Fairness in Electronic Commerce

by

N. Asokan

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1998

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Commerce over open networks like the Internet, sometimes referred to as *electronic commerce*, is becoming more widespread. This makes it important to study, and solve the security problems associated with electronic commerce. There are three prominent characteristics of commerce which are relevant in this respect. First, the crux of a commercial transaction is usually one or more *exchanges* of items of value. Second, players in a commercial transaction do not necessarily trust each other fully. Thus, protecting players from each other is as important as protecting them from outside attackers. Third, commercial transactions have legal significance. Therefore, it must be possible to gather sufficient evidence during the transaction to enable correctly behaving players to win any subsequent *disputes*.

This dissertation addresses the problem of *fairness* in electronic commerce. A system that does not discriminate against a correctly behaving player is said to be fair. Several protocols are proposed for performing exchanges fairly. The protocols are practical, and provide a high degree of fairness. The basic approach optimises for the common case that all players behave correctly. This is known as the *optimistic* approach. These protocols attempt to guarantee fairness during a protocol run. This is known as *strong fairness*. When strong fairness is not possible, one can fall back on gathering enough evidence so that fairness can be restored later by initiating a dispute. This is known as *weak fairness*. An analysis of the protocols leads to the conclusion that the exchange of *generatable* items can be guaranteed to be strongly fair. Various techniques to add generatability to items, including one technique which uses a cryptographic primitive called *verifiable encryption*, are presented.

In the case of weak fairness, a subsequent dispute is necessary to restore fairness. In general, disputes can occur even after a correctly concluded transaction. *Non-repudiation* techniques are used to gather evidence that can be later used in disputes. A novel non-repudiation technique called *server-supported signatures* is proposed.

The issue of handling disputes in electronic commerce is complex and hitherto not well-understood. Some aspects of the problem, within the limited context of electronic payment systems, are addressed. First, a unified definition of electronic payment systems, called the *generic payment service* is presented. Based on the generic payment service, a uniform way to express payment dispute claims is proposed. The need for a coherent framework for handling disputes in electronic payment systems is motivated.

Acknowledgements

Without the encouragement, support, flexibility, and guidance of my supervisors Prof. Jay Black and Dr. Michael Waidner, this dissertation could never have become reality. My sincerest thanks go to them first and foremost for making time for me despite their crowded schedules. When I was wandering aimlessly in the post course-work wilderness, Jay gave me my much-needed first break by agreeing to supervise my research. I thank him for taking a chance with me, for backing me up ever since, and for his invaluable guidance and feedback. I thank Michael for going beyond the call of duty in agreeing to co-supervise my work, for his patient reading of my write-ups (all the various versions of them!), and for teaching me the importance of rigour.

Bill Ince, software manager at the Math Faculty Computing Facility, walked into my office one day, several years ago, and asked me to take the responsibility for implementing a Kerberos infrastructure for the University of Waterloo. That was the day I became interested in security. I am grateful to Bill for letting me re-arrange my work schedule while I worked at MFCF. His flexibility enabled me to keep my research interests alive even while I worked full-time. I thank my former colleagues at MFCF, in particular Anil Goel, Robyn Landers, and Ken Wellsch, who helped me in my research in a variety of ways.

I thank Gene Tsudik and Phil Janson for giving me the opportunity to work with the network security group at the IBM Zurich Laboratory. I thank all my colleagues in the group for exposing me to various different aspects of security research. I am particularly indebted to Michael Steiner for his insightful feedback, for his willingness to listen to ideas, and for letting me use his beanbag chair — *Merci vielmal!* Part of this work was done within the context of Project SEMPER, funded jointly by the European Commission and the Swiss Federal Government. I thank the participants in the payment block work in SEMPER for their feedback.

I thank Professors Peter Landrock, Gord Agnew, Ken Salem, and Johnny Wong for agreeing to serve on my thesis committee. In particular, I thank Johnny for his support and guidance while he was on sabbatical leave in Zurich.

Thanks to José Abad-Peiro, Els van Herreweghen, Phil Janson, Matthias Schunter, Victor Shoup, Michael Steiner, and Gene Tsudik for collaborating with me on various publications. Special thanks to Matthias for his enthusiasm and support during our initial forays into the problem of fair exchanges.

Thanks to Hervé Debar, Ceki Gülcü, Els van Herreweghen, Mehdi Nassehi, Matthias

Schunter, Michael Steiner, Leena Tirkkonen, and Gene Tsudik for reading this dissertation, either wholly or in parts, and giving me valuable feedback. Several people, including various anonymous referees, provided feedback on papers related to this dissertation. I am grateful for their comments. In particular, feedback from Ivan Damgård, Birgit Pfitzmann, and Dave Presotto led me towards better techniques and presentation.

I thank my parents for teaching me to value fairness, and my sisters for egging me on to finish this dissertation. I thank Leena for her infinite patience and for always being there to support me through good times and bad.

The path I took to ultimately arrive at this dissertation has been, to say the least, unusual! Several people at Waterloo and Zurich helped along, always willing to accommodate my unusual requests. My sincerest thanks to all of them.

Trademarks

- DigiCash and ecash are registered trademarks of DigiCash bv.
- CyberCash is a registered trademark of CyberCash, Inc.
- Sun and Java are registered trademark of Sun Microsystems, Inc.
- CommercePOINT and IBM are registered trademarks of International Business Machines Corporation.
- FSTC is a trademark of Financial Services Technology Consortium, Inc.
- Mondex is a trademark of Mondex International Limited.
- Chipper is a registered trademark of Royal PTT Nederland NV and the Dutch Postbank.
- SET Secure Electronic Transaction and SET are trademarks owned by Visa International Service Association and MasterCard International, Inc.
- MANDATE is a trademark of the MANDATE consortium.
- SEMPER is a trademark of the SEMPER consortium.

To my sisters
Ahila, Kavitha, and Nalina
for their Friendship, and Love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Electronic Commerce

In the last decade, the Internet has grown dramatically in terms of the number of users as well as the types of services available. A notable new development is the ability to buy and sell goods and services over the Internet. The term *electronic commerce* refers to such commercial transactions carried out over open computer networks. The scope of electronic commerce should cover most of the various commercial scenarios in today's physical marketplaces. More importantly, electronic commerce research can enable functionality that is not present in non-electronic forms of commerce.[1] There are various reasons for the growing popularity of electronic commerce: reducing overhead costs, improving accessibility of services, providing new or improved services, and simply keeping up with a changing world in order to maintain a competitive advantage. Replacing paper-based processes with digital equivalents can reduce overhead costs: electronic cheques avoid the cost of having to store, protect, and securely transport paper cheques. Electronic commerce can enable new or better services: with appropriate hardware support, strong cryptography can be used to construct hard-to-forge digital signatures, at low cost.

Regardless of the motivating factors, the increase in commerce over open networks is reality. It is, therefore, important to study the security issues surrounding electronic commerce and seek effective solutions. Commercial transactions involve multiple players. Usually, the players *mutually distrust* one another. Protecting one legitimate player from another is as important as protecting legitimate players from intruders. Commercial transactions typically have legal significance — it must be possible for a correctly behaving player to gather sufficient evidence to win any subsequent *disputes*.

In 1995, the European Commission launched a research project called SEMPER (Secure Electronic Market Place for EuRope) [Wai96] to design a framework to enable secure electronic commerce. Several business scenarios were identified in the initial SEMPER deliverable [SEM96] including on-line purchase of goods, subscriptions, contract-signing, and auctions. All of these scenarios can be built up as a sequence of *exchanges*. This is the central premise on which the SEMPER architecture has been based. Frequently, but not always, these exchanges involve a transfer of value, also called a *payment*.

Most of the attention of electronic commerce research so far has been focussed on mechanisms for electronic payments. A large number of payment schemes with a wide

[1] *Micropayment schemes* [HSW96], which are cost-effective mechanisms for the repeated payment of very small amounts of money, is an example of such functionality.

range of security and other characteristics has been proposed over the last two decades. Several practical electronic payment systems have been implemented and deployed in recent years. However, payments rarely exist in a vacuum — usually one makes a payment in exchange for something. A purchase is an exchange of a payment (or value) in return for some goods.

1.2 An Example Scenario

To illustrate the issues more concretely, consider the following example scenario.

Alice wants to buy airline tickets for a forthcoming trip. There are several airlines selling tickets on the Internet. After browsing around, Alice decides to take up a promotional offer by BobAir, offering a reduced price provided the payment is made before January 12. The process of buying a ticket consists of two separate steps: ticket reservation, payment/delivery. Alice first sends a reservation request. BobAir reserves a place and sends an acknowledgement.

Alice and BobAir do not necessarily trust each other. Consider the potential trouble spots due to this distrust. After making a reservation, Alice may not come back to buy the ticket. BobAir loses money as the seat remained empty. On the other hand, after promising the seat to Alice, BobAir may have sold it to another traveller willing to pay the full price, leaving Alice without a seat. In each case, one player suffers a disadvantage in spite of following the protocol correctly. We say that such a protocol is not *fair*.

Alice's reservation request and BobAir's acknowledgements can be made *non-repudiable* by using, for example, digital signature techniques (for example, see [MvOV96, Ch. 11]). Thus, having sent a reservation request, Alice cannot deny having made the request. Similarly, having sent an acknowledgement, BobAir cannot deny having made a reservation for Alice. But this does not solve the problem completely. Suppose Alice sends a non-repudiable request first. If BobAir does not reply with an acknowledgement, Alice faces a dilemma. If she takes her business elsewhere, she runs the risk of paying for two seats; if she does not, she runs the risk of having no seat.

There are various possibilities for the next step. The most likely scenario using today's technology is that payment is made electronically in return for a digital receipt, but the actual ticket itself is delivered by non-digital means. However, eventually there will be electronic versions of air-line tickets as well (presumably down-loadable into a portable device). We will consider this possibility in our scenario. Alice used an "on-line ticket purchase" application to send the reservation request, and the subsequent purchase of the ticket. BobAir used the server version of the same application.

Throughout the rest of this dissertation, we will come back to this example and its variations to see how the proposed solutions help in avoiding the potential sources of unfairness.

1.3 Fairness

When a system involves the participation of multiple, mutually distrustful, players, a natural question is whether it meets the security requirements of all the players. A system that does not discriminate against a correctly behaving player is said to be *fair*. As long as a player behaves correctly, a fair system must ensure that other players will not gain any advantage over the correctly behaving player. The concrete meaning of fairness depends on what the protocol is intended to achieve. The notion of fairness is well established in some real life processes like elections or auctions.

In the case of exchanges, the meaning of fairness is equally clear. Consider the case of a payment for receipt. If the protocol requires Alice to send the payment first, it is an obviously unfair protocol. In order to be able to win any subsequent dispute, Alice requires that she be guaranteed to receive a receipt if the payment system has transferred her money to BobAir; at the same time, BobAir requires that no receipt be issued to Alice unless the money has been transferred. This is an instance of the generic problem of *fair exchange*. In electronic commerce scenarios, one can find other instances of exchange where fairness is a critical requirement: contract signing and certified mail are two examples. As with electronic payment systems, there has been some theoretical work on protocols for fair exchange. However, unlike the case of payments, there has been no practical systems for fair exchange with satisfactory security and performance characteristics.

Ideally, a protocol will *guarantee* fairness to all players, provided they follow the protocol correctly. But this may not always be possible or cost-effective. If some players suffer a loss of fairness after running a protocol, it may still be possible to *recover* fairness to them by starting a dispute. Such a protocol must ensure that sufficient evidence is accumulated during a run to enable correctly behaving players to win subsequent disputes. Even protocols that enable correctly behaving players to seemingly achieve strong fairness at the end of a normal protocol run, may need to generate evidence during the run in order to *retain* fairness to these players during any subsequent disputes. The term "fairness" has been used in other contexts in computer science. In Chapter 7, I provide a short summary of various usages of the term.

1.4 Scope of Research

1.4.1 Focus

The focus of the research described in this dissertation is to study fairness in electronic commerce. Since exchange protocols constitute a basic building block of electronic commerce, my primary emphasis is on investigating ways to guarantee fairness in exchange protocols. My secondary emphasis is on supporting disputes, as a means of recovering, as well as retaining fairness. In the interest of tractability, I limit myself to the problem of handling disputes in electronic payment systems. An orthogonal emphasis is on generality. For example, my approach to specifying dispute claims is independent of specific payment schemes, as it is based on the design of a generic payment service. The fair exchange protocols I describe in Chapter 2 and Chapter 3 are generic; concrete instances can be derived from them by making suitable assignments. The motivation for generality is both practicality and security: it enables both implementation and security analysis to be modular.

1.4.2 Outline of Dissertation

This dissertation is structured as follows. The problem of fair exchange is discussed in Chapter 2, starting with contract signing as a concrete problem instance. The contract signing protocol is then generalised into a protocol for the exchange of a large class of items called *forwardable items*. The relationship between the type of item and the achievable degree of fairness is also presented. This analysis leads to a generic fair exchange protocol for a class of items called *generatable items*, presented in Chapter 3. Various techniques for adding generatability to items are presented. In particular, *verifiable encryption* techniques are proposed to make items generatable.

The first step in supporting disputes is the collection of evidence. Typically, non-repudiation tokens constitute evidence about what happened in a transaction. In Chapter 4, a brief look at techniques for non-repudiation is followed by the presentation of a novel signature scheme, called *server-supported signatures* (S^3). S^3 provides signature service and has some additional properties that can make it useful in electronic commerce. The remainder of the dissertation focusses on handling disputes in payment systems. A simple model of electronic payment systems is presented in Chapter 5 and used as the basis for the design of a *generic electronic payment service*. A prototype of the generic electronic payment service has been implemented as part of the SEMPER project. In Chapter 6, a language for expressing payment dispute claims is presented. This is intended as the first step in designing a framework for dealing with disputes in payment systems. Different parties of a dispute may be required to prove or disprove certain *statements* about an alleged past transaction. Receipts and other items of evidence generated during a transaction will be used in such proofs. Chapter 7 closes the dissertation by summarising the contributions made.

Chapter 2

Fair Exchange

2.1 Introduction

The crux of a commercial transaction is usually an exchange of one item for another. We can find various instances of the general exchange scenario in different types of commercial activity:

- in a **purchase**, a *payment* is exchanged for a *receipt* of some valuable item,
- in **contract signing**, each player exchanges a non-repudiable commitment to the contract text in return for the other player's non-repudiable commitment to the same contract text,
- in **certified mail**, a message is exchanged for an acknowledgment of receipt, and
- in **barter**, an arbitrary item of value is exchanged for another item of value.

An important security requirement on exchanges is *fairness*. An exchange is fair if at the end of the exchange, either each player receives the item it expects or neither player receives any additional information about the other's item. In electronic commerce scenarios, exchanges have to be carried out over insecure networks. An attacker can gain control of the network or corrupt the systems used by the other player. Thus, carefully designed exchange protocols are necessary to *guarantee* fairness. Typically these protocols must also possess additional properties. For example, there may be subsequent disputes about what was exchanged during the transaction even if the exchange itself was completed fairly. In this case, sufficient evidence must be accumulated during the exchange to support the resolution of any future disputes.

In this chapter, I will discuss the problem of fair exchange and present new "optimistic" techniques for solving it. A definition of fair exchange and the properties required are presented in Section 2.1.1, followed by a brief survey of related work in Section 2.1.2. The idea of the optimistic approach is presented in Section 2.2. A protocol for solving an instance of the fair exchange problem (contract signing) using this approach is also presented. The issue of generic fair exchange protocols for exchanging arbitrary items is discussed in Section 2.3. The relationship between the nature of the items and the degree of fairness provided by the optimistic protocol is also discussed. Brief descriptions of some optimised instantiations of the generic protocol are presented in Section 2.4.

2.1.1 Requirements for Fair Exchange

P		Q
Input : i_P, d_Q, Q	$\xleftrightarrow{\text{fair exchange}}$	**Input** : i_Q, d_P, P
Output :		**Output** :
i_Q $(\mathsf{desc}(i_Q) = d_Q)$		i_P $(\mathsf{desc}(i_P) = d_P)$
	or	
aborted		*aborted*

Figure 2.1: A Successful Fair Exchange

We assume that there exists a function $\mathsf{desc}()$ that maps any exchangeable item to a string describing it in "sufficient" detail. (For example, a signature may be described by the message and the public key of the signer; a payment may be described by the payee, value, and currency.) Figure 2.1 shows the inputs and outputs of each player during a successful fair exchange. The inputs of P (Q) consist of an item $i_P(i_Q)$ and a description $d_Q(d_P)$ of the expected item. We formulate the following requirements in terms of one player, P — i.e., assuming P behaves correctly. The requirements of the other player, Q, can be stated similarly. P can consider a protocol run *completed*, when it is safe for P to terminate (and take no further part in the protocol). A protocol should indicate the points where it is completed. Any failures in the communication channel are subsumed in the notion of a misbehaving peer.

- **R1 - Effectiveness:** If Q also behaves correctly, and both P and Q do not want to abandon the exchange, then when the protocol has completed, P has i_Q such that $\mathsf{desc}(i_Q) = d_Q$.
- **R2 - Fairness**: Two notions of fairness are possible,
  - **R2a - Strong Fairness**: When the protocol has completed, either P has i_Q such that $\mathsf{desc}(i_Q) = d_Q$, or Q has gained no additional information about i_P, or
  - **R2b - Weak Fairness**: When the protocol has completed,
    - either P has i_Q such that $\mathsf{desc}(i_Q) = d_Q$, or Q has gained no additional information about i_P,

    * or P can prove to an arbiter that Q has received (or *can* still receive) i_P such that $\mathsf{desc}(i_P) = d_P$, without any further intervention from P.

- **R3 - Timeliness**: P can be sure that the protocol will be completed at a certain point in time. At completion, the state of the exchange as of that point is either final or any changes to the state will not degrade the level of fairness achieved by P so far.[1]

- **R4 - Non-repudiability**: After an effective exchange (i.e., P has received i_Q at the end of the exchange), P will be able to prove

  - **R4a - Non-repudiability of Origin**: that i_Q originated from Q, and
  - **R4b - Non-repudiability of Receipt**: that Q received i_P.

  Note that, in general, non-repudiability is not an integral requirement for fair exchange protocols. However, it is included here because it is useful in subsequent disputes after a fair exchange (even a successful one). In Chapter 4, we will take a closer look at non-repudiation techniques, and in Chapter 6, we investigate how non-repudiation tokens can be used in disputes.

Some fair exchange protocols may also satisfy other desirable security properties such as privacy and anonymity. On the other hand, depending on the context, non-repudiability may not be required. It is also possible that several parties may want to engage in a fair exchange. See [ASW96b] for a treatment of multi-party fair exchange. In this dissertation, I limit myself to the two-party case.

2.1.2 Related Work

Previous work on fair exchange falls into two categories: *third party* protocols which make use of a trusted, on-line third party, and *gradual exchange* protocols which gradually increase the probability of correctness over several rounds of communication.

The third party protocols all require a trusted (to various degrees) third party. The third party is *on-line* in that it is required to be actively involved in every exchange transaction. The basic approach is simple: each player sends its item and the description of the expected item to the trusted third party T; if the items match the expected

[1] For example, this implies that if P has already achieved strong fairness, any subsequent change in state should not cause P to lose its fairness or even degrade it to weak fairness.

descriptions, T forwards the items to the respective recipients; otherwise the exchange is aborted. Several variants of the same basic idea have been reported [CTS95, ZG96, DGLW96, FR97]. These protocols differ in attributes like their message flows and required degree of trust, but they all require a third party to be involved in each run of the exchange protocol. There are two problems with this approach: the third party needs to be trusted, and it must be on-line. These two problems exacerbate each other. The high trust requirements tend to minimise the number of third parties that can be candidates for providing the mediation service. The smaller the number of such third parties, the higher the risk of their becoming bottlenecks. On the positive side, the third party approach can be used to exchange arbitrary types of items (as long as they can be automatically verified against their descriptions), thereby covering all different types of exchanges, including those mentioned in Section 2.1.

The main goal of the gradual protocols is to achieve fairness without using an on-line third party. One approach is called the *gradual release of secrets*. The exchange protocol consists of many rounds of communication. In each round, each player releases a small portion of its item. Each portion must be verifiable as a valid component of the item. If a player detects misbehaviour, it stops the protocol run immediately. It is easy to see that the effectiveness property is guaranteed. If the protocol run is aborted, then each player can try to guess the remaining portions of their respective expected items. Thus, assuming that the players have roughly equal computational capabilities, and that portions of an item give no more information about the item itself, the strong fairness property is guaranteed in a probabilistic sense. Several protocols of this type have been proposed for contract signing and certified mail [Blu83, EGL85, BT94]. The main difficulties with this approach are the assumption of equal computational power (which is unreasonable when an individual user with limited resources engages in an exchange with a large organisation with substantial resources), and the extensive communication requirements.

The second gradual approach is called the *gradual increase of privileges*. This approach is useful only for contract signing [BOGMR90]. It does use a third party called the *judge* who rules on the validity of contract. However, the judge is not involved during the exchange itself. In every round, each player signs the contract specifying a probability parameter p. Given this signature, the judge will, with probability p, rule the contract valid. The value of p is gradually increased until both players end up with a fully binding contract. If a player detects misbehaviour, it stops the protocol run immediately. At this point, each player has a signed contract that has roughly the same chance of being declared valid. This protocol removes the assumption of comparable computational resources but suffers from the same extensive communication requirements, relies on the use of a trusted third party in case of disputes, and, as noted by Pfitzmann [Pfi95], fails to meet the timeliness requirement.

2.2 The Optimistic Approach

As we saw, none of the approaches described in Section 2.1.2 is satisfactory: the gradual protocols are not suitable for practical implementations due to their excessive communication requirements, while the third party protocols may result in bottlenecks and calls for high levels of trust in the third party.

However, in an environment where most players behave correctly most of the time, we can design efficient protocols for fair exchange by optimising for this common case. We will also rely on the use of a third party, but only in the case of an exception. The basic idea is as follows. First, both players agree on what is to be exchanged and which third party to use in case of an exception. This "agreement" is informal — it has no validity outside the context of the protocol. Then, one player (called the "originator") takes the risk of sending its item first, hoping that the other player will behave correctly and respond with its item. If the other player responds as expected, the protocol ends successfully. Otherwise, the originator contacts the third party to resolve the fair exchange. This is known as the *optimistic approach*.

Optimistic protocols for the exchange of a payment for a receipt or goods were first outlined by Bürk and Pfitzmann in [BP90]. The contract signing protocol of Ben-Or et al [BOGMR90] is also optimistic in nature. The optimistic approach is an obvious approach to performing fair exchange using trusted third parties. My contributions are twofold: I present detailed protocols and prove their properties with respect to the requirements listed earlier, and I analyse the relationship between the nature of the items being exchanged and the possible degrees of fairness.

A first attempt at designing detailed protocols for the optimistic fair exchange of generic items, along with an analysis of their security was presented in [ASW96a, ASW96b]. Independently Micali proposed similar protocols for certified mail [Mic97]. However, all these proposals failed to meet the timeliness requirement unless strong assumptions about the reliability of the communications channels are made. In both protocols, one player takes the risk by sending its item first. Only this player (originator) is allowed to invoke the third party. The *other* player (responder) cannot know the final state of the exchange until the originator invokes the third party. In [ASW96a, ASW96b], an overall time limit parameter was used to address this problem. The third party will not resolve an exchange after the specified time limit. However, this solution is not completely satisfactory because it is difficult to choose the right value for this time limit. If the time limit is too short, the originator may not be quick enough to start the recovery on time. Failure to start the recovery within the time limit will destroy the fairness for the originator, even if it is honest. Similarly, if the time limit is too long, the responder may have to wait without knowing what happened. Worse still, if the responder happens to crash during this time, it may not be able to prove later that it behaved correctly otherwise.

In the rest of this section, I will describe the optimistic approach in detail and present protocols which achieve the timeliness requirement in addition to being efficient and secure. The main highlight of the protocol is that at any time during a protocol run, either player can *unilaterally* choose to force the protocol to complete, without losing fairness.

As usual, ordinary players are not required to trust each other. But all players are required to trust the third party to a certain extent. Naturally, it is desirable to minimise this trust as much as possible. Making the third party *verifiable* (that is, if it misbehaves, its victim(s) can prove the fact in a dispute) is one way of reducing this trust. We introduce a new requirement which captures this property:

- **R5 - Verifiability of Third Party**: Assuming that the third party T can be forced to eventually send a valid reply to every request, the *verifiability of third party* property requires that if T misbehaves, resulting in the loss of fairness for P, then P can prove the misbehaviour of T to an arbiter (or verifier) in an external dispute.

In other words, each of the other players has a weak fairness guarantee even in the case of a misbehaving or corrupted third party.

2.2.1 Model

Two players O (the "Originator") and R (the "Recipient") want to engage in a fair exchange. The originator is the player that starts the optimistic fair exchange protocol by sending the first message. Hereafter, I will use the labels O and R when we need to distinguish between the originator and the responder. When there is no such need, we will use the label P for one of the players, and the label Q for the peer of P.

A third player T is known to both O and R. We assume that communication channels between any two players are *confidential*, meaning that eavesdroppers will not be able to determine the contents of messages travelling through these channels. Confidential channels can be implemented by encrypting messages for the recipient. We define two levels of quality for a communication channel.

Definition 2.2.1 *A communication channel between two correctly behaving players is* *operational* *if the messages inserted into it by the sender are received by the recipient within a known, constant (or constant factor) time interval.*

A communication channel between two correctly behaving players is ***reliable*** *if it is guaranteed to be always operational. An attacker will not be able to delay any message in a reliable channel, beyond the known upper bound.*

A communication channel between two correctly behaving players is ***resilient*** *if it is normally operational but an attacker can succeed in delaying messages by an arbitrary, but finite amount of time. In other words, a message inserted into a resilient channel will eventually be delivered.*

We will normally make the weaker assumption that the channel between any player and T is resilient. The resilient channel assumption leads to an *asynchronous* communication model: we can make no timing assumptions such as bounds on message delays, or deviations between local clocks.

Normally, we make no assumptions about the communication channel between O and R. The attacker may gain complete (and permanent) control of it. If either O or R is dishonest or wants to abandon the fair exchange, it may choose not to respond to any message, regardless of the quality of the communication channel used to send the message.

We assume that each player has the ability to compute and verify digital signatures using some arbitrary digital signature scheme

- in which each player P has a signing key S_P, and a corresponding verification key V_P,
- which has a signature algorithm sign such that, given a message m, $\mathsf{sign}(m, S_P) = Sig_P(m)$, where $Sig_P(m)$ is said to be a signature on message m with key S_P, and

- which has a signature verification algorithm verifySig such that, given a message m and a claimed signature s on m, verifySig(m, s, V_P) evaluates to true if and only if there exists a $Sig_P(m)$ which is equal to s.

The verification keys can be potentially anonymous — for example, if they are associated with short-lived pseudonyms. We also assume that anyone who has $Sig_P(m)$ also has m. This is easily done by always appending m to the signature on m. Further, we assume a collision-resistant one-way hash function $h()$. Intuitively, a one-way function $f()$ is a function such that given an input string x it is easy to compute $f(x)$, but given a randomly chosen y it is computationally infeasible to find an x' such that $f(x') = y$. A one-way hash function is a one-way function $h()$ that operates on arbitrary-length inputs to produce a fixed length value. The term x is called a *pre-image* of $h(x)$. A one-way hash function $h()$ is said to be *collision-resistant* if it is computationally infeasible to find any two strings x and x' such that $h(x) = h(x')$. The collision-resistance property also implies that given y and x such that $y = h(x)$, it is infeasible to find $x' \neq x$ such that $h(x') = y$. This property is called *second pre-image resistance*. A number of efficient and allegedly one-way hash functions, such as SHA-1[NIS95], have been invented. We will encounter the use of collision resistant one-way hash functions throughout this dissertation. Unless specified otherwise, we will assume the use of a one-way hash function like SHA-1 with fixed length inputs.

The notations used for various functions and objects are summarised in Table 2.1. (Some of the concepts are introduced later in the text.)

Notation	Explanation
V_P,S_P	Verification and signing keys of P
$Sig_P(m)$	Message m signed with S_P
verifySig()	Signature verification algorithm
$h()$	Collision-resistant one-way hash function
$Enc_P(m)$	Public-key encryption of message m recoverable only by P

Table 2.1: Notation Summary

2.2.2 An Example: Contract Signing

First, we illustrate the optimistic approach by considering a two-party contract signing scenario. In two-party contract signing, both players have initially agreed on some contract text. A valid contract consists of non-repudiation tokens on the contract text by each player. A fair contract signing protocol must ensure that either both players end up with valid contracts or neither does.

Protocol Description

O and R want to sign a contract so that they both expect to end up with each other's non-repudiation token on a previously agreed contractual text. The optimistic contract signing protocol has three sub-protocols: exchange, abort, and resolve. In the normal case, only protocol exchange is executed. The other two are used only if one of the two players decides to forcibly complete the protocol (presumably because it has decided that something has gone wrong). This is a non-deterministic choice made locally by a player. Note that, since our model is asynchronous, a player deciding something has gone wrong does not necessarily imply that any other player has behaved incorrectly. The protocol is shown in Figure 2.2. Places where a player can decide to give up and forcibly complete the protocol are marked with the tag "**give_up**?". For example, "**give_up**?: *abort*" means "if the player decides to give up, it should run protocol *abort*." The "**give_up**?" decisions are meant to be taken during an interval of time; in Figure 2.2, the positions of the "**give_up**?" decisions indicate the typical point of the interval concerned.

Step 1. We assume that O and R have already agreed on the text of the contract and then run protocol *exchange* with the contract text as input. O generates a random number o_O and computes $com_O = h(o_O)$. com_O will be used as a public *commitment* to the secret, o_O, meaning that once com_O is given to R, O cannot change the secret o_O. Note that $h()$ need not be a secure string commitment scheme (see Section 2.3.2) — $h(x)$ can leak information about x as long as it is computationally infeasible to determine x, given $h(x)$. O generates message m_1 as shown in Figure 2.2, and signs it to produce message me_1;[2] me_1 is sent to R.

[2]The messages are labelled so that the second letter of the label identifies the protocol, and the subscript identifies the message number. For example, me_1 is the first message of protocol *exchange*, and mr_2 is the second message of protocol *resolve*.

exchange protocol

O		R
Input : $V_R, \mathsf{T}, text$		**Input** : $V_O, \mathsf{T}, text$
$com_O = h(o_O)$	$\xrightarrow{me_1 = Sig_O(\overbrace{V_O\|V_R\|\mathsf{T}\|text\|com_O}^{m1})}$	**give_up**? : **quit**
give_up? : *abort*	$\xleftarrow{me_2 = Sig_R(\overbrace{me_1\|com_R}^{m_2})}$	$com_R = h(o_R)$
	$\xrightarrow{me_3 = o_O}$	**give_up**? : *resolve*
give_up? : *resolve*	$\xleftarrow{me_4 = o_R}$	

abort protocol

O		T
	$\xrightarrow{ma_1 = Sig_O(aborted\|me_1)}$	**if** resolved **then** $ma_2 = Sig_T(me_1\|me_2)$ **else** aborted = true $ma_2 = Sig_T(aborted\|ma_1)$
	$\xleftarrow{ma_2}$	

resolve protocol

O, R		T
	$\xrightarrow{mr_1 = (me_1, me_2)}$	**if** aborted **then** $mr_2 = Sig_T(aborted\|ma_1)$ **else** resolved = true $mr_2 = Sig_T(me_1\|me_2)$
	$\xleftarrow{mr_2}$	
Output : *contract* (Def. 2.2.2)		**Output** : *contract* (Def. 2.2.2)
	or	
abort_token		*abort_token*

Figure 2.2: Optimistic Protocol for Contract Signing

Step 2. If R decides to give up, it simply terminates the protocol run. In practice, R will decide this if it does not receive me_1 within a reasonable time. Note that we do not require globally synchronised clocks: it is entirely up to R to decide what is a "reasonable time" and when to time out.

If m_1 is not formed correctly, or if $\mathsf{verifySig}(m_1, me_1, V_O)$ evaluates to false, R ignores the message. In practice, R may send a negative acknowledgement to O and continue to wait for a valid message until timing out. In the rest of this description, I do not explicitly mention that malformed or incorrect messages are ignored.

Otherwise, R generates a random number r_O and computes $com_R = h(o_R)$. R generates message m_2 as shown in Figure 2.2, and signs it to produce me_2, which is then sent to O.

Step 3. If O decides to give up, it invokes T by running protocol *abort*. In practice, O decides to give up if it does not receive me_2 within a reasonable time.

Otherwise it sends o_O to R.

Step 4. If R decides to give up, it invokes T by running protocol *resolve*. Typically, R gives up if it does not receive an x in time, such that $h(x) = com_O$.

Otherwise it sends o_R to O.

Step 5. If O decides to give up, it invokes T by running protocol *resolve*. Typically, O gives up if it does not receive a y in time, such that $h(y) = com_R$.

Protocol *abort* is used by O to abort the exchange so that T will not resolve the exchange at a later time. Protocol *resolve* is used by either O or R to force a successful termination. Clearly *only one* of *abort* or *resolve* can succeed for a given instance of *exchange*. On T's system each of protocol *resolve* and protocol *abort* is guaranteed to be atomic.

In protocol *abort*, O sends a signed abort request to T. If the corresponding exchange has not already been resolved, T issues an *abort_token*, $Sig_T(aborted|ma_1)$. The *abort_token* is a guarantee by T that it has not and will not resolve this particular instance of *exchange*. If the protocol has already been resolved, T will simply forward the result of the protocol *resolve*.

In protocol *resolve*, either player sends me_1 and me_2 to T. If the exchange has already been aborted, T simply returns the *abort_token*. Otherwise, it issues a *replacement_contract* by counter signing me_1 and me_2 (i.e., $Sig_T(me_1|me_2)$).

Definition 2.2.2 *A valid contract is of the form*

1. $\{me_1, o_O, me_2, o_R\}$, *or*
2. $Sig_T(me_1|me_2)$

where,

$$\begin{aligned} me_1 &= Sig_O(V_O|V_R|\mathsf{T}|text|com_O), \\ me_2 &= Sig_R(me_1|com_R), \\ com_O &= h(o_O), \text{ and } com_R = h(o_R). \end{aligned}$$

Analysis

The "items" expected by both parties are the contract authenticators: o_O and o_R respectively. The descriptions of the items are com_O and com_R, respectively, along with the contract text itself.

Claim 1 *Assuming that the communication channel between* O *and* R *is resilient, the protocol satisfies the effectiveness requirement **R1**.*

Proof: It is easy to see that the effectiveness requirement is met under the additional assumptions specified in the statement of the claim. Since neither player wishes to abandon the protocol, they will not decide to give up and invoke T. The players are assumed to be behaving correctly; therefore, they will send messages out according to the protocol. Since we assume that the communication channel between them is resilient, the sent messages will be eventually received. From the description of protocol *exchange*, this implies that O and R will conclude the protocol with $\{me_1, o_O, me_2, o_R\}$. By Definition 2.2.2, this is a valid contract.

Claim 2 *Assuming that the communication channel between* T *and any other player is resilient, the optimistic contract signing protocol satisfies requirements **R2a, R3, R4,** and **R5** for both* O *and* R.

Proof:

- **R2a** (Strong Fairness): Assume that T behaves correctly. First consider the fairness for O — thus we assume that O behaves correctly as well. Since the execution of *resolve* and *abort* are atomic at T's system, there are only two ways in which R may end up with a valid contract:

- O sent o_O to R in message me_3. If O receives a correct me_4, then the fairness requirement for O is met. Otherwise, O can run protocol *resolve*. O is the only entity that could have aborted the exchange. Since we assume O behaves correctly, this would not have happened. Thus, T will respond with $Sig_T(me_1|me_2)$, which is a valid contract by definition. Thus the fairness requirement for O is met.
- O sent me_1 to R but did not receive a reply. In the meantime, R ran protocol *resolve* and obtained $Sig_T(me_1|me_2)$ from T. In this case, O will run protocol *abort*. Since the protocol was already resolved, T will respond with $Sig_T(me_1|me_2)$. Thus the fairness requirement for O is satisfied.

Thus, if T behaves correctly, strong fairness is guaranteed for O. Now, consider the fairness for R. There are only two ways in which O may end up with a valid contract:

- R sent o_R to O in message me_4. A correctly behaving R will do so only if it received a valid me_3. Thus the fairness requirement of R is met, in this case.
- R sent me_2 to O but did not receive a reply. In the meantime, O ran protocol *resolve* and obtained $Sig_T(me_1|me_2)$ from T. In this case, R will run protocol *resolve*. Since the protocol was already resolved, T will respond with $Sig_T(me_1|me_2)$. Thus the fairness requirement for R is satisfied.

Thus, if T behaves correctly, strong fairness is guaranteed for R.

- **R3** (Timeliness) : First, since we assume that the channel between a given player and T is resilient, both protocol *abort* and protocol *resolve* are guaranteed to be completed within a finite time. In Figure 2.2, notice that R can conclude the protocol in one of three ways:

  - simply terminating at any time before sending message me_2, or
  - terminating normally after sending message me_4, or
  - running protocol *resolve* at any other time.

  From **R2a** above, all of these will result in guaranteed fairness. The decision "**give_up**?" is entirely local. Notice that even if it is based on a timeout (e.g., whether to run protocol *resolve*), the timeout value is locally chosen and can be arbitrarily small. Thus, at the beginning of the protocol, R is guaranteed that the exchange will be completed at a finite point in time.

Similarly, O can conclude the protocol in one of three ways:

- terminating normally after receiving message me_4, or
- running protocol *abort* at any time before sending message me_3, or
- running protocol *resolve* at any other time.

Again, all decisions are local and both protocol *abort*, and protocol *resolve* are guaranteed to terminate at a finite point in time. Therefore, O has the timeliness guarantee.

- **R4** (Non-repudiability): Consider the non-repudiability of origin requirement of R. According to Definition 2.2.2, the first form of a valid contract contains

$$me_1 = Sig_O(V_O|V_R|\mathsf{T}|text|com_O), o_O$$

If the signature scheme is secure, the first component can be created only by O. If the one-way hash function used to compute com_O is such that it is infeasible to find second pre-images (if the one-way hash function is collision resistant, this property is implied), then no one can determine o_O unless O revealed it to them. Thus, if R can demonstrate possession of a valid contract containing me_1, o_O, then O must have sent it.

The second form of a valid contract is $Sig_T(me_1|me_2)$. If the digital signature scheme used is secure, this can be generated only by T. Thus, in both cases, R can prove the origin of a valid contract. A similar argument can show that the non-repudiability of origin requirement for O is met.

We already showed that the protocol guarantees fairness. If one player is able to produce a valid contract, it implies that the other player, had it followed the protocol correctly, will also possess a valid contract. Thus, the non-repudiability of receipt requirement is met for either player.

- **R5** (Verifiability of T): First, consider this requirement from the point of view of O. Suppose T behaves incorrectly so that the fairness property is lost for O. This means that R ends up with a valid contract while O does not. Since the channel to T is resilient, there are two cases when O finishes the protocol without a valid contract:

1. when it runs either protocol *abort* or protocol *resolve* but does not receive any reply from T, or
2. when it runs protocol *abort* and receives an *abort_token*.

Since the statement of the claim assumes that T is forced to eventually send a valid response to any request sent to it, case 1 is not applicable. In section 2.2.3, I discuss how reasonable this assumption is. For the rest of this proof, we only consider case 2: when T causes the loss of fairness for O by sending an *abort_token* to O while having sent (or later sending) a *replacement_contract* to R. This is an instance of T causing loss of fairness to one player by sending it a dishonest, but valid, reply.

If R has a valid contract of the form $\{me_1, o_O, me_2, o_R\}$, and $h()$ is collision-resistant, it must mean that O sent o_O to R in message me_3. But in this case our correctly behaving O would have run protocol *resolve*. The only possible reply by T is a *replacement_contract*, since O has not previously run protocol *abort* for this exchange. In this case, there is no loss of fairness for O. It follows that the only way in which O can lose fairness due to a misbehaving T is if R ends up with valid contract in the form of a *replacement_contract*, $Sig_T(me_1|me_2)$, given to R by T. If R never tries to enforce this contract, there is no loss of fairness for O. Therefore it must be the case that R attempts to enforce this contract. In this case, O can produce the *abort_token*. Both the *abort_token* and the *replacement_contract* are signatures under S_T. Thus, if the signature scheme is secure, only T could have generated them. According to protocol *abort* and protocol *resolve*, a correctly behaving T will not create an *abort_token* and a *replacement_contract* containing the same me_1. Therefore, the existence of such a pair can convince an arbiter that T was misbehaving. A similar argument can show that verifiability of T is guaranteed from the point of view of R as well.

Pfitzmann et al [PSW98] have shown that optimistic contract signing in an asynchronous communication model requires four messages in four "rounds." Therefore, this protocol is optimal. Pfitzmann et al show that removing any flow from the protocol in Figure 2.2 leads to a violation of the requirements. For example, consider eliminating the last flow by defining that a valid contract consists of me_2 and me_3. This could lead to a loss of fairness for R because O could receive me_2 and abort the exchange. Since it is R who takes the risk, we may transfer the right to abort the exchange to R. But this would result in a loss of the timeliness guarantee to O.

Invisibility of Third Party

In [Mic97], Micali described the notion of an *invisible* third party. According to Definition 2.2.2, there are two forms of a valid contract. The second form makes it obvious that T was involved in protocol *resolve*. T is said to be *visible* in such a protocol. If T were invisible, the end result (i.e., the form of a valid contract) would be the same regardless of whether T was run or not. We can easily change the protocol so that T is invisible to R: in the first step of the protocol, we can replace the one-way hash function $h()$ with the one-way trapdoor function $Enc_T()$, where the public encryption key of T is used to encrypt the random numbers o_O and o_R. A valid contract for R is $\{me_1, o_O, me_2, o_R\}$, where,

$$
\begin{aligned}
&me_1 = Sig_O(V_O|V_R|\mathsf{T}|text|com_O)\\
&me_2 = Sig_R(me_1|com_R)\\
&com_O = Enc_T(o_O|V_O|V_R), com_R = h(o_R)
\end{aligned}
$$

During protocol *resolve*, T simply decrypts com_O with its private decryption key to recover the random authenticator. To verify the contract, a verifier must recompute the encryption[3] com_O, and then verify me_1, and me_2. This version of the protocol satisfies the same set of requirements as the original version except for **R5** (verifiability of T). Our proof for the verifiability of T depended on its being visible. However, the real intent behind invisibility should be *non-invasiveness* — the contract-signing protocol described here is invasive because it dictated the structure of a valid contract. A truly non-invasive fair exchange protocol should be able to exchange *arbitrary* items such as tokens from electronic payment protocols (coins, cheques etc.), credentials, and contracts, without making any demands on its structure. In Section 2.3, we study the problem of fair exchange of arbitrary items. Typically, non-invasiveness implies invisibility of third party. It seems to be difficult to simultaneously achieve non-invasiveness and verifiability.

[3]Note that this is not possible in all encryption mechanisms.

2.2.3 Remarks on the Verifiability of Third Party

In the analysis of the contract signing protocol, we were able to prove verifiability of the third party only under the assumption that the third party can be forced to eventually send a valid response to any request sent to it. This is a rather strong assumption. To satisfy it, we need a fourth party in one form or another. For example, requests to T and its responses can be sent over a broadcast channel where a number of *witnesses* listen in. If T does not send a valid response to a request, the witnesses can vouch for this fact. Notice that the witnesses simply need to verify whether a response is a valid *abort_token* or *resolved_token*, without needing to know the entire contents of messages.

If this valid response assumption is not satisfied, we cannot achieve perfect verifiability as stated in the requirement. However if T does not reply, the victim will be able to *detect* it. Therefore, even without the valid response assumption, our protocol makes the third party verifiable with respect to *undetectable* cheating.

2.3 Generic Fair Exchange

Clearly, a fair exchange mechanism capable of exchanging arbitrary items is quite useful. In this section, I investigate how feasible it is to develop protocols for generic fair exchange and in particular what difficulties arise in using the optimistic approach for such protocols. I will attempt to generalise the protocol described in Section 2.2.2.

2.3.1 Properties of Exchangeable Items

In the simplest case, each item in an exchange can be represented as one or more strings. For example, parts of a valid contract in Section 2.2.2 were represented as strings. To transfer such an item, it is enough to simply send the string to the recipient. Consequently, an exchange of items is essentially an exchange of secret strings. However, in general, items are transferred using specific transfer protocols. For example, a transfer of value may involve a payment protocol (see Chapter 5). A transfer of a digital good from a seller to a buyer may involve an interactive fingerprinting protocol [PS96]. In theory, one may still be able to identify the communication steps which transfer critical secret strings in an arbitrary transfer protocol, and separate them. Alternatively, it may be possible to reduce the transfer of the item to the transfer of a specific secret string [AS98]. In these cases, we can still model any exchange as an exchange of secret strings. In practice, this is difficult because it will involve changing existing applications. Even if the transfer protocols support a generic token-based interface (see Chapter 5), they will need to be modified so that they indicate which tokens contain critical strings. Further, it may not be always possible to separate critical strings from complex transfer protocols without violating some properties provided by the protocol (such as unlinkability or confidentiality of message contents).

Therefore, in the interest of keeping our assumptions as weak as possible to begin with, we will assume that the transfer of items requires specific transfer protocols over which we have no control. In Section 2.3.2, I describe an asynchronous protocol for the optimistic fair exchange of a class of items which I call *forwardable* items. Whether it is possible to design optimistic fair exchange protocols for arbitrary items is an open question.

2.3.2 Optimistic Fair Exchange of Forwardable Items

We assume the same requirements, communication model, and cryptographic tools as in Section 2.2.1. In addition, we assume the following.

First, we assume that P can send the item directly to Q, or it can send it to T; T will be able to verify the correctness of the item (with respect to the stated description) and either store it or send it to Q. We also assume that the transfer protocols for these items are *idempotent*: re-sending the item by repeating the transfer protocol is harmless. If an item satisfies both properties, we call it a *forwardable item.* Simple strings are forwardable items.

Secondly, we assume the existence of a string commitment scheme with some special properties. A suitable string commitment scheme should consist of a function $\mathsf{commit}()$ which, given a string m, and a random key key, generates a commitment com, $\mathsf{commit}(m, key) = com$, and a function $\mathsf{verifyCom}()$ such that, given a string m, a commitment com, and a key key,

$$\mathsf{verifyCom}(com, key, m) = \mathsf{true} \iff \mathsf{commit}(m, key) = com$$

The idea is that once a commitment com of a string m is made, nobody can change the content m without invalidating com. Further, given com, nobody can obtain any information about m. We will use a commitment of an item and its key as a non-repudiation of origin token for that item. In the protocol description below, when we have to compute commitments on items that are not plain strings, we will use an empty string instead — we will still be able to retain the non-repudiability property, as explained in Section 2.3.3.

New Notation	
$\xRightarrow{m}$	transfer m using a sub-protocol

exchange protocol

O		R
Input : $i_O, d_R, V_R, \mathsf{T}$		**Input** : $i_R, d_O, V_O, \mathsf{T}$
	$me_1 = Sig_O(\overbrace{V_O \vert V_R \vert \mathsf{T} \vert com_O \vert h_O \vert d_R \vert d_O}^{m1})$ $\longrightarrow$	**give_up**? : **quit**
give_up? : *abort*	$\longleftarrow$ $me_2 = Sig_R(\overbrace{me_1 \vert com_R \vert h_R}^{m_2})$	
	$me_3 = i_O, key_O$ $\Longrightarrow$	**give_up**? : *resolve*
give_up? : *resolve*	$\Longleftarrow$ $me_4 = i_R, key_R, r_R$	
	$me_5 = r_O$ $\longrightarrow$	**give_up**? : *resolve*
Output : i_R, key_R, r_R **or** *affidavit_token*	**or**	**Output** : i_O, key_O, r_O **or** *affidavit_token*
abort_token		*abort_token*

Figure 2.3: Optimistic Fair Exchange (forwardable): Exchange Protocol

The optimistic protocol for the fair exchange of forwardable items is shown in Figures 2.3 and 2.4. The former depicts protocol *exchange* which is the only one used in a normal, fault-free case. The latter depicts the recovery protocols *resolve* and *abort* which are used in exceptional situations. Protocol *exchange* proceeds as follows.

Initially, O and R agree on the descriptions of the items they expect from each other: d_R and d_O describe the items expected by O and R respectively. Each player then runs protocol *exchange* with its own item and the description of the item it expects to receive.

New Notation	
P	Initiator of *resolve* (either O or R)
Q	Responder in *resolve* (either R or O)

abort protocol

O T

$\xrightarrow{ma_1 = Sig_O(aborted|me_1)}$

$\xleftarrow{\text{resolve}}$ **if** R-resolved

$\xleftarrow{ma_2 = Sig_T(aborted|ma_1)}$ **else**

aborted = true

resolve protocol

P T Q

$\xrightarrow{mr_1 = (V_P, me_1, me_2)}$

$\xRightarrow{i_P, key_P, r_P}$

$\xleftarrow{mr_2 = Sig_T(aborted|ma_1)}$ **if** aborted

else

P-resolved = true

$\xLeftarrow{i_Q, key_Q, r_Q}$ **if** Q-resolved

else $\xrightarrow{\text{resolve}}$

$\Longleftrightarrow$ (Q runs *resolve*)

$\xleftarrow{Sig_T(\text{affidavit}|mr_1)}$ **if give_up**?

$\xLeftarrow{i_Q, key_Q, r_Q}$ **else**

Figure 2.4: Optimistic Fair Exchange (forwardable): Recovery Protocols

Step 1. O generates a random number key_O from the domain of $\mathsf{commit}()$ and a random number r_O. Then it computes $com_O = \mathsf{commit}(i_O, key_O)$, $h_O = h(r_O)$, and $d_O = \mathsf{desc}(i_O)$; collects these pieces of data into a message m_1 as shown in Figure 2.3, and signs it to produce message me_1; me_1 is sent to R. As we noted earlier, if i_O is not a simple string, we compute com_O as a commitment of key_O only. In this case, key_O serves as a non-repudiation of origin token for any item described by d_O. If d_O is capable of uniquely identifying i_O, the two ways of computing com_O are equivalent. The same applies to the computation of com_R below.

Step 2. If R decides to give up, it simply terminates the protocol run. Typically, R gives up if it does not receive a valid me_1 within a reasonable time. As before, no clock synchronisation is required. Also, as before, all invalid messages are ignored. A valid message is such that the last but one component of m_1 is the same as $\mathsf{desc}(i_R)$, the last component of m_1 is d_O, and $\mathsf{verifySig}(m_1, me_1, V_O)$ evaluates to true.

Otherwise, R generates a random number key_R from the domain of $\mathsf{commit}()$ and a random number r_R. It then computes $com_R = \mathsf{commit}(i_R, key_R)$, $h_R = h(r_R)$ and then generates message m_2 as shown in Figure 2.3, and signs it to produce message me_2; me_2 is sent to O.

Step 3. If O decides to give up (e.g., if it does not receive a valid me_2 within a reasonable time), it runs protocol *abort*. A valid me_2 is such that $\mathsf{verifySig}(m_2, me_2, V_R)$ evaluates to true.

Otherwise it sends i_O and key_O to R in message me_3. Note that this operation consists of two parallel sub-protocols: a run of the transfer protocol of i_O and a simple message transfer for sending key_O, r_O. For simplicity, we indicate these as a single "transfer protocol" in Figure 2.4. The same applies to message me_4 below.

Step 4. If R decides to give up (e.g., if it does not receive a valid me_3 in time), it runs protocol *resolve*. A valid me_3 is such that $d_O = \mathsf{desc}(i_O)$, and $\mathsf{verifyCom}(com_O, key_O, i_O)$ evaluates to true, and $h(r_R) = h_R$. Again, note that *all* the sub-protocols mentioned above must succeed for R to conclude that it has received a valid me_3. The same applies to message me_4 below.

Otherwise it sends i_R, key_R, r_R to O in message me_4.

Step 5. If O decides to give up (e.g., if it does not receive a valid me_4 in time), it runs protocol *resolve*. A valid me_4 is such that $d_R = \mathsf{desc}(i_R)$, and $\mathsf{verifyCom}(com_R, key_R, i_R)$

evaluates to true.

Otherwise it sends r_O to R in message me_5.

Step 6. If R decides to give up (e.g., if it does not receive a valid me_5, such that $h(r_O) = h_O$) in time, it runs protocol *resolve*.

If the protocol completes without any exception, each player P will end up with:

- the item i_Q from the other player, Q,
- its non-repudiation of origin (NRO) token consisting of me_1, me_2, com_Q, key_Q, and
- a non-repudiation of receipt (NRR) token for i_P consisting of me_1, me_2, h_Q, r_Q.

Only O is allowed to initiate protocol *abort*. On receiving a request to abort a particular instance of an *exchange*, T will first check to see if R has already resolved. If it has, T will tell O to run protocol *resolve* instead. Otherwise, T will mark the exchange as aborted and issue an *abort_token* to O.

Protocol *resolve* proceeds as follows. Either O or R can run *resolve*. Let us denote the invoker by P and the other player by Q.

Step 1. First, the signed message me_1 and me_2 are sent to T along with the invoker's verification key V_P. Then P sends its item to T using the item using the transfer protocol, as well as all the non-repudiation tokens.

Step 2. If the protocol instance has been aborted before, T replies with the *abort_token*.

Step 3. Otherwise, T will mark the protocol as resolved by P and adds i_P, key_P, and r_P to its stable storage.

Step 4. If the exchange is already marked as resolved by Q, T will extract i_Q, key_Q, and r_Q from its stable storage and forward them to P using the item transfer protocol.

Step 5. Otherwise, T will send a message to Q asking it to run protocol *resolve*.

Step 6. If Q runs *resolve* successfully, i_Q, key_Q, and r_Q will be added to T's storage.

Step 7. If T finds that Q has resolved the exchange, it can forward i_Q, key_Q, and r_Q to P. If T decides to give up and force a termination (e.g., when Q does not complete protocol *resolve* within a reasonable time), it will issue an *affidavit_token* to P. P may choose to terminate at this point, in which case it has achieved weak fairness, or to re-run protocol *resolve* later.

The *affidavit_token* given to P is a statement by T that if Q wants, it can achieve strong fairness at any time after the issue of *affidavit_token*. As before, we assume that T will execute each of protocol *resolve* and protocol *abort* in a mutually exclusive manner. The idea behind weak fairness is that the player who ends up with only an affidavit can initiate a dispute outside the fair exchange system and use the affidavit as evidence.

Note that the interaction with T and Q (steps 5 and 6) is not necessary as far as meeting the weak fairness requirement is concerned. It is included as a practical measure only.

2.3.3 Analysis

Security of the Protocol

The items expected by P from Q are the actual item itself (i_Q), its non-repudiation of origin token (key_Q), and the non-repudiation of receipt token for i_P (r_Q). Now let us analyse the security of the protocol by arguing how far it meets the requirements identified in Section 2.1.1.

Claim 3 *Assuming that the communication channel between* O *and* R *is resilient, the protocol satisfies the effectiveness requirement (**R1**).*

Proof: Since both players are assumed not to want to abandon the exchange, neither will give up and invoke T. As both are honest, they will send their messages according to the protocol. The resilient channel between them will guarantee that the sent messages are eventually received. Under these conditions, from the description of protocol *exchange*, we see that at the end of *exchange* O has i_R, key_R, and r_R. Thus the effectiveness requirement for O is met. Similarly, the requirement is met for R as well.

Claim 4 *Assuming that the channel between* T *and any other player is resilient, and that* T *behaves correctly, the protocol satisfies the weak fairness requirement (**R2b**).*

Proof: First, we consider requirement **R2b** from the point of view of O. Since T will not run protocol *abort* and protocol *resolve* in parallel, there are only two ways in which R may end up with i_O:

- O sent it in flow me_3. If O receives me_4 in reply, then the fairness requirement is met. Otherwise O can run protocol *resolve*. Since O is the only entity that can run protocol *abort*, and we assume O behaves correctly, T will not respond with *abort_token*. In this case, T sends one of two replies:

– if R has successfully concluded protocol *resolve*, then T will already have i_R, key_R, and r_R in its storage. It simply forwards them to O. O therefore achieves strong fairness. Note that R may have run protocol *resolve* before O started its own run of protocol *resolve*. Or R may have been asked to initiate a run of protocol *resolve*.

– if R has not successfully concluded protocol *resolve*, then T will issue an affidavit to O. Thus, O will achieve weak fairness.

In either case, the fairness requirement of O is satisfied.

- R obtained it from T by running protocol *resolve*. The only way T can possess the item is if O sent it to T as part of a run of protocol *resolve*. But we showed above that in this case, O's fairness requirement is already met.

Thus given the assumptions of the claim above, the weak fairness requirement (**R2b**) is satisfied from the point of view of O.

Consider the requirement from the point of view of R. There are two ways in which O may end up with i_R:

- R sent it in message me_4. But in this case, R already received i_O in message me_3. Thus R has achieved strong fairness.

- T forwarded it to O during a run of protocol *resolve*. Since we assume that T behaves correctly, this means that the exchange was not already aborted when O ran protocol *resolve* (otherwise the "aborted" flag would have prevented the protocol *resolve* from succeeding). Call this "run 1". The only way T can come to possess i_R is if R deposited it during another run (run 2) of protocol *resolve*. Run 2 must have taken place before run 1. Thus, the aborted flag was not set during run 2 either. Therefore, T would not have replied with *abort_token* to R. This means that at the end of run 2, R received either i_O or an affidavit. In either case, the fairness requirement for R is met.

Claim 5 *Assuming that the channel between* T *and any other player is resilient, and that the underlying transfer protocols are guaranteed to complete within a finite time once they are begun, if the fair exchange protocol meets the weak fairness requirement (**R2b**), it also meets the timeliness requirement (**R3**).*

Proof: Now consider requirement **R3** from the point of view of O. First, since the channel to T is assumed to be resilient and the transfer protocols are timely, the recovery protocols initiated by O are guaranteed to be completed within a finite time. From the protocol description, notice that O can conclude the protocol in one of three ways:

- terminating normally after sending message me_4, or
- running protocol *abort* at any time before sending message me_3, or
- running protocol *resolve* at any other time.

All of these will result in guaranteed weak fairness. The "**give_up**?" decision is entirely local (as mentioned earlier, the "**give_up**?" decision by T during protocol *resolve* is not essential to the security requirements). In the first two cases, if R runs protocol *resolve* later, it can result in O receiving the items it expects. This upgrades the fairness for O from weak to strong. Thus, the timeliness requirement for O is met.

Now consider the same requirement for R. It can conclude the protocol in one of three ways:

- simply terminating at any time before sending me_2, or
- terminating normally after receiving me_5, or
- running protocol *resolve* at any other time.

Again, each of these will result in guaranteed weak fairness and the level of fairness will not be degraded by a subsequent attempt by O to run protocol *resolve*. Thus the timeliness requirement is satisfied for R as well.

Claim 6 *The protocol meets non-repudiability requirements (**R4a** and **R4b**).*

Proof: By the protocol description, at the end of a successfully concluded exchange, P will have

- its non-repudiation of origin token consisting of $(me_1, me_2, com_Q, key_Q)$, and
- a non-repudiation of receipt token for i_P consisting of (me_1, me_2, h_Q, r_Q).

The commitments com_Q and h_Q are embedded into the initial agreement message from Q (one of me_1 and me_2). Since this agreement message is a signature using S_Q, as long as the digital signature scheme is secure, only Q could have generated it. As long as the string commitment scheme is secure and the one-way hash function $h()$ is resistant against second pre-image finding,[4] if P possesses key_Q and r_Q, it can only be so because Q sent them to P. Since according to the protocol, Q always sends key_Q along with i_Q, $(me_1, me_2, com_Q, key_Q)$ is a valid non-repudiation of origin token. Similarly, since according to the protocol, Q will not release r_Q before receiving i_P, (me_1, me_2, h_Q, r_Q) is a valid non-repudiation of receipt token.

Remarks on Channel Quality

Throughout, we have assumed that the channel between T and each other player is resilient. It is reasonable to assume that semi-open networks such as corporate intranets or the telephone network are reliable. One way to implement reliable channels is to fall back to more reliable media (for example, starting with a packet-switched network, falling back to a dial-up line and then to a dedicated line).

The resilience assumption is reasonable even on open networks such as the Internet. If an attacker managed to disrupt a communication channel, it will eventually be detected and repaired. Also, protection against all benign network failures are covered under the resilient network assumption. We also assumed that an attacker can merely delay messages inserted into a resilient channel, but cannot remove them. In practice, this is achieved by using a datalink-layer protocol that takes care of message re-transmissions [Bla94].

Of course the resilience assumption does not place a bound on the delay before the channel becomes operational again. Therefore, the optimistic approach is inappropriate to exchange time-sensitive items (e.g., current stock quotes). Fair exchange of time-sensitive items over open networks is a difficult problem, regardless of the technique. Even if we used an online third party (instead of the optimistic approach), the protocol will not be secure against an attacker who can disrupt communication channels long enough. In any case, the optimistic approach is intended to be used in conjunction with a suite of other approaches such as those that use an on-line third party, so that depending on the type of the items to be exchanged, a suitable approach can be chosen for the exchange.

We also assumed that all communication channels are confidential. This is stronger than what we actually need in order to guarantee fairness. The fairness of the contract signing protocol does not require confidentiality for any of its message flows. In the

[4] Second pre-image finding (i.e., given $h(x)$ and x, finding a y so that $h(y) = h(x)$ as well) is harder than finding collisions.

exchange protocol for forwardable items, R must send message mr_1 confidentially, for otherwise, O can intercept i_R and then abort the exchange. Of course, if confidentiality of message contents against eavesdroppers is a requirement, then all messages need to be sent via a confidential channel.

External Dispute Resolution

If a player produces an affidavit during an external dispute, the other player may be able to refute it. To see this, consider the case when O is malicious and wants to frame R. It completes the exchange normally and then runs protocol *resolve* after cutting off the channel between T and R. Since R is not reachable, T will issue an affidavit to O. R will not even know that O has acquired an affidavit since R had no need to initiate protocol *resolve*. But R can produce the non-repudiation of receipt token obtained as a result of the successful exchange.

Practical Concerns

Once T becomes involved in an exchange (by aborting or resolving), it has to retain information about that exchange forever, since we imposed no time limits. In practice, however, T can limit the extent of its record-keeping to manageable levels. T may advertise a maximum lifetime value for its records. This lifetime can be very large (e.g., several years). Note that this does not change the timeliness guarantee: each player can complete an exchange as fast as it wants; a player who has completed an exchange to its satisfaction need not wait until the end of the lifetime.

Limiting the lifetime will also have an impact on the verifiability of T. T must embed a time-stamp in the affidavits it issues. For example, to prove T cheated in the contract signing protocol, a player has to produce an *abort_token* and a *replacement_contract* for the same exchange containing time-stamps which differ by a value less than the maximum lifetime.

2.3.4 Weak vs. Strong Fairness

Weak fairness is useful only if *both* O and R can be made to participate in the dispute resolution. Even though the contract signing protocol in Section 2.2.2 uses the same optimistic approach as the generic protocol, we were able to guarantee strong fairness.

This was because T could generate the contract by itself (even when it was used as an invisible third party). Let us call this property *generatability*. Contracts, and receipts for certified mail etc. are examples of generatable items.

Consider a purchase scenario: Suppose O wants to make a credit card payment to R and wants to receive some goods in return. If R receives the payment but does not respond, O will end up with an *affidavit_token* instead of the item. But the credit card payment system is a "pay-later" system — if the credit card issuer is willing to consider the *affidavit_token* by T trustworthy, it can cancel the payment. This will result in strong fairness for O. Let us call this property *revocability*.

When we use generatable or revocable items in an exchange, protocol *resolve* is changed so that instead of sending an affidavit, T will provide a replacement (in case of the generatable items) or have the item revoked (in case of revocable items).

Notice that if R is willing to forego the timeliness requirement (i.e., if it is willing to wait until it receives flow me_3), it can actually achieve strong fairness. Thus, by suitable assignment of roles, the optimistic protocol for generic fair exchange can be made to guarantee strong fairness. If the item of one player is generatable, then it should play the role of the responder (R). Similarly, if the item of one player is revocable, then it should play the role of the originator(O). In passing, also note that if one of the two players wishes to be anonymous, then it should play the role of the originator. Interestingly, it is also possible to transform a large class of items into generatable items. Techniques for such transformations and efficient protocols for the exchange of generatable items are discussed in Chapter 3.

2.4 Instantiations of the Generic Protocol

We can instantiate the generic protocol of Figures 2.3 and 2.4 to yield concrete protocols for specific types of fair exchanges. In this section, we briefly outline two such instantiations: for certified mail, and payment for receipt. Note that the generic protocol has provision for non-repudiation tokens built into it. In both instantiations described below, we will treat the non-repudiation tokens as the item expected by the originator. In the generic protocol, an item i_P is always sent along with its NRO token, key_P. Therefore, by treating key_P as i_P, the fairness properties are not affected.

2.4.1 Certified Mail

In certified mail, a sender wants to send a mail message m to a receiver. The sender requires that the receiver not be able to deny receiving the message. To achieve this, the sender needs a non-repudiation of receipt token from the receiver in exchange for the message. Thus certified mail is a fair exchange of the message and its non-repudiation of receipt token.

The mail message is neither generatable nor revocable. But it is forwardable. As in the contract signing example, we define a valid receipt for the message as either the tuple, $\{me_1,\ me_2,\ key_R\}$ or an *affidavit_token* consisting of T's signature on $me_1,\ me_2$. Thus, the mail sender takes the role of the originator and the receiver takes the role of the responder. The generic protocol can be instantiated as shown in Table 2.2. The receipt is, by definition, a generatable item.

Generic Protocol	Instantiation
i_O	mail
i_R	receipt text(t)
d_O	not needed
d_R	receipt text(t)
com_O	$\mathsf{commit}(mail, key_O)$
com_R	$\mathrm{h}(key_R)$
resolve for O	if unsuccessful, T issues replacement receipt
resolve for R	same as in Figure 2.4

Table 2.2: Instantiating the Generic Protocol for Certified Mail

With resilient channels, this instantiation makes T verifiable, and guarantees strong fairness and timeliness for O. R is guaranteed either strong fairness without timeliness or weak fairness with timeliness.

A slightly different instantiation, shown in Table 2.3 guarantees strong fairness and timeliness to both players, but does not make T verifiable. A valid receipt for the mail message is either $\{me_1,\ me_2,\ key_R\}$ or an affidavit from T. This instantiation is illustrated in Figure 2.5. The figure depicts the result of a direct instantiation according to Table 2.3, followed by the removal of unnecessary flows and fields. For example, since we do not make use of the non-repudiation of receipt tokens (the r_P values), the exchange protocol has only four flows. Protocol *abort* is the same as in Figure 2.4.

exchange protocol

Sender(O)		Receiver(R)
Input : mail, t, V_R, T		**Input** : t, *null*, V_O, T
$\xrightarrow{me_1 = Sig_O(V_O \| V_R \| \mathsf{T} \| Enc_T(\text{mail} \| key_O \| V_O \| V_R) \| \text{t})}$	**give_up**? : **quit**
give_up? : *abort*	$\xleftarrow{me_2 = Sig_R(me_1 \| h(key_R))}$
$\xrightarrow{me_3 = \text{mail}, key_O}$	**give_up**? : *resolve*
give_up? : *resolve*	$\xleftarrow{me_4 = key_R}$

resolve protocol for O

O		T
$\xrightarrow{mr_1 = (V_O, me_1, me_2, \text{mail}, key_O)}$
$\xleftarrow{mr_2 = abort\_token}$	**if** aborted
	else
	O-resolved = true
$\xleftarrow{Sig_T(\text{affidavit}, mr_1)}$

resolve protocol for R

R		T
$\xrightarrow{mr_1 = (V_R, me_1, me_2, key_R)}$
$\xleftarrow{mr_2 = abort\_token}$	**if** aborted
	else
	decrypt fourth component of me_1
	if V_O, V_R found in encryption
$\xleftarrow{\text{mail}, key_O}$	R-resolved = true

Figure 2.5: Certified Mail Protocol

Generic Protocol	Instantiation
i_O	mail
i_R	receipt text (t)
d_O	not needed
d_R	receipt text (t)
com_O	$Enc_T(\text{mail}\|key_O\|V_O\|V_R)$
com_R	$h(key_R)$
resolve for O	T recovers key_R
resolve for R	if com_O is correct, T recovers $mail, key_O$ in return for key_R

Table 2.3: Instantiating the Generic Protocol for Certified Mail

2.4.2 Payment for Receipt

In payment for receipt, a payer wants to make a payment and get a receipt in return. If the payment messages are forwardable (e.g., an electronic cheque), a simple instantiation is to allow T to issue a replacement receipt when payer O runs protocol *resolve*. This would guarantee timeliness for both players, strong fairness for payee R but only weak fairness for O.

Suppose that the payment mechanism used supports revocation. Table 2.4 shows an instantiation that guarantees strong fairness and timeliness for the payer. The payee has a choice of either timeliness combined with weak fairness or strong fairness without timeliness. The protocol is illustrated in Figure 2.6. If O does not receive a payment-receipt after sending the payment, it can invoke T to revoke the payment. If R does not receive an NRR token after sending a payment-receipt, it can invoke T to receive a replacement NRR token. With a resilient channel between R and T, a misbehaving O may succeed in revoking a payment after a successful exchange. But R can prove that the O misbehaved (in choosing to run protocol *resolve*) by producing the original NRR token for the payment-receipt itself (i.e., r_O). If O presents the payment receipt issued by R to a verifier, the verifier must always check with R or T to ensure that the payment has not been subsequently revoked.

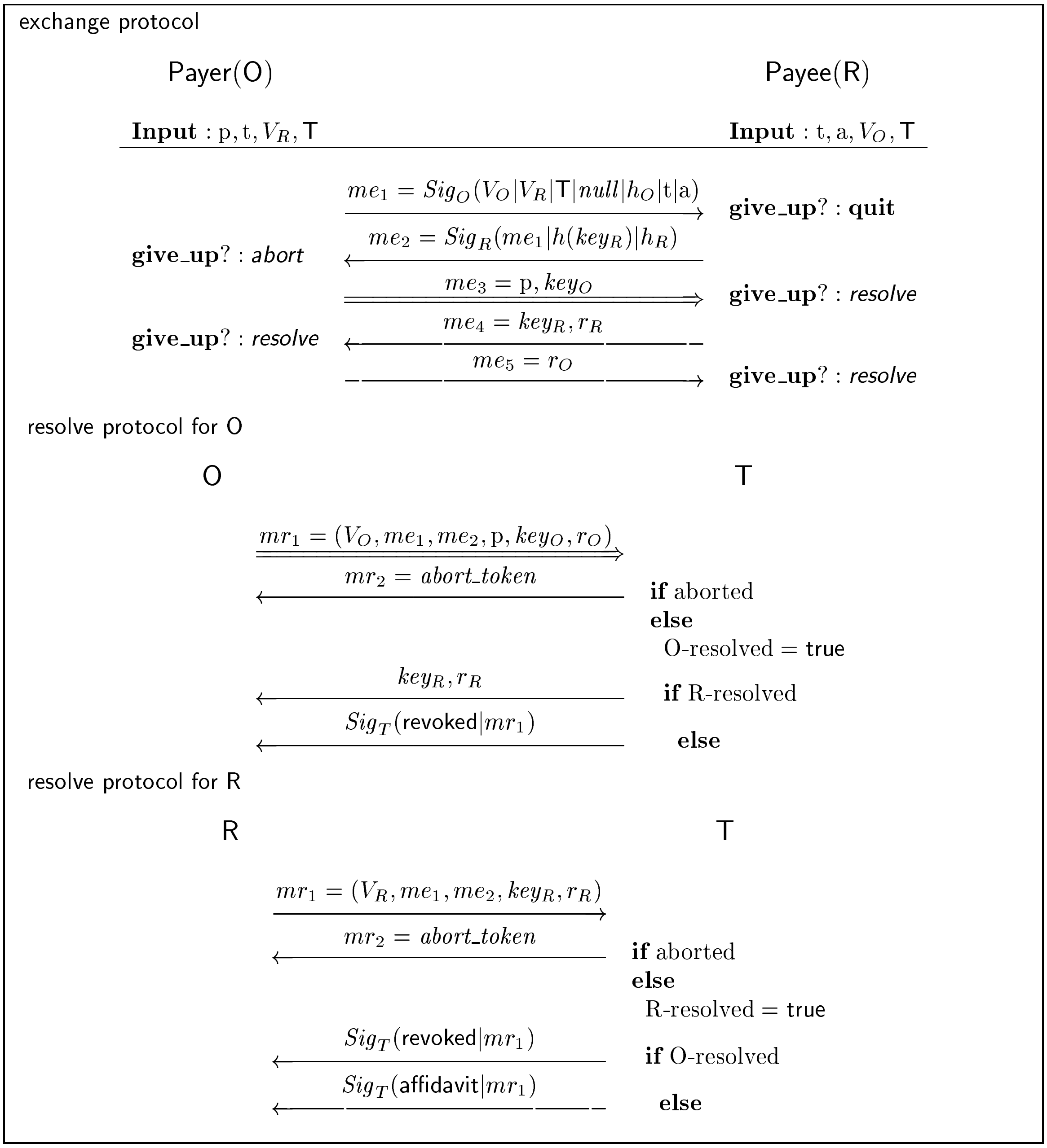

Figure 2.6: Payment for Receipt

Generic Protocol	Instantiation
i_O	payment (p)
i_R	receipt text (t)
d_O	amount (a)
d_R	receipt text (t)
com_O	not needed
com_R	$h(key_R)$
resolve for O	if unsuccessful, T revokes payment
resolve for R	if unsuccessful, T issues replacement NRR token for payment-receipt

Table 2.4: Instantiating the Generic Protocol for Payment-for-receipt

2.5 Summary and Conclusions

My contributions in this work are as follows. I have described the first **asynchronous protocol** for contract signing guaranteeing **timely completion**. I have generalised the idea to the exchange of **forwardable items**. In analysing this protocol, I have introduced the notions of **strong, and weak fairness**, as well as the notions of **generatability** and **revocability**. I showed that the optimistic approach can guarantee strong fairness, if the items to be exchanged are generatable or revocable.

In the case of weak fairness, it is necessary to use the collected evidence (e.g., *affidavit_token*) in an external dispute resolution procedure. Chapters 4 and 6 take a closer look at these issues. In Chapter 3, I show various techniques to add generatability to large classes of items, thereby making strong fairness possible while exchanging these items.

Chapter 3

Fair Exchange of Generatable Items

3.1 Fair Exchange of Generatable Items

3.1.1 Introduction

In Chapter 2, Section 2.3.4, we concluded that optimistic fair exchange protocols can guarantee strong fairness to a player if the item expected by that player is generatable. We also saw intuitive examples of generatable items such as replacement receipts or contracts. However, in these examples, the fair exchange protocol was invasive, because generatability depended on the ability to dictate the structure of the items. It would be desirable to find non-invasive ways of making items generatable.

In this chapter, the notion of a generatable item is formally defined in Section 3.1.2. Based on this definition, in Section 3.2, an optimistic protocol for the fair exchange of generatable items guaranteeing strong fairness, is presented. In Section 3.3, general techniques for transforming arbitrary items into generatable items are examined. Then *verifiable encryption* is described in Section 3.4 as a powerful non-invasive technique for adding generatability to a large class of items. Some variations are discussed in Section 3.5.

3.1.2 Definition of Generatable Items

The basic requirement of a generatable item is that T must be able to generate it even when the sender of that item is unavailable or uncooperative. Let us introduce the notion of a *permit* of an item. A permit will allow T to generate the corresponding item. Thus, a permit serves as a guaranteed promise of an item from its sender to its receiver. There are various ways of implementing a permit. In the contract signing example, the permit consisted of the initial messages me_1 or me_2. It simply indicated to T that it can issue a replacement contract. In Section 3.4, we will see a type of permit which contains the item *inside* it.

In the context of optimistic fair exchange, an additional requirement is that before T can generate an item, it may need to verify what item was expected in return for the item to be generated. There must be a way to securely *bind* this requirement into the permit.

I define a generatable item as one from which a permit can be constructed, using the following primitives. (We assume that a collision-resistant one-way hash function $h()$ exists.)

- A protocol *permit-trans* transfers a permit for an item from the sender to the receiver. The protocol is invoked via the service primitive bind(at the sender) and verify(at the receiver):

  - bind(*receiver*, *item*, *description*, *authenticator*, T) $\rightarrow$ *permit*

    bind() takes the identity of a receiver, an item, a description of another item, a number from the range of $h()$, and the identity of a third party as input, constructs a permit and sends it to the receiver, and returns the permit as output. Intuitively, a permit is a secure binding of the input parameters. The third parameter describes the expected item. For example, P will generate its permit p_P as bind(Q, i_P, d_Q, a, T). The identity Q may be replaced by a key, such as V_Q. The *authenticator*, as its name suggests, will be used in the optimistic exchange protocol (in Section 3.2.1) to authenticate the originator. It is also used to uniquely identify an exchange transaction. The usage is later explained in detail. T identifies the entity to be used as the third party.

  - verify(*sender*, *item*, *description*, T) $\rightarrow$ *permit*, *authenticator*

    verify() takes the identity of a sender, an item, a description of another item, and the identity of a third party as input, receives a purported permit from the sender, and, if the binding in the permit is valid, outputs the permit and the embedded authenticator (a number from the range of $h()$). For example, Q will receive and verify p_P by invoking verify(P, i_Q, d_P, T).

- extrAuth(*permit*) $\rightarrow$ *authenticator*

  extrAuth() takes a permit as input and returns the unique authenticator embedded in it. It is intended to be used by T.

- extrExp(*permit*) $\rightarrow$ *description*

  extrExp() takes a permit as input and returns the description of the expectation embedded in the permit. For example, extrExp(p_P) will yield d_Q. It is intended to be used by T.

- extrItem(*permit*, R_T) $\rightarrow$ *item*

  extrItem() takes a permit and a private key as input and returns the item represented by the permit. It is intended to be used *only* by T. For example, extrItem(p_P, R_T) will yield an item with the same description as i_P. R_T is a private key known only to T. Depending on the instantiation, R_T can be a signing key (S_T) or a decryption key (D_T).

- matchExp(*description*, *item*) $\rightarrow$ true/false

  matchExp() takes a description and an item, and returns true if the item matches the description. Typically, the implementation of matchExp() would be $description \stackrel{?}{=} \mathsf{desc}(item)$. But it can be slightly different in some instantiations.

The various extraction primitives (extrAuth, extrExp, and extrItem) are intended to be used by T during the recovery protocols, as explained in Section 3.2.1. In Section 3.2.2, I state the requirements on a permit in terms of the primitives defined here. However, I describe the protocol for the fair of exchange generatable items first, in order to motivate the requirements.

3.2 Fair Exchange Protocol for Generatable Items

3.2.1 Protocol Description

As usual, we assume that the communication channels are confidential. Optimistic fair exchange of generatable items consists of an exchange protocol (protocol *exchange*) and two recovery protocols (protocol *abort* and protocol *resolve*). Any malformed or incorrect messages are ignored by the receiver (in practice, the receiver will request a re-send). As in Chapter 2, each player may make an asynchronous, unilateral decision (indicated by "**give_up**?") to conclude the protocol. Typically, this decision is made if a valid response is not received from the peer within a reasonable, local, time limit. Also as in Chapter 2, the player who sends the first message is referred to as the originator (O). The other player is referred to as the responder (R). Protocol *exchange* proceeds as follows.

Step 1. O generates a random number r and computes an authenticator a_O as $h(r)$. The authenticator must be unique for each instance of the exchange. O will then run the *permit-trans* protocol using $\mathsf{bind}(\mathsf{R}, i_O, d_R, a_O, \mathsf{T})$. The *permit-trans* protocol transfers p_O and a_O to R in flow me_1. O will receive p_O as output.

Step 2. R participates in the *permit-trans* protocol using $\mathsf{verify}(\mathsf{O}, i_R, d_O, \mathsf{T})$. If it successfully returns a permit p_O and the authenticator a_O, R continues by initiating its own *permit-trans* protocol run using $\mathsf{bind}(\mathsf{O}, i_R, d_O, a_O, \mathsf{T})$, to create and send its own permit.

New Notation	
$\stackrel{m}{\Longrightarrow}$	transfer m using a sub-protocol
P	either one of O or R
i_P	item sent by P
d_P	description of i_P
p_P	permit sent by P

exchange protocol

O		R
Input : $\mathsf{R}, i_O, d_R, \mathsf{T}$		**Input** : $\mathsf{O}, i_R, d_O, \mathsf{T}$
$a_O = h(r)$		**give_up**? : **quit**
$\mathsf{bind}(\mathsf{R}, i_O, d_R, a_O, \mathsf{T})$	$\xrightarrow{me_1 = p_O, a_O}$	$\mathsf{verify}(\mathsf{O}, i_R, d_O, \mathsf{T})$
$\to p_O$		$\to p_O, a_O$? : **continue**
give_up? : *abort*		
$\mathsf{verify}(\mathsf{R}, i_O, d_R, \mathsf{T})$	$\xleftarrow{me_2 = p_R}$	$\mathsf{bind}(\mathsf{O}, i_R, d_O, a_O, \mathsf{T})$
$\to p_R, a$		$\to p_R$
$a \stackrel{?}{=} a_O$: **continue**		**give_up**? : *resolve*
	$\xrightarrow{me_3 = i_O}$	$\mathsf{matchExp}(d_O, i_O)$: **continue**
give_up? : *resolve*		
$\mathsf{matchExp}(d_R, i_R)$: **quit**	$\xleftarrow{me_4 = i_R}$	
Output :		**Output** :
i_R		i_O
	or	
abort_token		*abort_token*

Figure 3.1: Optimistic Fair Exchange of Generatable Items: Exchange Protocol

abort protocol

O		T
	$\xrightarrow{ma_1 = \mathsf{abort}, p_O, r}$	$\mathsf{extrAuth}(p_O) \neq h(r)?$: **quit** resolved$[h(r)]?$: $ma_2 = i_R$ **else** $ma_2 = Sig_T(aborted, p_O)$ aborted$[h(r)] = \mathsf{true}$
	$\xLeftarrow{ma_2}$

resolve protocol for O

O		T
	$\xrightarrow{mr_1 = p_R, r}$	$\mathsf{extrAuth}(p_R) \neq h(r)?$: **quit** aborted$[h(r)]?$: $mr_2 = Sig_T(aborted, p_O)$ **else** $mr_2 = \mathsf{extrItem}(p_R, R_T)$ resolved$[h(r)] = \mathsf{true}$
	$\xLeftarrow{mr_2}$

resolve protocol for R

R		T
	$\xRightarrow{mr_1 = p_O, i_R}$	$\neg$ $\mathsf{matchExp}(\mathsf{extrExp}(p_O), i_R)?$: **quit** $a = extrAuth(p_O)$ aborted$[a]?$: $mr_2 = Sig_T(aborted, p_O)$ **else** $mr_2 = \mathsf{extrItem}(p_O, R_T)$ store i_R; resolved$[a] = \mathsf{true}$
	$\xLeftarrow{mr_2}$

Figure 3.2: Optimistic Fair Exchange of Generatable Items: Recovery Protocols

The *same* authenticator a_O received from O is embedded into R's permit. If R decides to give up, it simply terminates the protocol run.

Step 3. O participates in this second run of the *permit-trans* protocol using $\mathsf{verify}(\mathsf{R}, i_O, d_R, \mathsf{T})$. If it returns successfully with p_R, and the authenticator a such that $a = a_O$, O continues by sending i_O to R in message me_3. If O decides to give up, it runs protocol *abort*.

Step 4. If R decides to give up, it invokes protocol *resolve*. If R receives a message me_3 such that $\mathsf{matchExp}(d_O, me_3)$ is true, it sends i_R to O in message me_4 and terminates with success.

Step 5. If O decides to give up, it runs protocol *resolve*. If O receives me_4 such that $\mathsf{matchExp}(d_R, me_4)$ is true, it terminates with success.

Protocol *abort* is used to tell T not to resolve this particular exchange. Only O is allowed to run protocol *abort*. To prove itself as the originator of an exchange, O will send the pre-image r of the exchange authenticator a_O in the abort request. If it has not already been resolved, T will comply by issuing an *abort_token* and marking the exchange as aborted. If it has already been resolved, T will give the expected item (i_Q) instead.

Protocol *resolve* is used to request T to generate the item from a permit. It can be run by either player. The invoker will send the permit received from the other party to T. In addition, it has to send an appropriate credential: when O invokes protocol *resolve*, it has to send r whereas when R runs protocol *resolve*, it has to send i_R. If the exchange has been already aborted, T will reply with *abort_token*. Otherwise, it will generate the item from the given permit and mark the exchange as resolved.

As before, we assume that the execution of protocols *abort* and *resolve* at T are atomic. O must choose a fresh r for each exchange it initiates. Normally, it must also keep r secret. (During recovery, r will be sent to T over a confidential channel.)

3.2.2 Requirements on Permits

Now, we are in a position to formally state the requirements a permit should satisfy, in terms of the primitives defined in Section 3.1.2.

Consider a permit p_P generated by P (i.e., either O or R) as $\mathsf{bind}(\mathsf{Q}, i_P, d_Q, a, \mathsf{T}) \rightarrow p_P$, in order to be sent to Q (i.e., either R or O) using our optimistic fair exchange protocol. The precise semantics of a permit are captured by the following requirements on these primitives. The statements of all requirements assume that the permit p_P was created as above, and that all players trust T. As usual, a requirement of the form "P requires ..." assumes that P behaves correctly.

- **Permit effectiveness**: Each player requires that T can extract a, d_Q, and i_P (or another item which matches the same expectation) from p_P, created as shown above. More concretely,

$$\mathsf{matchExp}(d_P, \mathsf{extrItem}(p_P, R_T)) \rightarrow \mathsf{true}$$
$$\mathsf{extrAuth}(p_P) \rightarrow a$$
$$\mathsf{extrExp}(p_P) \rightarrow d_Q.$$

  Further, each player requires that the receiver Q can receive and verify the correctness of the permit p_P (created using $\mathsf{bind}(\mathsf{Q}, i_P, d_Q, a, \mathsf{T})$).

$$\textbf{if } \mathsf{desc}(i_P) = d_P \textbf{ and } \exists\, i_Q \mid \mathsf{desc}(i_Q) = d_Q$$
$$\textbf{then } \mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T}) \rightarrow p_P, a$$

- **Permit integrity**: The issuer P requires that the receiver Q cannot tamper with p_P in such a way that it appears to bind a different description and/or authenticator with the same item. More concretely, if Q creates a p'_P such that

$$\textbf{if } \mathsf{extrAuth}(p'_P) \neq \mathsf{extrAuth}(p_P) \textbf{ or}$$
$$\mathsf{extrExp}(p'_P) \neq \mathsf{extrExp}(p_P)$$
$$\textbf{then } \mathsf{matchExp}(\mathsf{desc}(i_P), \mathsf{extrItem}(p'_P, R_T)) \rightarrow \mathsf{false}$$

- **Permit confidentiality**: The issuer P also requires that p_P reveals no information about i_P to Q other than its description, d_P.

- **Permit correctness**: The receiver Q requires that if a permit p_P passes the permit validity test, then it is guaranteed to be able to recover the expected item without any help from P. More concretely,

$$\begin{aligned}
&\textbf{if} \ \ \mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T}) \rightarrow p_P, a \\
&\textbf{then} \ \ \mathsf{matchExp}(d_P, \mathsf{extrItem}(p_P, R_T)) \rightarrow \mathsf{true} \\
&\textbf{and} \ \ \mathsf{extrExp}(p_P) \rightarrow \mathsf{desc}(i_Q) \\
&\textbf{and} \ \ \mathsf{extrAuth}(p_P) \rightarrow a
\end{aligned}$$

3.2.3 Analysis

Claim 7 *Assuming the communication channel between* O *and* R *is resilient, the protocol satisfies the effectiveness requirement **R1**.*

Proof: The argument is similar to that in Section 2.2.2.

Claim 8 *Assuming the communication channel between* T *and any other player is resilient, and* T *behaves correctly, the generic protocol for the fair exchange of generatable items satisfies requirements **R2a** (strong fairness), and **R3** (timeliness) for both* O *and* R.

Proof:

- **R2a** (Strong fairness) : First we prove that this requirement is met for O, assuming O behaves correctly. Since we assume that R cannot extract any information about i_O from p_O other than d_O (the "permit confidentiality" property), there are only two ways in which R receives an item matching d_O while O has **not** received an item matching d_R yet:

  – O sent it to R in message me_3. Since O is assumed to have behaved correctly, it follows that (a) O has not run protocol *abort* for this exchange, and (b) O has already received a permit p_R in me_2 that passed the validity test. We assume that T behaves correctly. T will honour abort requests only if the request is accompanied by a pre-image r of the authenticator a_O. Only O knows r. If the hash function $h()$ is one-way and resistant to collisions, R cannot determine a valid pre-image of a_O. Therefore, R cannot convince T to abort the exchange.

The "permit correctness" property implies that

$$\mathsf{matchExp}(d_R, \mathsf{extrItem}(p_R, R_T)) \rightarrow \mathsf{true}$$
$$\mathsf{extrAuth}(p_R) \rightarrow a_O$$

Therefore, if O runs protocol *resolve* by sending r and p_R, T will reply with an item matching description d_R. Thus, the strong fairness requirement is met for O in this case.

– T sent it to R when the latter ran protocol *resolve*. Since we assume T is honest, and protocols *abort* and *resolve* are atomic, it must be the case that this exchange has not been previously aborted. If O already has p_R, then this case reduces to the previous case, thereby satisfying the fairness requirement for O. If O does not yet have p_R, then it can run protocol *abort*. Now, from the "permit integrity" property and the fact the R successfully finished protocol *resolve*, it follows that it must have used p_O during that run. Thus, when O runs protocol *abort* with r, the authentication is guaranteed to succeed, and, T is guaranteed to possess i_R that satisfies O's expectation, d_R. Thus, the strong fairness requirement for O is met.

Thus, the protocol guarantees strong fairness for O.

Now, consider the requirement for R. There are again only two ways by which O can get an item matching d_R. Either R sent it in message me_4 or T sent it during recovery. In the former case, a correctly behaving R must have already received i_O thereby satisfying its fairness requirement. In the latter case, since T resolved the exchange, and the recovery protocols are atomic at T, T has not and will never mark the exchange as "aborted." Suppose R had already received the permit p_O which passed the verification test. In this case, the "permit correctness" property implies that R can successfully run protocol *resolve*, thereby satisfying the fairness requirement. Suppose R has not already received p_O. Then O does not have p_R either, and therefore could not have run protocol *resolve* itself. If O had run protocol *abort*, it would have succeeded, since neither player could have run protocol *resolve* before. But we assumed that O already got the item. Therefore, this last case does not arise.

Thus, the protocol guarantees strong fairness for R.

- **R3** (Timeliness): The argument is similar to that in Section 2.2.2.

We started with the requirement that all communication channels are confidential. This is stronger than what we actually need. Message mr_1 during R's execution of protocol *resolve* needs to be protected so that only T can see its contents. Otherwise, O can abort the exchange and still get i_R by passively observing this message. Similarly, mr_1 needs to be protected during O's execution of protocol *resolve*. Otherwise, R can steal r from O's resolve request, and use it to abort the exchange. None of the other messages in any of the three protocols needs to be confidential, as far as the fairness requirements are concerned. If data confidentiality against observers is a requirement, then messages me_3, me_4 of protocol *exchange* need to be sent confidentially, too.

3.3 Making Items Generatable

A straight-forward way to construct permits is to use digital signature schemes. As long as the digital signature scheme is secure, the binding property is maintained. In this section, I describe some ways of using such permits to render items generatable.

3.3.1 Replacement Tokens

Some items are either inherently generatable or can be defined so that they are generatable. We saw an example in the case of contract signing in Section 2.2.2. Table 3.1 shows how the generic definition of a generatable item can be instantiated in the case of replacement tokens. The symbol '|' indicates concatenation. We assume that concatenation does not remove information about the constituent strings. That is, if $z = x|y$, then given z, one can infer that its constituent parts are x and y.

Claim 9 *The construction described in Table 3.1 constitutes a valid permit according to the requirements in Section 3.2.2.*

Proof:

- **Permit effectiveness**: From the definition of p_P in Table 3.1, it is easy to see that the conditions for permit effectiveness are met.

Generic Definition	Instantiation
p_P	$Sig_P(\mathsf{desc}(i_P)\|d_Q\|a\|\mathsf{T})$
$\mathsf{bind}(\mathsf{Q}, i_P, d_Q, a, \mathsf{T})$	construct p_P as shown above, send it with a; return p_P
$\mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T})$	receive p_P, a, check $\mathsf{verifySig}(d_P\|\mathsf{desc}(i_Q)\|a\|\mathsf{T}, p_P, V_P)$; return p_P, a
$\mathsf{extrAuth}(p_P)$	extract d_P, d_Q, a from p_P, check $\mathsf{verifySig}(d_P\|d_Q\|a\|\mathsf{T}, p_P, V_P)$, return a
$\mathsf{extrExp}(p_P)$	extract d_P, d_Q, a from p_P, check $\mathsf{verifySig}(d_P\|d_Q\|a\|\mathsf{T}, p_P, V_P)$, return d_Q
$\mathsf{extrItem}(p_P, R_T)$	$\mathsf{sign}(\mathsf{affidavit}\|p_P, S_T)$
$\mathsf{matchExp}(desc, item)$	$(desc \stackrel{?}{=} \mathsf{desc}(item))$ **or** $\mathsf{verifySig}(\mathsf{affidavit}\|p_P, item, V_T)$, where $p_P, a \leftarrow \mathsf{verify}(\mathsf{P}, i_Q, desc, \mathsf{T})$

Table 3.1: Instantiating the Generic Protocol for Replacement Tokens

- **Permit integrity**: If the digital signature scheme is secure, no one can take the permit p_P and create a new fake permit p'_P such that $\mathsf{verifySig}(\mathsf{P}|d_P|d_Q|a, p'_P, V_P)$ evaluates to true. Thus the premise of the permit integrity condition cannot be true. Therefore, this instantiation satisfies the permit integrity condition.

- **Permit confidentiality**: From the definition of p_P in Table 3.1, p_P contains no information about i_P other than $\mathsf{desc}(i_P)$. Thus, this instantiation satisfies the permit confidentiality condition.

- **Permit correctness**: $\mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T})$ successfully returns the received permit p_P and authenticator a only if $\mathsf{verifySig}(d_P|d_Q|a|\mathsf{T}, p_P, V_P)$ evaluates to true. The same test is used in both $\mathsf{extrExp}(p_P)$ and $\mathsf{extrAuth}(p_P)$. Therefore these two primitives would successfully return $\mathsf{desc}(i_Q)$ and a respectively. $\mathsf{extrItem}(p_P, S_T)$ will return $Sig_T(\mathsf{affidavit}|p_P)$. $\mathsf{matchExp}(d_P, Sig_T(\mathsf{affidavit}|p_P))$ is trivially true. Therefore, this instantiation satisfies the permit correctness condition.

Plugging this instantiation into the generic fair exchange protocol of Figures 3.1 and 3.2 results in essentially the same protocol as the one in Figure 2.2. The variations are minor and actually the version here is slightly more efficient than the earlier version: instead of O making a second signature during the abort request, it uses the pre-image of the authenticator, thereby saving one signature. It is easy to see how the invisible third party variation can be instantiated.

3.3.2 Pre-arranged Deposits

Another straight-forward way for making items generatable is to deposit items with T ahead of time. For example, a merchant P who sells a software program i_P can deposit a copy of it with T and obtain a certificate $cert = Sig_T(d_P)$, from T. The permit for this item will include this certificate. Pre-arranged deposits are used in other contexts, such as software source-code escrowing, in order to facilitate transactions between mutually suspicious parties. This instantiation is shown in Table 3.2.

Generic Definition	Instantiation
p_P	$Sig_P(cert\|d_Q\|a\|\mathsf{T})$, $cert = Sig_T(d_P)$
$\mathsf{bind}(\mathsf{Q}, i_P, d_Q, a, \mathsf{T})$	construct p_P as shown above, send it with a; return p_P
$\mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T})$	receive p_P, a and check $\mathsf{verifySig}(cert\|\mathsf{desc}(i_Q)\|a\|\mathsf{T}, p_P, V_P)$ and $\mathsf{verifySig}(d_P, cert, V_T)$; return p_P, a
$\mathsf{extrAuth}(p_P)$	extract $cert, d_Q, a, \mathsf{T}$ from p_P, check $\mathsf{verifySig}(cert\|d_Q\|a\|\mathsf{T}, p_P, V_P)$, return a
$\mathsf{extrExp}(p_P)$	extract $cert, d_Q, a, \mathsf{T}$ from p_P, check $\mathsf{verifySig}(cert\|d_Q\|a\|\mathsf{T}, p_P, V_P)$, return d_Q
$\mathsf{extrItem}(p_P, R_T)$	extract $cert$ from p_P, $\mathsf{verifySig}(d_P, cert, \mathsf{T})$, retrieve corresponding i_P from storage, and return it.
$\mathsf{matchExp}(desc, item)$	$desc \stackrel{?}{=} \mathsf{desc}(item)$

Table 3.2: Instantiating the Generic Protocol for Pre-arranged Deposits

Claim 10 *The construction described in Table 3.2 constitutes a valid permit according to the requirements in Section 3.2.2.*

Proof:

- **Permit effectiveness**: From the definition of p_P in Table 3.2 , it is easy to see that the conditions for permit effectiveness are met.

- **Permit integrity**: If the digital signature scheme is secure, no one can take the permit p_P and create a new fake permit p'_P such that $\mathsf{verifySig}(cert|d_Q|a|\mathsf{T}, p'_P, \mathsf{P})$ evaluates to true. Thus the premise of the permit integrity condition cannot be true. Therefore, this instantiation satisfies the permit integrity condition.

- **Permit confidentiality**: From the definition of p_P in Table 3.2, p_P contains no information about i_P other than $\mathsf{desc}(i_P)$. Thus, this instantiation satisfies the permit confidentiality condition.

- **Permit correctness**: $\mathsf{verify}(\mathsf{P}, i_Q, d_P, \mathsf{T})$ successfully returns the received permit p_P and authenticator a only if $\mathsf{verifySig}(\mathsf{desc}(i_Q), cert, V_T)$ evaluates to true. The same test is used in both $\mathsf{extrExp}(p_P)$ and $\mathsf{extrAuth}(p_P)$. Therefore these two primitives would successfully return $\mathsf{desc}(i_Q)$ and a respectively. $\mathsf{extrItem}(p_P, S_T)$ returns i_P. $\mathsf{matchExp}(d_P, i_P)$ is trivially true. Therefore, this instantiation satisfies the permit correctness condition.

3.3.3 Other Techniques

Another way to achieve cryptographic binding is to use encryption. But unlike signatures, there is no easy way to verify an encryption without actually doing the decryption. The problem then is to find a way to make *verifiable* encryptions. At the outset, this might seem a contradiction in terms: the very goal of encryption is to leak no information about what is encrypted. In the next section, we see that we can make verifiable encryptions of certain classes of items without leaking additional information that might lead to an attacker recovering the items themselves.

3.4 Generatability via Verifiable Encryption

In general, a verifiable encryption scheme consists of:

- A function $\mathsf{desc}()$ as defined earlier that takes a string s and returns its description p.

- a verifiable encryption algorithm $\mathcal{VE}()$ and a corresponding decryption algorithm $\mathcal{VD}()$ such that
  - $\mathcal{VE}(s) \to c$, and
  - $\mathcal{VD}(c) \to s$.

- a protocol $\mathcal{V}()$ such that $\mathcal{V}(c, p) \to \mathsf{true}$ iff $\mathcal{VD}(c) = s$ **and** $\mathsf{desc}(s) = p$.

The intent is that the encryption and decryption algorithms behave as usual with any cryptosystem. In addition, each string in the domain of the encryption algorithm can be described by another string. Given a ciphertext a and a string purported to be the description of the corresponding secret plaintext s, the $\mathcal{VD}()$ predicate can be used to verify if a will indeed yield s on decryption.

As far as I know, the notion of *publicly verifiable encryption* was first mentioned by Stadler in [Sta96]. He used publicly verifiable encryption as a building block for achieving publicly verifiable secret sharing. He gave constructions for publicly verifiable encryption of discrete logarithms and e^{th} roots. However, there does not appear to be an easy way to embed additional information (such as the description of the expected items) into the encryption. Therefore, Stadler's techniques are not suitable for constructing permits for optimistic fair exchange. Also, his constructions were applicable to a specific encryption scheme only.

Here, I present a permit construction scheme based on verifiable encryption that can work with arbitrary public-key encryption mechanisms.

As a building block, we use a certified claw generation scheme (CCGS) [BG96]. A CCGS is a 2-out-of-2 verifiable secret sharing system. Given a secret s and a public description p of s, a CCGS will generate two *shares* (or "claws") s_0 and s_1 of s, such that anyone who knows p and a share can determine that it is a valid share but infer no more information about s. Someone who knows both s_0 and s_1 can combine them to recover s. More concretely, a CCGS consists of the following functions:

- a predicate $\mathsf{match}()$ that takes two strings p and s as arguments,
- a sharing algorithm $\mathsf{share}()$ which takes a string s as input and returns
  - two shares s_0 and s_1, and
  - a public string v,
- a share verification predicate $\mathsf{verifyShare}()$ corresponding to the secret sharing algorithm which takes one of the two shares as input, along with v and p.; After a valid sharing, $\mathsf{verifyShare}(i, s_i, v, p)$, where $i \in \{0, 1\}$, will evaluate to true iff $\mathsf{match}(p, s)$ is true and s_i and v were produced by $\mathsf{share}(s)$, and
- A reconstruction function $\mathsf{reconstruct}()$ corresponding to the secret sharing algorithm which takes both shares as input and returns the original secret (i.e., $\mathsf{reconstruct}(s_0, s_1)$ produces s iff s_0 and s_1 were produced by $\mathsf{share}(s)$).

The sets from which p and s are drawn and the domains and ranges of the various functions and predicates must be specified when a CCGS is instantiated concretely.

In addition, we assume an encryption scheme consisting of a probabilistic encryption algorithm $\mathcal{E}()$ and a corresponding decryption algorithm $\mathcal{D}()$ such that for a random r and plain-text x, $\mathcal{D}(\mathcal{E}(r,x)) = x$. We also assume a collision-resistant, one-way hash function $h()$.

Given the tools above, the generic *permit-trans* protocol for verifiable encryption permits is constructed as shown in Figure 3.3. The sender possesses a secret string s, and an arbitrary token t to be embedded in the permit. The purpose is to send a permit based on verifiable encryption of s to the receiver without revealing s. The receiver possesses the description of s (say p) and t. At the end of a successful protocol run, the receiver will have a permit which contains, with a high probability, s bound with t.

The sender starts by generating k pairs of certified shares of s. It probabilistically encrypts each share and commits to all $2k$ encryptions by hashing them together. The hash $\mathcal{H}$ is sent to the receiver, along with the set of k public strings $\mathcal{V} = \{v_i\}$, generated by share(). The receiver generates a random $k-$bit challenge and sends it to the sender. Depending on the value of the i^{th} bit of the challenge, the sender reveals the corresponding share of the i^{th} pair of certified shares, along with the random value used to probabilistically encrypt it, and the encryption of the other share.

The receiver first re-computes the opened encryptions by re-encrypting each opened share (s_{ib}) and its corresponding random value (r_i). It then re-computes the overall hash to make sure that the set of $2k$ encryptions is the same set that was committed in $\mathcal{H}$. Then it verifies that the share in the opened encryption is a valid share by using the verifyShare function.

From each pair of encryptions $\mathcal{E}_{i0} = \mathcal{E}(r_{i0}, s_{i0}|t), \mathcal{E}_{i1} = \mathcal{E}(r_{i1}, s_{i1}|t)$, the receiver will get the plain text of one of them (say $\mathcal{E}_{i0}$). The share from this opened encryption (s_{i0}) has to be stored. The un-opened share ($\mathcal{E}_{i1}$) is the i^{th} component of the permit. T can decrypt this permit component. When the share (s_{i1}) is recovered from that permit, it can be combined with the stored share using reconstruct() to yield the original secret s. Thus, at the end of the protocol, the receiver will have a permit in the form of set of k potential components. Only one of the components needs to be used.

The receiver checks only one encryption from each pair. However, the sender needs to compute and commit to all $2k$ encryptions before knowing which encryptions will be checked in each of the k pairs. Therefore, the probability that the sender can escape undetected after cheating (i.e., sending random strings which cannot be used as permit components) is 2^{-k}.

Sender		**Receiver**
Input : $\mathsf{VEPSend}(s, t, \mathcal{E}())$		**Input** : $\mathsf{VEPReceive}(p, t, \mathcal{E}())$
Generate k pairs of shares of s : $\{(s_{00}, s_{01}), \ldots, (s_{k0}, s_{k1})\}, \mathcal{V} = \{v_1 \ldots v_k\}$ Probabilistically encrypt each share : For $i = 1 \ldots k$: $\mathcal{E}_{i0} = \mathcal{E}(r_{i0}, s_{i0}\|t), \mathcal{E}_{i1} = \mathcal{E}(r_{i1}, s_{i1}\|t)$ Commit by hashing all encryptions: $\mathcal{H} = h(\mathcal{E}_{00}\|\mathcal{E}_{01}\| \ldots \|\mathcal{E}_{k0}\|\mathcal{E}_{k1})$	$\xrightarrow{\text{Commitment } \mathcal{H}, \mathcal{V}}$	
	$\xleftarrow{\text{Challenge } \mathcal{C}}$	Compute a challenge : $\mathcal{C} = \{c_i \in_R \{0, 1\} : i = 1 \ldots k\}$
Compute response : For $i = 1 \ldots k$: $r_i = (s_{ib}, r_{ib}, \mathcal{E}_{i\bar{b}})$, where $b = c_i$ $\mathcal{R} = \{r_i : i = 1 \ldots k\}$	$\xrightarrow{\text{Response } \mathcal{R}}$	
		Check for $i = 1 \ldots k, j = 0, 1$ (setting $b = c_i$) : compute and check all $\mathcal{E}_{ij}$ $\mathcal{E}_{ib} = \mathcal{E}(r_{ib}, s_{ib}\|t)$ $\mathsf{verifyShare}(b, s_{ib}, v_i, p)$? $\mathcal{H} \stackrel{?}{=} h(\mathcal{E}_{00}\|\mathcal{E}_{01}\| \ldots \|\mathcal{E}_{k0}\|\mathcal{E}_{k1})$ exit if any check failed
Add $p^i_{Sender} = \mathcal{E}_{i\bar{b}}$ to set p_{Sender}		Add $p^i_{Sender} = \mathcal{E}_{i\bar{b}}$ to set p_{Sender} Add s_{ib} to storage
Output : p_{Sender}		**Output** : p_{Sender}

Figure 3.3: Generic Protocol for Permit Generation using Verifiable Encryption

Let us name the input primitives of the sender and receiver VEPSend and VEPReceive respectively. Then we can use the instantiation shown in Table 3.3 to "plug" the verifiably encrypted permits into the generic protocol for the fair exchange of generatable items.

Generic Definition	Instantiation
p_P	$\mathcal{E}(r_{ib}, s_{ib}\|d_Q\|a), b = 0$ *or* $b = 1,\ i \in \{1 \ldots k\}$
bind(Q, i_P, d_Q, a, T)	send a; VEPSend($i_P, d_Q\|a, Enc_T()$); output $p_P = \{p_P^i, i = 1 \ldots k\}, a$
verify(P, i_Q, d_P, T)	receive a; VEPReceive(d_P, desc(i_Q)$\|a, Enc_T()$); output $p_P = \{p_P^i, i = 1 \ldots k\}, a$
extrAuth(p_P)	pick some p_P^i and decrypt it to recover $s_{ib}\|d_Q\|a$, return a
extrExp(p_P)	pick some p_P^i and decrypt it to recover $s_{ib}\|d_Q\|a$, return d_Q
extrItem(p_P, R_T)	pick some p_P^i and decrypt it to recover $s_{ib}\|d_Q\|a$, return $s_{ib}\|i$.
matchExp($desc, item$)	if $desc \stackrel{?}{=}$ desc($item$) return true; otherwise if $item$ was of the form $s'\|i$, then retrieve stored complement s'' corresponding to p^i_{sender}, compute s = reconstruct(s', s''), return $desc \stackrel{?}{=}$ desc(s)

Table 3.3: Instantiating the Generic Protocol for Verifiable Encryption Permits

Note that only T can execute extrExp() and extrAuth(), since they implicitly require access to T's decryption key. Also, in this instantiation, a post-processing step is required at the end of protocol *resolve*: the player who initiated the protocol needs to use the reconstruct() algorithm to combine the share received from T with the corresponding share added to the local storage during protocol *permit-trans*. Note also that protocol *permit-trans* results in a set of k permit components. Any one of these can be used during a subsequent run of protocol *resolve*. If the item reconstructed at the end of protocol *resolve* does not match expectations, protocol *resolve* be re-run so that T can use a different permit component.

Claim 11 *The construction described in Table 3.3 constitutes a valid permit according to the requirements in Section 3.2.2.*

Proof: Protocol *permit-trans* transfers k pairs of potential permit components. At the end of the protocol, k of these, one from each pair, become potential permit components. They are denoted by $\{p_P^i, i \in 1 \ldots k\}$.

- **Permit effectiveness**: From the definition of p_P in Table 3.3 , it is easy to see that the conditions for permit effectiveness are met.

- **Permit integrity**: If the encryption scheme is non-malleable [MvOV96, Section 8.7.3], the receiver cannot make any changes to the permit components created by the sender, without invalidating them. The receiver can of course generate a fake permit that satisfies the premise of the permit integrity condition; but in this case, the consequence of the condition is also true, since such a permit (supposedly created without knowing i_P beforehand) cannot yield a share that can be used to reconstruct i_P. Therefore, this instantiation satisfies the permit integrity requirement.

- **Permit confidentiality**: The sender will open only one of each pair. The shares in each pair are chosen randomly. Thus, knowing one share does not reveal any additional information about the secret itself. If the encryption scheme used to construct permits is secure, the receiver will not be able to retrieve the other share, embedded inside the unopened half of each pair. Thus, this instantiation satisfies the permit confidentiality condition.

- **Permit correctness**: If p_P^i is correctly formed, then the consequences of the rule is satisfied. If $\mathsf{verify}(\mathsf{Q}, p_P, i_Q, d_P, \mathsf{T})$ returns successfully, there is a 2^{-k} probability that all of the k permit components are incorrectly formed. Therefore, this instantiation satisfies the permit correctness condition, with probability 1-2^{-k}.

Now, I show how to construct CCGS for discrete logarithms and e^{th} roots. These constructions can be plugged into the generic protocol to yield specific permit generation schemes.

3.4.1 Verifiable Encryption of Discrete Logarithms

Bellare and Goldwasser [BG96] presented a CCGS for the Diffie-Hellman cryptosystem. We can plug their technique into our generic protocol. A CCGS for discrete logarithms is as follows. Choose prime numbers P and Q such that $P = 2Q + 1$. Pick an additive subgroup G of Z_P^* which has order Q. Let g be a generator of G. We have a secret value $s \ mod \ Q$ and a public value $p = g^s \ mod \ P$. Our goal is to verifiably encrypt s. We instantiate the generic CCGS as follows:

- $\mathsf{match}(p, s)$: $g^s \stackrel{?}{=} p \ mod \ P$,
- $\mathsf{share}(s)$:
  - two shares $s_0 \in_R Z_Q$ and $s_1 = s_0 - s \ mod \ Q$, and

  - a public string $v = g^{s_0}\ mod\ P$

- verifyShare(i, s_i, v, p): $p^i g^{s_i} \stackrel{?}{=} v\ mod\ P, i = \{0, 1\}$
- reconstruct(s_0, s_1): $s_0 + s_1\ mod\ Q$

In order to be a valid instantiation, it must be the case that one share alone does not reveal any additional information about the secret s. s_0 is chosen randomly from G. Thus, if there is an algorithm that takes s_i, p, and v as input and outputs s, then it can be used to determine the discrete logarithm of any number p in G. In other words, this construction is a computationally secure CCGS as long as it is computationally infeasible to find discrete logarithms in G.

3.4.2 Verifiable Encryption of e^{th} roots

The following construction can be plugged into our generic protocol to realise a permit generation scheme based on e^{th} roots. Let n be the product of two large distinct primes and $ed \equiv 1\ (\ mod\ \ \phi(n)\ )$ as in the usual setup of RSA. (All arithmetic is performed modulo n.) We have a secret value s and a public value $p = s^e\ mod\ n$ (implying that $s = p^d\ mod\ n$). We instantiate the generic protocol as follows.

- match(p, s): $s^e \stackrel{?}{=} p$,
- share(s):
  - two shares $s_0 \in_R Z_n$ (such that s_0 is relatively prime to n) and $s_1 = s/s_0$, and
  - a public string $v = {s_0}^e$
- verifyShare(i, s_i, v, p): if $i = 0$ $s_i^e \stackrel{?}{=} v$; if $i = 1$ $s_i^e v \stackrel{?}{=} p$
- reconstruct(s_0, s_1): $s_0 s_1$

In order to be a valid instantiation, it must be the case one share alone does not reveal any additional information about the secret s. s_0 is chosen randomly from Z_n. Thus, if there is an algorithm that takes s_i, p, and v as input and outputs s, then it can be used to determine the e^{th} root of any number p in Z_n, thereby breaking the RSA cryptosystem. In other words, this construction is a computationally secure CCGS as long as it is computationally infeasible to break the RSA cryptosystem in Z_n.

Any object that can be reduced to a verifiably shareable primitive object (such as discrete logarithms, e^{th} roots etc.) can be verifiably encrypted. Consequently, such objects can be exchanged fairly using the protocol in Section 3.2.1. In [ASW97c], Victor Shoup has shown how various types of digital signatures and coins in certain electronic payment schemes can be reduced to discrete logarithms or e^{th} roots.

3.5 Variations on the Theme

3.5.1 Verifiability of Third Party

As mentioned in 2.2.3, it is not clear how one can guarantee verifiability of third party while preserving non-invasiveness. Consider the following model as an example of a limited solution to this problem. All items are sent with non-repudiation of origin (NRO) tokens. Before a player attempts to "use" an item (e.g., enforcing a contract or cashing a cheque), he will be asked to produce the non-repudiation token. Inability to produce a valid NRO token for an item is considered incorrect behaviour. In this model, the third party can be held liable if it issues an *abort_token* to the originator of an item while extracting, and sending the item to the other player, along with an NRO token.

However, this solution is not entirely satisfactory. Firstly, there must be a way to associate the "use" of the item with having to produce an NRO token. It is possible to do this in some cases (e.g., electronic payment systems), even though it implies that the non-invasiveness property may no longer hold. Secondly, if a player is unable to produce an NRO token, it is clear that he has colluded with one other player. But it is not possible to unambiguously identify the collaborator – it may be the third party, or it may be the originator of the item, colluding with the receiver in an attempt to frame the third party. It remains an open problem to find a technique which guarantees verifiability of third party while being non-invasive at the same time.

3.5.2 Generatability via Off-line Coupons

We can avoid the relatively expensive cut-and-choose approach for verifiable encryption (i.e., where the recipient cuts out k encryptions and chooses the other k to be opened) if we allow the possibility of pre-processing. In the pre-processing stage, each player P generates a set of random strings. For each string s_0, it generates the corresponding public string v of the share procedure of a CCGS. Then, it presents the set of corresponding pairs $\{(s_0, v)\}$ to T. T encrypts the random string with its own public key, and issues a certificate containing the public string stating that it knows the secret share corresponding to it. A certificate looks thus: $cert(v) = Sig_T\{v|\mathcal{E}(s_0)\}$.

During the fair exchange of a secret string s (corresponding to a public string p known to both parties) P picks an s_0 for which it has a certificate, computes s_1 in such a way that share(s) procedure would have yielded s_0, s_1, and v, and presents s_1 along with the certificate for s_0 to the other player Q. Q can check that verifyShare$(1, s_1, v, p)$ is true and that $cert(v)$ is a valid certificate. In case of resolution, Q can present $cert(v)$ to T and be guaranteed that T can extract a s_0 from it.

This is essentially the same as generatability via pre-arranged deposits, with a subtle twist: the actual items need not be known at the time of deposit, and players do not have to reveal the actual items to T at any time.

3.6 Summary and Conclusions

3.6.1 Summary

In this chapter, I have given a formal definition of generatable items, and described an **optimistic fair exchange protocol for generatable items**, guaranteeing strong fairness. I have described some ways of adding generatability to arbitrary items. The most notable among these is the **use of verifiable encryption techniques**. I have been able to use well-understood cryptographic primitives in a novel manner to make items generatable.

3.6.2 Usage Scenarios

Now, we can revisit the scenario described in Section 1.2. The problem of making a reservation can be solved using the contract signing protocol of Section 2.2.2. The same contract serves as both Alice's non-repudiable reservation request, and BobAir's non-repudiable acknowledgement of reservation. The protocol provides strong fairness — therefore, either both BobAir and Alice end up with the contract or neither does. It may be necessary to separate the reservation request and its acknowledgement. If these two items are implemented in the form of publicly verifiable digital signatures, they are forwardable. Then we can use the exchange protocol for forwardable items in Section 2.3.2. If the items are implemented as digital signatures that can be reduced to discrete logarithms or e^{th} roots, we can use verifiable encryption to make them generatable.

The actual purchase of the ticket is more complicated. Assume that the ticket is a digital signature by BobAir. It can be down-loaded onto a portable device, such as a smartcard, or can be converted into a bar code and printed out on paper, which in turn can be later scanned in at the airport and verified. Such a ticket is forwardable, and can be made generatable, for example, using verifiable encryption.

Generatability of the payment depends on the particular payment system used. In some payment systems, the critical payment messages can be identified and reduced to a discrete logarithm or e^{th} root. This is the case with the various credit-card payment protocols (in Section B.0.1, I describe a simplified version of such a protocol called iKP [BGH+95]). The critical message in these payment systems is a digital signature, which can be verifiably encrypted. In this case, the exchange protocol for generatable items in Section 3.2 can be used.

Payment messages in some payment systems are forwardable. An example is an electronic cheque system such as the FSTC system [FST95]. Alice's cheque in favour of BobAir can be verified by T. BobAir can take the role of the originator and send the ticket first. If it doesn't hear from Alice, it can deposit the ticket with T and obtain an affidavit.

3.6.3 Conclusions

The techniques described in this chapter can add generatability to a large class of items, thereby enabling strong fairness while exchanging them. However, we still need to be able to support the possibility of subsequent disputes for two reasons.

Firstly, there still will remain cases where strong fairness is not possible using the optimistic approach. If non-optimistic approaches are not suitable (e.g., because the cost of the non-optimistic exchange exceeds the value of the items), then weak fairness will still be reasonable. However, as we discussed in Chapter 2, the optimistic fair exchange protocol merely collects evidence (in the form of *affidavit_token*). There must be an external framework where the evidence can be used.

Secondly, even after an exchange is concluded fairly, there may still be subsequent disputes. For example, along with an airline ticket, Alice might successfully purchase an online tour guide which matches a description like "Vacation in Wonderland, 123^{rd} Edition, 1998." Later, she might discover the book she received had portions missing. Alice can use the non-repudiation of origin token to prove the exact text she received during the exchange. In Chapter 4, we look at protocols for non-repudiation of origin and receipt.

Chapter 4

Non-repudiation

4.1 Introduction

4.1.1 Motivation

Non-repudiation is an essential service required for transactions with legal significance. The International Standardisation Organisation (ISO) has recently standardised techniques to provide non-repudiation services in open networks. Late versions of the draft ISO standards [ISO97] identify various classes of non-repudiation services. Two of these are of particular interest. *Non-repudiation of Origin (NRO)* guarantees that the originator of a message cannot later deny having originated that message. *Non-repudiation of Receipt (NRR)* guarantees that the recipient of a message cannot deny having received that message.

Non-repudiation for a particular message is obtained by constructing a *non-repudiation token*. The non-repudiation token must be verifiable by the intended recipients of the token (e.g., in the case of NRO, the recipient of the message; in the case of NRR, the originator of the message), and in case of a dispute, by a mutually acceptable *arbiter*.

4.1.2 Techniques

The draft ISO standards divide non-repudiation techniques into two classes. *Asymmetric non-repudiation techniques* are based on digital signature schemes using public-key cryptography. The non-repudiation token for a message is a digital signature on the message and some other pieces of related information. Non-repudiation is based on certification of the signer's signature verification key (in the rest of this chapter use of the term "verification key" means a signature verification key) by a *certification authority*(CA). Trust in this CA can be minimised by an appropriate *registration procedure*. For example, the signer and the authority may be required to sign a paper contract listing the signer's and CA's verification keys, responsibilities, and liabilities, possibly in front of a notary public. In the worst case, the CA could cheat the user by issuing a certificate with a verification key chosen by an attacker. But the supposed signer could deny all signatures based on this forged certificate by citing the contract signed during registration. Thus, trust in the CA is reduced to trust in the verifiability of the registration procedure.

Symmetric non-repudiation techniques are based on symmetric message authentication codes (MACs) and trusted third parties that act as witnesses. The trusted third party is responsible for generating and verifying non-repudiation tokens.

4.1.3 Issues

The main difficulty in using asymmetric techniques is the computational cost involved with public-key based digital signature schemes (henceforth referred to as "traditional digital signatures"). This is a particularly serious issue when anemic portable devices (like mobile phones or smartcards) are involved. Another difficulty is with key revocation. Information about revoked keys is disseminated using certificate revocation lists (CRL). Verifiers are expected to periodically refresh their copies of the CRL. Without an up-to-date CRL, a verifier runs the risk of accepting a signature made with a revoked key.

When non-repudiation is provided using only symmetric techniques, computational cost is not a problem — generating and verifying message authentication codes are typically low-cost operations compared to digital signature operations. However, the signer has to trust the third party *unconditionally*, which means that the third party could cheat the user without giving the user any chance to deny forged messages. One could reduce this trust by using several third parties in parallel or by replacing the third party by tamper resistant hardware. These two approaches increase both cost and complexity but neither of them solves the problem completely.

In this chapter, I present a novel non-repudiation technique called *Server-Supported Signatures, S^3*. It is based on one-way hash functions and traditional digital signatures. Its efficiency is comparable to the efficiency of symmetric techniques. The technique relies on the use of third parties called *signature servers* to aid in generating non-repudiation tokens. However, unlike with the ISO symmetric techniques, signature servers in S^3 are *verifiable*: if they misbehave, the victim can prove the fact in a dispute.

Ordinary users in S^3 need only be able to verify traditional digital signatures, but not generate them. Signature servers are responsible for generating digital signatures. For some signature schemes, such as RSA with a public exponent of 3, verifying signatures is significantly more efficient than generating them.

This chapter is organized as follows. The actual design of S^3 is described in Sections 4.2 and 4.3. Some variations are presented in Section 4.4. In Section 4.5, potential applications are outlined. Performance and storage costs are discussed in Section 4.6, and Section 4.7 concludes with a brief review of related work.

4.2 Server-Supported Signatures (S^3)

4.2.1 Model and Notation

There are three types of entities in an S^3 system.

- *Users* – participants in the system who wish to avail themselves of the non-repudiation service while sending and receiving messages among themselves.
- *Signature Servers* – special entities responsible for actually generating the non-repudiation tokens on behalf of the users.
- *Certification Authorities* – special entities responsible for linking verification keys with identities of users and servers.

Signature servers and certification authorities are verifiable third parties from the users' point of view.

One-way hash functions can be recursively applied to an input string. The notation $h^i(x)$ denotes the result of applying $h()$ i times recursively to an input x. That is,

$$h^i(x) = \underbrace{h(h(h(\ldots h}_{i\ times}(x)\ldots)))$$

Such recursive application results in a *hash chain* that is generated from the original input string:

$$h^0(x) = x, h^1(x), \ldots, h^i(x)$$

All entities agree on a collision-resistant one-way hash-function $h()$ and a traditional digital signature scheme. Further, the collision-resistance, and one-wayness properties are assumed to hold over iterations. Entities should "personalise" the hash function. For example, this can be done by always including their unique name as an argument: using $h(\mathsf{P}, x)$, where P is the entity computing the one-way hash. We use $h_P()$ to refer to the personalised hash function used by P. The users' security against its signature server depends on the one-way property of $h_P()$, which must hold even against the servers. In practice, this is not a problem because hash functions such as SHA-1 [NIS95] are one-way for *all* parties.

In order to minimize the computational overhead for users, $h_P()$ must be efficiently computable, and traditional digital signatures must be efficiently *verifiable*. Only signature servers and certification authorities need to have the ability to *generate* traditional signatures. SHA-1 as hash function and RSA with public exponent 3 as traditional signature scheme would be reasonable choices.

Each user, P generates a secret key, K_P, randomly chosen from the range of $h_P()$. Based on K_P, P computes the hash chain $K_P^0, K_P^1, \ldots K_P^n$, where

$$K_P^0 = K_P,\ K_P^i = h_P^i(K_P) = h_P(K_P^{i-1}).$$

$V_P = K_P^n$ constitutes P's *root verification key.* It will enable P to authenticate n messages. This is not a limitation because it is possible to link a new signature chain to the old one: before a signature chain is completely consumed, P can generate a new chain and send a non-repudiable request (signed with the verification key from the old chain) to the CA for a certificate on the new verification key.

4.2.2 Initialisation

To initialise the system, each signature server S generates a pair of signing and verification keys (S_S, V_S) of the digital signature scheme. Each certification authority, CA, does the same. The CA is responsible for verifiably binding a user O (server S) to her root verification key V_O (its verification key V_S). We assume that the registration procedure is constructed such that CA becomes a verifiable third party.

To participate in the system, a user O chooses a signature server S that will be responsible for generating signatures on O's behalf, generates a random secret key K_O, and constructs the hash chain. As described below, O can request S to transfer the signature generation responsibility to another signature server S', if required (e.g., because O is a mobile user who wishes to always use the closest server available).

O submits the root verification key $V_O = K_O^n$ to a CA for certification. A certificate for O's root verification key is of the form: $Cert_O = Sig_{CA}(\mathsf{O}|n|V_O|\mathsf{S})$. We ignore all information typically contained in a certificate but not relevant to the discussion at hand, e.g., organizational data such as serial numbers and expiration dates. The registration performed by O and CA must be verifiable, as discussed above. CA may make the certificate available to anyone via a directory service. O then deposits the certificate received from CA with S.

Each signature server S acquires a certificate containing V_S from its CA. As these are ordinary public key certificates, I do not describe them here. For the sake of simplicity, we do not include the certificates in the following protocols. They might be attached to other messages or retrieved using a directory service. We assume that the necessary certificates are always available to anyone who needs to verify a signature.

4.2.3 Generating NRO Tokens

The basic idea is to exploit the digital signature generation capability of a signature server to provide non-repudiation services to ordinary users. The basic protocol, providing non-repudiation of origin, is illustrated in Figure 4.1. We assume that a user O wants to send a message m along with an NRO token to some recipient R. The first protocol run uses $i = n$; i is decreased during each run.

1. O begins by sending (O, m, i) to its signature server S along with O's current verification key K_O^i in the first protocol flow. (In case O does not want to reveal the message to S for privacy reasons, m can be replaced by a randomised hash of m, computed using a collision-resistant hash function such that the hash does not leak *any* information about m.)

2. S verifies the received verification key based on O's root verification key (and O's certificate obtained from CA) by checking $h_O^{n-i}(K_O^i) \stackrel{?}{=} V_O$. S has to ensure that only one NRO token can be created for a given (O, i, K_O^i). If a message on behalf of O containing K_O^i has not yet been signed, S signs $(\mathsf{O}|m|i|K_O^i)$, records K_O^i as consumed, and sends the signature back to O in the second flow. $Sig_S(\mathsf{O}|m|i|K_O^i)$ is called the *candidate non-repudiation token.*

3. O verifies the received signature and stores it. It also records K_O^i as consumed by replacing i by $i-1$. The NRO token for R now consists of the pair: $(Sig_S(\mathsf{O}|m|i|K_O^i), K_O^{i-1})$. O produces this token, which actually authenticates m, by revealing K_O^{i-1}.

   In Figure 4.1, we assumed that the NRO token is sent to R via S in the third flow. Alternatively, O can send the token directly to R.

K_O^i is called the *i-th token verification key.* It corresponds to the $(n-i+1)$th non-repudiation token, $Sig_S(\mathsf{O}|m|i|K_O^i), K_O^{i-1}$. Note that O must consume the token verification keys in sequence and must not skip any of them. In particular, O must not ask for a signature using K_O^{i-1} as token verification key unless she has received S's signature under K_O^i. Otherwise, S could use that to create a fake non-repudiation token, which O cannot repudiate during a dispute.

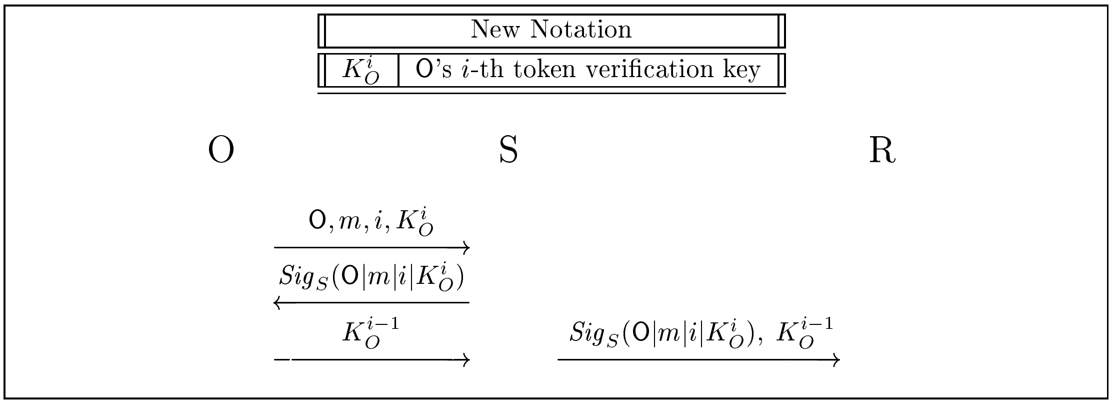

Figure 4.1: Protocol providing non-repudiation of origin.

4.2.4 Dispute Resolution

In case of a dispute, R can submit a pair (x, y) to an arbiter and claim that they constitute an NRO for message m using the i-th token verification key of O. The arbiter will do the following:

- extract the verification keys K_O^n and V_S, and verify that they are certified by CA,
- verify that S's signature on the token is valid: $\mathsf{verifySig}(\mathsf{O}|m|i|h(y), x, V_S)$?,
- that the token verification key K_O^i is in fact a hash of the alleged pre-image in the token, and
- that the root verification key K_O^n can be derived from the token verification key by repeated hashing: $h_O^{n-i}(K_O^i) \stackrel{?}{=} K_O^n$.

If these checks are successful, then the originator is allowed the opportunity to *repudiate* the token by proving that either CA or S cheated, by showing a different non-repudiation token corresponding to the same token public key.

If CA has cheated by certifying an attacker's public key in O's name, O can show its legitimate certificate by CA, on a different root verification key. Otherwise CA will be asked to prove that the root verification key was registered by O (i.e., by showing the signed contract with O). If S has cheated, O can prove it by showing a different non-repudiation token corresponding to the same token verification key.

When R receives an NRO token for a message from O, he will do the same set of tests listed above. The trust placed on this token by R depends on his trust on *both* CA and S because in a subsequent dispute, O may repudiate the message if she can convince a verifier that either S or CA cheated. In traditional digital signature schemes, the recipient of a signature takes a similar risk based on the trust placed on the certification authorities. This implies that R must have a certain trust relationship with CA and S. (For example, such trust can be based on malpractice insurance: then R knows that either O cannot repudiate its non-repudiation tokens, or, if O did manage to repudiate them by showing that S or CA is a cheater, then any losses that R may suffer as a result will be compensated for.)

If CA is honest, a cheating R has to produce an NRO token of the form: $(Sig_S(\mathsf{O}|m'|i|K_O^i),\ K_O^{i-1})$, in order to falsely claim that O has sent a message m'. If O has not revealed K_O^{i-1} yet, then one-wayness of $h_O()$ on iterates implies that anyone else will find it computationally infeasible to generate this NRO token, even if K_O^i is known. If O has already revealed K_O^{i-1}, it must have sent K_O^i to S before. According to the protocol, O reveals K_O^{i-1} only if she has received a signature from S under K_O^i which satisfied her. Therefore, O can show a different token corresponding to the same token verification key.

Suppose an adversary of O successfully breaks the one-wayness of $h_O()$ and obtains an NRO token of the form $(Sig_S(\mathsf{O}|m'|i|K_O^i),\ y)$, where $h_O(y) = h_O(K_O^{i-1})$. If y is different from K_O^{i-1}, then on being challenged with this NRO token, O can reveal K_O^{i-1}, proving that the system has been broken. This is known as the *fail-stop* property [Pfi96]. Assuming that h_O has a uniform distribution, the domain used must be larger than the range of h_O in order to achieve a reasonable level of fail-stop property. We can do this by slightly modifying the building procedure to include a random padding to the input of h_O during the computation of every link in the chain. Whenever O tries to repudiate a signature by claiming that an attacker has broken the one-wayness of $h_O()$, O can be required to reveal several successive pre-images from its hash chain. This implies that in each chain, some initial links are left unused during normal operation.

4.3 S^3 for Non-repudiation of Origin and Receipt

Non-repudiation of receipt (NRR) can be easily added to the basic protocol. Before sending K_O^{i-1} to R, S can ask R for an NRO token for NRR:m, which is then passed on to O.

The construct "NRR:m" is taken to mean "This is to acknowledge that I received m." This is illustrated in Figure 4.2. The NRR token consists of: $(Sig_S(\mathsf{R}|\mathsf{NRR}{:}m|j|K_R^j), K_R^{j-1})$.

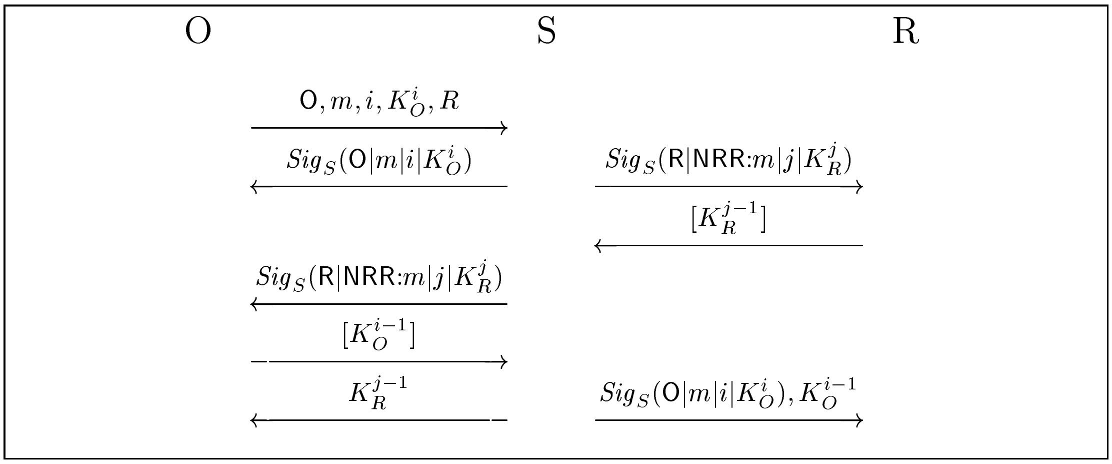

Figure 4.2: Protocol providing non-repudiation of origin and receipt.

Square brackets ([ ]) indicate that the message contained within them is sent via a confidential channel. As this protocol is just two interleaved instances of the basic NRO protocol, it still guarantees that O and R can repudiate all forged NRO and NRR tokens, respectively. Note that this protocol actually implements fair exchange of the NRO token for m and its NRR token, using S as an on-line trusted third party. If S behaves dishonestly, no fairness can be guaranteed: O might not receive the NRR token or R might not receive the NRO token.

The protocol as depicted in Figure 4.2 allows the possibility that R may refuse to send the NRR token after having received the candidate NRR token from S (the accompanying plain text of which contains m). An alternative approach is to include only a commitment to the message m in the candidate NRO token instead of the actual message itself. However, R has to trust that S will in fact send m *after* R has already acknowledged having received it. Note that if O and R happen to use different signature servers, additional inter-server message flows will be necessary.

The protocol in Figure 4.2 allows the possibility that either the NRO token or the NRR token may be optional, at the cost of an extra signature by S. The entire protocol has eight message flows. Further, the NRO and NRR tokens are linked only by the hash

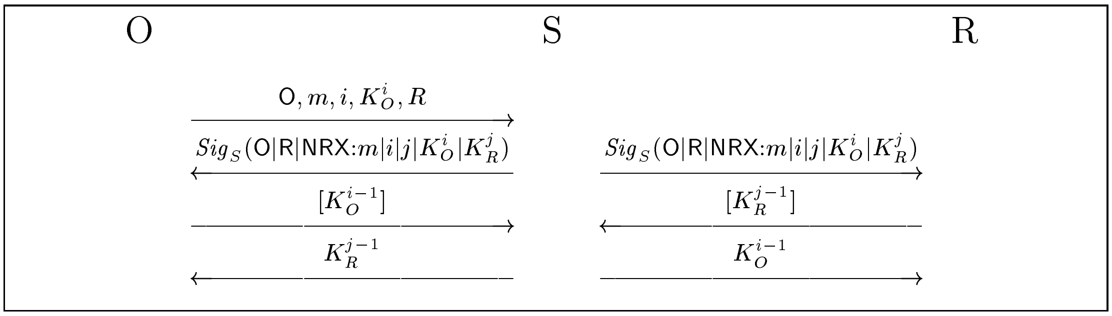

Figure 4.3: Protocol providing integrated non-repudiation of origin and receipt.

of the message. In environments where both NRO and NRR are mandatory, a modified protocol as shown in Figure 4.3 can be used. It results in a combined NRO and NRR token:

$$Sig_S(\mathsf{O}|\mathsf{R}|\mathsf{NRX}{:}m|i|j|K_O^i|K_R^j), K_O^{i-1}, K_R^{j-1}$$

The modified protocol has only seven message flows and requires only a single signature by S.

4.4 Variations on the Theme

4.4.1 Reducing Storage Requirements for Users

In order to deny forged non-repudiation tokens, O has to store *all* signatures received from S, which might be a bit unrealistic if O is not even able to compute signatures. One can easily avoid this storage problem by including an additional field H in S's signature that serves as a commitment on all the previous signatures made by S for that hash chain; i.e., an NRO token looks like: $NRO^i = (Sig_S(\mathsf{O}|m_i|i|K_O^i|H^i), K_O^{i-1})$.

The value H^i is recursively computed by $H^n = Cert_O$ and $H^{i-1} = f(H^i, NRO^i)$. The function $f()$ is a collision-resistant one-way hash function.[1] NRO^i is an NRO token on message m_i using the token verification key K_O^i.

[1]Note that $f()$ may be the same as $h_O()$. However, collision-resistance is a mandatory property for $f()$. If S succeeds in breaking the collision-resistance of $f()$, it can forge signatures which O may not be able to refute since O no longer retains all past signatures. As we saw in Section 4.2.4, collisions in $h_O()$ are not equally catastrophic from the point-of-view of O.

O has to store only the last value H^i and the last signature received from S. S has to store all signatures, and has to provide them to O in case of a dispute. If S cannot provide a sequence of signatures that fits the hash value contained in the last signature received by O, the arbiter allows O to repudiate all signatures and assumes that S cheated.

This idea of chaining previous signatures was used by Haber and Stornetta [HS91] for the construction of a time stamping service, based on the observation that the sequence of messages in H^i cannot be changed afterwards. One can combine their protocols with mine, using S as a time stamping server, as explained in Section 4.5.1.

4.4.2 Increasing Robustness

As mentioned above, S must sign exactly one message for a given user per verification key (K_O^i) in the hash chain. However, anyone can send a signature request in the form of the first flow, i.e., $(\mathsf{O}, m, i, K_O^i)$.

If S does not subsequently receive the corresponding pre-image of the current verification key (K_O^{i-1}), the current verification key is rendered invalid in any case. This implies that an attacker can succeed in invalidating an entire chain of a user by generating fake signature requests in her name.

An obvious solution would be to require O and S to share a secret key to be used for computing (and verifying) a message authentication code over the first protocol flow. An alternative solution is to give users the ability to invalidate token verification keys without having to create a new chain. The construction is only slightly more complicated than the basic protocol: instead of one chain, each user generates *two* chains: $K_O^n, \ldots, K_O^0$ and $\hat{K}_O^n, \ldots, \hat{K}_O^0$.

Each token verification key is now a pair of hash values, say, $(K_O^i, \hat{K}_O^j)$. If O receives the candidate token $Sig_S(\mathsf{O}|m|i|K_O^i|\hat{K}_O^j)$, she can either *accept* or *reject* it. O accepts by revealing K_O^{i-1}. The next token verification key is $(K_O^{i-1}, \hat{K}_O^j)$. O rejects by revealing $\hat{K}_O^{j-1}$. The next token verification key is $(K_O^i, \hat{K}_O^{j-1})$. On receiving K_O^{i-1} or $\hat{K}_O^{j-1}$, S creates the *non-repudiation token* $Sig_S(\mathsf{O}|m|i|j|K_O^{i-1}|\hat{K}_O^j)$ or the *invalidation token* $Sig_S(\mathsf{O}|\mathsf{INV}{:}m|i|j|K_O^i|\hat{K}_O^{j-1})$ respectively.

The additional signature by S is necessary because for one signature $Sig_S(\mathsf{O}|m|i|j|K_O^i|\hat{K}_O^j)$, it can easily happen that both K_O^{i-1} and $\hat{K}_O^{j-1}$ become public. That is, the combination of the first signature with one pre-image would not be unambiguous and recipient R could not depend on what he receives. Instead of making two signatures, we could use the same trick as in Chapter 3 to save a signature: S can instead include two commitments $h(a_{NRO})$ and $h(a_{INV})$ of two random numbers a_{NRO} and a_{INV} in the first signatures. Then, S can release one of the two random numbers to O. The random number together with the first signature serves as either the NRO token or the invalidation token. Note that a cheating S could generate both tokens for the same token verification key; but S still remains verifiable because O could then easily prove that S cheated, by showing the token received.

4.4.3 Support for Roaming Users

In the basic protocol, the trust placed on the signature server is quite limited — it is trusted only to protect its secret key from intruders and to generate signatures in a secure manner.[2] This limited trust enables a mobile user to make use of a signature server in foreign domains while travelling. Normally the signature server in the user's home domain will be in charge of the user's hash chain. Whenever the user requests to be transferred to a signature server in a different domain, an agreement could be signed by the user and the old signature server authorising the transfer of charge of the user's hash chain. As usual, the pre-image of the current token verification key, used to sign this agreement will become the next token verification key.

In other words, instead of having a single root verification key certificate (which includes the identity of the "home" signature server), a chain of verification key certificates could be used. The chain consists of the root verification key certificate signed by the home CA and one *hand-off certificate* every time the charge for the user's verification key has changed hands:

$$Sig_{CA}(\mathsf{O}|n|K_O^n|\mathsf{S}_0)$$
$$Sig_{S_{l-1}}(\mathsf{O}|n_l|K_O^{n_l}|\mathsf{S}_l),\ \ for\ 0 < n_l < n,\ \ l > 0$$

where, $\mathsf{S}_0 = \mathsf{S}$ and n_l is the index of the token verification key used to sign the request for the l^{th} hand-off (from S_{l-1} to S_l).

To effect a change in charge during a hand-off, the following procedure is carried out.

1. The user O sends a hand-off request to both the current signature server S_{l-1} and the intended signature server S_l. As this request must be non-repudiable, this step

[2] As mentioned in Section 4.2.4, the recipients of NRO tokens require additional assurance that their losses can be compensated in case signature servers are shown to have behaved incorrectly.

is essentially a run of the basic protocol to generate a NRO token with a message that means "hand-off from S_{l-1} to S_l requested." S_{l-1} will issue a candidate NRO token for the request using the current token verification key $K_O^{n_l}$ and O will validate the token by revealing $K_O^{n_l-1}$.

2. When the NRO token is received and verified by S_{l-1}, it generates a corresponding hand-off certificate described in the previous paragraph and sends it to both O and S_l. It will no longer generate signatures on behalf of O for that hash chain unless charge is explicitly transferred back to it at some point. In addition, it will store both the hand-off certificate and the corresponding NRO token.

3. When S_l has received both the NRO token and the hand-off certificate, it will be ready to generate signatures on behalf of O, starting with $K_O^{n_l-1}$ as the first token verification key.

During a dispute, suppose the verifier is presented with two different chains of certificates. If the chains are completely different, the dispute has to be resolved by verifying the initial registration process by CA. Otherwise, the chains must diverge at some point. There must be a single signature server who issued two different hand-over certificates for the same chain. This is the server that needs to be ultimately held responsible.

4.4.4 Key Revocation

As with any certificate-based system, there must be a way for any user O to revoke her hash chain.[3] If the currently secret portion of O's hash chain (say K_O^i, for $i = p-1,\ p-2,\ \ldots\ 1$) has been compromised, O will detect this when she attempts to construct an NRO the next time for the token verification key K_O^p: S will return an error indicating the current token verification key $K_O^q (q < p)$ from S's point of view. O can attempt to limit the damage by doing one of the following:

1. invalidate all remaining token verification keys K_O^i $(i = q,\ q-1,\ \ldots\ 1)$ by requesting NRO tokens for them, or

2. notifying S to invalidate the remaining hash chain by sending it a non-repudiable request to that effect and receiving a non-repudiable statement from S stating that

[3] Revocation by authorities is not an issue in this system because the user has to interact with the signature server for the generation of every new NR token anyway.

the hash chain has been invalidated. This can be implemented similar to the invalidation tokens described in Section 4.4.2 — except in this case the token would invalidate the entire chain and not just a single key.

4.4.5 General Signature Translation

In a more general light, the signature server in S^3 can be viewed as a "translator" of signatures: it translates one-time signatures based on hash-functions into traditional digital signatures. The same approach can be used to combine other techniques such that the result provides some features that are not available from the constituent techniques by themselves.

For example, one could select a traditional digital signature scheme (say D_1) where signing is easier than verification (e.g., DSS) and one (say D_2) where verification is easier than signing (e.g., RSA with a low public exponent) and construct a similar composite signature scheme. The signing key of an entity X in digital signature scheme D is denoted by S_X^D. To sign a message m, an originator O would compute $Sig_O^{D_1}(m)$ and pass it along with m to the signature server S. If S can verify the signature, it will translate it to $Sig_S^{D_2}(m, Sig_O^{D_1}(m))$ In other words, the composite scheme allows digital signatures where both signing and verification are computationally inexpensive.

4.5 Applications

4.5.1 Building a Secure Time Stamping Service

In Section 4.2.4, we saw that S^3 meets the standard requirements of a signature scheme. The structure of the non-repudiation tokens results in an additional property: non-repudiation tokens issued by a given user have a strict temporal ordering among them.

Recall the structure of the non-repudiation tokens described in Section 4.4.1:

$$
\begin{aligned}
& NRO^i = (Sig_S(\mathsf{O}|m_i|i|K_O^i|H^i), K_O^{i-1}) \\
where, \quad & H^n = K_O^n, \quad H^{i-1} = f(H^i, NRO^i)
\end{aligned}
$$

If $f()$ is collision-resistant, the chaining factor H^i imposes an order among the messages signed. Let us call this a *token chain.* Suppose that

$$\{NRO^i\}, i = n \ldots p, p-1 \ldots q$$

indicates the token chain of NRO tokens issued by a certain user O at a given time. It is easy to see that all $NRO^q, q < p$, must have been created after NRO^p. As NRO^p is a commitment on m_p, it follows that O knew m_p and showed it to S before NRO^q was created. O and S, either by themselves or in collusion, cannot create NRO^p after NRO^q was created. This enables us to build a time stamping service based on S^3.

Ideally, a time stamping system must be able to impose a total order on all the messages time stamped. We can adapt the approach used in [HS91], where S generates a chaining factor from a *single*, global chain. Every signature generated by S has a chaining factor from this global chain. To verify a given time stamp, one needs to know the owners of the previous (and if necessary the subsequent) time stamps generated by the time stamping server.

In Section 4.3, we discussed a protocol that results in a combined NRO/NRR token. Chaining factors can be included in this token as well. The resulting NR token will be

$$\begin{aligned}
&NR^{ij}_{AB} = Sig_S(\mathsf{A}|\mathsf{B}|\mathsf{NRX}{:}m_{ij}|i|j|K^i_A|K^j_B|H^i_A|H^j_B), K^{i-1}_A, K^{j-1}_B,\ \textit{where}\\
&H^n_A = K^n_A,\quad H^{i-1}_A = f(H^i_A, NRO^i_A),\\
&H^m_B = K^m_B,\quad H^{j-1}_B = f(H^j_B, NRO^j_B).
\end{aligned}$$

and NR^{ij}_{AB} is the same as NRO^i_A and NRO^j_B. Each time A sends a message to B using the modified protocol resulting in a combined NRX token, the token chains of A and B are "synchronised:" any $NRO^p_A, p > i$ must have been created before any $NRO^q_B, q < j$. Thus, when A's and B's chains are synchronised, they become *witnesses* to each others non-repudiation tokens made before the synchronisation point. This temporal linking of chains is transitive: if B's chain is later synchronised with C's chain, C indirectly becomes a witness to the earlier synchronisation between A and B. Although this does not necessarily result in a total order, the more chains are synchronised after a message has been signed, the greater the number of witnesses to the time of signing.

Thus, S^3 can be used to temporally link the tokens generated by S on behalf of multiple users. A practical implementation of a time stamping service can be constructed by requiring that S include a time stamp in each signature generated. The aim of this construction is not to provide an absolute guarantee that the time stamp in a document is precisely correct. Instead, the temporal ordering property of S^3 signatures is used to verify if the time stamp is plausible. In case of a dispute about the time stamp on a signature by A, the token chains of all parties synchronised to A's chain after the signature was made are examined (suppose there are n such parties). If these token chains satisfy all the temporal ordering relationships discussed above *and* if there is a sufficient number of honest parties among those linked to A's token chain, then the time stamp is correct. In this scenario, all of the $n+2$ parties involved (the recipient, S, and n witnesses) involved must collude in order to produce a fake time stamp which cannot be proved to be a fake.

When digital signatures are used as a means to provide accountability, it is crucial to have unforgeable time stamps embedded in the signatures. The usual technique to achieve this is to use a separate, external time stamping service in conjunction with a traditional digital signature mechanism. The structure of S^3 makes it a signature scheme with an integrated unforgeable time stamping facility.

4.5.2 Applications with a Fixed Recipient

As the role of the signature server is verifiable, the recipient can also play that role. This is useful in applications where several non-repudiable messages need to be sent to the same fixed recipient. An example of this is a "home-banking" (or electronic funds transfer) application, where customers send signed payment orders to their bank.

Payment messages are of the form $m = (payee,\ amount,\ date)$. When a payer wants to make a payment, he constructs a message of the above form and executes the normal S^3 protocol with the bank, resulting in an NRO token for the message. The bank then transfers *amount* to the account of *payee* and issues a special NRR token, which can be its signature on the entire NRO token. Optionally, it may also get an S^3 NRR token from the payee and forward it to the payer. The non-repudiation tokens serve as evidence of the transaction.

The idea of using a hash chain for repeated, *fixed-value* payments was suggested recently [Ped96, HSW96]. We have been able to use S^3 for payments of arbitrary values because S^3 provides non-repudiation of origin for arbitrary messages.

4.6 Analysis

Computation: Ordinary users of S^3 need to be able to compute one-way hashes and to verify traditional digital signatures. Only the signature servers and CAs are required to generate traditional digital signatures. Key generation for ordinary users is also relatively simple: the user needs to be able to generate a random number. In contrast, key generation in traditional digital signature systems is typically more complex, involving, for example, the generation of large prime numbers. In summary, the computational requirements for ordinary users of S^3 are less than those of a traditional digital signature scheme offering comparable security.

Storage: Using the improvement described in Section 4.4.1, users need to store only the last signature received from S, the secret key, the sequence number of the current token public key, and the verification keys needed to verify certificates.

Signature servers need to store all generated signatures in order to provide them to the users on demand. The stored signatures are necessary only in case of a dispute. Therefore, they can be periodically down-loaded to a secure archive.

Communication: The communication overhead of S^3 is comparable to that of standard symmetric non-repudiation techniques because a third party is involved in each generation of a non-repudiation token.

In non-repudiation techniques based on traditional digital signature schemes, the involvement of third parties can be restricted to exception handling, whereas token generation is usually non-interactive. The price to be paid for this gain in efficiency is that revocation of signing keys becomes more complicated. In S^3, revoking a key is trivial: O simply has to invalidate the current chain.

Security: In the preceding sections, I demonstrated that as long as the registration procedure, the digital signature scheme, and the one-way hash function are secure, both users and signature servers are secure with respect to their individual objectives. Furthermore, the security of originators depends on the strength of the hash function and not on the security of the digital signature scheme.

Note that in practice, traditional digital signature algorithms are not applied directly to arbitrarily long messages. Instead, a collision-resistant, one-way hash function is first applied to the message to produce a fixed-length digest or fingerprint which is then signed using the signature algorithm. Thus, even in traditional digital signature schemes, the overall security therefore depends on both the signature algorithm and the hash function. Signature servers typically have significantly more computational resources available to them than do ordinary users. For a given digital signature scheme, signature servers can afford to choose a higher grade of security (e.g., longer signature keys) than can ordinary users. The primary advantage of S^3 is that it gives ordinary users the ability to produce stronger signatures than they could have been able to by using traditional signatures by themselves in the standard way.

4.7 Related Work

Although non-repudiation of origin and receipt are among the most important security services, only a few basic protocols exist. See [For94] for a summary of the standard constructions. The efficiency problem as addressed by specific designs of signature schemes was mainly motivated by the limited computing power of smart cards and smart tokens. Schneier [Sch96] lists most known proposals. Typically they are based on pre-processing, or on some asymmetry in the complexity of signature generation and verification (i.e., either sender or recipient must be able to perform complex operations, but not both). Note that although S^3 uses a signature scheme that is asymmetric with respect to signature generation and verification, ordinary users are *never* required to generate signatures; thus, both the sender and the recipient are assumed to be computationally weak.

In his well-known paper [Lam81], Lamport proposed using hash chains for password authentication over insecure networks. There had been other, earlier proposals to use one-way hash functions to construct signatures. Merkle has presented an overview of these efforts [Mer87]. The original proposals in this category were impractical: a proposal by Lamport and Diffie requires a "public key" (i.e., an object that must be bound to the signer beforehand) and two hash operations to sign *each* bit. Using an improvement attributed to Winternitz involving a single public key (which is the n^{th} hash image of the private key) and n hash operations, one can sign a single message of size $log_2 n$ bits. Merkle introduced the notion of a tree structure [Mer87]; in one version of his proposals, with just a single public key, it is possible to sign an arbitrary number of messages. Nevertheless, it requires either a large number of hash operations or a large amount of storage in order to sign more than a handful of messages corresponding to the same public key.

Motivated by completely different factors, Pfitzmann et al [PPW91][Pfi96, Section 6.3.3] proposed a fail-stop signature protocol which uses the same ideas as S^3. There, the signature server is also the recipient of the signature (which is a sub-case in the scope of S^3), and the goal is to achieve unconditional security for the signer against the server (in the sense of fail-stop signatures). The protocol has a similar structure to the one in Section 4.2. Because of the specific security requirements, all parties have to perform complex cryptographic operations, and signatures are not easily transferable.

4.8 Summary and Conclusions

Non-repudiation techniques are important in providing accountability and legal significance to commercial transactions. In this chapter, I have presented a **new and efficient non-repudiation technique**. It has several useful properties. It allows ordinary users to make use of the superior resources of large servers to generate stronger signatures, while keeping the servers **verifiable**. The structure of the non-repudiation tokens makes it easy to provide secure time-stamping with little additional effort. Key revocation by authorities is instantaneous. The basic scheme can be modified to support user mobility.

How can S^3 signatures be used in fair exchange? In contract signing, the usage is straight-forward with the version of S^3 described in Section 4.4.2.

Consider the protocol in Figure 2.2. If the signature server S is trusted by the signer O, then me_1 can simply be $Sig_S(\mathsf{O}|m|i|K_O^i|\hat{K}_O^j)$, where m is $(V_O, V_R, text, K_O^i)$, and me_3 is K_O^{i-1}. To abort the exchange, O releases $\hat{K}_O^{j-1}$ to S and obtains a revocation token, which it forwards to T in protocol *abort*. However, if S is not trusted by O, this is not acceptable because S can collude with R and convince T to issue a replacement token. In this case, O could use a special type of S^3 chain where only even numbered token public keys are valid for usual signatures. Odd numbered token public keys are used in message me_1 of the exchange. In other words, O obtains two signatures from S for two successive token public keys K_O^{i+1} and K_O^i. It releases the first candidate non-repudiation token along with K_O^i in me_1 and K_O^{i-1} in me_3.

The pre-images are forwardable strings. So, S^3 signatures can also be used in the fair exchange protocol for forwardable items. There does not seem to be a way to use verifiable encryption with S^3 signatures, as long as ordinary one-way hash functions are used.

The purpose of collecting non-repudiation tokens during a protocol run is to be able to support claims during subsequent disputes about what happened during the protocol run. Whether the non-repudiation tokens are sufficient to win a dispute depends on the framework within which disputes would be conducted. Such a dispute handling framework has legal, policy, and technical aspects to it. Without a clear picture of how eventual disputes are going to be conducted, there is no guarantee that the collected non-repudiation tokens are of any use. A comprehensive framework for handling disputes in electronic commerce will evolve over time. In Chapter 6, I will discuss one aspect of the technical infrastructure for handling disputes in electronic payment systems. To set the stage for this discussion, in Chapter 5, I will present the design of a generic payment service.

Chapter 5

Design of a Generic Payment Service

5.1 Introduction

An important aspect of many commercial transactions is *payment*. A payment is the transfer of monetary value from one player to another. In the example scenario from Section 1.2, the initial reservation request is followed by Alice making a payment for the ticket. Ever since money was invented as an abstract way of representing value, systems for making payments have been in place. In the course of time, new and increasingly abstract representations of value were introduced. A corresponding progression of value transfer systems, starting from barter, through bank notes, payment orders, cheques, and later credit-cards, has finally culminated in electronic payment systems. Mapping between these abstract payments and the transfer of "real value" is still guaranteed by banks through the financial clearing systems. Several electronic payment systems have been proposed and implemented in the past few years [AJSW97]. The many different payment systems are incompatible with each other. Each individual player will have the ability to use only a subset of these payment systems. When Alice wants to make a payment to BobAir, they must first identify what payment systems they have in common, and then pick a suitable one from among them for the payment.

In the example from Section 1.2, Alice and BobAir used a standard on-line ticket purchase application for the reservation and sale of the airline ticket. We will use the term *business application* to refer to such a generic application which implements a certain business process. Ideally, the business application should be able to make use of any of the several common means of payment available to Alice and BobAir. Currently, the developer of such a business application has to worry about:

- making sure that the application knows how to use all the various different payment systems its users are likely to have available, and

- in case multiple payment instruments are available, providing a way to choose one of them.

A unifying framework enabling business applications to use different payment systems in a transparent manner will greatly ease the task of business application developers. In the rest of this chapter, I describe the design and implementation of such a framework called the *generic payment service*. The primary component of this generic service is a coherent hierarchy of application programming interfaces (APIs) for the transfer of monetary value. The flow of information during transactions leads to a classification of actual payment systems into a set of payment models. The API hierarchy reflects this separation into different payment models: it consists of a base API common to all payment models and extensions specific to each model. In addition to the unified interfaces, the generic service also provides mechanisms for automatic selection of the specific payment instrument to be used in a transaction; this will enable the business applications to be concerned just with the questions "how much to pay?" and "to whom?" but not with "what payment instrument to use?" More interestingly, these applications can specify requirements on payment instruments to be used for the requested transaction — for example, in terms of the security services supported by an instrument. In other words, instead of saying "pay using ABC brand credit-card," the application can say "pay using an instrument that provides payer anonymity and non-repudiation of receipt."

Thus, the design of a generic payment service is an interesting and useful effort in its own right. But it also helps in the task of developing a framework for handling disputes in electronic commerce. In our example, there are several points about which disputes can arise. There may be disputes about the reservation itself, as already mentioned in Section 1.2; there may be disputes about the subsequent payment by Alice; there may be disputes about the delivery. Developing a complete framework for handling all types of disputes is a complex task. It has legal, policy, and technical aspects to it. In order to keep the problem tractable, I will limit myself to discussing the technical aspects of payment disputes only. In Chapter 6, I discuss the issue of handling disputes in electronic payment systems. But before discussing the types of disputes that can arise in electronic payment systems and how they can be handled, we need to have a definition of the service provided by such systems. The generic payment service therefore is a pre-requisite to the discussion on handling disputes.

This chapter is organised as follows. In Section 5.2, a simple classification of various models of electronic payment systems is presented. The design of a generic payment service, based on this classification, is presented in Section 5.3. Section 5.4 describes how a new payment system can be plugged into the generic payment service. Section 5.5 describes how the a business application can use the generic payment service. A different, more general design approach is presented in Section 5.6, and its advantages are discussed.

5.2 Models of Electronic Payment Systems

There are several different electronic payment systems. All of them have the same basic purpose of facilitating the transfer of value among different parties. They differ in various aspects such as the point at which an electronic transaction is linked to the movement of real monetary value in the financial clearing system, and the degree of security provided by the system [AJSW97]. In this section, I present an intuitive model of electronic payment systems as a first step in the design of the generic payment service.

5.2.1 Players

Electronic payments involve a *payer* and a *payee*. The intent of the payment is to transfer monetary value from the payer to the payee. This is accomplished by a payment protocol. The process also requires at least one financial institution which links the data exchanged in the payment protocol to transfers of monetary value. The financial institution may be a bank which deals with monetary value represented in terms of real money; or it may be some organisation that issues and controls other forms of representation (e.g., loyalty points). Here, I use the term "bank" to mean all different types of financial institutions and the term "real money" to cover all forms of non-digital value representations used by financial institutions. Typically, banks participate in payment protocols in two roles: as *issuers* (interacting with payers) and as *acquirers* (interacting with payees). Finally, an *arbiter* may be involved in resolving disputes in the payment system.

The basic set of players involved in a payment system is illustrated in Figure 5.1. In most systems, the presence of the arbiter is not explicit: even if the necessary proofs of evidence are produced, dispute handling is done outside the payment protocol and often not even specified. Sometimes, it is not even possible to define dispute handling at the protocol level since the resolution of disputes may be subject to policy decisions of the users and financial institutions (a full-fledged payment system built on top of a given payment protocol should however provide appropriate dispute management services).

Certain payment systems might involve more players, e.g., registration and certification authorities, or other trusted third parties that provide anonymity [LMP94, Cha81] or enforce receipts for payments (for example, using a fair exchange protocol).

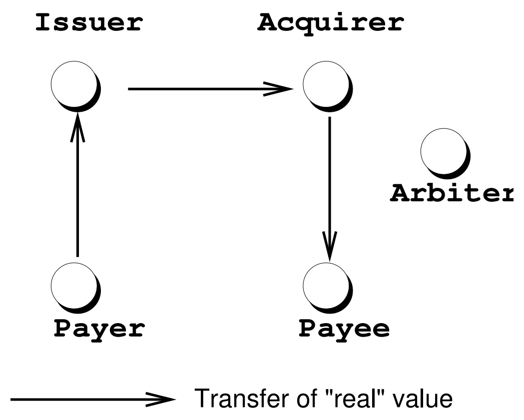

Figure 5.1: Players of a Payment System

5.2.2 Payment Models

We can classify payment systems according to the flows of information between players. Figure 5.2 lists, without claiming completeness, the four most common payment models and their information flows.

One criterion for distinction is whether the communication between the payer and the payee is *direct* or *indirect*. In the latter case, the payment operation is initiated by one player and involves only the initiator and the bank(s). The other player is notified by its bank at the completion of the transaction. An example of direct payment is paying by cash or cheque. An example of indirect payment is paying by means of a standing order or wire transfer. Most currently proposed Internet payment systems implement direct payments. Consequently, I will focus only on these systems.

A second criterion is the the relationship between the time the payment initiator considers the payment as finished, and the time the value is actually taken from the payer. There are three possibilities, identified by the names "pay-before," "pay-now," and "pay-later" payment systems. In pay-before payment systems, real value is removed from the payer ahead of time. This implies that some sort of electronic value token is issued to the payer at this time. These tokens are used during the electronic payment transaction. Since this is similar to cash in the physical world, I will use the term *cash-like* to describe the class of payment systems of this model.

Pay-now and pay-later payment systems are quite similar: in both cases, the user must have some sort of an "account" with the bank and a payment is always done by sending some sort of "form" from payer to payee (cheque, credit card slip, etc.). Thus, we can treat these two cases as instances of the same model. I will use the term *cheque-like*[1] to describe the class of payment systems of this model. Similar informal models of payment systems have been used by various others [BP89, NM95, Mas96].

A large number of proposed or existing payment systems can be grouped into these two categories. Examples of cash-like payment system include ecash,[2] NetCash [MN93], CAFE [BBC+94] and Mondex.[3] Examples of cheque-like systems include credit card protocols like SET [MV97], iKP [BGH+95], CyberCash [EBCY95] and electronic cheque schemes like FSTC [FST95].

The process of defining a generic payment service goes hand in hand with the development of a formal definition of a secure payment system and the properties it should possess. Such a formal definition will be a useful framework for verification and comparison of security properties of payment systems.

5.3 Design of the Generic Payment Service

5.3.1 Scope and Terminology

The main functionality of any payment system is to provide **value transfer services** consisting of moving electronic value from a payer to a payee, moving it back from the payee to the payer in case of a payment reversal, and converting "real money" into electronic value ("loading") or vice versa ("deposit"). The players may specify certain security attributes for this value transfer.

In the simplest case, the transfer of value happens between two end points. Such an end point is called a *purse*.[4] A purse corresponds to a single instance of a specific payment system and contains all the user information related to that instance. For example, a user who has a credit card account, an instance of a stored-value card and an ecash account will

[1] In the prototype implementation, I used the term "account-based." It was somewhat confusing because certain practical implementations of cash-like payment systems, such as DigiCash's ecash also have a notion of an "account" in the bank. Thus, in the interests of avoiding confusion, I use the term "cheque-like" here.

[2] See *http://www.digicash.com/* for more information.

[3] See *http://www.mondex.com/* for more information.

[4] Sometimes, a different term – *pocket* – is used to denote the same concept.

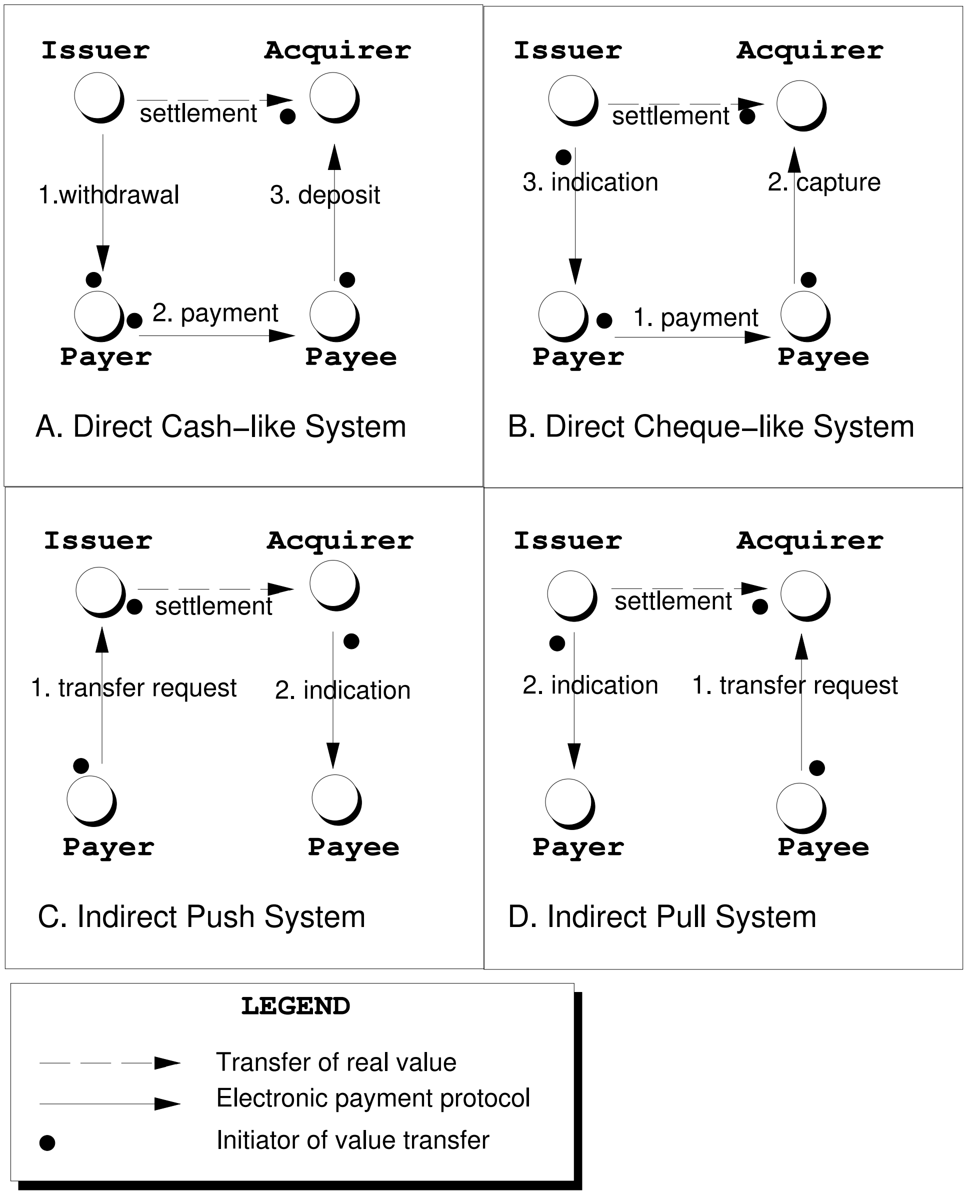

Figure 5.2: Payment Models

have three separate purses, one associated with each of the above. **Purse management services** allow a user to set-up, configure, manage, and delete purses.

Each value transfer in progress is embodied in a separate *transaction*. A purse may be involved in several concurrent transactions. **Transaction management services** allow transactions to be queried for their status, cancelled, or recovered from a crash. Before beginning a transaction, each player must choose a suitable purse. This selection may have two parts: a local decision based on preferences and requirements and a mutual decision based on negotiations. The services that enable this decision making are collectively known as **purse selection services**.

In addition to purses and transactions, a separate entity called the *payment manager* manages the overall operation of the generic payment service. Each player will have one active payment manager managing its purses and transactions. **Information services** enable the retrieval of information on the state of the payment manager, or a specific purse: for example, a list of previous transactions or statistics on all payments received and made in a certain period of time.

Finally, **dispute management services** allow the user to make claims about (alleged) past transactions as well as prove or disprove them to an arbiter. None of the payment systems introduced so far has integrated dispute handling features. Most limit themselves to the collection of evidence alone. In Chapter 6, I discuss dispute handling in greater detail.

Figure 5.3 shows the entities in the generic payment service and the services they provide. When a business application wants to make a value transfer, it first identifies a suitable purse, using purse selection services. It then asks the selected purse to create a new transaction. Transactions are treated as transient entities. Each transaction is associated with a longer-lived transaction record where all relevant information about the transaction is maintained. Some of the services are distributed over more than one entity (e.g., information services are provided jointly by the payment manager, purses, and transaction records).

5.3.2 Design Overview

To define each of the above services more concretely, I have adopted the following approach. For a given class of services (e.g., value transfer services):

1. First, identify the primitives for this service that are common to most payment systems. Describe these in the form of a *base* service interface. For example, the

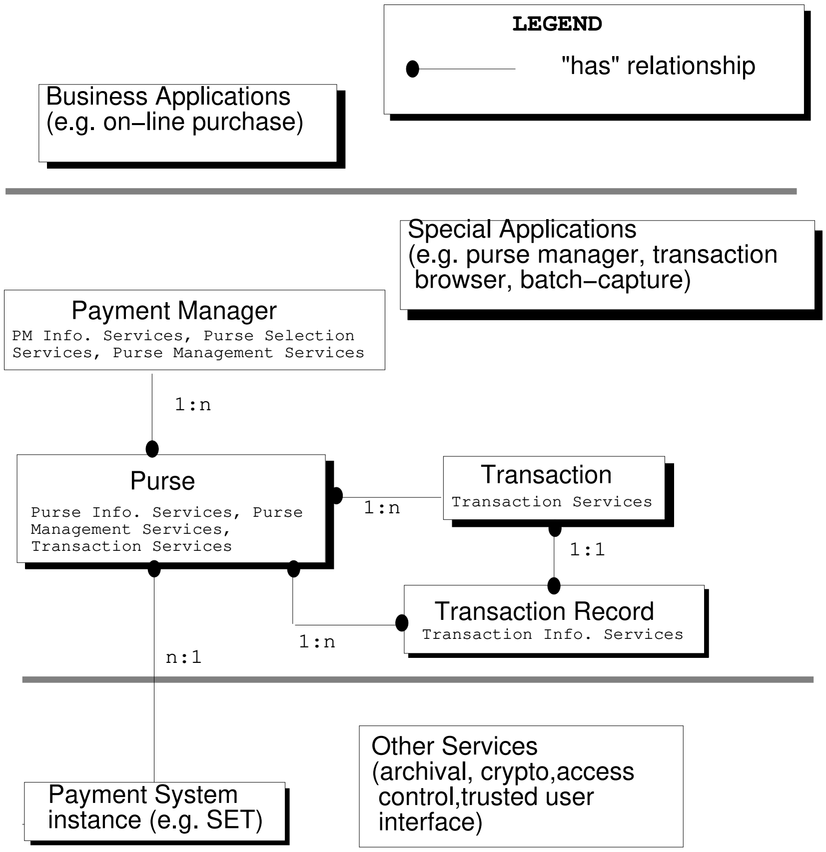

Figure 5.3: Generic Payment Service in Action (Entities in a Typical Instance)

ValueTransferServices[5] interface contains primitives like pay().

2. Then for each payment model, identify any additional primitives, not already covered in the base interface but common to all payment systems of that model. Describe these in the form of a sub-interface. For example, the sub-interface CashLikeValueTransferServices for the cash-like model defines primitives like withdraw(), that are relevant only in the cash-like model.

Some services, such as purse selection, are provided by the generic payment service itself. Other services, such as value transfer, are provided by the various payment systems. To incorporate a specific payment system into the generic payment service, a system-specific *adapter* must be built. The adapter uses the services provided by the payment system to implement services defined in the generic payment service interface. To introduce a new model, a new (possibly empty) sub-interface will have to be defined for each service interface. In the next section, I will describe the services interfaces in detail.

The high-level design described so far can be implemented in a variety of ways. I opted for an object-oriented approach. I describe the entities and services required in the generic payment service in terms of base classes and interfaces. The four main types of entities identified in the previous section (purses, transactions, transaction records, and the payment manager) are described by four different classes. Each service interface corresponds to an interface or abstract class. Concrete implementations for the services that are independent of payment systems (such as services for purse selection) are provided by the payment manager or related classes. Adapters for specific payment systems can then provide implementations for the remaining interface and abstract class methods. For example, the Transaction class in an adapter is expected to implement the services defined in the ValueTransferServices interface as well as the TransactionServices interface. The ValueTransferServices interface has model-specific extensions. The adapter for a given payment system should implement the branch of the ValueTransferServices interface corresponding to the model of that payment system. Figure 5.4 illustrates the classes that constitute an adapter and the services they implement.

The users of the generic payment service (such as Alice's and BobAir's generic on-line purchase application) can treat the various objects (such as purses and transactions) as instantiations of the generic base classes. In the following sections, I describe the services

[5]In the prototype implementation, this hierarchy was named PurseServices. Here I opt for a more intuitive name.

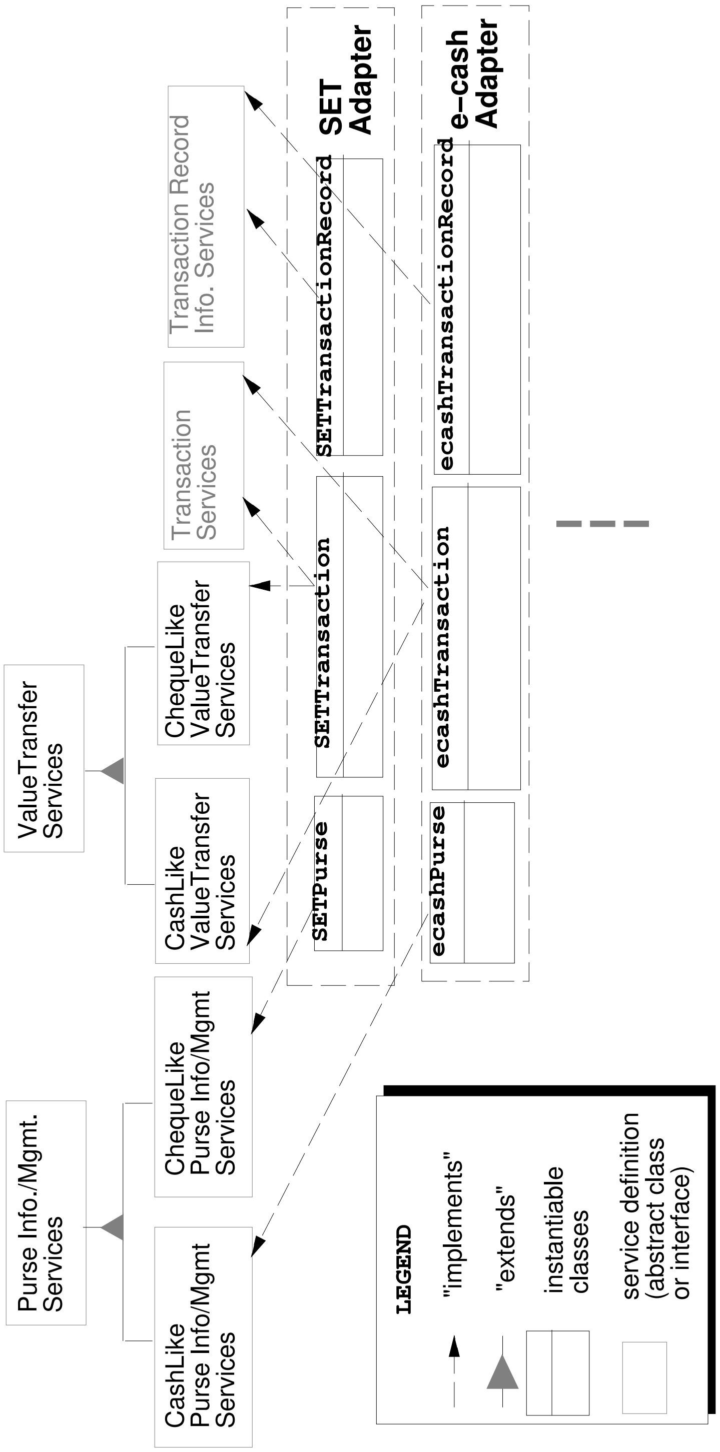

Figure 5.4: Generic Payment Service (Classes and Interfaces)

of the generic payment service and the objects that provide them (shown in Figure 5.3) in more detail.

5.3.3 Services

The primitives of the value transfer services interface are described briefly in Table 5.1. The complete set of primitives in the generic payment service API appears in Appendix A. Square parentheses ([ ]) indicate optional parameters. Exceptions and errors are not shown. Concrete Java bindings of the service descriptions can be found in [SEM96]. Note that each transaction object knows its transaction record object, which in turn knows the purse object which created them.

Primitive	Input	Output	Description
	implemented by *Transaction*		
pay	*payee, amount, options, ref.*[a]		send a payment
receivePayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment
reversePayment	*transaction record*		ask/get a refund
reverseReceivedPayment	*transaction record*		make a refund
Additions for cash-like model			
withdraw	*amount, options, ref.*		load money into purse
deposit	*amount, options, ref.*		unload money from purse
Additions for cheque-like model			
receiveRawPayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment (defer authorisation)
authorise			authorise a previous raw payment
capture	[*amount*]		capture a previous raw payment
multiCapture	*list of transaction records*		capture a set of previous raw payments

Table 5.1: Generic Payment Service: Value Transfer Services

[a] ref. allows this payment to be tightly linked to its context, e.g., an order and its description

5.3.4 Purses

A purse is an abstraction of an instance of a payment system that is available to the user. It is necessary to have services

- for creating a purse (i.e., a constructor to instantiate a purse object),
- for configuration and set-up, which will be used by purse management applications (e.g., to associate a purse with a credit card and to register with a Certification Authority),
- for initialisation, which are invoked during start-up to activate the purse,
- for creating transactions (see Section 5.3.5), and
- for information (e.g., answers to questions like "does this purse provide non-repudiable receipts for payments?").

These services are part of the purse management, purse information, and transaction services, and are implemented by the Purse class hierarchy. The base Purse class defines the aforementioned services and provides default implementations for some of them. For each payment model, the base Purse class is extended to a model-specific sub-class (e.g., ChequeLikePurse class). Adapter-writers are required to extend a model-specific Purse class and override/extend default implementations as necessary. For example, to adapt the SET payment system (a protocol for making credit-card transactions over the Internet; SET Secure Electronic Transactions [MV97]), one can define a class SETPurse which extends ChequeLikePurse.

Additionally, the Purse class hierarchy also provides services for information management corresponding to these purses (e.g., answers to questions like "what is the user name associated with this purse?" or, where applicable, "what amount is associated with this purse?").

A further classification of purses can be made based on the subset of operations supported by the purse

- **pay-only** purse for a purse that can be used to make but not to receive payments,
- **receive-only**[6] purse for a purse that can be used to receive but not to make payments, and
- **pay-and-receive** purse or just purse for a purse that can be used for making and receiving payments.

[6]Sometimes the term *till* is used to denote a receive-only purse.

Note that most current payment systems do not support pay-and-receive purses. Ecash and Mondex are examples of payment systems that do support pay-and-receive purses.

5.3.5 Transactions and Transaction Records

As mentioned, the base ValueTransferServices interface defines value transfer services that are common to all payment models. Some example services defined in this interface are: pay() makes a payment from a purse to a designated recipient, receivePayment() is the counterpart of pay(); it receives an incoming payment. Model-specific sub-interfaces may define additional services. For example, the sub-interface for the cash-like model has a service to withdraw money from the bank into the purse.

Every instance of a value transfer service is abstracted by a transaction. The PaymentTransaction class implements the value transfer services described in one branch of the ValueTransferServices interface hierarchy. Information associated with a transaction (both transient information such as state that is relevant only while the transaction is active and permanent information such as receipts or other evidence that is relevant long after the transaction is completed) is kept in a related PaymentTransactionRecord object. This can be used in crash recovery and dispute management as well as for informational purposes.

The base PaymentTransaction defines general transaction services such as trying to abort an on-going transaction or retrieving its current status. Each sub-class of the base class implements a leaf interface of the ValueTransferServices interface hierarchy (e.g., SETTransaction extends PaymentTransaction and implements the ChequeLikeValueTransferServices interface). Each leaf Purse class provides a startTransaction() method which creates a new transaction of the appropriate type (e.g., in the SETPurse class, the startTransaction() method will instantiate a SETTransaction object).

5.3.6 Payment Manager

The payment manager provides services for purse selection as well as to retrieve management information. It keeps track of the currently available purses, known payment module adapters, etc. To maintain and manage this information, the payment manager provides various services such as creation and registration of a purse, deletion of a purse, registration of a new adapter. Additional services are provided to make this information available to other objects and applications in a variety of useful ways. The manager is also responsible for initialising all the relevant components on start-up.

Selection of a purse to be used in a transaction is based on several factors: requirements for the transaction (e.g., security requirements), static user preferences, negotiation with a peer payment manager, and manual selection by user. Except negotiation, the remaining factors are all local. The payment manager provides various services to facilitate this local selection.

Negotiation with the peer for selection of the payment instrument can be done in several ways. But all negotiation protocols consist of simple request-response exchanges. Currently, negotiation is restricted to tuples containing two parameters.

- *Payment System Name*: "payment system name" is defined as follows: two purses that report the same payment system name can potentially engage in a payment transaction between themselves. Typically, the payment system name corresponds to a single <protocol, brandname> pair; e.g., SET:MasterCard and SET:Visa will be two different payment systems. It is up to the adapter to determine the payment system name associated with a purse as long as it satisfies the definition above, and

- *Amount* (value and currency).

I have designed and implemented a simple negotiation protocol which can support various negotiation policies. Two example methods, selectPayingPurse() and selectReceivingPurse() implementing a default policy, are provided: the payer is the initiator of negotiation, the payee is allowed to adjust the amounts in its reply (e.g., the merchant may add a surcharge for using a credit-card or give a discount for using ecash). It is also possible to enforce other negotiation policies.

5.4 Adapting a Payment System

In order to incorporate a new payment system into the generic payment service, a suitable adapter has to be designed (Figure 5.4 indicates what constitutes an adapter). The following steps are required in this process:

- Identify the model to which the payment system belongs (e.g., SET belongs to the cheque-like model),

- Implement a sub-class of the Purse class corresponding to the payment model identified (e.g., SETPurse extends ChequeLikePurse). This implies providing implementations for all abstract services defined in the ancestor Purse classes (e.g., Purse and ChequeLikePurse) and overriding default implementations therein, where necessary. In particular, the new class must provide a proper implementation of the setup() method: this method should allow the user to carry out all configuration necessary for the payment system,

- Implement a sub-class of PaymentTransaction class which implements the value transfer services defined in the leaf of the ValueTransferServices interface hierarchy corresponding to the payment model identified (e.g., SETTransaction implements ChequeLikeValueTransferServices and inherits from PaymentTransaction).

In addition, if any special action needs to be taken during the installation of the adapter, a suitable installation hook must be provided. A standard installation application is available as part of SEMPER. It performs two actions: installing the contents of the adapter module in the correct locations, and registering the name of the new purse class with the payment manager. If there are any adapter-specific installation procedures, they must be implemented in the form of an installation hook defined in the ModuleInstallHook interface.

5.5 Using the Generic Payment Service

Once a user has installed one or more payment instruments along with their adapters on her system, two types of usage are possible: making payment transactions from business applications, and using *special purpose applications* which are meant to manage the the configuration and operation of the generic payment service.

5.5.1 Payment Transactions

The primary use of the generic payment service is via business applications making payment transactions.

A user will initiate payment transactions using some sort of a high-level business application (e.g., a web-browser or a CD-catalogue reader). Figure 5.5 shows the object interactions that take place at the payer end during an execution of a typical payment transaction. The important things to note are:

- the user need not specify the payment instrument to use if he does not want to; the payment service can be configured to prompt him for selection of payment instrument if it cannot do so by itself, and
- the application is not aware of the specific payment instrument being used; it deals with a generic `Purse` and `PaymentTransaction` objects. It need not even know the model to which the chosen purse belongs.

The sequence of events at the payee side is similar, with minor differences. The payee application is probably an unattended merchant server. Thus, there will be no user interaction. There may be interactions with third parties during the transaction. For example, in a cheque-like system, the payee's adapter may contact the acquirer for authorisation. One can also imagine a payment system where the payer's adapter has to obtain some sort of a credential from the issuer before each payment. All such communication with third parties are carried out within the adapter — the calling applications are typically unaware of them.

This example is also intended to give an idea about how the generic payment service enables business application development. The primary services used by the business applications are purse selection and value transfer between payer and payee. Both of these are common to all payment systems. Thus, a large class of applications using the generic payment service need not be aware of system- or model-specific details. Certain special applications (see next section) will make use of the model-specific components of the generic payment service.

5.5.2 Special Applications

The second category of usage is via special applications. The most important special application is the purse management tool.

Purse Management

Before being able to use an installed payment instrument, a purse corresponding to it must be created and configured. A special *purse management application* is provided for this purpose. Changes to purses are written out to stable storage. Purse management is an infrequent activity (typically, once a purse is created and configured, it can be used in several subsequent payment transactions). Purse management makes use of a `setup()` method provided by the `Purse` class in an adapter. This method must implement all the

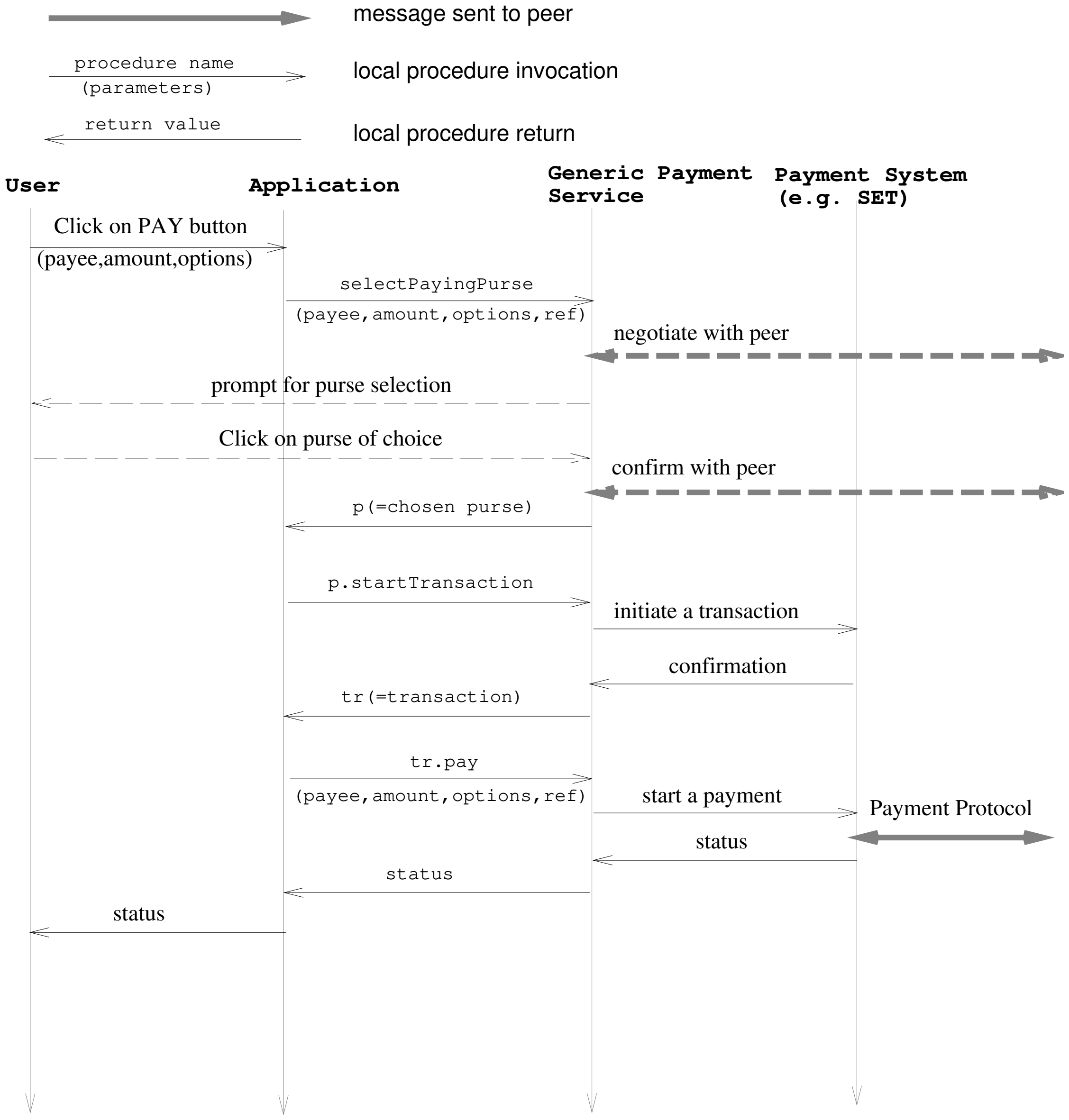

Figure 5.5: Interactions During a Payment (dotted lines indicate optional flows)

necessary configuration for that payment system. For example, the setup() method of the SETPurse allows the user to enter the credit card information (cardholder name, brand, number, expiry date) which are then stored as part of the purse state.

Other Applications

There can be a number of other special applications. Some of these are model-specific. A *batch capture* application can be used by the merchant to capture a set of received payments for cheque-like purses; typically this will be used as part of end-of-day processing. A *withdrawal* application can be used to load money into cash-like purses. The SEMPER prototype implementation comes with two model-independent special applications: a *transaction browser* allows the user to browse through accumulated transaction records; a *module installer* allows a user to install a new payment instrument along with its adapter.

5.6 Token-based Interface Definition

In the original design, I assumed a synchronous model since the first version of the SEMPER architecture did the same [SEM96]. However, I have defined a *token-based* interface which can support an asynchronous model in a straight-forward manner. The token-based interface is inspired by the GSS-API [Lin97] approach. It has two types of methods:

- one "starter" method for each different type of protocol; the starter method returns a token containing the first message of the protocol, and
- a common "processor" method; this takes a token as input and depending on the internal state of the protocol run, may return another token as output.

In the token-based model, the payment service does not engage in any direct communication with the peer. Instead, the caller is expected to take care of the communication. The payment service is still responsible for maintaining the state of a protocol run. The initiating caller invokes an appropriate starter method in the payment service API to start a protocol. Typically, these starter methods will return a token as output. The initiating caller application is expected to communicate this token to its peer entity, the responding caller application. The latter in turn will invoke the processor method on its instance of the payment service and give the received token as input. From this point

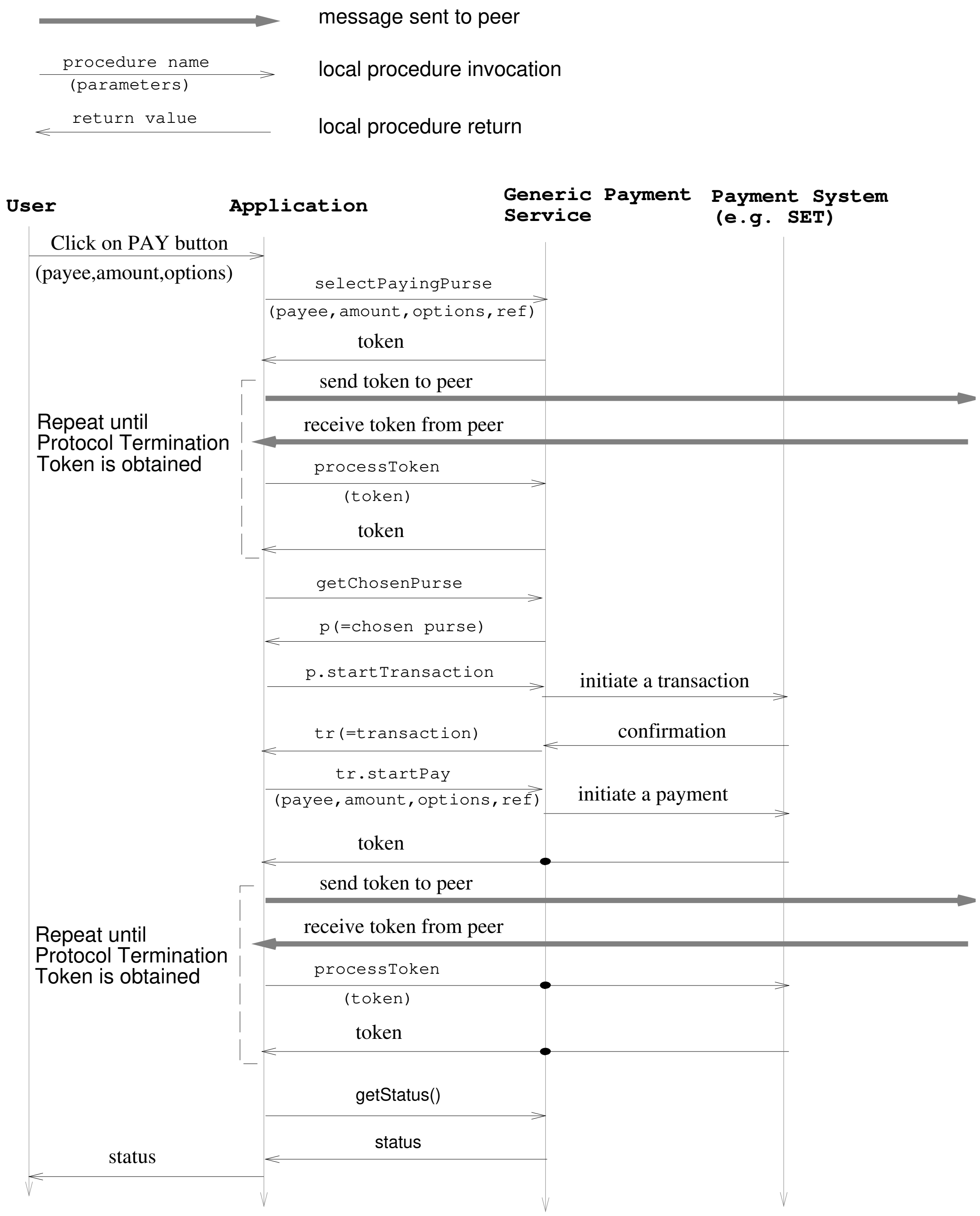

Figure 5.6: Interactions During a Payment in the Token-based Model (Dotted lines indicate optional flows)

on, whenever a caller entity receives a token as output from the processor method, it will send the token to its peer; whenever a caller entity receives a token from its peer, it will invoke the processor method on its payment service giving the received token as input.

A token-based version of value transfer services is defined in an interface hierarchy called TValueTransferServices parallel to the ValueTransferServices interface hierarchy. For each method (e.g., pay()) in the latter, a corresponding starter method (e.g., startPay()) is defined in the former. In addition, a common processor method processToken() is defined in the TValueTransferServices interface. Figure 5.6 illustrates object interactions in the same scenario depicted in Figure 5.5, but with a token-based interface for negotiation and value transfer.

Since there is no peer-to-peer communication taking place inside the generic payment service, the caller does not have to block on service invocations. The designer of the calling application has the freedom to use an asynchronous implementation architecture. More importantly, the token-based approach can allow an application to supplement the level of security provided by a payment system by transporting the tokens via a channel with particular security attributes. For example, even though payment protocol messages in SET are encrypted, an eavesdropper may be able to determine and link the identity of the payer and payee by watching the network addresses in the payment messages. With a token-based interface, if the applications were able to establish an untraceable communication channel between them, they can extend the untraceability to SET payments as well.

In the current implementation, the interface TValueTransferServices is optionally implemented by sub-classes of the PaymentTransaction class. Since the token-based version is more general than the synchronous version, it deserves to be the default value transfer services interface.

5.7 Related Work

U-PAI [KGMP+96] is being developed as part of the Stanford Digital Libraries project.[7] Their focus is on providing a unified interface to payment services. They do not address negotiation for parameters before a payment transaction begins; nor do they explicitly address issues like refunds. They also appear to assume a distributed object infrastructure such as CORBA and do not have a very clear security and trust model.

[7] See *http://www-diglib.stanford.edu* for more information.

The Joint Electronic Payments Initiative (JEPI) of the W3C Consortium focussed only on defining the protocol for the negotiation of various payment related parameters such as the payment system. The scope of my work roughly corresponds to the scope of these two projects taken together.

In 1996, Sun announced their **Java Electronic Commerce Framework (JECF)**.[8] The framework is still in the process of being defined. Their emphasis is on the payer side: payers will be able to down-load different "payment cassettes" (a cassette roughly corresponds to a payment instrument and its adapter in our terminology) and integrate them into their JECF installation. They also propose a sophisticated general access-control scheme which can be used in my design.

The **E-CO System project** had roughly the same scope [Bah96] as my work although their main focus, until the project was discontinued, was on establishing APIs and mechanisms for payment negotiation [BN96].

5.8 Summary and Conclusions

In this chapter, I have described the design and implementation of a **generic payment service**, which includes the definition of a common application programming interface for payment systems. The main advantages of the generic payment service framework are **generality**, **transparency** and **abstraction**. It is general because any payment system can be incorporated into the framework. The unified service definition hides the details of particular payment systems from business applications, allowing them to be implemented independent of payment systems. Developers of these applications do not have to know about details of particular payment systems. Finally, they can specify abstract requirements on the payment instrument to be used for their value transfer transactions.

Complete Java bindings of the payment service interfaces can be found in the web page of the SEMPER deliverable D03 [SEM96]. A prototype of the generic payment service with all the basic functionality has been implemented as part of the SEMPER project and tested using a "dummy" payment system. This work served as a basis for the design of IBM's Internet payment framework, SuperSET. Adapters for a variety of other payment systems, including SET, ecash, a stored-value card system called Chipper,[9]

---

[8]See *http://www.javasoft.com/commerce* for more information.

[9]See *http://www.chipper.com/* for more information.

and an electronic cheque system called MANDATE, have been developed (or are under development) by various partners in the SEMPER consortium.

In Chapter 6, I will describe a generic approach to handling disputes in payment systems, based on the generic payment service.

Chapter 6

Handling Disputes in Payment Systems

6.1 Introduction

6.1.1 Disputes in Electronic Commerce

In the previous chapters, we saw that support for handling disputes is necessary for two reasons. First, electronic commerce transactions typically have legal significance in the real world. This means that even if a transaction is concluded successfully, there may be subsequent disputes about what happened during the transaction (or whether in fact an alleged transaction took place). Second, practical considerations may sometimes render it desirable to settle for weak fairness: even if the system is not able to make the fairness guarantee to a correctly behaving player during normal operation, it must arrange to gather sufficient evidence so that the player can recover fairness by initiating a dispute. "Standing order" payments is an example of such a system. The payer instructs his bank to allow the payee to request periodic value transfers (e.g., for paying utility bills). While the payer cannot prevent the payee from transferring more money than necessary (e.g., the amount of the monthly bill), he can prove the amount transferred by obtaining a statement from the bank. Earlier, we saw other examples of systems with weak fairness — e.g., in the fair exchange protocols in Chapter 2, or in the case of fairness with respect to the signature server in the S^3 signature scheme in Chapter 4.

Thus, the ability of an honest player to win any dispute about a past or current transaction is often an important requirement. In a dispute, there is a set of (one or more) players called *initiators* who start the dispute and another set of players called *responders* who participate in it. A special player called the *verifier* or *arbiter* makes the final decision regarding disputes, according to some well-defined procedures which can be verified by anyone. The initiator(s) try to convince the verifier of a *claim*. Initiators may support their claims by producing evidence or engaging in some sort of a proof protocol. Responders may attempt to disprove the claims. The verifier analyses the claims made and the evidence presented and makes a judgment as to whether a dispute claim is valid or not.

6.1.2 Handling Disputes

Even those systems which have accountability as one of their major goals (signature systems such as RSA [RSA78], payment systems such as SET [MV97], or integrated electronic purchase systems such as NetBill [CTS95]), usually limit themselves to the generation and collection of evidence. Analysis of these systems may include proofs which demonstrate that the collected evidence is enough to win any disputes. It is assumed that such evidence can be used in some dispute resolution procedure external to the system. Obviously, this is not practical if the system needs to make decisions based on the outcome of disputes.

Evidence tokens are essentially part of the internal structure of the system. For instance, a payment receipt in the form of a digital signature is outwardly just a string of bits. The semantics to the evidence is added by the system itself. Outside the system (that is, from the point of view of the user of a system), what is necessary is to know what the evidence *means*, and how it can be *used* in disputes. The structure and raw contents of the evidence itself are not relevant outside the system. Thus, a system should support a *dispute service* just like it provides its primary service (e.g., payment service). The dispute service should specify how to initiate and resolve disputes for the given primary service.

It is unlikely that completely automated dispute resolution is always feasible, or even desirable. But complete automation should not be our intent. Instead, the dispute handling service should be used as a tool in human-driven dispute resolution. There are several projects that attempt to investigate the legal aspects of human-assisted on-line arbitration [Kat96]. The verifier in the dispute handling service is perhaps an expert witness who will use the dispute service to aid him in his testimony. It may not necessarily be a legally competent authority. For example, it may be an entity like the Online Ombuds Service [Kat96] which helps players to resolve their disputes. In this case, the verifier does not make the final decision. Instead, it presents an analysis of the evidence (in terms of likely decisions, and the set of assumptions that support them) to a human judge. The verifier may even be one of the players themselves. For example, if a customer claims to a merchant that he has already made a payment, the first step would be to try resolving the dispute without using an external verifier. In this case, the merchant may ask the customer to prove his claim: they run the dispute resolution protocol with the merchant playing the role of the verifier. Even before complaining to the merchant, the payer may have run the dispute resolution protocol playing the role of the verifier himself so as to confirm that his claim is provable. Also, the same evidence may be interpreted differently by different verifiers (depending on their policies, trust assumptions, and context).

The first step in developing a coherent approach to dispute handling is to figure out how to define a dispute handling service given the description of some primary service. To keep the problem tractable, we focus on handling disputes in payment systems. In

Chapter 5, a generic payment service was developed, abstracting away details specific to individual payment systems. Such a generic service definition enables the users (either human users or applications acting on their behalf) to invoke services in a system-independent fashion. The advantages of having a generic service definition are diminished if the handling of disputes is system-specific. Therefore, our aim is to head towards the development of a unified framework, rather than focussing on a specific payment system. Even though our focus is on payment systems, I will attempt to stay general as far as possible so that the approach outlined here has the potential of being extended so that it is applicable to other types of generic service definitions as well.

This chapter is organised as follows. In Section 6.2, the problem of how to express dispute claims is studied. As we saw, dispute claims in a generic service must be generic as well. I will use the generic payment service as an example to illustrate dispute claims. In Section 6.3, the problem of how to map evidence tokens to dispute claims is investigated. This mapping is system-specific and needs to be done internally in every system (see Section B.0.1 for an illustration).

6.2 Expressing Dispute Claims

6.2.1 Model and Notation

We use the ISO-OSI approach of modelling a distributed system [Lin83] by defining the **service** provided by it. The purpose of a system is to provide some capabilities to its users. In a distributed system, users are typically not co-located. A user accesses a system at a **service access point**. The interaction between a user and the system is carried out by invoking **service primitives** at access points. A service primitive can be either an input to the system or an output from the system. Each service primitive may be associated with a set of **service parameters**. The service parameters represent the information exchanged between the user and the system during a service primitive invocation. The **service** of the distributed system is essentially a statement of the capabilities provided by it. In concrete terms, the service is defined by the interface between the users and the system as the complete set of service primitives.

The problem I address is as follows: given the description of a *primary service*, how can we derive the description of a corresponding *dispute service*?

6.2.2 What to dispute?

Consider a payment system which implements the services defined by a generic payment service described in Chapter 5. Assume that the system has already been appropriately initialised. The primary purpose of the generic payment service is the transfer of value from payer to payee. The description of value transfer services is given in Table 5.1.

To make a value transfer, the payer tells the system who the payee is, what amount is to be transferred, and certain other parameters. Consider the example in Section 1.2. Alice (or the application that she uses for on-line ticket purchase), will tell the payment system,

- pay \$200 to BobAir ("#434: for flight 822 on Jan 19").

In the generic payment service, this is represented by the service primitive pay which takes the following pieces of information as parameters: payee, amount, ref (the external reference strings enables the payment transaction to be linked to an external context). Figure 6.1 illustrates the interface events during a payment. The service primitives involved, along with their parameters specify the characteristics of the value transfers.

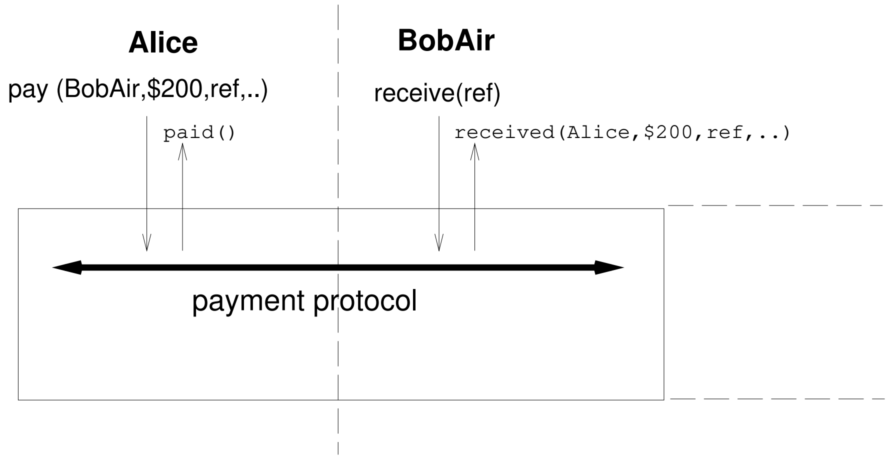

Figure 6.1: An Example Payment Transaction

Before formally defining a language to express dispute claims, let us attempt to get a feel for the kinds of claims that need to be expressed. What sorts of disputes, related to the above value transfer, does the payer (Alice) expect to be able to initiate and win? For example, Alice may want to claim that

- she paid \$200 to BobAir (if BobAir refused to send the tickets claiming no payment was made), or
- her payment was made before Jan 12 (if BobAir refused to sell reduced fare tickets claiming that Alice did not make the payment before the deadline, Jan 12).

Some disputes may be about negative claims: for example, BobAir may want to prove that

- BobAir did **not** receive \$200 from Alice.

In other words, dispute claims are statements about the characteristics of value transfer. These characteristics are determined by the service parameters used, and additional contextual information (such as the time of value transfer).

In addition to the payer and payee, a financial institution may be involved in creating a digital representation of money or converting it back to real value. Thus the value transfer may involve two or more sub-protocols involving different pairs of players. For example, in the cheque-like model, the payer sends a "form" (e.g., a cheque or a credit card slip) to the payee using a payment protocol and the payee may use a deposit protocol to claim the real value. This leads to two other types of dispute claims.

- Recall that BobAir made an offer to Alice for a cheap ticket if she made the payment before Jan 12. Alice goes through the steps of the payment protocol (e.g., sending a credit-card slip). However, BobAir changes its mind after receiving the credit-card slip — he does not "capture" Alice's payment. Alice cannot of course prove that the value transfer took place. But if she has a signed acknowledgement from BobAir, she can prove that the value transfer *could* have taken place without further help from Alice, if BobAir had wanted.
- Suppose Alice pays \$200 to BobAir using a debit card. Later, she finds an entry in her monthly statement indicating a debit of \$300. Alice may now want to start a dispute with the bank claiming that she approved a debit of only \$200. In other words, a single original transaction could lead to two different types of disputes: one involving the payer and the payee and the other involving the payer and the bank.

6.2.3 Value Transfers as Primitive Transactions

Users expect a system to provide a certain service. Therefore, disputes in the system are about an instance of the service that was or could be provided. The primary service provided by the generic payment service is value transfer from one player to another. The model in Chapter 5 assumes four types of players involved in value transfer: the payer, the payee, the issuer, and the acquirer. Value transfers between the issuer and acquirer are carried out over traditional banking systems; this transfer is outside the scope of the generic payment service. Thus, for the purpose of our disputes, we will consider the issuer and acquirer as a single entity, called the *bank*. This leads to three types of value transfer as shown in Figure 6.2.

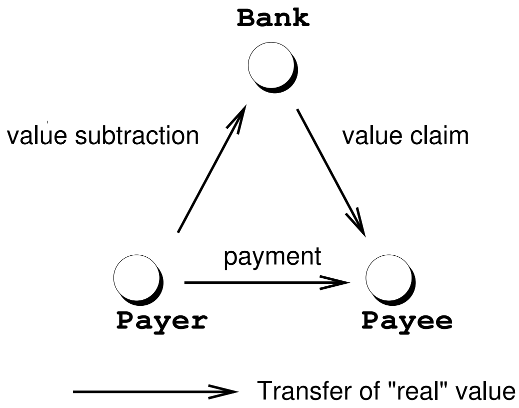

Figure 6.2: Value Transfer Transactions

- In *value subtraction*, a user allows the bank to remove "real value" from the user; this implies the user's right to spend "electronic value." In cash-like payment systems, this corresponds to the withdrawal of coins. In cheque-like payment systems, this is implicit in the payment itself. The players involved in a value subtraction are a bank and a user.

- In *value claim*, a user requests that the bank gives "real value" to the user. In cash-like payment systems, this corresponds to the depositing of coins. In cheque-like payment systems this is either part of the payment or corresponds to the separate capture transaction. The players involved in a value claim are a bank and a user.

Primitive (See Table 5.1)	Types of Value Transfer
cheque-like model [a]	
pay	*payment, value subtraction*
receivePayment	*payment, value claim*
reversePayment	*payment, value subtraction*
reverseReceivedPayment	*payment, value claim*
receiveRawPayment	
capture, multiCapture	*value claim*
cash-like model	
pay	*payment*
receivePayment	*payment, value claim*
reversePayment	*payment*
reverseReceivedPayment	*payment, value claim*
withdraw	*value subtraction*
deposit	*value claim*

[a] authorise is not treated here since it does not involve a value transfer. But if the authorisation process involves non-repudiable evidence (e.g., guarantee from the bank to hold an amount for a certain time), it may be relevant in disputes. It would correspond to the *feasibility* of a value transfer.

Table 6.1: Types of Value Transfer

- In *payment*, the payer transfers value to the payee. The players involved in a payment are the payer and the payee.

This is the approach taken in [PW96]. The relationship between the service primitives of Table 5.1, and the types of value transfer identified here is shown in Table 6.1. We will not refer to specific protocols or service primitives in order to express dispute claims. Instead, we will state the claims in terms of whether an intended service did or could take place. For this purpose, we introduce the notion of a *primitive transaction*. Each instance of a value transfer described above corresponds to a primitive transaction. In some cases, a primitive transaction corresponds to an actual protocol run of the underlying payment system.

For example, a withdrawal protocol run in a cash-like system is a *value subtraction* primitive transaction (Figure 6.3). In other cases, it may only represent the *view* of a subset of the players. For instance, a payment protocol run in a cheque-like payment system is seen by the set {payer, bank} as a *value subtraction* primitive transaction while it is seen by the set {payer, payee} as a *payment* primitive transaction (Figure 6.4).

Note that these subgroups correspond to the different applications that a player may use. For example, Alice used the standard on-line ticket purchase application in our example scenario (Section 1.2). This application knew about BobAir, but does not have to know about Alice's bank (in fact, it need not even know which payment system was used to pay for the ticket). The purchase will be viewed by this application as a *payment* primitive transaction, involving only Alice and BobAir. Alice may also have a banking application which she uses to interact with her bank in managing her bank account. This application does not know about BobAir. It will see the purchase as a debit from Alice's account — in other words, as a *value subtraction*.

If a value transfer is reversed (e.g., the payee refunded the payer), it is equivalent to the value transfer not having taken place at all. However, it must still be possible to express a claim like "Alice did pay $200 to BobAir in the past" which must be true, even if the payment was later refunded by BobAir. If this is not sufficient, we can make reverse value transfers to be distinct primitive transactions.

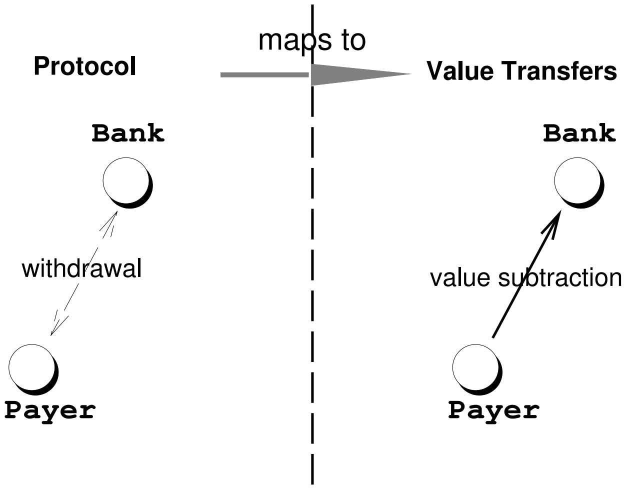

Figure 6.3: Primitive Transactions in a Cash-Like System for *withdrawal*

6.2.4 Statements of Dispute Claims

Syntax

We express a statement of a dispute claim as a formula in a first-order logic with certain modal extensions. The language of the logic consists of the following symbols: logical connectives, typed variables, typed constants, and relational connectives (which are functions of variables/constants of the appropriate type).

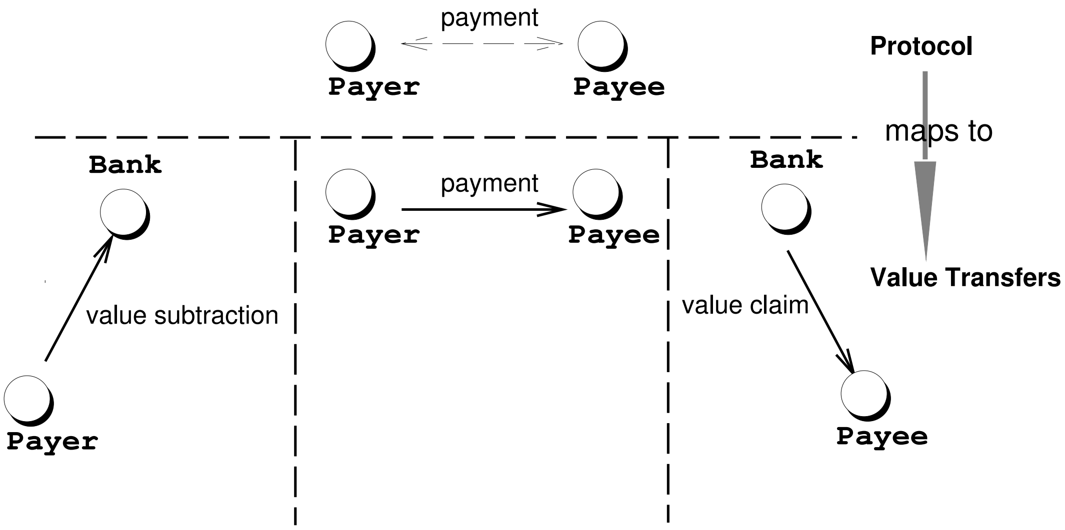

Figure 6.4: Primitive Transactions in a Cheque-Like System for *payment*

There are three types of variables: primitive transaction (pt), roles, and attributes. The pt variable can take its value from a well-defined enumerated set. In the case of the payment service, this set consists of PAYMENT, VALUE_SUBTRACTION, and VALUE_CLAIM. Each primitive transaction has a set of well-defined *role variables* associated with it. For example, PAYMENT has payer and payee as associated role variables. The role variables belong to a type called *id_val*, which represents distinguished names according to some well-defined naming scheme (e.g., account numbers or certified e-mail addresses). Each primitive transaction also has a well-defined set of *attribute variables* associated with it. For simplicity, we assume that all value transfer primitive transactions have the same set of attribute variables: amount, time,[1] and ref. The attribute variables are typed — they take their values from the appropriate domains. In the case of the payment service, we assume that amount, time, and ref take values from domains named *amount_val, time_val, and ref_val* respectively. Table 6.2 summarises the set of variables for the generic payment service.

Each attribute, depending on its type, has a finite set of relational operators associated with it. Table 6.3 lists the variables in the generic payment service, their domains, and

[1]There may be several different timestamps involved; for simplicity, we assume that there is only one instant at which the transaction is considered to have taken place.

Primitive Transaction	Roles	Attributes
VALUE_SUBTRACTION	user, bank	amount, time, ref
VALUE_CLAIM	user, bank	amount, time, ref
PAYMENT	payer, payee	amount, time, ref

Table 6.2: Dispute Claim Variables for the Generic Payment Service

applicable relational operators. We also allow two logical connectives: $\wedge$ (conjunction) and $\neg$(negation), a parenthetical operator for specifying precedence, and modal operators called **can_without**, **could_without**, and **once**. Two additional operators called **never** and **always** are also used. A comma indicates concatenation.

variable	domain	relational operators
pt	{PAYMENT, VALUE_SUBTRACTION, VALUE_CLAIM }	$=$
$<\mathit{role}>$	*id_val*	$=$
amount	*amount_val*	$<\ \leq\ =\ \geq\ >$
time	*time_val*	$<\ \leq\ =\ \geq\ >$
ref	*ref_val*	$=$

Table 6.3: Attributes and Operators of Primitive Transactions

The rules to construct valid dispute claim statements are described in the family of grammar specifications shown in Table 6.4. Angle brackets indicate place holders which must be instantiated to derive a concrete grammar. Ellipses indicate that in a concrete instantiation, the rule may contain a repetition of same type of terms as the preceding term. Given a particular service, such as the generic payment service, the place-holders are instantiated, leading to a concrete, single grammar, as shown in Table 6.5. From now on, we will simply write PRIMITIVE_TRANSACTION_NAME to denote the predicate 'pt=PRIMITIVE_TRANSACTION_NAME.' Also, when a conjunction ($\wedge$) is obvious, we omit it. For example, 'PAYMENT payer=Alice payee=BobAir' is shorthand for 'pt=PAYMENT $\wedge$ payer=Alice $\wedge$ payee=BobAir'.

claim	::=	role **claims** claim_stmt
claim_stmt	::=	modal_stmt $\Vert$ $\neg$ modal_stmt $\Vert$ (modal_stmt) modal_stmt $\wedge$ modal_stmt
modal_stmt	::=	possibility_stmt $\Vert$ certainty_stmt $\Vert$ basic_stmt
certainty_stmt	::=	**always** basic_stmt $\Vert$ **never** basic_stmt
possibility_stmt	::=	role_set **could_without** role_set basic_stmt $\Vert$ role_set **can_without** role_set basic_stmt $\Vert$ **once** basic_stmt
role_set	::=	role $\Vert$ role, role_set
basic_stmt	::=	role_part $\wedge$ attr_part
role_part	::=	pt=$<$*primitive tx*$>$ $\wedge$ role_list $\Vert$ ...
role_list	::=	role=*id_val* $\Vert$ role=*id_val* $\wedge$ role_list
attr_part	::=	true $\Vert$ attr_val_pair $\wedge$ attr_part
attr_val_pair	::=	$<$*attr*$>$ $<$OP$>$ $<$value$>$ $\Vert$ ...
role	::=	$<$*role*$>$ $\Vert$ ...

Table 6.4: Family of Grammars for the Dispute Claim Language

claim	::=	role **claims** claim_stmt
claim_stmt	::=	modal_stmt $\Vert$ $\neg$ modal_stmt $\Vert$ (modal_stmt) modal_stmt $\wedge$ modal_stmt
modal_stmt	::=	possibility_stmt $\Vert$ certainty_stmt $\Vert$ basic_stmt
certainty_stmt	::=	**always** basic_stmt $\Vert$ **never** basic_stmt
possibility_stmt	::=	role_set **could_without** role_set basic_stmt $\Vert$ role_set **can_without** role_set basic_stmt $\Vert$ **once** basic_stmt
role_set	::=	role $\Vert$ role, role_set
basic_stmt	::=	role_part $\wedge$ attr_part
role_part	::=	pt=PAYMENT $\wedge$ payer=*id_val* $\wedge$ payee=*id_val* $\Vert$ pt=VALUE_CLAIM $\wedge$ user=*id_val* $\wedge$ bank=*id_val* $\Vert$ pt=VALUE_SUBTRACTION $\wedge$ user=*id_val* $\wedge$ bank=*id_val*
attr_part	::=	true $\Vert$ attr_val_pair $\wedge$ attr_part
attr_val_pair	::=	amount relop *amount_val* $\Vert$ time relop *time_val* $\Vert$ ref=*ref_val*
relop	::=	$<$ $\Vert$ $\leq$ $\Vert$ $=$ $\Vert$ $\neq$ $\Vert$ $\geq$ $\Vert$ $>$
role	::=	payer $\Vert$ payee $\Vert$ user $\Vert$ bank

Table 6.5: Grammar for the Payment Dispute Claim Language

Semantics

First, let us try to capture the intuitive semantics of the language for expressing dispute claims. During the execution of a protocol, the system as a whole goes through a series of well-defined *global states*. The global state consists of the initial secrets (e.g., private keys) of all the players involved, all message exchanges up to that point, and all sources of randomness. Therefore it has enough information to determine an assignment to all the variables that can possibly appear in a dispute claim phrased in our claim language. Given a dispute claim phrased in our claim language, a verifier who knows the entire global state can decide with certainty if the claim is true or not. However, it is extremely unlikely that any verifier can know the entire global state (e.g., private keys of other players). The goal of dispute resolution is for the verifier to attempt to partially reconstruct the *sequence of global states* (along with their contents) that the system has gone through, arriving at the *current state*. A verifier can do this based on *evidence* presented to it. It would require a payment system-specific function to interpret the evidence. Based on this interpretation, the verifier can assign values to *some* of the variables. We will call this a *partial assignment*. A partial assignment may be contingent on the trustworthiness of the entities involved in the creation of the evidence. The claim is evaluated with respect to the current state, and/or the sequence of states traversed so far.

Once a sufficiently complete interpretation of a state is available, the verifier can determine if a given basic_stmt s is true in that state. The meanings of the modal operators are intuitive. The statement "**always** s" is true at a state S if s is true in S as well as in every state reachable from S. The meaning of "**never** s" is analogous. The statement "P **can_without** Q s" is true if there is a state S' where s is true, *and* it is possible for P to cause the transition from S to S' without any action from Q. Given a path $p = \{S_0, \ldots S_n\}$, the statement "P **could_without** Q s" is true in p, if (a) "P **can_without** Q s" at some state S in p, and (b) if at some later state in p, it was no longer possible to reach a state where s is true, then P was responsible for this change.

Now, let us try to define the semantics more formally. Our model consists of a set of *states* S, a set of *roles* R, a set of *transitions* $T \subseteq S \times S$, and an interpretation L which assigns a value of the appropriate type to variables in the language.

A role uniquely identifies a service access point (e.g., payer). We assume that there is an infrastructure that enables a verifier to unambiguously identify and authenticate the identity of a player (which is some value from the domain *id_val*) playing any given role. We assume that a multi-party protocol can be described by a directed acyclic graph

(DAG), the edges of which correspond to message transmissions between the players, and the nodes correspond to internal global states. We can capture this property by defining that each state has a cardinality, defined by the function $card() : S \rightarrow N$, where N is the set of natural numbers. The cardinality is used to impose a partial temporal order over S. If $(S_1, S_2) \in T$, then $card(S_2) > card(S_1)$. If the protocol is specified in the form of local finite state machines, they can be first unwound into a set of DAGs which can then be combined into a single DAG. The sender of a message is the agent associated with the edge that represents the message in the DAG. A function $agent() : T \rightarrow R$ identifies the role associated with a given transition. (Recall that an interpretation will associate an *id_val* with a role, thereby associating an *id_val* with each transition as well.) Given a *path* p, in the form of a sequence of states $\{S_0, S_1, \ldots, S_n\}$, the function $agents(p)$ returns the union of $agent(S_i, S_{i+1})$, for $i = 0 \ldots n-1$.

The verifier has a payment system-specific evaluation function which can be used to associate a partial assignment with a given state. The relational operators have the usual semantics. Thus, given a sufficiently complete partial assignment, and a proposition involving an attribute, a relational operator, and a value, it is possible to evaluate if the proposition is true. Given a global state S with a partial assignment, and a basic_stmt s of the form "role_part attr_part" (where role_part and attr_part are conjunctions of propositions as described in the previous section), s is true in S, if role_part and attr_part evaluate to true after the partial assignment is made. Since the assignment is partial, the verifier may not always be able to decide whether a claim is true or not, for example, if the evidence supplied is incomplete.

Figure 6.5 illustrates the semantics of the modal operators. The figure shows the DAG description of a protocol, including the states where a certain basic_stmt s is true. Suppose that the verifier has concluded, based on the evidence submitted, that the system went through states S_0, S_1, S_2, S_3. Then,

1. If the statement '**always** s' is true in a state (e.g., S_{100}), then it is also true in all states reachable from it, in all possible paths. Similarly, if '**never** s' is true in a state (e.g., S_3, S_{201}), it is also true in all states reachable from it.

2. The statement 'P_2 **can_without** P_1 s' is true in S_1 because P_2 can cause the transfer to S_{100}.

3. The statement 'P_2 **could_without** P_1 s' is true in the path $\{S_0, S_1\}$ because of 2. It is also true in the path $\{S_0, S_1, S_{110}\}$ even though s itself cannot be true in S_{110}

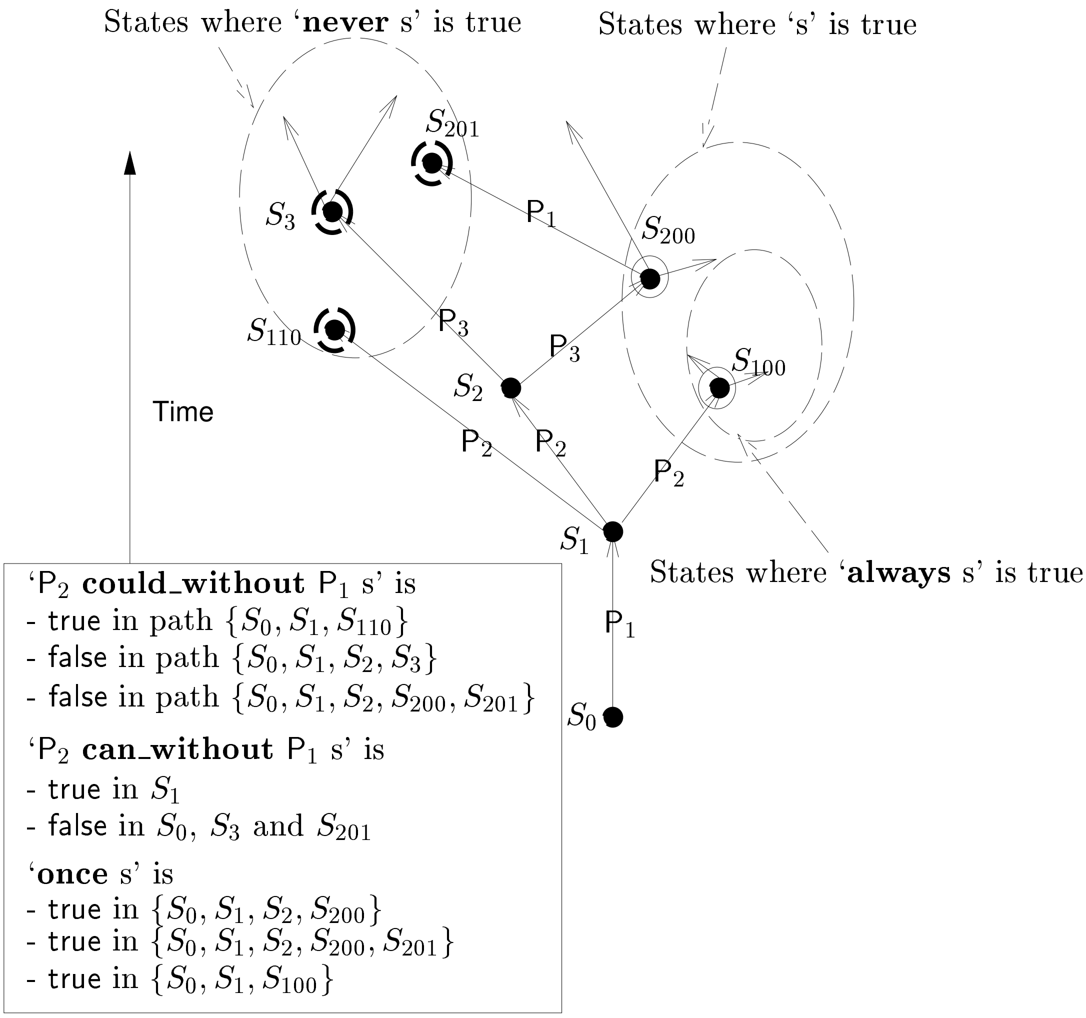

Figure 6.5: Semantics of Dispute Statements

or any state reachable from S_{110}. This is because, in S_1 it was still possible to reach a state where s would have been true (S_{100}) and it was P_2 which chose not to effect that transition.

4. Given a path, the statement '**once** s' has the usual meaning in linear temporal logic — for example, given $\{S_0, S_1, S_2, S_{200}, S_{201}\}$, '**once** s' is true in S_{200} and S_{201}.

More formally, we can define the semantics of the **can_without** operator as follows:

$$
\begin{array}{lll}
S \vdash \mathit{PSET}\ \textbf{can\_without}\ \mathit{QSET}\ s, \text{ iff } \exists\ S_n \text{ such that} & & \\
\quad & S_n \vdash s & \\
\wedge & \exists\ p = \{S_0 = S, S_1, \ldots S_n\} \text{ such that} & \\
 & \quad (S_i, S_{i+1}) \in T,\ i = 0 \ldots n-1 & \\
 & \wedge \quad \mathit{agent}(S, S_1) \in \mathit{PSET} & \\
 & \wedge \quad \mathit{QSET} \cap \mathit{agents}(p) = \phi &
\end{array}
$$

That is, given a state S, if there is valid path p leading to a state S_n, such that s is true in S_n, and the first transition in p can be made by a member of $PSET$ and no one from $QSET$ is required to make any transition in p, then the statement '$PSET$ **can_without** $QSET$ s' is true in S. The **can_without** operator is used to make a statement about the possible future states of the system. In contrast, the **could_without** operator is more general. It is defined with respect to a path (more precisely, with respect to a state and a *specific* path leading to that state). It can be used to make a statement about the system at the end of the path about where it *can* go in the future., as well as where it *could have* gone in the past. The semantics of the **could_without** operator can be defined in terms of the **can_without** operator: (In the following, I do not explicitly mention that all paths considered must consist of valid transitions only.)

For a path $p = \{S_0, S_1, \ldots S_n = S\}$,

$$
\begin{array}{lll}
p \vdash \mathit{PSET}\ \textbf{could\_without}\ \mathit{QSET}\ s, \text{ iff} & & \\
\quad & S \vdash \mathit{PSET}\ \textbf{can\_without}\ \mathit{QSET}\ s & \\
\vee & \exists\ S_i \in p, i = 0 \ldots n-1 \text{ such that} & \\
 & \quad S_i \vdash s & \\
 & \wedge \quad S_{i+1} \vdash \neg\ s & \\
 & \wedge \quad \mathit{agent}(S_i, S_{i+1}) \in \mathit{PSET} &
\end{array}
$$

$$
\begin{array}{ll}
\vee & \exists\, S_i \in p, i = 0 \ldots n-1 \text{ such that} \\
& \quad S_i \vdash \mathit{PSET}\ \textbf{can\_without}\ \mathit{QSET}\ s \\
& \quad \wedge \quad \exists\, S_j \in p,\ j = i \ldots n-1 \text{ such that } S_j \vdash \mathit{ALL}\ \textbf{can\_without}\ \mathit{QSET}\ s \\
& \quad \wedge \quad S_{j+1} \vdash \neg\ \mathit{ALL}\ \textbf{can\_without}\ \mathit{QSET}\ s \\
& \quad \wedge \quad \mathit{agent}(S_j, S_{j+1}) \in\ \mathit{PSET}
\end{array}
$$

The set ALL represents the set of all roles for the primitive transaction referred to in s. The rule identifies three disjunctive conditions to evaluate the truth of the statement 's_{could}=$PSET$ **could_without** $QSET$ s' with respect to a path p. The first disjunction says s_{could} is true if 's_{can}=$PSET$ **can_without** $QSET$ s' is true in the last state of p. The second disjunction says that if s was true at some state in p, but not at the state immediately following it, *and* the transition was caused by someone in $PSET$, then s_{could} is true with respect to p. The third disjunction is a little more complicated. The intent is to capture the following case. Sometime in the past, it was possible to reach a state where s is true (i.e., there was a path, say p' from some state S_i in p, making s_{can} true at S_i). The agents of p' consist of some members of $PSET$ but none from $QSET$. It may also consist of *other* entities, not in either of the above sets. The statement '($PSET$ $\cup$ $agents(p')$) **can_without** s' should therefore hold at every state in p subsequent to S_j. If it fails to hold after a certain transition, *and* that transition was caused by a member of $PSET$, then we can assert that s_{could} is true in p. In the interest of simplicity, we relax the definition a little by using the set of all roles (ALL) instead of ($PSET$ $\cup$ $agents(p')$). The last disjunction captures this property.

Similarly,

$$
\begin{array}{l}
p \vdash \textbf{once}\ s \text{ iff } \exists\, S \in p \text{ such that } S \vdash s \\
S_0 \vdash\ \textbf{always}\ s, \text{ iff } \forall\, p = \{S_0, \ldots\},\ p\ \vdash\ \Box\, s \\
S_0 \vdash\ \textbf{never}\ s, \text{ iff } \forall\, p = \{S_0, \ldots\},\ p\ \vdash\ \Box\, \neg\, s
\end{array}
$$

The notation "$p\ \vdash\ \Box\, s$" has the usual meaning in linear temporal logic: in the sequence of states p, the formula s holds true in every state. In Section B.0.1, we will look at an example payment system to see how we can build a global state transition diagram.

Extensions

The claim language described in the previous section constitutes the set of symbols and grammar rules necessary for *specifying* dispute claims in the generic payment service. The language constructs deal with the possibility of multiple time-lines. Therefore, it belongs to the family of branching temporal logics [Eme90]. It does not include all possible temporal logic constructs: for example, the **until** operator. I have only included the constructs that appeared to be necessary to express the type of claims described earlier. However, I have included constructs like **can_without** and **could_without**, which are not standard branching temporal logic constructs.

The language can be extended as needed. One generic extension is the inclusion of additional primitive transactions which do not represent value transfer but may be relevant in disputes. One example is the transaction started by the authorise primitive in cheque-like systems. Another example is a "statement of account" transaction where a bank issues a statement indicating the current balance on a user's account.

A second type of generic extensions can be used to derive the language to describe the messages in a generic dispute protocol between the verifier and the player. One extension is the addition of a function operator **witnessed**, as described in Section 6.3.2 below. Another extension is to allow a statement to refer to multiple instances of one or more primitive transactions by using functions, such as summation (e.g., statements like "the sum of the amounts involved in all VALUE_SUBTRACTION primitive transaction between Jan 1 and Jan 20 is less than 400").

Finally, individual payment system adapters may introduce additional primitive transactions, and role/attribute variables necessary to describe their protocols. Such additions will not be visible at the interface to the payment dispute service; but they can be useful internally.

Examples

The five dispute claims mentioned in Section 6.2.2 correspond to the following statements in our language:

- PAYMENT payer=Alice payee=BobAir amount=$200
- PAYMENT payer=Alice payee=BobAir amount=$200 time $<$ Jan12
- $\neg$ PAYMENT payer=Alice payee=BobAir amount=$200

- payee **could_without** payer PAYMENT payer=Alice payee=BobAir amount=$200 time < Jan12
- ¬ VALUE_SUBTRACTION *user*=Alice *bank*=CarolBank amount=$300 ref ="#434: for flight 822 on Jan 19:payment to BobAir"

If players have evidence proving that a certain value transfer transaction has reached a guaranteed final state, they may choose to make stronger claims, using the **always** or **never** operators. For example, if Alice has a receipt for the payment made using a payment system which does not support refunds, she may claim "**always** PAYMENT payer=Alice payee=BobAir amount=200" instead.

6.3 Supporting Claims with Evidence

6.3.1 Architecture for Dispute Handling

Overview

Recall that there are three types of players in a dispute: the *initiator* who starts the dispute by making a claim, the *verifier* who co-ordinates the dispute handling and possibly makes the final decision about the validity of the claim, and a set of *responders* who may be asked by the verifier to participate in the process.

At the access point of each player, there is a "user" part (which may be the human user, or an application program acting on his behalf) and a "system" part (which is an implementation of the dispute service: e.g., an iKP implementation of the generic payment dispute service). The verifier's system has two parts: one receives a claim, and associated non-repudiation tokens, analyses them, and determines the conditions under which the claim is true, and the conditions under which it is false; the second part uses the result of the first part to make a final decision. The second part is essentially a policy engine, and may be a human arbiter, external to the dispute handling service. It needs to use trust assumptions. In the simplest case, this may be in the form of a blacklist of untrusted principals. It is possible that the trust assumptions require a more elaborate representation (for example, as in PolicyMaker [BFL96]).

Figure 6.6 illustrates the various interactions during a dispute transaction. To begin a dispute, the initiator constructs a dispute claim. The claim is sent to the verifier. The verifier's system decides which players need to prove which statements in order to resolve this dispute claim. This may involve interaction with the systems of players, asking them what evidence they can produce. Once the verifier's system has made a decision, the players' systems are informed of the statements they are required to prove.

Each player's system then engages in a proof protocol with the verifier's system. Recall that during the dispute protocol, the verifier's goal is to determine both the current state of the payment system, and the sequence of states through which it has progressed, and that the validity of the claim should be evaluated with respect to this state and sequence. At the end of the protocol, the verifier's system returns an analysis of the claim. The analysis consists of a likely decision (yes or no) as well as the set of players who were witnesses to the decision and the set who were against it. If there is insufficient evidence, the verifier's system may throw an exception. The verifier makes the final decision by combining this analysis with his trust assumptions.

This leads to the following requirements on the design:

- The verifier needs the following service primitives:
  - map: Takes a *claim* as input and returns a list of (*player*, *statement*) pairs. Each player is required to prove the corresponding statement.
  - analyse: Takes a *claim*, a set of players, and the statements they need to prove as input, engages in some proof protocol (we assume that it is as simple as receiving one or more non-repudiation tokens, interpreting them, and making an inference on the statements proved), and returns the set of players according to which the claim is true, the set of players according to which it is false, and the set of players who have been found to be cheating.
  - decide: Takes a *claim*, and two sets of players as input, and returns one of yes, no, or cannot-decide, based on the verifier's trust relationships as output. This is the result of the dispute.
- Each player needs the following primitives:
  - constructClaim: which allows the user to construct a claim, and
  - prove: which takes a statement as input and attempts to prove it (maybe as simple as retrieving the pieces of evidence and returning them to the caller)

Enhancements

We have limited ourselves to disputes in a generic payment service where there is only one service boundary. The system is "below" the boundary and the user or his application

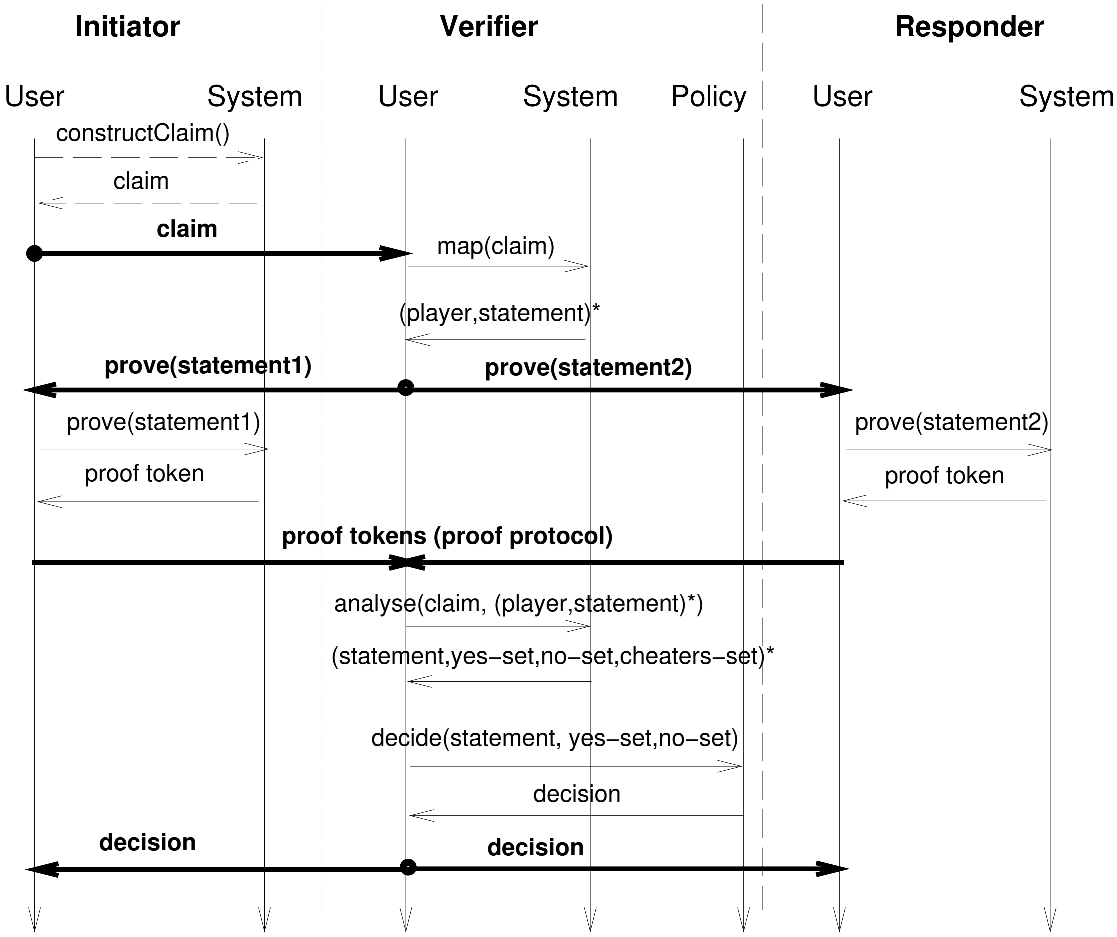

Figure 6.6: Basic Dispute Protocol

is "above" the boundary. Comprehensive electronic commerce frameworks such as SEMPER [Wai96] are structured into multiple layers. A payment usually take place in the context of a higher layer transaction (e.g., a fair exchange) which in turn may take place in the context of a transaction in the layer above (e.g., an instance of an on-line purchase application). A dispute claim made in a higher layer needs to be suitably mapped to corresponding claims in the lower layers. The running of the dispute protocol needs to be co-ordinated among the different levels. This is left as an open problem.

6.3.2 Evidence and Trust

During a dispute, players have to support or counter dispute claims by proving certain statements to the verifier. The ability to prove statements comes from pieces of evidence (evidence tokens) accumulated during a transaction of the primary service. The simplest form of evidence is a non-repudiation token that can be verified by anyone who has the necessary public keys, certificates, certificate revocation lists etc. The proof protocol in this case is to simply present the token to the verifier. There can be more involved proof protocols, such as in "undeniable signature schemes" [CvA89]. In the following, we will assume only simple proof protocols consisting of the presentation of non-repudiation tokens.

Often, non-repudiation tokens cannot substantiate an absolute statement. In general, an evidence token corresponds to a non-repudiable assertion by one or more players that they believed that the protocol reached a certain state (with an associated partial assignment). Such an assertion is a *witnessed statement*. During the dispute protocol, the verifier asks various players to prove witnessed statements (in the proof request messages in the dispute protocol of Figure 6.6). We can extend the claim language in Section 6.2.4 to express witnessed statements by adding a function operator called **witnessed**. The grammar is extended by adding the following rule.

witnessed_stmt	::=	role **witnessed** asserted_stmt
asserted_stmt	::=	basic_stmt ‖ certainty_stmt

The **witnessed** operator takes two parameters: an asserted_stmt s, and a role P. The statement "P **witnessed** s" is true if P has non-repudiably asserted that the transaction reached a certain state, with an associated partial assignment.

As we saw, a dispute claim is a statement of certain attributes of a primitive transaction. Consider a dispute claim discussed earlier: Alice claims to have made a payment of $200 to BobAir. There is no way to say with absolute certainty whether the transaction actually took place. A receipt from BobAir proves the statement

BobAir **witnessed** PAYMENT payer=Alice payee=BobAir amount=$200

Whether a verifier can infer

PAYMENT payer=Alice payee=BobAir amount=$200

depends on the context, the trust assumptions of the verifier, and even on the actual payment system allegedly used for the value transfer. For example, in a dispute against BobAir, the verifier could accept a signed receipt from BobAir as sufficient evidence to conclude that the latter claim is true. However, if BobAir and Alice are in collusion and want to convince a third party (say the tax authorities), BobAir's signed receipt alone is not sufficient. In this case, if the verifier trusts the bank, it can accept a signed statement from the bank as sufficient evidence.

In general, to decide the validity of a dispute claim, the verifier asks various players to prove various statements. Which players need to prove which statements depends both on the internal design of the particular payment system and the trust assumptions of the verifier. We assumed that the proof protocols consist of forwarding non-repudiation tokens. The analyse method will consult the inference engine and will return a result of the form:

$$\bigvee \{[\neg]\ \langle \text{claim\_stmt} \rangle\ \text{yes}=\{\ldots\}\ \text{no}=\{\ldots\}\ \text{cheating}=\{\ldots\}\}$$

Each component of this disjunction contains either the original statement in the claim or its negation. In addition, each component contains three optional sets of players: the set of cheating players, the set of players whose statements are in agreement with the conclusion and the set of players whose statements are contrary to the conclusion. The verifier will pick one of these disjunctions, depending on the set of players he trusts. If the evidence presented is insufficient to decide one way or the other, the system may raise an exception. Further, it may also be able to detect if some player has cheated (for example, if the same player has witnessed a certain statement and its exact opposite, it may imply that the player had cheated). Note that a carefully designed, secure, payment system can be rendered insecure if the inference engine used by the verifier is wrong. It is important to make sure that the inference engine used not degrade the security of the protocol.

For instance, in the example presented at the beginning of this section, a conclusion of the form:

PAYMENT <role_part> <attr_part> yes={payer,payee} no={}

will be applicable in the first case but not in the case of Alice and BobAir colluding, whereas a conclusion of the form:

PAYMENT <role_part> <attr_part> yes={bank} no={}

will be applicable in both cases.

A player may have gathered several evidence tokens during a particular transaction. Not all of them may be necessary to substantiate a dispute claim about some aspects of the transaction. Thus, given a statement to prove, a system should use only the minimal subset of the relevant evidence.

6.4 Summary and Conclusion

I have attempted to show why a generic dispute service is needed for payment systems. I have developed a **language to express payment dispute claims**. In Section B.0.1, I show the how evidence tokens in an example payment system can be used to analyse claims made in this language. A crucial part of the dispute handling framework is the verifier's inference engine. When a new payment system is adapted to the framework, the critical step is to identify the inference rules applicable to that system. In general, the process of deriving the inference rules is equivalent to proving the payment system correct. It is also necessary to verify that the default inference rules do not violate the security of the system. At the current state of knowledge, this appears to be a hard task.

On the other hand, the aim is not complete automation of dispute resolution. Thus, even with an incomplete set of inference rules, a payment system can be incorporated into the framework. The most trivial inference rule is "ask the human user!" Another impact of the framework is that it can help the designers of new payment systems to derive the inference rules as an integral part of the design process.

Chapter 7

Conclusion

7.1 Effectiveness of Solutions

I have presented a set of fair exchange protocols and proved their properties. The network security group at the IBM Zurich Research laboratory is currently working on implementing these protocols. The implementation experience will provide further validation of the effectiveness of the solutions to the fair exchange problem. IBM is in the process of applying for a United States patent for the design of the optimistic fair exchange protocol and the techniques for the application of verifiable encryption.

I have provided informal arguments for the security of the S^3 signature scheme. It was not validated by an implementation.

The generic payment service has been validated by a prototype implementation. Adapters for several payment systems have already been designed and implemented. The design and the prototype implementation also served as the basis for a software product.

I presented a language to express payment dispute claims. (In Section B.0.1, I outline how this can be used with an example payment system.) This is only a first step. The effectiveness of the language can be judged only in the course of applying designing and implementing a full-fledged dispute handling framework. The network security group has recently embarked on a project to build and experiment with dispute handling support in electronic payment systems. This project will use the work described in Chapter 6 as the starting point.

7.2 Other Aspects

Throughout this dissertation, I have made use of third parties which are trusted to perform specific functions. Using such third parties makes it possible to provide functionality that would otherwise be prohibitively expensive, or even impossible. There is no such thing as a "trusted third party" which is trusted in general – it is important to clearly define the specific functions with respect to which a given third party is trusted. Where possible, trust in such third parties can be further constrained by, for example, (a) designing systems that include support for verifiability of third party, and (b) implementing the third party in a distributed manner. The essential role of third parties is to perform trusted computation.

Another way to achieve trusted computation is to use a trusted computing base. Typically, it is implemented in the form of tamper-resistant hardware. Performing trusted computation this way makes off-line transactions possible. Recent trends in digital signature legislation indicate that trusted personal hardware devices will likely become mandatory for electronic transactions.

I have addressed only one aspect of the security issues associated with electronic commerce. There are a number of other aspects to electronic commerce. An important concern is *privacy.* There is a large body of work addressing the issue of user anonymity [Cha85, Cha92], though practical implementations are still not widespread [GT96, RR97]. Another critical issue is *copyright protection* [CMPS95, BS94, PS96]. If Alice buys a book from Bob, and illegally redistributes it to others who have not paid Bob, is it possible to detect Alice's incorrect behaviour? Is it possible to prevent it? *Trust management* is an overarching problem. This is tied to the problem of *certification* of names [BFL96, RL96]. If Alice receives a message which is successfully verified by V_{Bob}, how much assurance does Alice have that it is really Bob who signed it? If Alice and BobAir do not have a mutually acceptable third party *a priori*, can they find one on the fly, based on recommendations or guarantees from their respective trusted third parties?

Ford and Baum provide an excellent overview of the current concerns on making electronic commerce viable [FB97]. They correctly observe that there is a long way to go before the legal and business practice controls necessary for making widespread use of electronic commerce are well understood. They assert that "from the technological perspective, there are no major outstanding challenges." However, as electronic commerce becomes more widespread, issues that are considered less important now (such as providing fairness, or supporting various levels of privacy) will grow in their urgency and importance. There is still plenty of scope for interesting technical work in the area.

The term "fairness" has been used in other contexts in computer science. Micali introduced the notion of *fair public-key cryptosystems* in [Mic93]. According to his definition, a fair public-key system is one which guarantees that a specially designated player can understand all messages encrypted with a key using this public-key system, even without the co-operation of the entity that created the messages. The intent behind fair public-key cryptosystems is that the designated player will access encrypted messages only under "proper circumstances envisaged by the law." In general, this notion of fairness is not the same as ours. However, if the "proper circumstances" correspond solely to incorrect behaviour by the creator, then this notion of fairness is similar to ours. Camenisch et al [CPS95] introduced the notion of *fair blind signatures.* A blind signature protocol allows a player A to obtain a signature from a signer S, so that S's view of the protocol does not allow it to link the message and its signature. Thus, A can send the signature to someone else without S being able to link this message transmission to the original blind signature protocol run. In fair blind signature schemes, a trusted third party can provide information which will enable S to link the two events. Blind signatures are used to build anonymous cash-like payment systems, where a coin is represented by a blind signature. Fair blind signatures allow the anonymity of A to be revoked with the help of the trusted third party. Again, assuming that the trusted third party behaves honestly, fair blind signature schemes enable fairness in our sense of the word.

The notion of fairness is also used in scheduling schemes. A fair scheduling scheme guarantees that a waiting client is eventually served. Unlike in our case, malicious behaviour is usually not a concern in these scenarios.

7.3 Summary of Contributions

I have made the following contributions in this dissertation.

- Proposed and analysed an **optimistic protocol** for contract signing, which is the first such protocol to guarantee timely conclusion under the assumption of resilient communication channels (which is weaker than assuming reliable communication channels).

- Generalised the approach to the exchange of **forwardable items** and analysed the resulting fair protocol, identifying characteristics (**generatability** and **revocability**) that make it possible to provide a strong fairness guarantee,

- Proposed and analysed a specific optimistic protocol for the fair exchange of generatable items.

- Proposed and demonstrated the use of **verifiable encryption** to add generatability to items.

- Proposed a new non-repudiation technique called **server-supported signatures**, which enables users with limited computational resources to construct strong digital signatures and showed how this technique can be used to provide several services required in electronic commerce, including the creation and gathering of evidence that can be used in disputes.

- Proposed and prototyped an architecture for a **generic payment service** which enables
  - development of business applications (which use payment services) independent of specific payment systems,
  - transparent selection of a payment instrument to be used in a transaction, based on various factors, including an abstract specification of security services, and
  - the possibility of improving security services provided by specific payment systems.
- Argued the importance of a systematic approach to handling disputes and proposed a language to express dispute claims that may arise in the course of using the generic payment service.

7.4 Directions for Future Research

There are several directions to continue research. In fair exchange, the use of verifiable encryption enables the exchange of signatures. A remaining open question is whether it is possible to design optimistic protocols to exchange arbitrary items (such as encrypted data) while guaranteeing strong fairness? The verifiable encryption technique in Chapter 3 uses the cut-and-choose method, which requires the construction and transmission of a large number of candidate permit components, half of which are rendered useless at the end of the protocol. Is it possible to avoid the cut-and-choose approach in the verifiable encryption protocol? Another avenue of research is to explore the issues in distributing the role of the third party among multiple entities. Distribution will lead to high availability and increased trust, both of which are important in the scenarios where fair exchange protocols will be applied.

The dispute claim language presented in this dissertation is only a first step in designing a framework for handling disputes in electronic commerce. Can the language be validated by using it with many different real payment systems? Could the approach generalise to constructing claim languages for other services such as a non-repudiation service? I did not address the complex task of constructing inference engines, leaving it to be done separately for each adapter. An interesting, and useful, contribution would be to investigate whether a common part of the inference engine can be identified, and separately defined. It may be possible to find inference rules that are common to all payment systems of a given model, or indeed to all payment systems in general. Defining a partial inference engine will ease the task of the adapter designers. Finally, a complete framework for dispute handling must incorporate appropriate legal mechanisms.

Epilogue

Part of the work described in this thesis was carried out in the context of the SEMPER project. SEMPER, a three year project funded by the European Commission and the Swiss Federal government, began in Fall 1995. The Network Security and Electronic Commerce group at the IBM Zurich Research Laboratory provides technical leadership for SEMPER. I joined this group in November 1995, shortly after SEMPER was launched. My work was supervised primarily by Dr. Michael Waidner who is the manager of the group. In the following, I acknowledge contributions and help from colleagues in my work. Except where mentioned, the collaborators in my work are all from the Network Security and Electronic Commerce group.

The work on optimistic protocols for fair exchange was initiated by Michael Waidner early 1996. Initial attempts at a solution were carried out as a joint effort between Matthias Schunter (Universität des Saarlandes) and me, supervised by Michael Waidner. This work was reported in [ASW96a, ASW96b, ASW97a]. I conceived the idea of using verifiable encryption to make items generatable. Victor Shoup and I carried out a literature survey to identify the techniques that can be used to implement publicly verifiable encryption. We also jointly designed the improved version of the optimistic fair exchange protocols which provides the timeliness guarantee in an asynchronous model. This improved protocol was reported in [ASW97b, ASW98]. Chapter 2 is based on this work. The notion of verifiable encryption, and its use in fair exchange of digital signatures were reported in [ASW97c, AS98]. Chapter 3 is based on my contributions to the work described in [ASW97c]. Ceki Gülcü has begun an implementation of the fair exchange protocols described in this dissertation.

The original idea of combining a hash-chain and a traditional digital signature scheme was conceived by me. I developed the idea further under the supervision of Gene Tsudik (then at IBM Zurich Research Laboratory) and Michael Waidner resulting in the S^3 protocol. This work was reported originally in [ATW96] and expanded in [ATW97]. Chapter 4 is based on the latter work.

Work on a generic payment service for SEMPER was begun in October 1995 by Michael Waidner and José Luis Abad-Peiro. I took over primary responsibility for this work in December 1995. In the first phase of the payment service work, I developed a detailed design that included

- the notion of a hierarchy of service interfaces based on models of payment systems, and

- selection of payment instrument based on three classes of factors: user preferences, requirements, and negotiations.

I carried out the proto-type implementation of the generic payment service in SEMPER. Several other participants in SEMPER worked on designing and implementing adapters for various payment systems. In late 1996, IBM began work on a merchant-side generic payment service product. They used the SEMPER payment service block as a starting point. Discussions with the software engineers at IBM Secure Electronic Payments group at Research Triangle Park, Raleigh, NC, led to the conclusion that support for an asynchronous model is essential. Inspired by GSS-API [Lin97], I designed the token-based interface definition as a solution to this problem. My design was used as the basis for the IBM CommercePOINT e-till product. Several ideas I used in the design of the generic payment service are being adapted by designers of other blocks in SEMPER. These include the notion of transactions and transaction records, protocol for negotiation of parameters and the token-based definition of services. A survey of electronic payment systems was reported in [AJSW97]. The design of the generic payment service was described in [APASW96, APASW98]. Chapter 5 is based on this work.

The work on the dispute handling framework is primarily my contribution, with help from Els van Herreweghen, and Michael Steiner. This work was described in [AHS98a, AHS98b].

Appendix A

Generic Payment Service API

Primitive	Input	Output	Description
implemented by *Transaction*			
pay	*payee, amount, options, ref.*[a]		send a payment
receivePayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment
reversePayment	*transaction record*		ask/get a refund
reverseReceivedPayment	*transaction record*		make a refund
Additions for cash-like model			
withdraw	*amount, options, ref.*		load money into purse
deposit	*amount, options, ref.*		unload money from purse
Additions for cheque-like model			
receiveRawPayment	*[payer,] [amount,] options, ref.*	*payer, amount*	receive a payment (defer authorisation)
authorise			authorise a previous raw payment
capture	[*amount*]		capture a previous raw payment
multiCapture	*list of transaction records*		capture a set of previous raw payments

Table A.1: Generic Payment Service: Value Transfer Services

[a] ref. allows this payment to be tightly linked to its context, e.g an on order and its description

Primitive	Input	Output	Description
implemented by *Purse*			
startTransaction		*transaction*	create a transaction of this payment system
implemented by *Transaction/Transaction Record*			
abort			abort the transaction, if possible
suspend			suspend the transaction
resume			resume a suspended transaction
getState		*state*	retrieve the current state of a transaction

Table A.2: Generic Payment Service: Transaction Management Services

Primitive	Input	Output	Description
implemented by *Payment Manager*			
createPurse	*payment system name, purse name*	*purse*	create a purse of the specified payment system and name it as indicated
deletePurse	*purse*		delete the specified purse
implemented by *Purse*			
init			perform any necessary initialisation to make the purse usable in the current session
disable			disable an initialised purse so that it is no longer available in the current session
setup			perform any necessary configuration to make the purse usable

Table A.3: Generic Payment Service: Purse Management Services

Primitive	Input	Output	Description
implemented by *Payment Manager*			
selectCandidatePurses	*context*[a]	*list of purses*	select a list of purses, as per *context*
selectPurse	*context*	*purse*	select a single purse as per *context*
selectPayingPurse	*context*	*purse*	select a purse for payment
selectReceivingPurse	*context*	*purse*	select a purse for receiving payment

[a] *context* contains information like the identity of the peer, type and amount of transaction, security requirements, etc.

Table A.4: Generic Payment Service: Purse Selection Services

Primitive	Input	Output	Description
implemented by *Payment Manager*			
getListOfPurses	*criteria*[a]	*list of purses*	list of usable purses
getListOfAllPurses	*criteria*	*list of purses*	list of all purses
getListOfTransactions	*criteria*	*list of transactions*	list of transactions
getListOfKnownPurseClasses		*list of adapters*	list of known payment systems
getListOfEnabledPurseClasses		*list of adapters*	list of usable payment systems

[a] *criteria* can be used to limit the form (e.g. names of payment systems) and scope (e.g. only SET:Visa purses) of the returned list

Table A.5: Generic Payment Service: Information Services (Payment Manager)

Primitive	Input	Output	Description
		implemented by *Purse*	
getAmount		amount	amount of money in this purse[a]
getBank		*bank*	bank associated with this purse
getBrandName		*brand name*	brand name
getPaymentSystemName		*payment system name*	name of the payment system
getPurseType		*type*	type (e.g. pay-only)
getPurseAddress		*purse address*	get the address of this purse[b]
getPrintableStatusInfo		*string*	get the state of the purse as a printable string
getListOfTransactions		*list of transactions*	list of transactions made with this purse
isEnabled		*boolean*	is this purse usable in this session?
isRegistered		*boolean*	is this purse usable?
isSecurityServiceOffered	*security service*	*boolean*	is the specified service supported?
isCurrencySupported	*currency*	*boolean*	is the specified currency supported?
supportedCurrency		*currency*	which currency is supported?
offeredSecurityServices		*list of security services*	which security services are available?
Additions for cheque-like model			
getAmountFromBank		*amount*	what is the amount in the bank account?

[a] where applicable

[b] *purse address* contains information like the name of the purse, name of the owner, communication address (if relevant)

Table A.6: Generic Payment Service: Information Services (Purse)

Primitive	Input	Output	Description
implemented by *Transaction Record*			
getAmount		*amount*	amount of the transaction
getPeer		*purse address*	identity of the peer purse
getPurseName		*purse name*	name of the purse
getType		*service type*	type of the transaction
getState		*state*	state of the transaction
isCompleted		*boolean*	is the transaction completed?
isActive		*boolean*	is the transaction active now?
getStartTime		*time*	start time of the transaction
getEndTime		*time*	time of the last state change

Table A.7: Generic Payment Service: Information Services (Transaction and Transaction Records)

Appendix B

Disputes in iKP

B.0.1 An Example: Evidence Tokens in iKP

We will now use a simplified version of the iKP[1] payment protocol to illustrate how to represent the global states in the protocol in the form of a DAG, and how to associate evidence tokens with global states. iKP was designed as a solution for securing credit card payments over open networks. The original protocol was described in [BGH+95]. A more detailed definition with some improvements is available in [Tsu96]. In this section, I present a simplified version of the 3-party iKP (3KP) where all three players are assumed to have signature and encryption key pairs. We will see how the receipts gathered during an iKP protocol run can be mapped to dispute statements defined for the generic payment service in Section 6.2.4.

Protocol Description

In our simplified version of iKP, there are three players: Customer (Payer), Merchant (Payee), and Acquirer (Bank). Before the transaction begins, the customer and merchant agree about the amount of payment ("price") and the description ("desc") of what the payment is for. The first half of Table B.1 depicts the initial information of each player.

To begin the transaction, the payee:

- generates two random nonces v and vc; these nonces will later be used as part of the receipt(s) from the payee,
- collects the common information ("com"). The common information consists of all the pieces of information that will be known to all the parties at the end of the transaction,[2] and
- generates a signature Sig_M containing hashes of the data items mentioned above and sends the signature along with any necessary information to the payer. This is the signed *offer* from the merchant.

The signature on the offer makes it non-repudiable. But this is not relevant to our discussion.

[1] SET [MV97] is the proposed standard for credit-card payments on the Internet. However, we will use iKP here since its simplicity helps illustrate the approach more clearly.

[2] One item in the common information is a randomised hash of the description ($\mathcal{H}^R(\mathrm{desc})$). The payer and payee already know the description. The randomised hash allows the bank to confirm that the payer and payee agree on the description without the bank's having to know the actual text of the description.

Data		Functions	
M	Merchant (payee)	$\mathcal{H}()$	A one-way hash function
C	Customer (payer)	$\mathcal{H}^R(m)$	A randomised one-way hash of m
A	Acquirer (bank)	$\mathcal{E}_X^R(m)$	A randomised encryption of m with the public key of X
TID	Transaction ID		
date	Time stamp	$S_X(m)$	Signature of X on m
price	Amount information		
desc	Description of payment		
v	Confirmation Authenticator		
vc	Cancellation Authenticator		
CHI	Card-holder Information		
com	Common information $(M, C, \text{price}, \text{TID}, \text{date}, \mathcal{H}^R(\text{desc}), \mathcal{H}(\text{v}))$		

Payment

$$\textit{offer} : C \xleftarrow{\text{TID}, \text{date}, \mathcal{H}(\text{v}), \text{Sig}_M = S_M(\mathcal{H}(com), \mathcal{H}(\text{v}), \mathcal{H}(\text{vc}))} M$$

$$\textit{order} : C \xrightarrow{\text{Enc} = \mathcal{E}_A^R(\text{price}, \mathcal{H}(com), \text{CHI}), \text{Sig}_C = S_C(\text{Enc}, \mathcal{H}(com))} M$$

$$\textit{auth-request} : M \xrightarrow{\text{TID}, \text{date}, \mathcal{H}(\text{v}), \mathcal{H}(com), \mathcal{H}^R(\text{desc}), \text{Enc}, \text{Sig}_M, \text{Sig}_C} A$$

$$\textit{auth-response} : M \xleftarrow{\text{yes}\|\text{no}, \text{Sig}_A = S_A(\text{yes}\|\text{no}, \mathcal{H}(com))} A$$

$$\textit{confirm} : C \xleftarrow{(\text{v}, \text{yes})\|(\text{vc}, \text{no}), \text{Sig}_A} M$$

Cancellation

$$\textit{cancel} : C \xleftarrow{\text{vc}} M$$

Refund

$$\textit{refund-request} : M \xrightarrow{\text{Sig}_M^{refund} = S_M(\mathcal{H}(com), \text{price}, \text{“refund”})} A$$

$$\textit{refund-response} : M \xleftarrow{\text{Sig}_A^{refund} = S_A(\mathcal{H}(com), \text{price}, \text{“refund”})} A$$

Figure B.1: Simplified iKP Protocol

The payer then sends an *order* message. This is the authorisation by the customer to make the payment. The order is a signature Sig_C of the payer on two pieces of information,

- $\mathcal{H}(com)$, and
- an encryption of the price, the customer's account number ("CHI"), and $\mathcal{H}(com)$.[3]

The payee forwards the *order* along with the *offer* to the acquirer requesting authorisation. The acquirer replies indicating whether the authorisation succeeded or not. For simplicity, let us assume that the acquirer immediately transfers the money from payer to payee if the authorisation is successful. The authorisation response Sig_A from the acquirer is signed.

Once the payee receives the authorisation response, he will send a confirmation to the payer. The confirmation contains the acquirer's authorisation response and the authenticator v. The pair (Sig_M, v) constitutes a receipt by the payee that he received the payment. If a payee decides to cancel a payment, he can issue a cancellation receipt to the payer by sending him vc. If the payer possesses the pair (Sig_M,vc), it is proof that the payee agreed to cancel the payment. If the payee cancels an already authorised payment, he can contact the acquirer and arrange for a refund.

The second half of Table B.1 lists the pieces of information that are collected by the players at the end of a successful protocol run. Again, items within square parentheses are available only under certain circumstances.

Note that instead of using v and vc, the *confirm* and *cancel* flows from the payee to payer can be signed by the payee. The use of v and vc avoids the payee having to make two or more signatures by allowing the original signature to be "extended."

In the actual iKP protocol, the acquirer transfers the money from the customer to merchant during a "capture" transaction. The merchant can also capture a different (lower) amount than was previously authorised. The refund transaction is essentially a negative capture. Depending on the policies of the players, some of the receipts may be omitted. All these variations are not relevant to our discussion. Therefore they are left out from our simplified version.

---

[3] Parts of the card-holder information is considered "secret" information (e.g., like a credit card number). The encryption in the *order* can be opened only by the bank — thus, the payee will not be able to determine CHI.

Initial Information	
payer	desc, price, CHI
payee	desc, price, TID, date, v, vc
bank	
Collected Information	
payer	com, $\mathcal{H}$(v), Sig_M, [Sig_A], v **or** vc
payee	com, Enc, Sig_C, [Sig_A], [Sig_A^{refund}]
bank	[com, Enc, CHI, Sig_M, [Sig_M^{refund}], Sig_C]

Table B.1: Information of Players in a Completed iKP Transaction

Mapping iKP receipts to Dispute Statements

Table B.1 lists the pieces of information known to each player at the end of a successful transaction. We can now try to extract evidence tokens from these pieces and identify the dispute claims they can support.

The payer can prove that a payment transaction took place by producing (Sig_M, v). However, the payment may have been cancelled later. The payee does not get a receipt from the payer acknowledging a cancelled payment. However, the cancellation must have involved the running of a refund protocol with the bank. Thus, an honest payee can produce (Sig_A^{refund}) to counter the payer's claim. Line 1 of Table B.2 expresses this scenario. The rest of Table B.2 is an exhaustive list of all the pieces of evidence in iKP and their mapping to dispute statements.

Notice that all parties have to reveal *all* the pieces of the common information ("com") to the verifier. It is possible that a dispute claim may involve only the "price" attribute. Nevertheless, the players are forced to reveal the "date" attribute as well. The description of what the payment is for ("desc"), however need not be revealed because "com" contains a commitment (in the form of a randomised hash of "desc") of "desc" only. If the dispute was about the description as well, the commitment can be opened (by revealing the random factor used in computing the hash). If the same technique was applied to the rest of the common information, i.e., if it consisted of:

$$\mathcal{H}^R(M), \mathcal{H}^R(C), \mathcal{H}^R(\text{price}), \text{TID}, \mathcal{H}^R(\text{date}), \mathcal{H}^R(\text{desc}), \mathcal{H}(\text{v}),$$

it would have been possible to reveal only the necessary information to the verifier. In

Role	Evidence	Statement		Possible Counter
		witness	Primitive Transaction	
payer	1. $\text{Sig}_M, \text{v}, \text{com}$	payee	PAYMENT	8
	2. $\text{Sig}_A, \text{yes}, \text{com}$	bank	VALUE_SUBTRACTION	10
	3. $\text{Sig}_A, \text{no}, \text{com}$	bank	**never** VALUE_SUBTRACTION	
	4. $\text{Sig}_M, \text{vc}, \text{com}$	payee	**never** PAYMENT	11
payee	5. $\text{Sig}_C, \text{com}, \text{Enc}$	payer	PAYMENT	3, 4, 10
	6. $\text{Sig}_A, \text{yes}, \text{com}$	bank	VALUE_CLAIM	10
	7. $\text{Sig}_A, \text{no}, \text{com}$	bank	**never** VALUE_CLAIM	
	8. $\text{Sig}_A^{refund}, \text{com}$	bank	**never** VALUE_CLAIM	
bank	9. $\text{Sig}_M, \text{Sig}_C, \text{com}$	payee	VALUE_CLAIM	8
	10. $\text{Sig}_M^{refund}, \text{com}$	payee	**never** VALUE_CLAIM	
	11. $\text{Sig}_C, \text{com}, \text{CHI}$	payer	VALUE_SUBTRACTION	3, 4

Table B.2: Mapping Evidence to Dispute Statements in iKP

general, I recommend supporting the possibility of *least disclosure of information* as a principle of good practice in designing disputable protocols.

With the information in Table B.2 and Figure B.1, we can represent the global states in a run of the iKP payment protocol in the form of a DAG as in Figure B.2. Since iKP is a cheque-like system, all three primitive transactions are completed at the same time. The states where the primitive transactions are succesfully completed are marked with a circle. The states where '**never** s' is true (s is any basic_stmt with any of the primitive transactions) are marked with a thick broken circle. Notice how the verifier can use this graph to implement the analyse method (Section 6.3.1): while 'PAYMENT *rest_of_the_claim*' is false in S_{300}, S_{400}, and S_7, the statement 'payee **could_without** payer PAYMENT *rest_of_the_claim*' is true in S_{300} and S_7 (but not in S_{400}).

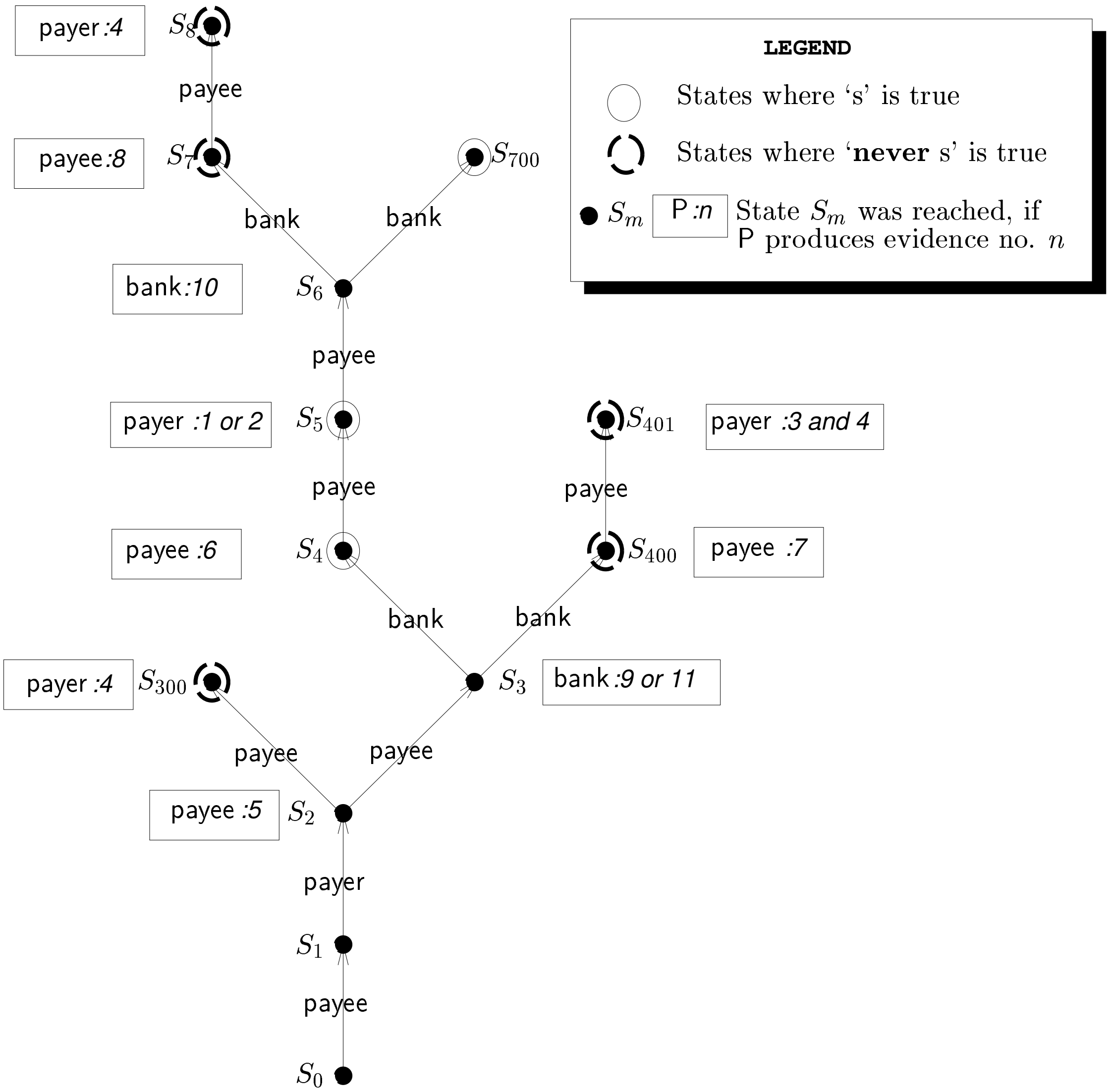

Figure B.2: Global States in iKP

Bibliography

[AHS98a] N. Asokan, E. Van Herreweghen, and M. Steiner. Towards a framework for handling disputes in payment systems. Research Report RZ 2996, IBM Research, March 1998.

[AHS98b] N. Asokan, Els Van Herreweghen, and Michael Steiner. Towards a framework for handling disputes in payment systems (to appear). In *Third USENIX Workshop on Electronic Commerce*, page (to be assigned), Boston, Mass., September 1998. USENIX.

[AJSW97] N. Asokan, Phil Janson, Michael Steiner, and Michael Waidner. State of the art in electronic payment systems. *IEEE Computer*, 30(9):28–35, September 1997. A Japanese translation of the article appears in pp 195-201, *Nikkei Computer* (http://nc.nikkeibp.co.jp/jp/) issue of March 30, 1998.

[APASW96] J. L. Abad-Peiro, N. Asokan, M. Steiner, and M. Waidner. Designing a generic payment service. Research Report RZ 2891 (# 90839), IBM Research, December 1996. A later version is also available [APASW98].

[APASW98] J. L. Abad-Peiro, N. Asokan, Michael Steiner, and Michael Waidner. Designing a generic payment service. *IBM Systems Journal*, 37(1):72–88, January 1998.

[AS98] N. Asokan and Victor Shoup. Optimistic fair exchange of digital signatures (to appear). In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT '98*, number to be assigned in Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 1998. A longer version is available [ASW97c]; A related paper [ASW98] is available as well.

[ASW96a] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. Research Report RZ 2858, IBM Research, September 1996. A shorter version [ASW97a] was presented at the 4th ACM Conference on Computer and Communications Security, Zurich 1997.

[ASW96b] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for multi-party fair exchange. Research Report RZ 2892 (# 90840), IBM Research, December 1996.

[ASW97a] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In Matsumoto [Mat97], pages 6, 8–17.

[ASW97b] N. Asokan, Victor Shoup, and Michael Waidner. Asynchronous protocols for optimistic fair exchange. Research Report RZ 2976 (#93022), IBM Research, November 1997. A related paper [ASW97c] is available as well.

[ASW97c] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. Research Report RZ 2973 (#93019), IBM Research, November 1997. A related paper [ASW97b] is available as well.

[ASW98] N. Asokan, Victor Shoup, and Michael Waidner. Asynchronous protocols for optimistic fair exchange (to appear). 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. A minor bug in the proceedings version was fixed. An errata sheet, (to be) distributed at the conference, is available at http://www.zurich.ibm.com/Technology/Security/publications/1998/ASW98-errata.ps.gz; A related paper [AS98] is available as well.

[ATW96] N. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. In Elisa Bertino et al., editors, *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS)*, number 1146 in Lecture Notes in Computer Science, pages 131–143. Springer-Verlag, Berlin Germany, September 1996. An improved version is available as [ATW97].

[ATW97] N. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures. *Journal of Computer Security*, 5(1):91–108, 1997.

[Bah96] Alireza Bahreman. Generic electronic payment services: Framework and functional specification. In *Second USENIX Workshop on Electronic Commerce* [USE96], pages 87–103.

[BBC+94] Jean-Paul Boly, Antoon Bosselaers, R. Cramer, R. Michelsen, S. Mjølsnes, F. Muller, T. Pedersen, B. Pfitzmann, P. de Rooij, B. Schoenmakers, M. Schunter, L. Vallée, and M. Waidner. The ESPRIT project CAFE - high security digital payment systems. In Dieter Gollmann, editor, *Proceedings of the Third European Symposium on Research in Computer Security (ESORICS)*, number 875 in Lecture Notes in Computer Science, Brighton, UK, November 1994. Springer-Verlag, Berlin Germany.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [IEE96].

[BG96] Mihir Bellare and Shafi Goldwasser. Encapsulated key escrow. Technical Report 688, MIT Laboratory for Computer Science, Apr 1996. presented at rump session of EuroCrypt 96, revised version at http://www-cse.ucsd.edu/users/mihir/index.html.

[BGH+95] Mihir Bellare, Juan Garay, Ralf Hauser, Amir Herzberg, Hugo Krawczyk, Michael Steiner, Gene Tsudik, and Michael Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce* [USE95], pages 89–106.

[Bla94] U. D. Black. *Data Link Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Blu83] M. Blum. How to exchange (secret) keys. *ACM Transactions on Computer Systems*, 1:175–193, 1983.

[BN96] Alireza Bahreman and Rajkumar Narayanaswamy. Payment method negotiation service. In *Second USENIX Workshop on Electronic Commerce* [USE96], pages 299–314.

[BOGMR90] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest. A fair protocol for signing contracts. *IEEE Transactions on Information Theory*, 36(1):40–46, January 1990.

[BP89] Holger Bürk and Andreas Pfitzmann. Payment systems enabling security and unobservability. *Computers & Security*, 8(5):399–416, August 1989.

[BP90] Holger Bürk and Andreas Pfitzmann. Value exchange systems enabling security and unobservability. *Computers & Security*, 9(8):715–721, 1990.

[BS94] Dan Boneh and James Shaw. Collusion-secure fingerprinting for digital data. Technical Report TR-468-94, DCS, Princeton University, NJ 08544, 1994. Submitted for patent application.

[BT94] Alireza Bahreman and J.D. Tygar. Certified electronic mail. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 3–19. Internet Society, February 1994.

[Cha81] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.

[Cha85] David L. Chaum. Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[Cha92] David Chaum. Achieving electonic privacy. *Scientific American*, pages 96–101, August 1992.

[CMPS95] A. K. Choudhury, N.F. Maxemchuk, S. Paul, and H.G. Schulzrinne. Copyright protection for electronic publishing over computer networks. *IEEE Network Magazine*, 9(3):12–21, May/June 1995.

[CPS95] Jan L. Camenisch, Jean-Marc Piveteau, and Markus A. Stadler. Fair blind signatures. In L. Guilloy and J-J. Quisquater, editors, *Advances in Cryptology – EUROCRYPT '95*, number 921 in Lecture Notes in Computer Science, pages 209–219. Springer-Verlag, Berlin Germany, 1995.

[CTS95] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce* [USE95].

[CvA89] David Chaum and Hans van Antwerpen. Undeniable signatures. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, number 435 in Lecture Notes in Computer Science, Santa Barbara, CA, USA, August 1989. Springer-Verlag, Berlin Germany.

[DGLW96] Robert H. Deng, Li Gong, Aurel A. Lazar, and Weiguo Wang. Practical protocols for certified electronic mail. *Journal of Network and System Management*, 4(3), 1996.

[EBCY95] Donald E. Eastlake, Brian Boesch, Steve Crocker, and Magdalena Yesil. CyberCash credit card protocol version 0.8. Internet Draft, July 1995.

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637, June 1985. 647.

[Eme90] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[FB97] Warwick Ford and Michael S. Baum. *Secure Electronic Commerce : Building the Infrastructure for Digital Signatures and Encryption.* Prentice Hall PTR, Upper Saddle River, New Jersey 07458, USA, April 1997.

[For94] W. Ford. *Computer Communications Security.* Prentice-Hall, 1994.

[FR97] Matthew K. Franklin and Michael K. Reiter. Fair exchange with a semi-trusted third party. In Matsumoto [Mat97], pages 1–5,7.

[FST95] FSTC. Electronic check proposal. Technical report, Financial Services Technology Consortium, 1995.

[GT96] C. Gulcu and G. Tsudik. Mixing e-mail with babel. In *1996 Symposium on Network and Distributed System Security*, February 1996.

[HS91] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *Advances in Cryptology – CRYPTO '90*, pages 437–455. Springer-Verlag, Berlin Germany, 1991.

[HSW96] Ralf Hauser, Michael Steiner, and Michael Waidner. Micro-payments based on iKP. Research Report 2791 (# 89269), IBM Research, Feb 1996.

[IEE96] IEEE Computer Society,Technical Committee on Security and Privacy. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, May 1996. IEEE Computer Society Press.

[ISO97] ISO/IEC JTC1, Information Technology SC 27. Information technology - security techniuqes - non-repudiation. ISO/IEC JTC 1/SC 27, 1997. Contains 3 parts; Current version dated March 1997.

[Kat96] M. Ethan Katsch. Dispute resolution in cyberspace. In *Connecticut Law Review Symposium: Legal Regulation of the Internet*, number 953 in 28, 1996. Available from http://www.umass.edu/legal/articles/uconn.html.

[KGMP+96] Steven P. Ketchpel, Hector Garcia-Molina, Andreas Paepcke, Scott Hassan, and Steve Cousins. U-pai: A universal payment application interface. In *Second USENIX Workshop on Electronic Commerce* [USE96], pages 105–121.

[Lam81] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[Lin83] Peter F. Linington. Fundamentals of the layer service definitions and protocol specifications. *Proceedings of the IEEE*, 71(12):1341–1345, December 1983.

[Lin97] J. Linn. Generic security service application program interface, version 2. Internet Network Working Group, Standards Track, Request for Comments: RFC 2078, January 1997. Obsoletes RFC 1508.

[LMP94] S. Low, N. Maxemchuk, and Sanjoy Paul. Anonymous credit cards. In Jacques Stern, editor, *2nd ACM Conference on Computer and Communications Security*, pages 108–117, Fairfax, Virginia, November 1994. ACM Press.

[Mas96] Francisco Fernandes Masaguer. Security in electronic trading over open networks: a detailed analysis and comparison. In *14th Worldwide Congress on Computer and Communications Security Protection*, pages 39–66, C.N.I.T Paris-La Defense, France, June 1996.

[Mat97] Tsutomu Matsumoto, editor. *4th ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997. ACM Press.

[Mau96] Ueli Maurer, editor. *Advances in Cryptology – EUROCRYPT '96*, number 1070 in Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 1996.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, number 293 in Lecture Notes in Computer Science, pages 369–378, Santa Barbara, CA, USA, August 1987. Springer-Verlag, Berlin Germany.

[Mic93] Silvio Micali. Fair public-key cryptosystems. In *Advances in Cryptology – CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, Berlin Germany, 1993.

[Mic97] Silvio Micali. Certified e-mail with invisible post offices. Available from author; an invited presentation at the RSA '97 conference, 1997.

[MN93] Gennady Medvinsky and B. Clifford Neuman. NetCash: A design for practical electronic currency on the internet. In Victoria Ashby, editor, *1st ACM Conference on Computer and Communications Security*, pages 102–106, Fairfax, Virginia, November 1993. ACM Press.

[MV97] Mastercard and Visa. *SET Secure Electronic Transactions Protocol*, version 1.0 edition, May 1997. Book One: Business Specifications, Book Two: Technical Specification, Book Three: Formal Protocol Definition. Available from http://www.mastercard.com/set/#down.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press series on discrete mathematics and its applications. CRC Press, 1996. ISBN 0-8493-8523-7.

[NIS95] NIST National Institute of Standards and Technology (Computer Systems Laboratory). Secure hash standard. Federal Information Processing Standards Publication FIPS PUB 180-1, April 1995.

[NM95] Clifford Neuman and Gennady Medvinsky. Requirements for network payment: The NetCheque Prespective. In *Proceedings of IEEE Compcon '95*, San Francisco, March 1995.

[Ped96] Torben P. Pedersen. Electronic payments of small amounts. In *Cambridge Workshop on Security Protocols*, volume 1189 of *Lecture Notes in Computer Science*, pages 59–68. Springer-Verlag, Berlin Germany, April 1996.

[Pfi95] Birgit Pfitzmann. Contract signing. Personal communication, June 1995.

[Pfi96] Birgit Pfitzmann. *Digital Signature Schemes — General Framework and Fail-Stop Signatures.* Number 1100 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1996.

[PPW91] Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. Practical signatures where individuals are unconditionally secure. Unpublished manuscript, available from the authors (pfitzb@informatik.uni-hildesheim.de), 1991.

[PS96] Birgit Pfitzmann and Matthias Schunter. Asymmetric fingerprinting (extended abstract). In Maurer [Mau96], pages 84–95.

[PSW98] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Optimal efficiency of optimistic contract signing (to appear). In *17th Symposium on Principles of Distributed Computing (PODC), ACM, New York*, 1998.

[PW96] Birgit Pfitzmann and Michael Waidner. Integrity properties of payment systems, December 1996. Private Communication of work in progress; contact the authors for the current status of the work.

[RL96] Ronald L. Rivest and Butler Lampson. SDSI – a simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, April 1996. SDSI Version 1.0.

[RR97] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. Technical Report 97-15, DIMACS, August 1997. Revised version.

[RSA78] Ron L. Rivest, A. Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Journal of the ACM*, 21(2):120–126, February 1978. US Patent 4,405,829: Cryptographic Communications System and Method, Public Key Partners PKP.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc, 1996.

[SEM96] SEMPER Consortium. Basic services: Architecture and design. Deliverable D03 of ACTS Project AC026, Public Specification, S eptember 24, 1996, September 1996. Available from *http://www.semper.org/deliver/d03.*

[Sta96] Markus Stadler. Publicly verifiable secret sharing. In Maurer [Mau96].

[Tsu96] Gene Tsudik. Zürich iKP prototype: Protocol specification document. Technical report, IBM Research, February 1996. IBM Research Report RZ-2792.

[USE95] USENIX. *First USENIX Workshop on Electronic Commerce*, New York, July 1995.

[USE96] USENIX. *Second USENIX Workshop on Electronic Commerce*, Oakland, California, November 1996.

[Wai96] Michael Waidner. Development of a secure electronic marketplace for Europe. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS)*, number 1146 in Lecture Notes in Computer Science, Rome, Italy, September 1996. Springer-Verlag, Berlin Germany. also published in: EDI Forum 9/2 (1996) 98-106.

[ZG96] Jianying Zhou and Dieter Gollmann. A fair non-repudiation protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* [IEE96], pages 55–61.