaged into `MixDataMessages` in encrypted form. Each message is automatically decrypted one time. Depending on the type of the node, the action is as follows:

- if the decrypting node is a MIX:
  - the decryption result is a `MixDataMessage`: queue it for output.
  - the decryption result is a `MixDummyMessage`: discard it and notify the `Coordinator` (may result in sending a dummy message).
  - the decryption result is a control message: process it.
- if the decrypting node is the responder or initiator:
  - if the decryption result is a `MixDataMessage`: return its content to the application using the anonymous connection.
  - if the decryption result is a control message: process it.

In case the initiator wishes to address one of the participating MIX servers, encryption may only be applied to the appropriate layer so the control message for the MIX is auto-decrypted to clear text at the MIX. Figure 122 shows the message flow through an anonymous connection with the `SecComPoints` that address MIX servers. A `WrapComPoint` in the figure is a class that takes objects as input and outputs `MixDataMessage` objects. It is used to store cyphertext from an encryption step into a `MixDataMessage`.

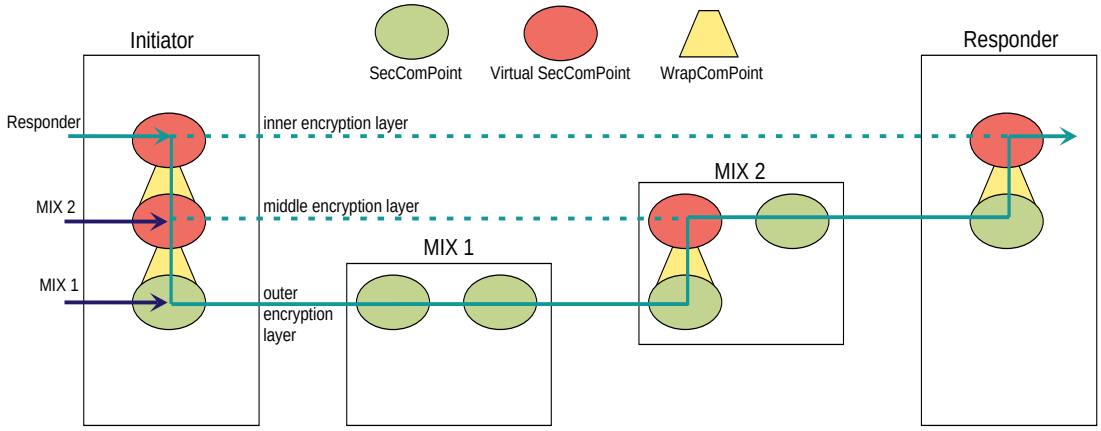

Figure 122: Message flow and encryption.

6.3.5.4 Step 3: Shutdown

Channel shutdown has to be done on all nodes that participate in creating the anonymous connection. As the initiator is the only node that knows about the participating nodes, it is the initiator's responsibility to send `MixCloseMessages` to the other nodes. Any node that wants to shut down the channel must send a `MixCloseMessage` to the initiator. In Figure 123, the responder requests

connection shutdown by sending a message to the initiator, which in turn instructs the two MIX serves to close the connection.

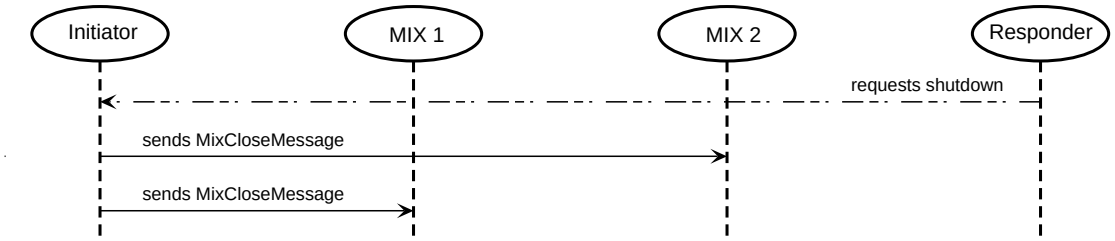

Figure 123: Shutdown of anonymous connections.

On connection timeout, every node is free to shut down its own channel bridges.

6.3.6 Behavior of the MIXes depending on the Policy

Although the current `Coordinator` implementation `CoordLight` does not provide any special functionality to the MIX automata, any future class implementing the `Coordinator` interface has a variety of privileges to control the automaton and the anonymous channel associated with it. This section describes how different policies for automaton and channel behavior can be enforced using a `Coordinator`-compatible class. The following sections describe the ways the `Coordinator` class can control the MIX automata's behavior and thereby control the anonymous connections.

6.3.6.1 Limiting bandwidth

Anonymity increases in the SEMPER scenario when all participants of anonymous communication use a standardized, limited bandwidth for communication. The `Coordinator` interface provides a comfortable and easy way to limit bandwidth without modification of the automaton states by controlling the output message flow. This way, message forwarding happens in intervals defined by the coordinator object.

6.3.6.2 Control of dummy message flow

The flow of dummy messages can be entirely controlled by the coordinator object.

On reception of a dummy message, the automaton grants permission to send a dummy message to the corresponding output channel. If permission is granted, the automaton creates a new dummy message and queues it for output.

6.3.6.3 Control of channel establishment

Opening a new channel is reported to the coordinator object. If for some reason the MIX is unable to open further connections, a `MixFullException` is thrown. If the automaton is ready to open a new outgoing channel, it returns the method call. Before returning, the new channel can be hidden by opening some dummy channel.

6.3.6.4 Control of channel shutdown

Channel shutdown messages are reported to the coordinator object. A `MixCloseMessage` contains linger intervals for the l-channel and the r-channel that can be inspected by the coordinator object. The method call returns a new (or the original) closing message that is then processed by the automaton.

6.3.6.5 Error handling

Errors and exceptions that can happen at various parts of the MIX automata are reported to the coordinator object by invoking the `have_error()` method.

6.3.6.6 Channel statistics

By means of the coordinator object, statistics about every channel of a MIX node can be gathered. These statistics can involve channel usage, channel data rate, errors, average usage of the MIX, current number of channels on a MIX, the capacity of a MIX for new channels and other data.

Obviously, collecting statistic information is dangerous regarding attacks. Therefore, the channel automaton forwards the request to the coordinator. The coordinator then either throws a `MixStatusProhibitedException` or returns a newly created `MixStatusMessage` with some content.

7 The Fair Internet Trader

(G. Papadopoulos / ERC & T. Hecht / FOG)

7.1 A new type of E-commerce: Person to person trade

In the following we describe shortly the concept of a person-to-person e-commerce tool enabling two individual persons execute business transaction spontaneous and securely. This implies the challenging question, what has to be provided to give the human operator the assurance he needs. The motivation into this application was two-fold: On the one hand we noticed that especially with higher values the decisions which drive the business process will be based on individuals but to over knowledge no such tool exist. The high security demands of such an application brought us on the other hand the opportunity for a nice demonstrator where we could easily show the important and distinguishing concepts of the SEMPER architecture.

Please note that person to person trade happens already. E-mail is heavily used to communicate business messages. To ensure confidentiality and authentication of sender and recipient S/MIME and PGP are used. This means that there exist person-to-person communication and by this commerce. But to our knowledge nobody has systematically looked into the security requirements of electronic person-to-person trade. Obviously security in that context does not only mean a secure message transport, but also the multi-party security aspects have to be consider. A main focus is fairness during the trade. This might involve arbitrary combinations of of trust, accountability and maybe also fair-exchanges. The whole process has to provide well-defined liabilities, clear semantics of business messages and the necessary evidence to keep parties accountable. As evaluating all possible business processes and providing a generic solution was clearly beyond our scope we limited ourselves to a prototype highlighting the necessary security components of person-to-person e-commerce and making first step towards a real solution.

In the following we focus only on the design of our prototype. The interested reader is referred to the final report of SEMPER [D13] for more detailed information on the involved requirements and problems of such manual sale.

7.2 The design in brief

In the previous chapter we illustrated the look and feel by describing a usage example.of such a manual sale trading tool. The following paragraph gives some background on the design and implementation design.of our prototype.

7.2.1 The scenario

The "*Fair Internet Trader*" implements a specific business scenario: the selling of knowledge. It is expected that it will be a quite common area of Internet e-commerce and that this scenario applies to a number of other e-commerce practices.

The scenario assumes a three-stage development of the trading process: The "**Negotiation**" phase, the "**Contract Signing**" phase and the "**Fulfilment**" phase.

The scenario is depicted in Figure 124.

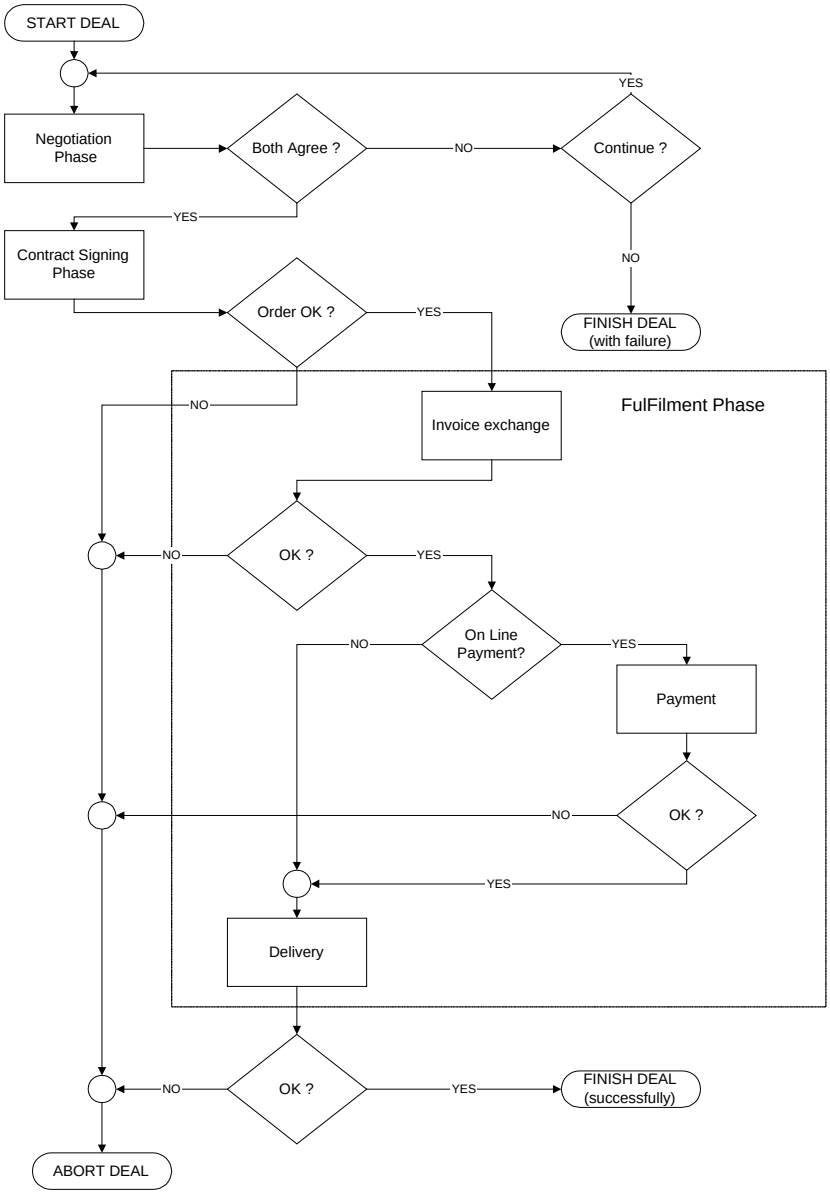

Figure 124.: The "*Fair Internet Trader*" business scenario.

- During the "**Negotiation**" phase, the Buyer and the Seller are negotiating by exchanging a series of "*Request For Offer*" and "*Negotiate Offer*" messages. The negotiation includes the prices, the currency as well as the *Terms and Conditions* of the contract. The messages exchanged at this phase are unsigned.
- During the "**Contract Signing**" phase, both parties agree on an "*Order*" based on a signed "*Offer*" and a signed confirmation. All these messages are non-reputable. Essentially, this is a contract.
- During the "**Fulfilment**" phase, there is the "*Invoicing*" the "*Payment*" and the "*Delivery*" according to the contract signed by both parties in the previous phase. The "*Payment*" fair exchange, the "*Invoice*", the "*InvoiceReceipt*" and the

"*DeliveryReceipt*" carry evidence in case of dispute. Therefore, these are signed messages. The "*Delivery*" does not need to be signed.

7.2.2 Design Overview

The *"Fair Internet Trader*" is composed of three sub-systems implemented by three class hierarchies: the "**Messages**" sub-system, the "**Display**" sub-system and the "**Flow**" sub-system (Figure 125).

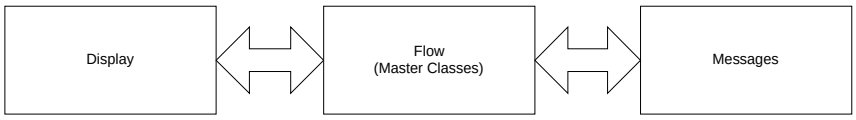

Figure 125: The *"Fair Internet Trader"* overall architecture.

This separation into three parts is proper for this specific area of e-commerce since it allows easier modifications to each sub-system without affecting the others.

7.2.2.1 The "Messages" sub-system

The "**Messages**" sub-system is composed of a class hierarchy derived from the Commerce Layer classes. These messages encapsulate the messages exchanged between the buyer and the seller and they carry the data of their corresponding GUI forms. Their relationship with their corresponding forms of the "**Display**" sub-system is determined and managed by the "**Flow**" sub-system.

There are two main class hierarchies of messages: the "`SignedMessage`" and the "`UnsignedMessage`" class hierarchy. In principle, the messages that carry binding and non-reputable data are signed (They are also expensive in performance terms). The rest, non-binding messages, are unsigned. Their contents have already been described in form of screenshots. (See the above Usage example).

7.2.2.2 The "Display" sub-system

The "**Display**" sub-system is composed of the hierarchy of classes that encapsulate the user interaction forms of the "*Fair Internet Trader*". It uses its own window, although it could have used the TINGUIN. Its design philosophy is to split the display of the "*Fair Internet Trader*" into two sections:

- The "**Business Form**" section which displays the GUI forms of the deal. In other words, it is the section that displays the **contents** and the **controls** that implement the **business** semantics. The "**Business Form**" section is located at the upper part of the display.
- The "**Control**" section, which contains the contents and the controls that manage the "*Fair Internet Trader*" **application**. (E.g. the "*About*" button). The "**Control**" section is located at the lower part of the display.

In addition to the plain display, some other relevant functionality is integrated at the "**Display**" sub-system:

- The "*Show Differences*" function indicates that there is a mismatch between any of the data sent and the data received at the response form, including the contents of the table.
- The "*Security Visualisation*" function highlights the form with a red frame if the form displays a signed message.

A capture of a "*Fair Internet Trader*" form is depicted in Figure 126.

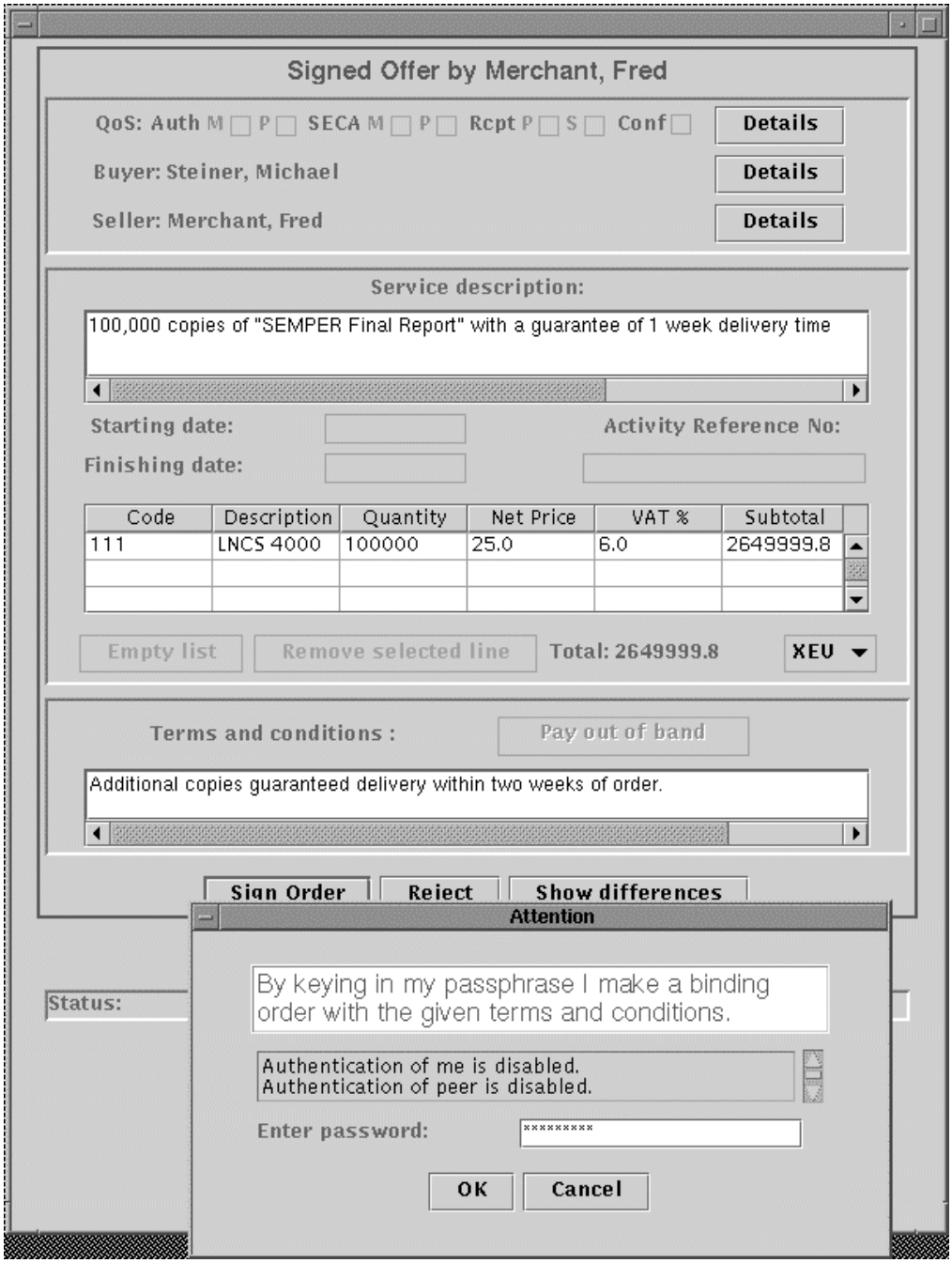

Figure 126: A signed Offer.

7.2.2.3 The "Flow" sub-system

The "**Flow**" sub-system is composed of the hierarchy of classes that encapsulate the business logic at the Buyer's and at the Seller's side.

7.2.2.3.1 The business flow

The flow of the forms for the Buyer and the Seller as well as the events that trigger their creation and their destruction is depicted in Figure 127. Essentially, this is also a state transition diagram for both the client part and the server part of the "*Fair Internet Trader*".

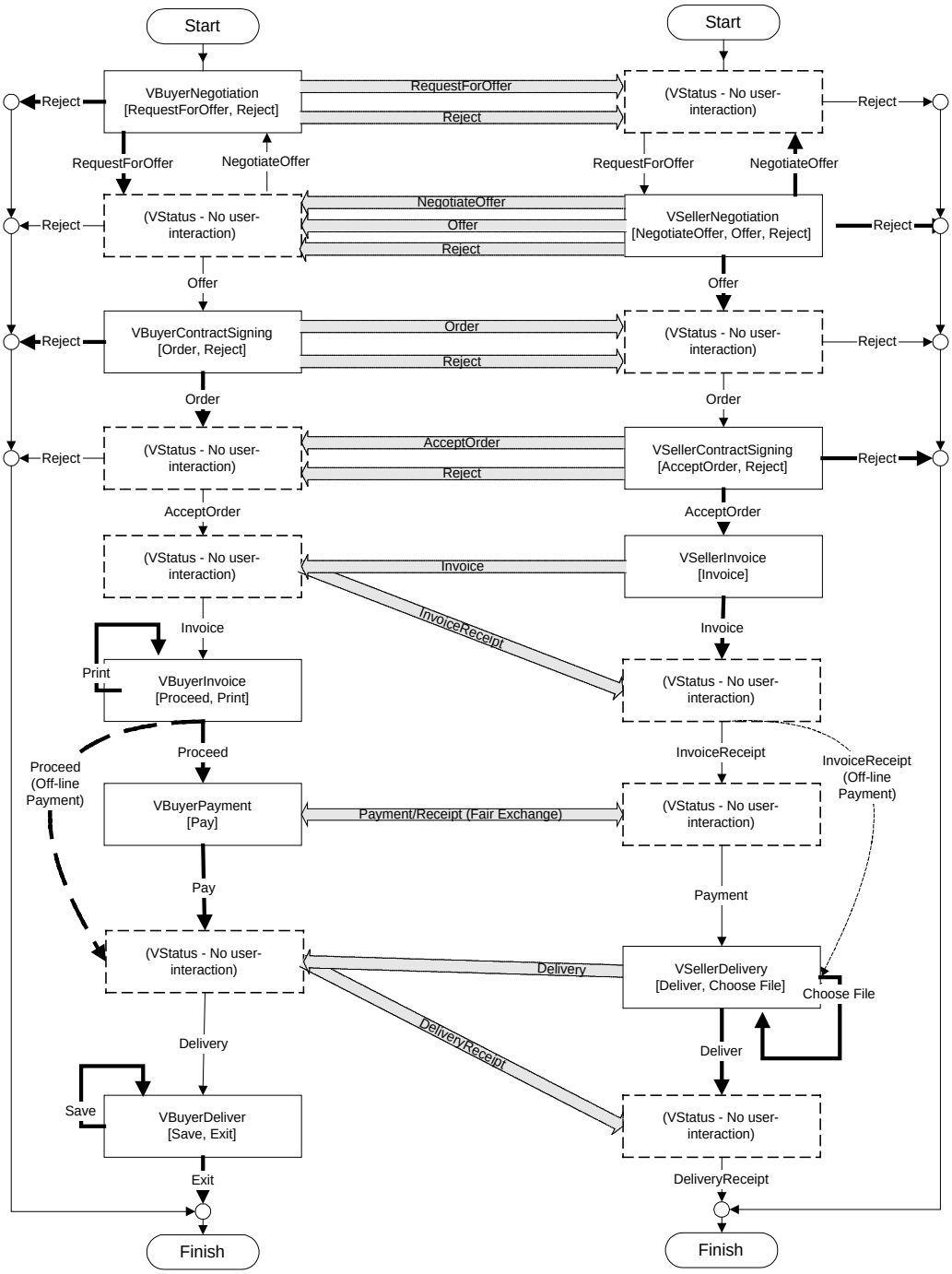

Figure 127: The business flow.

The forms are represented as boxes labelled with their corresponding names and they are raised due to a new message or due to a user interaction.

- The solid-line boxes represent the forms that expect user interaction to proceed. User interaction is validated through the pressing of one button of the form (e.g. Pressing the "*Request For Offer*" button).
- The dashed-lined boxes represent forms that require no user interaction. These forms usually contain informational messages about the status of the deal process (e.g. "*Waiting for Seller's response to the recent Order...*")

The transition between two forms can be either due to a user-driven event or due to a new message arrival. The transition due to a user-driven event is noted with a thick black line whereas the transition due to a message is noted in thin black line. The transition can also trigger a new message to the other party. The messages are noted in grey very thick lines. The transition from a non-input (status) form to a normal user-driven form due to the arrival of a new message is noted in thin black line.

It is noted that this diagram is complete regarding only the business semantics. It does not accommodate exceptional circumstances such as communication errors, software errors and the improper exit from the application by pressing the "Exit" button. A complete diagram would be too complex for the average reader.

All these "improper exit" transitions can occur in every form and their result is assumed to lead the process to the "*Finish*" state.

7.2.2.4 The execution model

The "*Fair Internet Trader*" is a multithreaded application. As such, it is composed of a number of threads that "travel" through objects by executing their methods.

According to the execution model, there is one "*Fair Internet Trader*" thread, which is identical for both the Buyer and the Seller, and two distinct **business flow** threads ("*Flow Threads*"), which differentiate the flow of execution between the Buyer and the Seller. The "*Flow Thread*" is responsible for implementing and running the business flow at the high-level (e.g. send offer, receive order etc.).

Each form runs in its own thread, the "*Form Thread*", which is responsible to carry out the processing of the details of each high level action (e.g. sum the sub-totals into total at the "*offer family*" forms, ensure that the recipient's name and address have been entered at all forms etc.). This split makes the design much simpler and easier to administer. For example, it is quite easy to implement another business scenario by changing only the corresponding buyer's and merchant's "*Flow Threads*".

Figure 128 depicts an overview of the execution model logic. The ellipses represent execution threads. The lines represent special milestones.

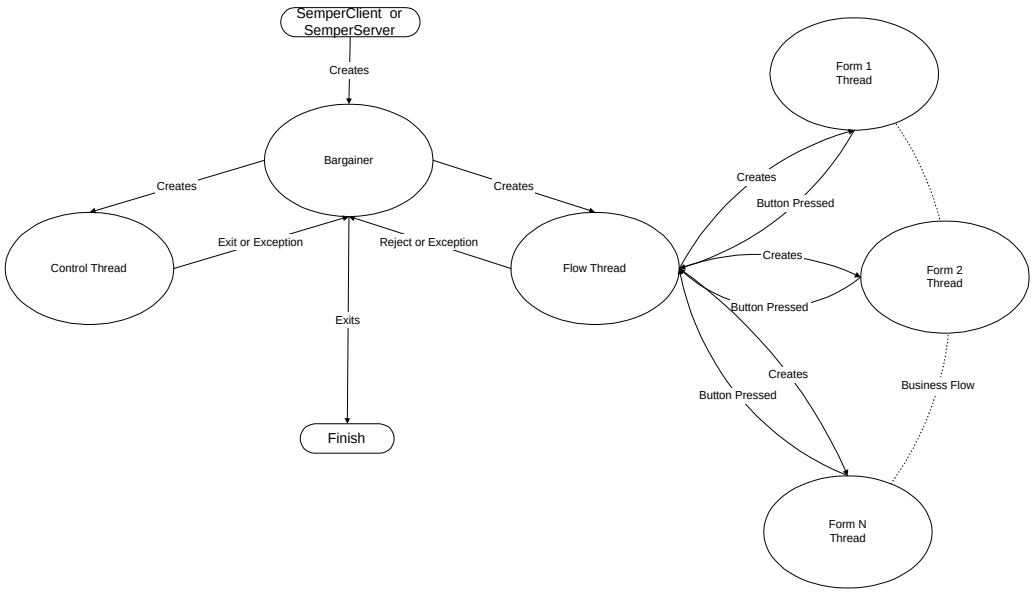

Figure 128: The "Fair Internet Trader" execution model.

Initially, the "`SemperClient`" (or the "`SemperServer`") creates a thread ("`FairInternetTraderThread`") and starts the execution of the "*Fair Internet Trader*" which:

- Creates a new thread ("`FlowThread`") that is concentrated on implementing the business flow semantics. It sends and receives the messages and it drives the creation and destruction of the GUI forms at the "**Business Form**" section of the "**Display**" sub-system. The events originated from the "`FlowThread`" are typically a "*Flow Finish*" or an "*Flow Aborted*" event, which represent a normal and an abnormal finish of the deal respectively. Note that the "`FlowThread`" is different for the Buyer and the Seller since each party implements a different logic. In practice, there are two different classes: the "`BuyerFlowThread`" and the "`SellerFlowThread`". For the sake of simplicity we will call either as "`FlowThread`".
- Creates a new thread ("`ControlThread`") at the "**Control**" section of the "**Display**" sub-system. The events originated from the "`ControlThread`" are typically the pressing of the "*Exit*" or the "*About*" button. Note that the "**Control Form**" is always available even while waiting for the other party's response.

After the creation of the above threads, the "*Fair Internet Trader*" thread sleeps waiting for an event from either thread.

The "*Fair Internet Trader*" execution model in the time domain is depicted in (Figure 129). The solid lines represent the flow of execution whereas the dashed lines represent waiting state.

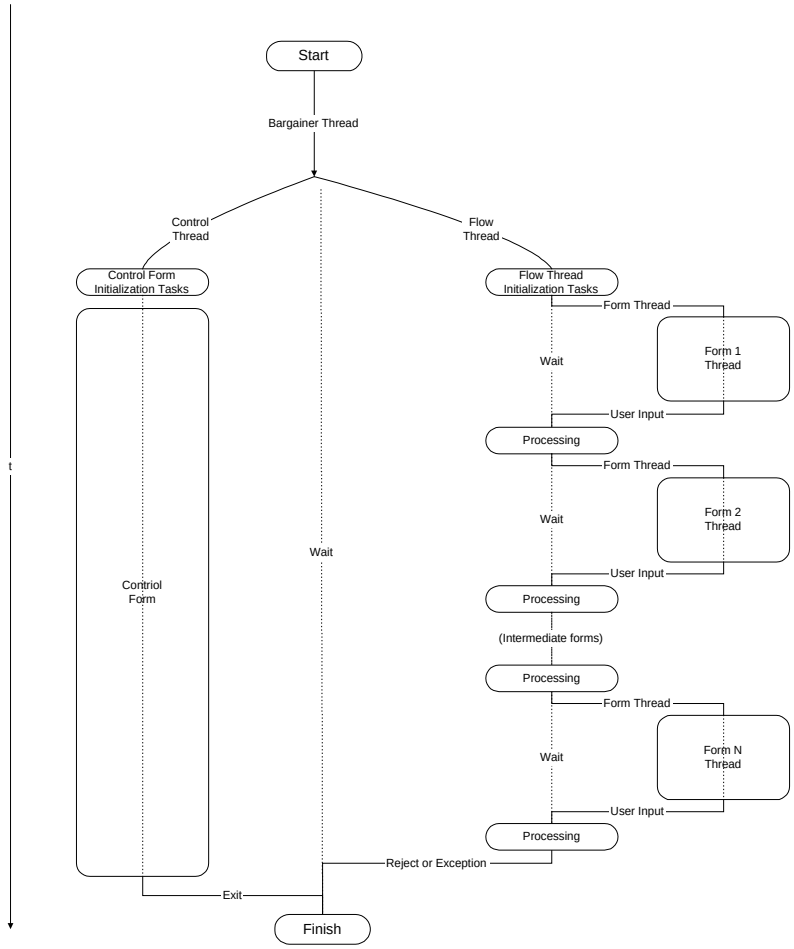

Figure 129: The Fair Internet Trader "run" execution model.

7.2.2.5 The business-context system

It is worth mentioning the business-context module, which is implemented as a part of the flow sub-system. The business-context is the repository of all the business-critical data such as the price of the offer, the currency etc.

The business rule section (which is a part of the "`FlowThread`") interacts with the business context while applying and checking the rules.

It is reminded that the business rule adopted for the "*Fair Internet Trader*" was to indicate differences between the previous user's "proposal" (e.g. a "`RequestForOffer`" during negotiations) and the other party's response (e.g. "`Offer`").

The core structure of the business-context mechanism is currently a HashTable, which holds all the information that the user sends to the other party. This information is updated whenever the user sends a form to the other party, possibly overwriting some previous values.

Once the response is received, the corresponding values of the form and the HashTable are compared. If there are any differences, a report is generated containing both the value of the HashTable ("*old*" value) and the form ("*new*" value) for the fields that differ.

The lifetime of the business-context is throughout the entire Deal. Therefore, its contents can be used at the fulfilment phase. For example, during negotiations both parties agree on an amount and currency. This information is stored at the business-context. Later, when payment occurs, the amount and the currency paid can be checked against the corresponding values that had been stored during the Contract Signing phase.

The HashTable structure that is currently used to implement the business context may change in the future to a "`BusinessContext`" class.

7.3 Open issues and lessons learned

7.3.1 The universal "plug-n-play" approach

During the analysis and the specification of the "*Fair Internet Trader*" it became evident that the "*universal plug-n-play"* approach was towards the wrong direction. Unless the applications drive the business cases, it is impossible to construct a single application that covers every business case. This is due to the fact that the possible number of business cases is a **countable** but **infinite** set. It is, therefore, impossible to know a priori all the requirements at the specification time and thus to integrate them at the development.

7.3.2 The interface to a business application

According to the previous paragraph, each class of business cases requires its own business application. The rules that should be encapsulated within it should be expressed in a way that the computer can understand. In the current approach, this is done through the Java programming language. However, is hard for the average potential user of SEMPER (e.g. merchant) to express his own rules by this way. A friendlier interface would make easier to disseminate SEMPER as a medium of performing e-commerce.

SEMPER is currently providing to the business application writer a **design** service. However, the application writers would prefer to be provided a **run-time** service. For example, it would be easier to express the business rules with a scripting (interpreted) language. This would make SEMPER friendlier to its potential users. We strongly believe that this approach will be a standard in the future. Therefore, more effort should have been placed on the interface of SEMPER to the business applications. However, the resources were limited and this remains an open issue.

8 Conclusions

This report has described the design and implementation of the *SEMPER* architecture for electronic commerce. The layered architecture is designed to support a wide range of electronic commerce applications and payment protocols while anticipating the need to support more complex applications and security requirements in the future. As the report shows, *SEMPER* will use standard libraries and tools as modules to achieve rapid development and platform portability. Various trials have shown [D05, D16, D12] that we are on the right track with of our architecture. Three years and small-scale trials are though clearly not sufficient to give the final answers for such a waste problem space. The future will tell how much we could lead the world towards secure and fair electronic commerce through our architecture, design and experience gained in the trials.

Appendix A: References

AASW98 J.L. Abad Peiro, N. Asokan, M. Steiner, M. Waidner, “Designing a Generic Payment Service” in IBM Systems Journal, 37(1): 72-88, 1998.

Adams96 C. Adams: „The Simple Public-Key GSS-API Mechanism (SPKM)“, RFC 2025, Jan 96

AJSW97 N. Asokan, P. Janson, M. Steiner, M. Waidner, “State of the Art in Electronic Payment Systems” in IEEE Computer, 30(9):28-35, September 1997.

AsScWa97 N. Asokan, Matthias Schunter, Michael Waidner: “Optimistic Protocols for Fair Exchange”; 4th ACM Conference on Computer and Communications Security, Zürich, April 1997, 6-17.

AsScWa98 N. Asokan, Victor Shoup, and Michael Waidner. Asynchronous protocols for optimistic fair exchange. In Proceedings of the IEEE Symposium on Research in Security and Privacy, Research in Security and Privacy, Oakland, CA, May 1998. IEEE Computer Society Press. Pp 86-99.

BahNar96 A. Bahreman and R. Narayanaswamy, “Payment Method Negotiation Service,” in Proceedings of the Second USENIX Workshop on Electronic Commerce, USENIX, Oakland, CA , Nov. 1996, 299-314

Bahrem96 A. Bahreman, “Generic Electronic Payment Services: Framework and Functional Specification,” in Proceedings of the Second USENIX Workshop on Electronic Commerce, USENIX, Oakland, CA , Nov. 1996, 87-103

Bauspi96 Bauspieß (ed.): „MailTrusT Spezifikation, Version 1.1“, 12/96

Booch94 G.Booch; “Object Oriented Analysis and Design”, Addison -Wesley, 1994.

BürPfi89 Holger Bürk, Andreas Pfitzmann: “Digital Payment Systems enabling Security and Unobservability”; Computers & Security 8/5 1989, 399-416

CAPI96 Microsoft Corporation. “Application Programmer's Guide Microsoft CryptoAPI”, January, 1996. See also http://www.microsoft.com.

CDSA97 Open Group Technical Standard: “Common Security: CDSA and CSSM”, December 1997.

CDSA98 Intel Corporation. “Common Data Security Architecture Specification”, February 1998. See also http://www.intel.com

CFN88 D. Chaum, A. Fiat and M. Naor: “Untraceable Electronic Cash”. Proceedings of Crypto’88. Springer-Verlag, LNCS 403. 1987. 319-327.

Chau81 D. Chaum: “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms” in Communications of the ACM 24/2 (1981) 84-88.

D03 *SEMPER* Consortium: “Basic Services: Architecture and Design”, SEMPER Deliverable D03; La Gaude, 1996

D05 *SEMPER* Consortium: “Survey Findings, Trial Requirements, and Legal Framework -- Results from First Year of Project SEMPER”, SEMPER Deliverable D05; La Gaude, December 1996

D13 *SEMPER* Consortium: “Final Report of Project SEMPER”, SEMPER Deliverable D13; To be published.

D14 *SEMPER* Consortium: “Architecture of Payment Gateway”, SEMPER Deliverable D14; La Gaude, November 1996

D15 *SEMPER* Consortium: “New Payment Instruments Prototype”, SEMPER Deliverable D15; La Gaude, December 1997

DES80 NBS FIPS Pub 81: „DES Modes of Operation“, National Bureau of Standards, U.S. Department of Commerce, Dec 1980

DES88 NBS FIPS PUB 46-1: „Data Encryption Standard“, National Bureau of Standards, U.S. Department of Commerce, Jan. 1988

DifHel76 W. Diffie, M. Hellman: „New Directions in Cryptography“, IEEE Transactions on Information Theory, v. IT-22, n. 6, Nov. 1976, pp. 644-655

Dobber96 H. Dobbertin: „Welche Hash-Funktionen sind für digitale Signaturen geeignet?“, Tagungsband „Digitale Signaturen“, Vieweg-Verlag, 1996, ISBN 3-528-05548-0, pp. 81-92

Dobber97 Dobbertin: „Digitale Fingerabdrücke - Sichere Hashfunktionen für digitale Signaturen“, DuD 2/97, Vieweg, pp. 82-87, 1997

DoBoPr96 Dobbertin, A. Bosselaers, B. Preneel: „RIPEMD-160: A strengthened version of RIPEMD“, Fast Software Encryption, Cambridge Workshop, LNCS 1039, Springer, 1996, pp. 53-69
corrected version via ftp://esat.kuleuven.ac.be/pub/COSIC/bosselae/ripemd/

DSS94 National Institute of Standards and Technology (NIST): “Digital Signature Standard (DSS)”. Federal Information Processing Standards Publication 186 (FIPS-186), 19th May, 1994

EDIFACT Electronic Data Interchange for Administration, Commerce and Transport - (EDIFACT) - Application Level Syntac Rules. Part 5. ISO 9735 -5.

Flanag97 David Flanagan; “JAVA in a nutshell(JAVA 1.1)” 2nd edition, O'Reilly & Associates, Inc,1997.

FowSco97 Martin Fowler and Kendall Scott, “UML Distilled: Applying the Standard Object Modeling Language,” Addison-Wesley Longman Inc., 1997 (ISBN 0-201-32563-2)

FrJP97 E. Franz, A. Jerichow, A. Pfitzmann: “Systematisierung und Modellierung von Mixen” , Verläßliche IT-Systeme, GI-Fachtagung VIS '97, DuD Fachbeiträge, Vieweg, Braunschweig 1997, 171-190.

GCS96 X/Open Company Limited. “Generic Cryptographic Service API (GCS-API)”. January, 1996. See also http://www.xopen.com.

GHJV95 E. Gamma, R. Helm, R. Johnson, J. Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, Reading, Mass., 1995.

GlKoWi98 P. Glöckner, S. Kolletzki, M. Wichert: „Signed Unique References“, to appear in the proceedings of JENC8

GoJoSt96 James Gosling, Bill Joy, and Guy Steele: “Java Language Specification”., section 15.11, *Method Invocation Expressions* http://java.sun.com/docs/books/jls/html/-15.doc.html#20448, published in August 1996.

Gold98 Theodore Goldstein. “The Gateway Security Model in the Java Commerce Client. Sun Microsystems.” February 1998.

GoRS96 D. Goldschlag, M. Reed, P. Syverson: “Hiding Routing Information” Information Hiding, Preproceedings, Workshop, Isaac Newton Institut, University of Cambridge, UK, 30 May - 1 June 1996, 125-142.

GrKn97 M. Grand, J. Knudsen: “Java: Fundamental Classes Reference”, O'Reilly, Reading, Cambridge, 1997.

GrRe93 J. Gray, A. Reuter: “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann Publishers Inc., San Francisco, 1993.

GüTs96 C. Gülcü, G. Tsudik: “Mixing E- mail with BABEL”; ISOC Symposium on Network and Distributed System Security, 1996, 2-16.

Hogref89 Dieter Hogrefe: Estelle, “Lotos und SDL - Standardspezifikationssprechen für verteilte Systeme”, Springer Compass, Springer-Verlag, Berlin 1989.

Holzne97 Steven Holzner; “Mastering Netscape IFC”, SYBEX Inc, USA.

IBMCA IBM, “IBM Registry SET User’s Guide and Reference,” Jan 1997.

IBMPG Advantis, IBM, “Payment Gateway Application. Overview Document,” Version 2.0, 1997

IBMSET IBM, “IBM SET White Paper,” http://www.software.ibm.com/commerce/payment/set-paper.html, Jul 1998.

IPSEC S. Kent, R. Atkinson: “Security Architecture for the Internet Protocol”, Internet Draft, <http://search.ietf.org/internet-drafts/draft-ietf-ipsec-arch-sec-05.txt>, May 1998

JMPP97 A. Jerichow, J. Müller, A. Pfitzmann, B. Pfitzmann, M. Waidner: ” Real-Time Mixes: A Bandwidth-Efficient Anonymity Protocol”. In IEEE Journal on Selected Areas in Communications, special issue "Copyright and privacy protection", April 1998.

JNI1.1 JavaSoft, “Java Native Interface Specification,” http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/index.html, Release 1.1, Nov 1996.

KGPS96 Steven Ketchpel et al, “U-PAI: A Universal Payment Application Interface,” in Proceedings of the Second USENIX Workshop on Electronic Commerce, USENIX, Oakland, CA , Nov. 1996, 105-121

Knudse98 J. Knudsen. “JAVA Cryptography”. Published by O’Reilly, The JAVA Series. May, 1998

Kollet96 S. Kolletzki: „Secure Internet Banking with Privacy Enhanced Mail“, Computer Networks and ISDN Systems 28 (1996) 1891-1899

Lai92 X. Lai: „On the Design and Security of Block Ciphers“, ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992

LDAP University of Michigan Information Technology Division: „Windows Binary Distribution (contains LDAP32.DLL, LIB and header files)“ ftp://terminator.rs.itd.umich.edu/x500/ldap/windows/

LDAPa University of Michigan Information Technology Division: „LDAP servers, client library and sample text based UNIX clients“ ftp://terminator.rs.itd.umich.edu/x500/ldap/ldap-3.3.tar.Z

Linn93 J. Linn: „GSS API“ RFC's 1508 and 1509 (C-bindings), Sep. 93

LMWF94 N. Lynch, M. Merritt, W. Weihl, A. Fekete: “Atomic Transactions”, Morgan Kaufmann Publishers Inc., San Mateo CA, 1994.

Lynch96 Nancy A. Lynch: “Distributed Algorithms”; Morgan Kaufmann, San Francisco 1996.

MANDATE TEDIS II report. “MANDATE”.

MAS7304 J. P. Boly et al. “Extended Protocols”. Deliverable, MAS7304 of ESPRIT Project 7023, CAFE. June 1995.

MeOV97 A. Menezes, P. van Oorschot, S. Vanstone: “Handbook of Cryptography”, CRC Press Inc., 1997.

Neuman95 Peter G. Neumann: “Computer Related Risks”; Addison Wesley - ACM Press, Reading Massachusetts 1995.

PEM93 J. Linn: „Message Encryption and Authent. Procedures“ RFC 1421, Feb 93
S. Kent: „Certificate Based Key Management“ RFC 1422, Feb 93
D. Balenson: „Algorithms modes and identifiers“ RFC 1423, Feb 93
B. Kaliski: „Key Certification and related Services“ RFC 1424, Feb 93

Pfit90 A. Pfitzmann: “Diensteintegrierende Kommunikationsnetze mit teilnehmerüberprüfbarem Datenschutz” in IFB 234, Springer-Verlag, Berlin 1990.

Pfitzm96 Birgit Pfitzmann: “Digital Signature Schemes — General Framework and Fail-Stop Signatures”; LNCS 1100, Springer-Verlag, Berlin 1996.

PfPW86 A. Pfitzmann, B. Pfitzmann, Michael Waidner “Technischer Datenschutz in diensteintegrierenden Digitalnetzen - Warum und wie?” in Datenschutz und Datensicherung DuD 10/3 (1986) 178-191.

PfPW91 A. Pfitzmann, B. Pfitzmann, Michael Waidner, “ISDN-MIXes – Untraceable Communication with Very Small Bandwidth Overhead” in Proc. Kommunikation in verteilten Systemen, IFB 267, Springer-Verlag, Berlin 1991, 451-463.

PfSsWa98 Birgit Pfitzmann, Matthias Schunter, Michael Waidner: “Optimal Efficiency of Optimistic Contract Signing”; 17th Symposium on Principles of Distributed Computing (PODC), ACM, New York 1998, 113-122.

PfWa87 A. Pfitzmann, M. Waidner: “Networks without user observability” in Computers & Security 6/2 (1987) 158-166.

PKCS10 RSA Laboratories Technical Note: "PKCS \#10: Certification Request Syntax Standard", November 1993.

PKCS11 RSA: „PKCS#1-#11: Public Key Cryptography Standards“, http://www.rsa.com, revised Nov. 1993

PKIX “Public-Key Infrastructure (X.509) (pkix)"

PPSW97 Andreas Pfitzmann, Birgit Pfitzmann, Matthias Schunter, Michael Waidner, “Trusting Mobile User Devices and Security Modules,” in IEEE Computer, 30(2):61-68, February 1997.

ReRu97 M. K. Reiter, A. D. Rubin: “Crowds: Anonymity for Web Transactions” in DIMACS Technical Report 97-15, April 1997, revised August 1997, <http://www.research.att.com/~reiter/papers/dimacs-tr9715-revised.ps.gz>

RiShAd78 R. Rivest, A. Shamir, L. Adleman: „A method for obtaining Digital Signatures and Public-Key-Cryptosystems", Communications of the ACM, v.21,n.2, Feb 1978, SS. 120-126

Rivest92 R. Rivest: „The MD4 Message Digest Algorithm“, RFC 1320, Apr. 1992

Rivest92a R. Rivest: „The MD5 Message Digest Algorithm“, RFC 1321, Apr. 1992

Schnei96 Bruce Schneier: “Applied Cryptography: Protocols, Algorithms, and Source Code in C”; John Wiley & Sons, (2nd ed.) New York 1996.

SECUDE97 GMD: „SECUDE 5.1 - Hyperlink Documentation“, 1997, http:www.darmstadt.gmd.de/secude/doc/index.htm

SET1.0 Mastercard, Visa. “SET Secure Electronic Transactions Protocol,” version 1.0, May 1997.

SETSP1.0 Mastercard, VISA, “External Interface Guide to SET Secure Electronic Transactions,” http://www.setco.org/set_specifications.html, Sep 1997.

Shneid87 Ben Shneiderman; “Designing the User Interface: Strategies for effective Human-Computer Interaction”, Addison-Wesley Publishing

SHS National Institute of Standards and Technology (NIST): „Secure Hash Standard“ (SHS). Federal Information Processing Standards Publication 188-1 (FIPS-188-1)

SSL A. Freier, P. Karlton, P. Kocher: “The SSL Protocol Version 3.0”, Netscape Communications Corporation, March 1996

SSLeay E. Young: Documentation of the SSLeay library, available from <ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL>

Sun98 “The Java Wallet Architecture White Paper—An Open, Extensible Framework for Electronic Commerce in the Java Programming Language.” Sun Microsystems. March 1998.

TEDIS93 TEDIS II B7 report. “Security in Open Environments”. 1993

Thimbl90 Harold Thimbleby; “User Interface Design”, ACM Press, Addison-Wesley Publishing Company, 1990.

X509 “Information Technology - Open System Interconnection - The Directory - Part8: Authentication Framework”. CCITT Recommendation X509.

X509v3 ISO/IEC 9594-8: "Information technology -- Open Systems Interconnection--The Directory: Authentication Framework", 1995.

Appendix B: Glossary

Acquiring bank
The bank used by the recipient of electronic money.

Active deal
Denoting a commerce deal that is accessed in the active mode.In this mode it is possible to add new commerce transactions to the deal.All participants of the deal must be willing to co-operate for a deal to be accessed in the active mode.See Section 4.1.1.2.

Anonymously transferable standard values
A payment scheme proposed in [BürPfi89]. It is based on signatures and anonymous communication.

Audit trail
Logging information which is the basis for an audit of the use of a service or system. The audit trail may contain non-repudiation tokens to resolve disputes.

Authentication
A proof of identity. This proof can be related to a message being sent meaning that a proof of origin is sent along (see also digital signatures and message authentication codes).

Authorisation
The process of determining the rights associated with a particular principal or role, e.g. rights to access data or services on a computer system.

Basic Communications
provides an interface that makes application development independent of the underlying communication protocols.

Business session
A business session is a sequence of commerce layer primitives that operate on the same context, including all case decisions, e.g., for error handling. The primitives could support, e.g., offer, order, invoice and payment.

Certificate
A certificate is a digitally signed statement about a person. *SEMPER* distinguishes three types of certificates: key certificates linking a public key to a person, attribute certificates linking a particular property to a person and hybrid certificates linking both a key and an attribute to a person.

Certification Authority (CA)
The CA is responsible for issuing and maintaining certificates belonging to users. In the certification procedure a user will at some point receive a valid certificate. It is essential that the CA corresponds to the RA in order to check the identity of the user before a certificate can be issued.

Channel Multiplexing
allows each address in the address space of a higher-level protocol using the communication block be used for communication between more than on pairs of entities.

Commerce deal
The commerce transaction service introduces the notion of a commerce deal as a representation of a business context.A deal comprises a representation of an association between the participants, a history of commerce transactions exchanged between the participants and any private data stored by the participants.See Section 4.1.1.2.

Commerce layer
The commerce layer is the collection of services that interface to the business applications in *SEMPER*.It comprises the commerce transaction service, the business application framework and the dispatcher.

Commerce transaction service
This is the service that *SEMPER* business applications use to access the *SEMPER* services and realise business scenarios.In addition to providing services for building business applications, the commerce transaction service also maintains the user's security policy.

Commerce transaction
Commerce transactions represent the service primitives in the commerce transaction service.A commerce transaction encapsulates a protocol to be executed between the participants of a commerce deal.This protocol can be the simple delivery of a message or it can be a complex payment protocol.Commerce transactions must always exist in the context of a commerce deal.

Conditional access
Access to a service, which is restricted to entities having certain properties or rights.

Confidentiality
Protecting a message against eavesdropping such that it is only meaningful to the intended set of entities.

Container
A data structure used in *SEMPER* to transfer or exchange information and payments. It is a data type structered in the form of a tree. The leaves are nodes specifying or containing (protected) information or payments. Internal nodes contain security attributes.

Context
The context is the local state of an entity. The context evolves from the configuration and preferences set by the user and the results of negotiating parameters with other entities.

Credentials
The personal information provided by a person or institution when registering. (In the literature, the term credential is also sometimes used instead of hybrid certificate).

Customer
The role played by someone wanting to buy something at a merchant.

Decision procedure
A procedure which is used to solve disputes between participants - usually based on audit trails. The decision procedure defines which messages serve as evidence of what and how they are verified.

Design pattern
A design pattern [GHJV95] is a simple and elegant solution to specific problems in object-oriented design.

Device
A physical object, such as a personal computer. Usually, a device belongs to the player (or set of players) who relies on it.

Digital signature
A digital signature is an electronic counterpart of handwritten signatures. It is verified against a public key, and if the signature system is secure, this serves as a proof that the signed message originates from someone knowing the corresponding secret key

Directory Authority (DA)
The DA is responsible for making information about certificates available so that whoever is interested in communicating with a user can easily retrieve his certificate.

Dispute handler
See Exception handler.

Electronic money
Electronic money is the electronic counterpart to the conventional money.

Entity
Any active element in the *SEMPER* architecture is called an entity. Thus an entity could be a service block, a manager, a module or a collection of these.

Exception handler
An exception handler has to resolve an exception raised by a party. In a first approach the exception handling bases on the assumption that all parties are honest. If this optimistic approach fails, the parties are in a dispute and a more pessimistic approach must be used, usually involving arbiters and finally courts. Exception handling requires all parties to keep sufficient audit trails. For real disputes, the audit trail has to contain evidence, e.g. non-repudiation tokens.

Exchange

An exchange is a protocol whereby a number of parties can exchange business items (payment, information, vouchers, etc.) . In the simplest case these are two transfers.

Extended finite state machine

An extended finite state machine [Lynch96] is a finite state machine[27] where each message and each state may contain a finite number of variables and the state transitions not only change to another state out of the finite set but also perform arbitrary computations on the variables contained in the state, the message received, and the message sent (this is the "extension"). The purpose of modeling the infinite state of a Turing machine as a combination of a small finite set of states together with an assignment of values to a given finite set of variables is to enable more intuitive protocol specifications: Each of the finite number of states and messages has an intuitive meaning whereas the variables make the model as powerful as Turing machines.

External Interface

The interface of the service block to the upper layers. This is also often called an Application Programming Interface (API).

Fair exchange

A special type of exchange in which each party beforehand specifies what he wants to send and receive. The exchange is fair if the participant is guaranteed to receive an item fulfilling this specification if he sends the promised item.

Generic Payment Service Framework

A SEMPER framework enabling electronic commerce applications to use a variety of payment systems. (within SEMPER, also known as the payment block)

Hash functions

A function used to compute a fixed length digest of any bit string. Unless stated otherwise a hash function is always assumed to be *collision-resistant,* meaning that it is infeasible in practice to find two different bit string with the same digest.

Identification

A process whereby an entity proves its identity.

Integrity protection

The protection of information against changes by intruders (tampering)

---

27 A finite state machine is a tuple (S, M, s_0, F, d) where S is a finite set of states, M is a finite set of inputs (i.e., messages), s_0 is the starting state, F are the final states, and $d: SxM \rightarrow SxM$ is the transition function which get a state and a message as input and produces a follow-up state and a message to be sent as output.

Internal Interface
The interface to the service manager to the service modules. This is also often called a Service Provider Interface (SPI).

Issuing bank
The bank of the payer in an electronic payment systems.

Layer
A (virtual) collection of entities in all devices that perform functions of a similar degree of abstraction.

Layer Interface
The interface between two layers. It is the union of all interfaces of the entities of this layer.

Non-repudiation tokens
A non-repudiation token is a proof that can later be used as a proof that a certain event took place. These tokens are often necessary in order define a decision procedure. Of certain interest are non-repudiation of origin proving that a message originated from a certain entity and non-repudiation of delivery proving that a message has been delivered.

Payment Instrument
An instance of one player's component of a payment system.

Payment Manager
Overall controller of an instance of the Generic Payment Service.

Payment system provider
A party (such as a financial institution) that makes a payment system available.

Payment System
A collective name for one "way" of making a value transfer, consisting of protocols, contractual agreements, and data structures

Payment Transaction Browser
User application to examine transaction records.

Peer entity
Peer entities are two entities of the same layer but located on different devices. Usually, they interwork to provide a certain service.

Player
A player is a real-world person or body participating in the electronic marketplace, e.g. a buyer, a seller or a third party like a payment system provider, a notary service, etc.

Protocol
A description how a service is provided by means of interactions of peer entities via lower layer services.

Purse Management Application
User application to create and manage purses in the Generic Payment Service

Purse
Representation of a payment instrument within the Generic Payment Service Framework.

Registration Authority (RA)
The RA is responsible for the correctness of the user's identity. It checks the attributes a user wants to certify. The importance of an RA varies with the level of assurance needed. In the strongest case, the role of the RA will be filled in by a notary public and the applicant has to show up in person and manually sign a contract with the RA.

Registration
A procedure by which a participant (usually a real person) registers his or her credentials with a so called registration authority. In return the participant gets a certificate as an electronic proof of the registration.

Revocation
The process of withdrawing the validity of a certificate. A certificate may be revoked by the certification authority. The reason could be that the key is compromised, lost, or was registered by an impostor or that the user abused the granted rights.

Role
The function of an entity in a certain transaction. All transaction protocols are specified for roles, but performed by entities 'playing' the corresponding roles. Roles are also needed for access control. The access rights usually depend on both the entity and its role.

Security attributes
Security attributes describes the security level that a certain transfer or exchange should provide. Examples are confidentiality, non-repudiation and anonymity.

Service Access Point
The access point for an entity to access the services of any lower layer.

Service block
A service block consists of a manager and a number of modules (possibly 0).

Service interface
A subset of the layer interface, consisting of service access points and a syntactical definition of the service primitives that can be exchanged via these service access points.

Service manager
A service manager provides a common interface to the modules of the service blocks (the *external interface*), plus methods for negotiation and selection of an appropriate module. In addition the service manager provides services providing

robustness against fault tolerance. The manager is always part of the *SEMPER* software.

Service module
The service module implements the services provided by the service block. It is expected that modules will be external (i.e., modules will only be implemented as part of *SEMPER* if necessary). The service module must support the *internal interface* in order to be used by the manager.

Service
Services are offered by the joint functionality of a layer to its upper interface. One layer can offer several services, e.g. contract signing or fair exchange of goods against payment are both services of the exchange layer.

Session
A session is a set of primitives that operate on the same context, e.g. a business transaction. Sessions can be nested: a session on a higher layer may start several sessions on lower layers and exchange parts of its context with them.

Sub-transaction
A transaction which is invoked by an ongoing transaction.

Suspended deal
Denotes a commerce deal that is accessed in the suspended mode.In this mode it is not possible to add new transactions to the deal.The deal can be inspected and private data can be updated.The co-operation of the other participants of the deal is not required to access a deal in suspended mode.See Section 4.1.1.2.

Third party
A third party is an entity supporting a business transaction without being involved as business partner.

Transaction
A *SEMPER* transaction is a sequence of method invocations which should result in a state transition from one consistent state into another.

Transfer
The process of transferring any business items (payment, information, vouchers, etc.) from one player to another, usually in a secure way. A transfer is realized by sending or receiving a container.

Type of module
A service block may support modules implementing the desired functionality in quite different ways and therefore the manager must support different internal interfaces. Modules supporting the same internal interfaces are said to be of the same type.

User
A user is anyone using the marketplace (e.g., buyer or seller).

Appendix C: Index

Appendix D: URL to JAVADOC

The complete documentation of the *SEMPER* application programming interface (*SEMPER*-API) is public and available on-line. This appendix describes how it can be accessed.

The documentation has been generated from the source code using JAVADOC which produces document in the HTML format. To access the documentation, one needs to input the so called URL as “Location” to the browser.

An HTML copy of the overview over the *SEMPER* Architecture with links to the API documentation of all service blocks can be found at URL:

<http://www.semper.org/deliver/d10/javadoc/index.html>.

This overview contains a link to a complete description of all service blocks. Alternatively, this description can be accessed directly at URL:

<http://www.semper.org/deliver/d10/javadoc/packages.html>.