Design and Implementation of Extensible Middleware for Non-repudiable Interactions

Thesis by

Paul Fletcher Robinson

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

UNIVERSITY OF
NEWCASTLE UPON TYNE

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

2006

(Submitted 2006)

© 2006
Paul Fletcher Robinson
All Rights Reserved

To Claire

Acknowledgements

The three years of my PhD have been some of the best years of my life (although I really hope they are superseded!). Several people have been influential during these years and without them my PhD would not have been possible:

Firstly I would like to thank my supervisor, Professor Santosh Shrivastava. He gave me exactly the right level of encouragement and support I needed, whilst allowing me to be independent. Santosh's ability to give constructive criticism without causing offense helped me to stay focused.

I would also like to thank Nick Cook who I worked with closely during my research. Nick and I had many discussions; several of which resulted in the solving of problems that would have taken weeks to solve had I been alone. Nick also kept me on track by quickly pointing out flaws in potential avenues of research.

I was very lucky to have been awarded a studentship from EPSRC and a CASE studentship from the Hewlett Packard Arjuna Lab. These studentships allowed me to enjoy the lifestyle of an employed researcher, rather than that of a student researcher. As a result, I was never distracted from my research with money related worries. For this I am extremely grateful.

Many friends have been exceptionally helpful during my PhD. From developing ideas on a white-board, to relaxing with a drink after a long days work. Without them my PhD wouldn't have been as enjoyable. There are too many to mention individually but each knows how much I value their friendship. Having said that, I would like to acknowledge one friend in particular, Simon Woodman, who has provided a large amount of input into this thesis by reviewing drafts and discussing ideas (unfortunately[1] many of these ideas were forgotten due to the discussions being held in the pub).

Most importantly I would like to thank my parents. Without their support and encouragement I would never have been in a position to begin a PhD, let alone complete it.

[1]...or maybe fortunately as these ideas were probably of dubious quality.

Abstract

Non-repudiation is an aspect of security that is concerned with the creation of irrefutable audits of an interaction. Ensuring the audit is irrefutable and verifiable by a third party is not a trivial task. A lot of supporting infrastructure is required which adds large expense to the interaction. This infrastructure comprises, (i) a non-repudiation aware run-time environment, (ii) several purpose built trusted services and (iii) an appropriate non-repudiation protocol. This thesis presents design and implementation of such an infrastructure. The runtime environment makes use of several trusted services to achieve external verification of the audit trail. Non-repudiation is achieved by executing fair non-repudiation protocols. The Fairness property of the non-repudiation protocol allows a participant to protect their own interests by preventing any party from gaining an advantage by misbehaviour. The infrastructure has two novel aspects; extensibility and support for automated implementation of protocols.

Extensibility is achieved by implementing the infrastructure in middleware and by presenting a large variety of non-repudiable business interaction patterns to the application (a non-repudiable interaction pattern is a higher level protocol composed from one or more non-repudiation protocols). The middleware is highly configurable allowing new non-repudiation protocols and interaction patterns to be easily added, without disrupting the application.

This thesis presents a rigorous mechanism for automated implementation of non-repudiation protocols. This ensures that the protocol being executed is that which was intended and verified by the protocol designer. A family of non-repudiation protocols are taken and inspected. This inspection allows a set of generic finite state machines to be produced. These finite state machines can be used to maintain protocol state and manage the sending and receiving of appropriate protocol messages.

A concrete implementation of the run-time environment and the protocol generation techniques is presented. This implementation is based on industry supported Web service standards and services.

Contents

Acknowledgements **5**

Abstract **7**

1 Introduction **19**

1.1 Research issues and definitions . . . 20

1.1.1 Security . . . 20

1.1.2 Non-repudiation . . . 21

1.1.3 Definitions . . . 22

1.1.3.1 Evidence . . . 22

1.1.3.2 Fairness . . . 22

1.1.3.3 Trusted third parties . . . 23

1.1.4 Common notation . . . 25

1.2 Thesis overview . . . 25

1.3 Thesis contribution . . . 29

2 Literature Review **31**

2.1 Non-repudiation . . . 31

2.1.1 Certificate management . . . 32

2.1.2 Fault tolerance . . . 33

2.2 Fair non-repudiation protocols . . . 34

2.2.1 Gradual exchange protocols and probabilistic protocols . . . 36

2.2.2 Inline TTP based protocols . . . 37

2.2.2.1 Coffey-Saidha protocol . . . 38

2.2.3 On-line TTP based protocols . . . 40

2.2.3.1 Zhou Gollmann protocol with on-line TTP . . . 40

2.2.4 Off-line TTP-based protocols . . . 43

2.2.4.1 Zhou Gollmann protocol with off-line TTP . . . 44

2.2.5 Transparent off-line TTP based protocols . . . 47

2.2.5.1 G Wang protocol . . . 48
2.2.6 Non-repudiation protocol comparisons . . . 51
2.3 Non-repudiation middleware and applications . . . 53
2.3.1 CORBA Non-repudiation Service . . . 53
2.3.2 TY*SecureWS . . . 54
2.3.3 DataPower XML Security Gateway and Verisign Trust Gateway . . . 54
2.3.4 FIDES . . . 55
2.3.5 B2B Objects . . . 55
2.4 Automatic security protocol implementation . . . 56
2.5 Business interactions . . . 57
2.5.1 RosettaNet PIPs . . . 57
2.5.2 Contracts . . . 58
2.6 Discussion . . . 59

3 Non-Repudiation Protocol Hierarchy **61**
3.1 The non-repudiation protocol hierarchy . . . 62
3.1.1 Domain specific message exchange patterns . . . 63
3.1.1.1 RosettaNet - request purchase order . . . 64
3.1.1.2 RosettaNet - notify of purchase order update . . . 65
3.1.1.3 Business protocol decomposition . . . 65
3.1.2 Generic MEPs . . . 65
3.1.3 Protocol level . . . 68
3.1.4 Example protocol composition . . . 71
3.2 Delivery-agent based fair exchange . . . 72
3.2.1 Protocol overview . . . 73
3.2.2 Assumptions . . . 73
3.2.3 Notation . . . 74
3.2.4 Fair exchange for light-weight end users . . . 76
3.2.5 Fair exchange with light-weight delivery agent . . . 80
3.2.6 TTP message validation . . . 81
3.3 Summary . . . 81

4 Protocol Representation **83**
4.1 Supported non-repudiation protocols . . . 83
4.2 Finite state machine representation . . . 84
4.2.1 Exchange protocol FSMs . . . 84
4.2.1.1 A's exchange protocol FSM . . . 85

4.2.1.2 B and TTP's exchange protocol FSM . . . 86
4.2.2 Exception handling FSMs . . . 87
4.2.2.1 A and B's exception handling FSM . . . 87
4.2.2.2 TTP's exception handling FSM . . . 88
4.3 Non-repudiation protocol notation . . . 90
4.3.1 Assumptions . . . 90
4.3.2 Notation for describing protocols . . . 91
4.3.2.1 Example . . . 95
4.3.3 Protocol message creation . . . 96
4.3.4 Protocol message verification . . . 98
4.4 Example creation and verification steps . . . 100
4.5 Summary . . . 101

5 Middleware Implementation 103
5.1 Web services and supporting standards . . . 103
5.1.1 SOAP . . . 104
5.1.2 WSDL . . . 104
5.1.3 XML Signature . . . 105
5.1.4 XML Encryption . . . 106
5.1.5 WS-Security . . . 107
5.1.6 WS-Reliability & WS-Reliable Messaging . . . 109
5.1.7 WS-Addressing . . . 109
5.1.8 XML Key Management Service (XKMS) . . . 109
5.1.9 Digital Signature Service (DSS) . . . 109
5.2 WS-NRExchange architecture . . . 110
5.3 Protocol messages . . . 111
5.3.1 WS-NRExchange interface . . . 111
5.3.2 Protocol message format . . . 112
5.3.3 Protocol message creation . . . 114
5.3.4 Protocol message verification . . . 122
5.4 WS-NRExchange service implementation . . . 126
5.4.1 Domain specific MEP layer . . . 126
5.4.2 Generic MEP layer . . . 129
5.4.3 Protocol layer . . . 132
5.5 Finite state machine based protocol handler . . . 133
5.5.1 Protocol handler implementation . . . 135

5.5.2 FSM implementation 135
5.6 Evaluation 137
5.6.1 Automation 137
5.6.2 Configuration 138
5.6.3 Confidence in implementation 138
5.7 Summary 139

6 Summary and Future Work 141
6.1 Non-repudiable business protocols (Chapter 3) 141
6.2 Protocol representation (Chapter 4) 142
6.3 Middleware implementation (Chapter 5) 142
6.4 Future work 142
6.4.1 Deadlines and Timing 142
6.4.2 Fault tolerance 143
6.4.3 Reliable messaging 143
6.4.4 DA Protocol verification 144
6.4.5 Multi-party non-repudiation 144
6.4.6 NR Information sharing 145
6.4.7 Support for other protocol classifications 146

Bibliography 147

A Examples 155
A.1 Protocol notation 155
A.1.1 Exchange protocol 155
A.1.2 Exception handling protocols 156
A.1.2.1 Protocol initiator's (A) abort protocol 156
A.1.2.2 Protocol responder's (B) abort protocol 156
A.1.2.3 Protocol initiator's (A) resolve protocol 156
A.1.2.4 Protocol responder's (B) resolve protocol 157
A.2 Finite state machines 157
A.2.1 Exchange protocol FSMs 158
A.2.1.1 Protocol initiator (A) 158
A.2.1.2 Protocol responder (B) 159
A.2.1.3 Trusted third party (DA) 160
A.2.2 Exception handling protocol FSMs 160
A.2.2.1 Protocol initiator and responder (A and B) 160

A.2.2.2 Trusted third party (DA) 161
A.3 Soap protocol messages 161
A.3.1 Exchange protocol message 1 162
A.3.2 Exchange protcol message 2 163
A.4 Protocol message schema 165

List of Figures

1.1 Traditional protocol development cycle 26
1.2 Proposed new development cycle 27

2.1 Fair exchange protocol hierarchy 36
2.2 Coffey-Saidha Protocol 39
2.3 Improved Coffey-Saidha Protocol 40
2.4 Zhou Gollmann protocol with on-line TTP 41
2.5 Extended Zhou Gollmann protocol with on-line TTP 43
2.6 Zhou Gollmann protocol with off-line TTP - exchange protocol 45
2.7 Zhou Gollmann protocol with off-line TTP - abort sub-protocol 46
2.8 Zhou Gollmann protocol with off-line TTP - resolve subprotocol 46
2.9 G Wang - exchange protocol 49
2.10 G Wang - abort protocol 49
2.11 G Wang - resolve protocol 50

3.1 Non-repudiation protocol hierarchy 62
3.2 PIP3A4 Process Diagram 64
3.3 PIP3A4 Sequence Diagram 65
3.4 PIP3A7 Process Diagram 66
3.5 PIP3A7 Sequence Diagram 66
3.6 Validated Message 66
3.7 Validated Request/Response 67
3.8 Single Message 67
3.9 Request/Response 68
3.10 Send protocol 68
3.11 Validated Send protocol 68
3.12 NR-Send protocol 69
3.13 Validated NR-Send protocol 69
3.14 Fair NR-Send protocol 69

3.15 Validated Fair NR-Send protocol 70
3.16 Two compositions for achieving non-repudiation of PIP3A4 71
3.17 Executing a validated fair NR-send MDP through a delivery agent 72

4.1 A's Exchange Protocol FSM 85
4.2 B's Exchange Protocol FSM 86
4.3 Generic exception handling FSM 88
4.4 TTP Exception Handling FSM 89

5.1 SOAP Envelope 104
5.2 Example simplified WSDL document 105
5.3 XML Purchase Order 105
5.4 XML Signature Example 106
5.5 XML purchase order with encrypted credit card number 107
5.6 Purchase Order SOAP Message 108
5.7 WS-Security Protected Purchase Order SOAP Message 108
5.8 WS-NRExchange and Web service standards 110
5.9 WS-NRExchange architecture 110
5.10 Extract of NRExchange WSDL 112
5.11 General form of ProtocolMessage 113
5.12 General form of ProtocolStateMessage 114
5.13 domain specific MEP layer implementation 127
5.14 PIP3A4 to message exchange pattern mapping 128
5.15 PIP3A7 to message exchange pattern mapping 128
5.16 Message exchange pattern layer implementation 129
5.17 Single message to fair non-repudiation protocol mapping 130
5.18 Validated single message to fair non-repudiation protocol mapping 131
5.19 Request/response to fair non-repudiation protocol mapping 131
5.20 Validated Request/response to fair non-repudiation protocol mapping 131
5.21 NRExchange Service Internals 132
5.22 Protocol implementation process 137

A.1 Exchange protocol FSM for initiator of DA protocol 158
A.2 Exchange protocol FSM for responder of DA protocol 159
A.3 Exchange protocol FSM for TTP of DA protocol 160
A.4 Exchange protocol FSM for initiator of DA protocol 160
A.5 Exchange protocol FSM for TTP of DA protocol 161

List of Tables

1.1 Common protocol notation 25

2.1 Coffey-Saidha protocol - additional notation 39
2.2 Zhou Gollmann protocol with on-line TTP - additional notation 40
2.3 Evidence for extended Zhou Gollmann protocol with on-line TTP 43
2.4 Zhou Gollmann protocol with off-line TTP - additional notation 45
2.5 G Wang protocol - additional notation 48
2.6 Comparison of TTP based non-repudiation protocols 52
2.7 References to protocol descriptions in this thesis 52

3.1 Notation for protocol elements 75
3.2 Definition of non-repudiation tokens 75

5.1 State transition table for Delivery Agent protocol - participant DA 136

Chapter 1

Introduction

Non-repudiation is an aspect of security that is concerned with the creation of irrefutable audits of an interaction. Ensuring the audit is irrefutable and verifiable by a third party is a challenging task. A lot of supporting infrastructure is required which adds large expense to the interaction. This infrastructure comprises, (i) a non-repudiation aware run-time environment, (ii) several purpose built trusted services and (iii) an appropriate non-repudiation protocol. This thesis presents design and implementation of such an infrastructure. The runtime environment makes use of several trusted services to achieve external verification of the audit trail. Non-repudiation is achieved by executing fair non-repudiation protocols. The Fairness property of the non-repudiation protocol allows a participant to protect their own interests by preventing any party from gaining an advantage by misbehaviour. The infrastructure has two novel aspects; extensibility and support for automated implementation of protocols.

Extensibility is achieved by implementing the infrastructure in middleware and by presenting a large variety of non-repudiable business interaction patterns to the application (a non-repudiable interaction pattern is a higher level protocol composed from one or more non-repudiation protocols). The middleware is highly configurable allowing new non-repudiation protocols and interaction patterns to be easily added, without disrupting the application.

Implementing non-repudiation protocols is not a trivial task and manual implementation can often introduce flaws. This can be due to programmer error, programmer mis-interpretation and also flaws in the environment. Formal methods can be used to add confidence in the correctness of the protocol, however, this correctness can be broken by an incorrect implementation. This thesis presents a rigorous mechanism for implementing non-repudiation protocols which ensures that the protocol being executed is that which was intended and verified by the protocol designer. A family of non-repudiation protocols are taken and inspected. This inspection allows a set of generic finite state machines to be produced. These finite state machines can be used to maintain protocol state and manage the sending and receiving of appropriate protocol messages.

The thesis also presents a concrete implementation of the runtime environment and the protocol

generation techniques. This implementation is based on industry supported Web service standards and services.

1.1 Research issues and definitions

Achieving non-repudiation is intellectually demanding. In order to achieve non-repudiation of a particular action, it must be impossible for the originator of the action to deny responsibility. Achieving this will rely on the generation of irrefutable evidence that can be used to prove that the action was taken after it has occurred. All parties involved in the interaction must obtain this evidence to allow them to prove that the interaction took place. An unfair situation would occur if some but not all parties receive this evidence. If the interaction is of high value, it is conceivable that one or more parties may wish to cheat by preventing another party from obtaining this evidence. In order to maintain fairness, a fair non-repudiation protocol must be used. Fair non-repudiation protocols maintain fairness by introducing a trusted third party (TTP). The TTP is also used to ensure that the evidence is undeniable by its creator (This is explained further in Section 2.1.1). Fair non-repudiation protocols are notoriously difficult to develop and, as a result, the vast majority of published protocols are flawed. Formal methods can be used to find protocol flaws and, as a result, can be used to increase confidence that the protocol does not contain any.

For a non-repudiation protocol to be useful it must be implemented and run within some suitable framework. Implementing non-repudiation protocols is a difficult task, one which is time consuming, tedious and error prone. As a result, a non-repudiation protocol, believed to be free of flaws, can become flawed due to its implementation. By utilising a systematic approach to non-repudiation protocol development, these development-introduced flaws can be removed.

As stated earlier, non-repudiation protocols need to be executed within a suitable framework. This framework allows applications to make use of non-repudiation, without having to implement their own non-repudiation protocol. To be useful to the applications, such a framework should provide a collection of non-repudiable interaction patterns. These non-repudiable interaction patterns can be used whenever the application needs to undertake a particular interaction that requires non-repudiation. *Request / response* is an example of one such interaction pattern.

The remainder of this section describes some basic concepts used throughout this thesis. There is also a set of definitions and notation frequently used.

1.1.1 Security

Security is a rather broad term and typically comprises five attributes.

- **Authentication**. The process of ensuring a principal is indeed who they purport to be.

- **Authorisation**. The process of establishing what actions a (authenticated) principal is allowed to undertake.
- **Confidentiality**. The process of preventing information from being discoverable by unintended principals.
- **Integrity**. The process of ensuring modification of a piece of information can be detected.
- **Non-Repudiation**. Inability of a principal to refute a particular action within a particular context.

Although non-repudiation is expressed as its own attribute, it does in fact encompass many of the other attributes. Non-repudiation can only be achieved if modification to the information in question can be detected. Thus non-repudiation also provides integrity. Non-repudiation is concerned with which principals were involved in the interaction. Thus non-repudiation also provides authentication. It is common for non-repudiation to be used in high value interactions. As a result, it would be highly likely that the mechanism used to achieve non-repudiation would also provide confidentiality. From this reasoning it can be seen that authorisation is the only aspect not of concern to non-repudiation. This is because non-repudiation is concerned with establishing exactly what happened during an interaction, not whether the action was allowed to happen. If a party were to undertake an action they were not authorised to do, it could easily be detected as the non-repudiation property would ensure the party in question could not refute the action. This is not to say that authorisation is not required, more that it would be achieved separately to non-repudiation.

1.1.2 Non-repudiation

As stated above, non-repudiation is the inability of a principal to refute an action within a given context. In practice, non-repudiation is typically concerned with the sending of a message from one principal to another. Non-repudiation ensures the originator (A) of the message (msg) cannot deny sending it and the recipient (B) of the message cannot deny receiving it. Non-repudiation is achieved by generating evidence at various points during the sending of a message. To prevent the evidence from being refuted, it is digitally signed by the originator. To achieve non-repudiation, A would generate evidence for non-repudiation of msg's origin. On B's receipt of msg, B can generate evidence for non-repudiation of receipt for msg. It is important that A receives the non-repudiation of receipt evidence and that B receives the non-repudiation of origin evidence and msg. It is these evidences that are required to prove what happened, during the interaction, to a trusted third party (judge) in case of dispute. An unfair situation can occur if one party receives their evidence and the other does not. Assume B receives evidence for non-repudiation of origin but A does not receive the evidence for non-repudiation of receipt. In this scenario B may repudiate receiving msg from A

because A cannot prove it was received. To prevent unfair situations a *fair non-repudiation protocol* is used to ensure that either both parties obtain their expected items or neither does.

There are other facilities required by a non-repudiation protocol, other than evidence generation and fairness. These are discussed further in Section 2.1.

1.1.3 Definitions

This section presents definitions of phrases and symbols commonly used throughout this thesis. Those not present here are required for localised use and are thus defined in situ.

1.1.3.1 Evidence

Four types of non-repudiation evidence are used throughout this thesis. A description of each follows along with the intended purpose.

- **Non-repudiation of origin (NRO)**. This protects against the originator's false denial of having sent the message. This would typically comprise a signed and zero or more unsigned parts. The signed part is referred to as the evidence of origin (EOO).
- **Non-repudiation of receipt (NRR)**. This protects against the recipient's false denial of having received the message. This would typically comprise a signed and zero or more unsigned parts. The signed part is referred to as the evidence of receipt (EOR).
- **Non-repudiation of submission (NRS)**. This provides evidence that the originator submitted the message to some intermediary. This would typically comprise a signed part and zero or more unsigned parts. The signed part is referred to as the evidence of submission (EOS).
- **Non-repudiation of validation (NRV)**. This provides evidence that the originator has validated the message and also as to the outcome of this validation. This would typically comprise a signed part and zero or more unsigned parts. The signed part is referred to as the evidence of validation (EOV).

The NRO, NRR and NRS are commonly used throughout the literature. The NRV is used in a protocol proposed in Section 3.2.

1.1.3.2 Fairness

Fairness ensures that at the end of a non-repudiation either one of two situations occur:

1. All parties get expected items.

2. No parties get any items of any value.

More specifically, four types of fairness can be expressed. Kremer et al. [47] provide definitions within the context of fair non-repudiation protocols:

- **Weak Fairness**. A non-repudiation protocol provides weak fairness if it ensures that if a participant A (or B) does not obtain its evidence, while the other participant B (or A) did, then A (or B) will receive a proof of this fact.
- **Strong Fairness**. A non-repudiation protocol provides strong fairness if and only if at the end of a protocol execution either A got the non-repudiation of receipt evidence for msg, and B got the corresponding msg as well as the non-repudiation of origin evidence for this msg, or neither of them got any valuable information.
- **True Fairness**. A non-repudiation protocol provides true fairness if and only if it provides strong fairness and, if the exchange is successful, the non-repudiation evidences produced during the protocol are independent of how the protocol is executed.
- **Probabilistic Fairness**. A non-repudiation protocol is e-fair if and only if the probability that at the end of a protocol execution either A got the non-repudiation of receipt evidence for the message msg, and B got the corresponding message msg as well as the non-repudiation of origin evidence for msg, or none of them got any valuable information, is $\geq 1 - e$.

These definitions are useful for expressing fairness properties of non-repudiation protocols. Markowitch et al. [55] define two additional properties, of a non-repudiation protocol, that impact fairness. These properties are as follows:

- **Timeliness**. A non-repudiation protocol is said to be timely if it ensures that both A and B will always have the ability to reach, in a finite amount of time, a point in the protocol where they can stop the protocol while preserving fairness.
- **Abuse-Free**. A non-repudiation protocol is said to be abuse free if it is impossible for a single entity, at any point in the protocol to be able to prove to an outside party that he has the power to either terminate (abort) or successfully complete the protocol.

Timeliness is a key goal of some of the more modern fair non-repudiation protocols; where as the abuse-free property tends to attracts less attention.

1.1.3.3 Trusted third parties

Zhou and Gollmann [91] have presented a clear definition of what comprises a trusted third party. They also define the different roles that a trusted third party can take:

A trusted third party (TTP) is a security authority or agent that is trusted by all parties involved within a particular interaction. TTPs play important roles in non-repudiation services. Depending on the non-repudiation protocol used and the non-repudiation policy in force, TTPs may be involved in different ways to assist participants to generate, verify, or transfer non-repudiation evidence and resolve disputes.

The level of involvement required by a TTP (during a non-repudiation protocol) can vary. Kremer et al [47] provide definitions for five levels of involvement.

- **Inline TTP**. A TTP involved in each protocol message's transmission during the protocol is said to be inline.
- **Online TTP**. A TTP involved during each session of the protocol but not during each (protocol) message's transmission, is said to be online.
- **Offline TTP**. A TTP involved in a protocol only in case of an incorrect behavior of a dishonest entity or in case of a network error, is said to be offline.
- **Neutral TTP**. A TTP is known as neutral if the assistance that it brings to the successful realisation of a protocol is not conditioned by its knowledge of the information exchanged.
- **Transparent TTP**. An offline TTP providing evidences indistinguishable from the evidences exchanged in a faultless case, is said to be transparent.

Zhou and Gollmann define the following roles that a TTP can take:

- **Certification Authority**. A certificate authority generates key certificates which guarantee the authenticity of verification keys to be used for non-repudiation purposes. They also maintain a certificate revocation list (CRL) in order to determine the validity of existing verification keys. Certificate authorities are always required when digital signatures are used for evidence generation. They will usually be offline in a non-repudiation service.
- **Notary**. A notary is trusted by the communicating parties to provide correct evidence on their behalf or verify evidence correctly. Properties about the message, such as its origin and integrity, can be assured by the provision of a notarisation mechanism.
- **Delivery Agent**. A delivery agent is trusted to deliver a message from one party to another and provide them with corresponding evidence. A delivery agent is usually inline and can also be used to provide evidence regarding the time of message submission and delivery.
- **Adjudicator**. The ultimate purpose of a non-repudiation service is to resolve disputes about the occurrence or non-occurrence of a claimed event or action. An adjudicator is a judge capable of resolving disputes by evaluating the evidence against a non-repudiation policy. An

adjudicator is only involved in a non-repudiation service if a dispute occurs. The dispute is resolved on presentation of non-repudiation evidence. Evidence not presented is assumed not to exist.

These definitions are used frequently throughout this thesis to describe involvement of trusted third parties.

1.1.4 Common notation

Table 1.1 presents notation commonly used throughout this thesis.

Notation	Description
msg	the application message
A	the originator of msg
B	the intended recipient of msg
TTP	the trusted third party
X, Y	concatenation of two tokens X and Y
rn	a secure pseudo random number.
$h(X)$	a one way hash function applied to token X
K	symmetric key
$eK(X)$ and $dK(X)$	encryption and decryption of X with symmetric key K
$P \rightarrow Q : msg$	principal P sends message msg to principal Q.
$P \leftarrow Q : msg$	principal P gets message msg from principal Q.
$sig_P(X)$	principal P's digital signature on X.
$enc_P(X)$	X encrypted with principal P's public key.

Table 1.1: Common protocol notation

1.2 Thesis overview

This thesis presents a framework to render a business interaction non-repudiable. In order to render the business interaction truly non-repudiable it must be impossible for any party to refute their participation. Furthermore, the evidence generated must be sufficient to convince a judge in case of dispute.

At the heart of a non-repudiation service is the fair non-repudiation protocol. It is this protocol that ensures valid evidence is generated and that the intended parties receive the items they are supposed to. Traditionally, a non-repudiation protocol will go through a set of processes before

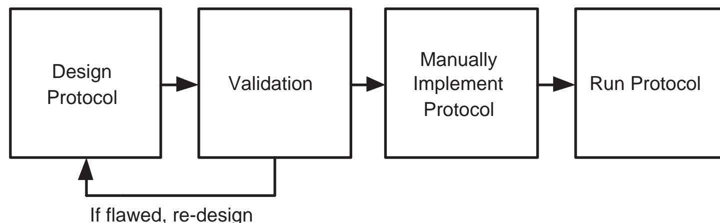


Figure 1.1: Traditional protocol development cycle

it is used in an application. It should be noted that this process is the same for any software engineering task. Figure 1.1 shows the traditional development cycle for a protocol. Initially, a protocol design is created. This design is then validated to improve confidence that it is free of flaws (this validation could be achieved by using automated formal methods based tools). Once the protocol has been validated, a protocol developer would take the design and implement the protocol. This implementation process is open to programmer error and also at the mercy of the programmer's interpretation of the design. Once implemented, the protocol is executed within a suitable framework. By the time the protocol is used in a real application, the confidence gained from the protocol's validation is lost due to the uncertainties introduced by the manual protocol implementation.

There are two main issues that can be raised by inspecting the traditional protocol development life cycle. The first is that the method of going from a verified protocol design to a faithful protocol implementation is difficult to achieve. The second issue regards the mentioned "suitable framework". Again, creating a framework capable of executing a non-repudiation protocol and providing all required services is non-trivial. To address the first issue, a new protocol development cycle is proposed. For the second issued, a protocol execution framework is presented.

Figure 1.2 shows the proposed protocol development cycle. A validated protocol design is developed in the same way as in the traditional approach. This validated protocol design is then used automatically to create a protocol implementation. The protocol implementation is then passed to the execution framework for execution and use within applications requiring non-repudiation. The execution framework is likely to be rather complex and thus beyond the capabilities of formal verification techniques. Section 5.6.3 describes a mechanism that could be used to increase confidence in the framework. This thesis is mainly concerned with the processes enclosed in the grey shaded area. However, in Chapter 3 a novel extension of a particular fair non-repudiation protocol will be presented — this work lies within the "design protocol" box. The protocol validation process has been researched heavily and is outside the scope of this thesis.

In Chapter 3, the protocol execution framework is presented. In order for this framework to be useful to application developers, a rich set of interaction patterns must be provided. To achieve this,

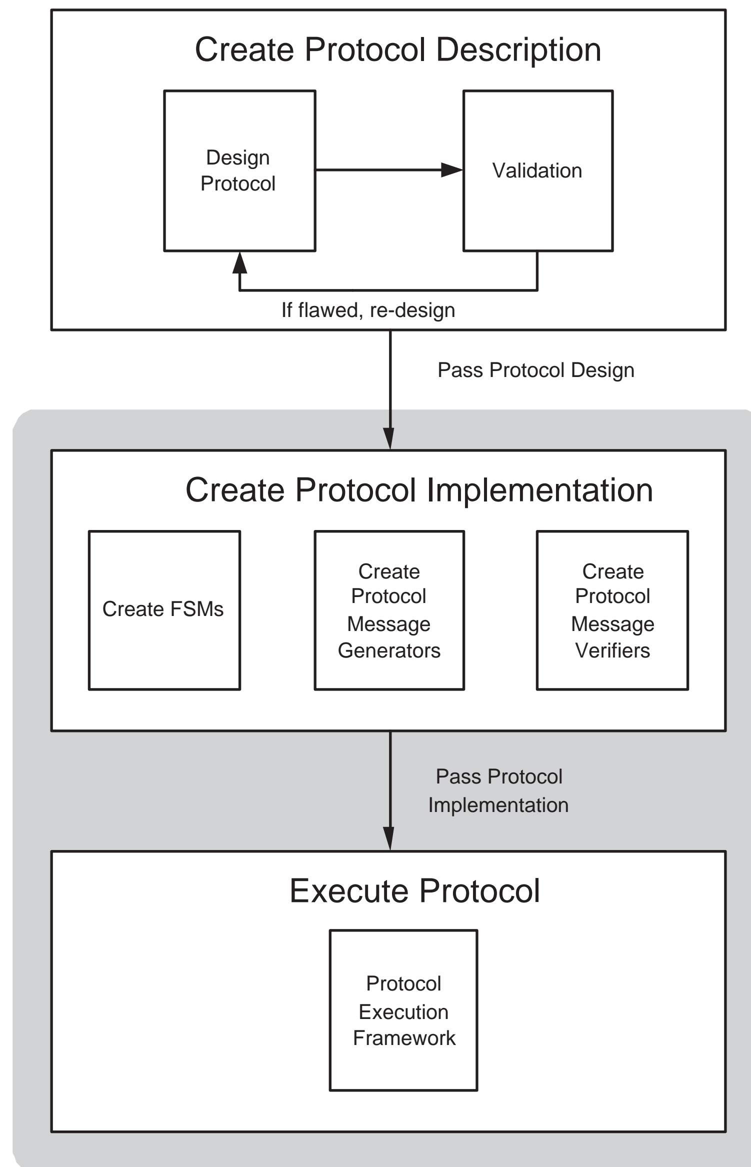


Figure 1.2: Proposed new development cycle

the framework presents an extensible non-repudiation protocol hierarchy. This hierarchy provides a wide range of non-repudiable interaction patterns for composing applications. Some of these are designed for very specific commonly used business interactions such as raising a purchase order or sending an invoice. Other interaction patterns, of a more general nature, are also presented. These provide facilities for undertaking interactions such as non-repudiable request/response. The hierarchy is composed in such a way that high level protocols are composed from lower level protocols. At each level of the hierarchy, new protocols can be added and old protocols can be removed. Furthermore, existing protocols can be re-configured such that their composition can be modified. Each of these re-configurations can be made without modifying the application interface and thus without causing disruption.

Business interactions are typically of high value and, as a result, it could be desirable to enact purpose built (professional) services to undertake some of the more demanding requirements of non-repudiation. As a result, critical services can be delegated to an expert. For example, auditing could be undertaken by a third party auditing service. Audits may be required to store non-repudiation evidence for many years; as a result, an intangible amount of data needs to be stored. This data must be stored reliably and in such a way that a particular piece of evidence can be obtained when needed. As a result, evidence storage can become rather cumbersome to undertake on an individual bases and may be better left to an expert. To support a service based architecture, a novel fair non-repudiation protocol is presented in Section 3.2. This protocol facilitates a (configurable) service based architecture whilst achieving other preferable features of a non-repudiation protocol.

Chapter 4 presents a process for creating a design faithful non-repudiation protocol implementation. This chapter shows how a class of non-repudiation protocols, designed using notation commonly found in the literature, can be represented using finite state machines (FSMs). These finite state machines also consider what to do if an exception occurs. Transitions in the FSM occur when a valid protocol message is received or when a signal is received from the protocol execution framework. For example, signals can be used to start the protocol and to signal exceptions. When a transition is made, zero or more protocol messages are sent. The finite state machines rely on mechanisms for verifying received protocol messages and for generating new ones. These requirements are achieved by careful inspection of the protocol design notation and by a set of mapping functions. This process can be totally automated and, as a result, removes the potential for error introduced by the manual protocol development present in the traditional protocol development cycle.

Chapter 5 presents a concrete implementation of the protocol execution framework and the process for automatic development of non-repudiation protocols. The non-repudiation protocol hierarchy is also present in the implementation. The implementation is middleware-based and makes use of Web service standards for the protocol messages exchanged between participants. The chapter shows mapping functions that are used to map protocol messages from the protocol design onto real

messages. The inverse is also presented, this allows the values of protocol tokens to be taken from real protocol messages.

1.3 Thesis contribution

This thesis makes several contributions to the research work on non-repudiation and B2B systems design. The main contributions are as follows:

- A flexible and extensible hierarchy for composing (non-repudiable) high level interactions from fair non-repudiation protocols.
- A novel protocol for service oriented fair non-repudiation (situated in the "Design protocol" phase of Figure 1.2).
- A novel approach for rigorous implementation of a class of fair non-repudiation protocols (situated in the "Create protocol implementation" phase of Figure 1.2).
- A concrete framework for executing non-repudiation protocols (situated in the "Execute protocol" phase of Figure 1.2).
- A design section describing how the above contributions could be implemented.

Chapter 2

Literature Review

This chapter presents a review of research relevant to this thesis conducted by various academic and industry based research groups. The chapter begins by discussing research based directly on non-repudiation and also covers the related issues of certificate management, fairness and fault tolerance. The next section takes an in depth look at fair non-repudiation protocols and concludes with a comparison of all (that we are aware of) published TTP based fair non-repudiation protocols. The following section presents various solutions for achieving non-repudiation. This section includes various middleware solutions for achieving non-repudiation and one application for achieving fair exchange. There follows a discussion of various methods for automatically generating implementations of security protocols. A discussion of business interaction related issues follows. Here the work by RosettaNet is presented as well as research regarding contract enforcement. The final section provides discussion of the research presented in this chapter. This includes an overview of the key points discovered and a comparison of existing research to that presented in this thesis.

2.1 Non-repudiation

Non-repudiation is the inability to deny a particular action. In the context of distributed systems, non-repudiation can be applied to the sending and receiving of messages. Consider an interaction where A sends a message *msg* to B. The following shows the non-repudiation evidence that is required.

1. Non-repudiation of origin (NRO). This provides undeniable proof that A was the originator of *msg*.

2. Non-repudiation of Receipt (NRR). This provides undeniable proof that B received *msg*.

The NRO provides B with a guarantee that A did in fact send the message. This is typically implemented using public key cryptography techniques. A will use their private key to digitally sign the message. As it is assumed that only A has access to their private key, it provides B with

undeniable evidence as to the message's origin. The NRR provides A with a guarantee that B did in fact receive the message. Again this would be implemented using digital signatures, where B would sign an acknowledgement of receipt and return to A.

To prevent possible denial of evidence creation, the NRO and NRR must be time-stamped by a trusted third party to ascertain when the evidence was created. Also, to ensure both parties receive the evidence they require, the NRO and NRR must be exchanged fairly. These issues are discussed below.

2.1.1 Certificate management

Digital signatures can be used to authenticate the origin and integrity of an associated message and are used for creating non-repudiation evidence. Thus, the integrity of the evidence is dependent on the security of the digital signatures. It is not sufficient for the signature scheme to use strong cryptographic techniques — the signature keys must also be well managed.

For a digital signature to be valid, the signature key must not be compromised. In practice this could be difficult to ensure and, as a result, signature keys are revocable. When a key is suspected of being compromised it should be revoked. The act of revocation involves adding the key details to a certificate revocation list (CRL). When validating a signature, the CRL should be consulted to ensure the signature key has not been compromised. This, however, presents a problem for non-repudiation evidence. Consider a situation where principal A generates evidence e at time t using key k. A then revokes k at time $t+1$. As the time of creation cannot be proved from e, A may deny creating e, claiming that it was produced at some time t' where $t' > t+1$. This problem can be resolved by ensuring e contains t. However t needs to be added to e in such a way that it can't be spoofed by the holder of A's signature key.

One aproach to solving this problem is to use an online trusted time stamping authority (TSA). Each newly created signature is time-stamped by the TSA so that a latest time of signature creation is established. Various schemes are based on this idea [3, 13, 24]. Timestamping can prove too inefficient and expensive for low value transactions. This is because the TSA has to be involved every time a signature is created. Although expensive, this mechanism provides the highest level of security as an exact time-stamp is provided. Thus, for some high value transactions, this may be the best option.

In response to the high expense associated with online time stamping services, various schemes have been proposed that reduce or eliminate the TTP requirements. In 1999, Zhou and Lam [94] proposed a scheme based on temporary certificates that reduced the TTP requirements to that of an online TTP. This is an improvement as the TTP needs to be available, but not every time a signature is created. In the scheme, a principal has two types of keys: Revocable signature keys and irrevocable signature keys. Revocable signature keys are issued by a certificate authority (CA)

and can be revoked as usual. Irrevocable signature keys are issued by users themselves and are time stamped by a TSA. As the name suggests these keys can't be revoked before their expiry. The revocable signature key is used as a long term master key to issue irrevocable signature keys. Signatures created using the irrevocable signature key are only valid during the validity period of the irrevocable signature key. As a result, digital signatures created with the irrevocable signature key do not need to be timestamped by a TSA. This scheme is only recommended for low value transactions. This is due to two properties. The first is that signatures are only valid during the validity period of the associated irrevocable signature key. Thus all disputes must be resolved before the irrevocable key expires. Secondly, due to the key being irrevocable, the owner must protect it or otherwise be liable for any misuse. This threat can be reduced by the irrevocable key owner destroying the key when it is no longer needed. Later in 1999, Zhou and Lam [93] proposed another scheme; however this scheme didn't involve a TTP. Here, every signature created by a user is linked such that any change in order of the signatures or insertion of a new signature can be detected. Signature keys can be revoked by terminating the chain. This prevents any new signature being added to the end of the chain with the revoked key. Although more efficient (no TTP involvement), this approach is only appropriate for regular business transactions as the scheme assumes a link of signatures can be created. In 2002, Itkis and Reyzin [42] proposed *Intrusion-Resilient Signatures* — another TTP-less aproach. This scheme has a strong mathematical basis, which was found in [86] to be too impractical for real application.

For high value transactions, the trusted timestamping aproach is the only method offering high enough levels of security. On the other hand for low value transactions a more efficient TTP-less method can be used.

2.1.2 Fault tolerance

A fair exchange protocol is fault-tolerant if it ensures no loss of fairness to an honest participant, even if the participant's node experiences failures of the assumed type. In other words, an honest user does not suffer a loss of fairness because of their node failure. This is not the case with most of the fair exchange protocols studied in the literature, including the ones presented in this chapter. Ezhilchelvan and Shrivastava (in [29]) describe various approaches to preserving fairness in the presence of node crashes and recovery. Here they develop a family of fair exchange protocols with differing abuser, fault tolerance and communication assumptions. There follows a description of each of these assumptions.

- **Abuser assumptions**. An abuser is a user of the fair exchange application that can misbehave. An abuser may be malicious or restricted. A malicious abuser has total control over the application executing the fair exchange protocol. A restricted abuser may only observe

the protocol messages exchanged as part of the protocol run. Furthermore, a restricted abuser has no access to any keys used by the application, encrypted messages can't be decrypted and signed messages can't be tampered with.

- **Failure assumptions**. A node executing the fair exchange protocol is said to be crash-tolerant if after a crash (the node stops functioning) the node recovers within some finite (but unknown) amount of time. The node has access to a stable store whose contents survive the crash. A node executing the fair exchange protocol is said to be reliable if it never fails.
- **Communication channel assumptions**. A communication channel can be synchronous or asynchronous. In a synchronous communication channel all messages are bounded by a known delay. In an asynchronous communication channel, the maximum delay is finite, but unknown.

The family of protocols consists of eight similar protocols. Each protocol has differing assumptions. These assumptions are derived from every permutation of the above assumptions. Four of the permutations show how a fair exchange protocol can be made crash-tolerant in the presence of restricted abusers and malicious abusers communicating via asynchronous and synchronous communication channels. The fair exchange protocols are rendered crash tolerant by slightly changing the behaviour of each participant. Each crash-tolerant participant must immediately persist protocol messages (on their receipt), to stable storage. This ensures the protocol messages survive the crash and the protocol can be continued when the crashed participant recovers. Furthermore, the TTP must maintain the state of all protocol runs; this allows a recovered participant to find out the final outcome of the protocol (should it have completed).

2.2 Fair non-repudiation protocols

Exchanging non-repudiation evidence is essential for proving what happened during an interaction based on message exchange. However, it is necessary that during any part of the interact no party is disadvantaged. For example, consider a situation where B receives NRO, but A does not receive the NRR. This would be unfair, as B could choose to deny receiving msg but A could not prove sending it. Fairness can be achieved by operating a fair exchange protocol in which A exchanges NRO and msg with B for NRR. Markowitch et al [55] present a precise definition of fairness:

> The communication channel's quality being fixed, at the end of the protocol run, either all involved parties obtain their expected items or none (even a part) of the information to be exchanged with respect to the missing items is received.

A fair exchange protocol is used in a scenario where two parties wish to exchange items without losing fairness. Several categories of fair exchange protocols exist, depending on the information exchanged [55]:

- Electronic purchase of digital goods: Exchange of an electronic item against an electronic payment (issued by the client).
- Digital contract signing: Exchange of signatures on a given electronic document.
- Non-repudiable message delivery: Exchange of an electronic item and its proof of origin against a proof of receipt.
- Certified e-mail: Exchange of an electronic message against proof of receipt.
- Barter: An electronic item of value is exchanged against another electronic of similar value.

It should be noted that there exists a distinction between non-repudiable message delivery protocols and certified e-mail protocols. That being that in the latter case the recipient of the e-mail should not know the sender's identity when deciding whether to accept the message or not [46].

Non-repudiable message delivery can be achieved by generating non-repudiation evidence at various points in the interaction. The originator A of the message can be verified by appending non-repudiation of origin (NRO) evidence to the message. However, to provide evidence of receipt NRR, the recipient B of the message must respond with a signed acknowledgement. There are two possible reasons why A may not receive the NRR:

- An unreliable communication channel. Here, B sent NRR to A, but the message delivery failed.
- A dishonest participant. B decides not to send NRR to A.

In either of these cases, B may repudiate receipt of the message. To ensure A receives NRR **and** B receives *NRO,* a fair exchange protocol must be used. Protocols for achieving non-repudiable message delivery, whilst maintaining fairness, are called *fair non-repudiation protocols.* The following definition by Zhou and Gollmann [90] states what makes a non-repudiation protocol fair:

> A non-repudiation protocol is *fair* if it provides the originator and the recipient with valid irrefutable evidence after completion of the protocol, without giving a party an advantage over the other party in any possible incomplete protocol runs.

Figure 2.1 shows a hierarchy of fair exchange type interactions and a hierarchy of protocols used to implement non-repudiable message delivery. Each of the interaction types under "fair exchange" can be implemented using a type of fair exchange protocol. A fair non-repudiation protocol is a fair exchange protocol that achieves non-repudiable message delivery. Fair non-repudiable message delivery can by achieved by a gradual exchange, probabilistic or deterministic non-repudiation protocol. This thesis is interested in the deterministic protocols (in the grey shaded boxes). These

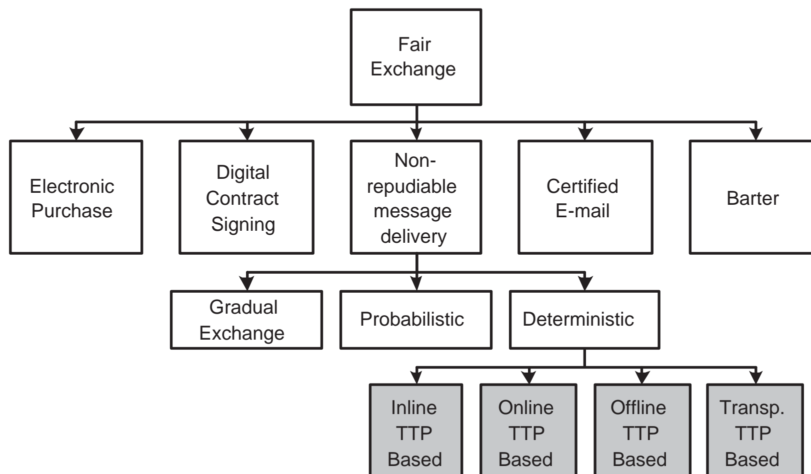


Figure 2.1: Fair exchange protocol hierarchy

protocols can be further sub-divided based on the level of involvement of the TTP. There follows a summary of recent research into non-repudiation protocols, categorised by the level of involvement by the TTP.

2.2.1 Gradual exchange protocols and probabilistic protocols

Kremer et al. undertook an intensive survey of fair non-repudiation protocols [47]; this has been the basis for much of the discussion of two party fair non-repudiation protocols that follows.

With regard to *fair exchange* protocols, those without a TTP were the first to be proposed. During the mid 1980's several such protocols appeared in the literature [8, 14, 15, 22, 62]. However, it was not until 1999 that the first *fair non-repudiation* protocol without a TTP was proposed [54]. The first *fair exchange* protocols without a TTP were developed for exchanging secrets, such as secret keys, between two parties. The basic idea was that both parties took it in turns to transmit a bit of their secret until both secrets had been exchanged in full. This idea assumed both secrets were of equal size. The amount of computation needed to retrieve the secret would halve with each round of the protocol. If one party quit the protocol before completion, there could be at most a factor of two difference in the time needed for each participant to obtain their secret. This time difference was reduced by in [78, 77] where it was shown how a fraction of a bit could be exchanged with each round. With each of these protocols it must be possible to detect whether a party has cheated by sending incorrect bits — this problem was solved in [78, 12, 28].

Each of these protocols was based on a rather unrealistic assumption — that each party would have equivalent computational power. The first protocol not based on this assumption was proposed by Or et al [30] in 1990. The protocol was designed for contract signing, in which parties exchanged privileges rather than the actual secret. An entity is more privileged than another if it has a greater

ability to convince a judge that the contract has been signed by all participants. In the two party protocol, participants exchange messages stating the following:

With a probability P the contract will be valid at the time T

With each round of the protocol, the probability P increases. The protocol ends when one party withdraws, P reaches 1 or time T is met. After time T each party can present the last received message to the judge. The judge will then decide whether or not the contract is valid (signed by all parties). Here the judge picks a random number n ($0 \leq n \leq 1$). If $P > n$ the judge declares the contract to be valid, otherwise the contract is invalid.

In 1998 Syverson proposed protocols [76] for the fair exchange of low value items. Here each item is encrypted using a secret key and sent to the other participant. The secret key is then encrypted using a weaker form of encryption that can be broken in some known (based on the computational power of the participants) period of time. The participants then exchange these encrypted keys. The protocol continues by each participant exchanging their encrypted key under successively weaker encryption until it becomes trivial to break. Again, this protocol makes the rather unrealistic assumption that both participants have equal processing power. Furthermore, at some point in the protocol run, one participant will be able to acquire the key in time where as the other will not, thus fairness is not maintained.

Two party protocols are often disregarded for practical use. This is because they only give probabilistic fairness and they tend to be rather inefficient due to the large number of messages that need to be exchanged. Furthermore, it was shown in Section 2.1.1, that for high value transactions a TTP is required to time stamp evidence. Therefore a seemingly two party protocol, requires a TTP if the value of the transaction is high.

2.2.2 Inline TTP based protocols

The first fair exchange protocol utilising an inline TTP was proposed in 1994 [8]; this was in the context of certified email. The first non-repudiation protocol utilising a TTP was proposed in 1996 by Coffey and Saidha [16]. In this protocol, all communication was done through the TTP. The protocol begins with the TTP collecting the message and non-repudiation evidence. Once the TTP has all the required items, it distributes them to the intended parties. The protocol includes steps dedicated to time stamping the evidence. However, this only adds confusion. If signatures were assumed to be time stamped (by a trusted time stamping authority), these steps could be removed whilst maintaining correctness. A detailed discussion of the protocol is presented in Section 2.2.2.1; here the steps required to time stamp the non-repudiation evidence have been removed. The protocol does not consider timeliness, in that no party may unilaterally bring the run to completion. This can break fairness as both participants will only be willing to wait a particular amount of time before

quiting the protocol. Once one party has quit the protocol, the other may complete the protocol and potentially gain an advantage.

In 1997 Zhou and Gollmann [92] discuss several issues present in the original Coffey and Saidha protocol. The original protocol ensured that all protocol messages remained confidential. This is unnecessary and thus inefficient. In actual fact only two protocol messages need to remain confidential. Furthermore, public key cryptography is used to realise the confidentiality requirements — this is inefficient compared to symmetric key alternatives. The original protocol used nonces to prevent replay attacks; Zhou and Gollmann commented on how this was unnecessary as replay is not of concern to a non-repudiation protocol. This is because the *NRO* and *NRR* are easy to duplicate, thus the number of copies does not represent the number of times the message was sent or received. Coffey and Saidha make the following statement:

> If TTP holds EOR, then the data exchange is deemed to have taken place

Zhou and Gollmann claim this statement is incorrect for the following reason. The TTP will hold EOR before the data exchange actually takes place. The data exchange does not occur until TTP has all required evidence (including the EOR). Thus, if the communication channel is unreliable either participant may not receive their expected items. However, it is normal to assume eventual message delivery during the run of a non-repudiation protocol, in which case the statement holds.

Protocols employing an inline TTP are commonly viewed as having too many disadvantages [85, 49]. The disadvantages are due to the large burden placed on the TTP:

- High communication requirements. The TTP is involved in every step of the protocol.
- High storage requirements. The TTP must store all evidence and the application message for some previously agreed period of time.
- High security requirements. The TTP is required to store sensitive information. For a correctly functioning TTP, this information must remain confidential.

However, in some circumstances it could be desirable to place high demands on the TTP. Essentially, the TTP can provide a high value service by removing burden from the end user participants. For example the TTP could provide a secure storage and archival service that would prevent the end user participants from needing their own mechanism.

There follows a detailed description of the Coffey and Saidha protocol.

2.2.2.1 Coffey-Saidha protocol

Coffey and Saidha [16] proposed a protocol for non-repudiation with mandatory proof of receipt. The protocol uses an inline TTP. Figure 2.2 shows a simplified protocol description. The description

is simplified in that the steps required to apply a trusted time stamp to the digital signatures have been removed. The simplified version is taken from [85]. In the protocol, principal A wishes to send message *msg* to principal B and receive a receipt. A and B do not communicate directly; instead all communication is via TTP, who acts as an inline trusted third party. For this protocol, the following additional notation is used:

Notation	Description
req	request to initiate the protocol
$n_A \, and \, n_B$	nonces chosen by TTP
$EOO = sig_A(A, B, msg)$	evidence of origin of msg
$EOR = sig_B(A, B, H(EOO))$	evidence of receipt of a message satisfying $h(EOO)$

Table 2.1: Coffey-Saidha protocol - additional notation

$$
\begin{array}{lll}
1 & A \rightarrow TTP & : & req \\
2 & TTP \rightarrow A & : & enc_A(n_A) \\
3 & A \rightarrow TTP & : & enc_{TTP}(n_A, A, B, msg, EOO) \\
4 & TTP \rightarrow B & : & enc_B(n_B, A, B, H(EOO)) \\
5 & B \rightarrow TTP & : & enc_{TTP}(n_B, A, B, EOR) \\
6 & TTP \rightarrow B & : & enc_B(A, B, msg, EOO) \\
7 & TTP \rightarrow A & : & enc_A(A, B, EOR)
\end{array}
$$

Figure 2.2: Coffey-Saidha Protocol

The protocol begins with A submitting a request to TTP to begin the protocol. If accepted, TTP will reply with the nonce n_A. In step 3, A sends the nonce n_A, the message *msg* and the evidence of origin *EOO*. *nA* is used by TTP to ensure the freshness of the message. In step 4, TTP sends the nonce n_B and a digest of the *EOO*. Given *H(EOO)*, B is able to generate *EOR* despite not receiving *msg*. Under normal circumstances, B would not be willing to submit *EOR* before receiving the associated *msg*. However, B trusts TTP and believes that *EOR* will only be disclosed to A should *msg* be available to B. B responds in step 5 by returning the *EOR* along with the nonce n_B to *TTP*. Again, the nonce $n_B$is used to ensure freshness of this message. TTP now has both the *EOR* and the *EOO*. Step 6 involves TTP sending the *EOO* and the message *msg* to *B*. Similarly TTP send the *EOR* to A in step 7.

Figure 2.3 shows the improved protocol taking into account Zhou and Gollmann's comments in [92].

Notation	Description
$C = eK(M)$	Commitment (cipher text) for message M.
$L = H(M, K)$	A unique label linking C and K.
$f_i(i = 1, 2, ...)$	Flags indicating the intended purpose of a signed message
$EOO_C = sig_A(f_1, B, L, C)$	Evidence of origin of C.
$EOR_C = sig_B(f_2, A, L, C)$	Evidence of receipt of C.
$sub_K = sig_A(f_3, B, L, K)$	Authenticator of K provided by A.
$con_K = sig_{TTP}(f_4, A, B, L, K)$	Evidence of confirmation of K issued by TTP.

Table 2.2: Zhou Gollmann protocol with on-line TTP - additional notation

$$
\begin{array}{lllll}
1 & A \rightarrow TTP & : & enc_{TTP}(A, B, msg, EOO) \\
2 & TTP \rightarrow B & : & A, B, H(EOO) \\
3 & B \rightarrow TTP & : & enc_{TTP}(A, B, EOR) \\
4 & TTP \rightarrow B & : & A, B, msg, EOO \\
5 & TTP \rightarrow A & : & A, B, EOR
\end{array}
$$

Figure 2.3: Improved Coffey-Saidha Protocol

2.2.3 On-line TTP based protocols

The first non-repudiation protocol employing an online trusted third party was proposed in 1983 by Rabin [69]. This protocol only provided probabilistic fairness. In 1996 Zhang and Shi [83] published a protocol that provided strong fairness; however, it did not provide timely termination. Also in 1996, Zhou and Gollmann [90] proposed a protocol that provided both strong fairness and timely termination. The aim of this protocol was to reduce the work required by the TTP to a minimum. A detailed description of this protocol follows. This protocol is chosen for discussion as it is a commonly studied protocol and appears frequently in the literature.

2.2.3.1 Zhou Gollmann protocol with on-line TTP

The basic idea of the protocol is to split the message M into two parts, the commitment C and the key K where $M = dK(C)$. A will submit C to B and K to TTP. Both A and B then need to retrieve the confirmed K from the TTP as part of the evidence generation. The additional notation is specified in Table 2.2. Figure 2.4 shows a description of the protocol. Flags (fn) are used to describe the purpose of each message.

1	$A \rightarrow B$	:	f_1, B, L, C, EOO_C
2	$B \rightarrow A$	:	f_2, A, L, EOR_C
3	$A \rightarrow TTP$	:	f_3, B, L, K, sub_K
4	$B \leftarrow TTP$	:	f_4, A, B, L, K, con_K
5	$A \leftarrow TTP$	:	f_4, A, B, L, K, con_K

Figure 2.4: Zhou Gollmann protocol with on-line TTP

A description of the protocol follows:

1. A begins by sending C and EOO_C to B. The protocol remains fair as C is incomprehensible without K. B must verify EOO_C and store it before continuing.

2. B responds with the EOR_C. Fairness is maintained as EOR_C only proves receipt of the ciphered M which is useless without K. A must verify and store EOR_C before continuing.

3. A now submits K and sub_k to TTP. B could eavesdrop this step and thus retrieve M. However, as eventual message delivery is assumed, A will eventually be able to send K and sub_K to TTP in exchange for con_k. After receiving K and sub_K from A, TTP will generate con_K and then make the following publicly accessible: f_4, A, B, L, K, con_K.

4. B can now fetch K and con_K from TTP. B then obtains M by computing $M = dK(C)$ and saves con_K as evidence that K originated from A. As eventual delivery is assumed, B will always be able to fetch K and con_K from TTP.

5. Here A fetches con_K from the TTP and saves it as proof that K is available to B.

The protocol remains fair as A can only obtain EOR_C and con_K if and only if A sends C to B and K to TTP. Also, if A can access con_K, then so can B. If eventual message delivery is assumed, the protocol remains fair at each step. The TTP used in this protocol is inline and acts as a lightweight notary. The only requirements are that it must notarise keys on request and maintain a publicly accessible directory service. This makes for a very lightweight TTP for several reasons. Firstly, the TTP is only required to deal with keys and never receives the message (ciphered or plain); in most cases the keys will be smaller than the message. Secondly, it is the participants' responsibility to retrieve con_K. Thus the TTP does not need to keep resending con_K until it it is acknowledged. Finally, the message remains confidential to both A and B, providing a secure communication channel is available.

There are two possible disputes that may occur; (i) A repudiates the origin of M, (ii) B repudiates the receipt of M. In the first case, B must present (EOO_C, con_K, M, C, K, L) to the arbiter. In evaluating the evidence the arbiter must ensure:

1. $EOO_C = sig_A(f_1, B, L, C)$

2. $con_K = sig_{TTP}(f_6, A, B, L, K)$

3. $L = h(M, K)$

4. $M = dK(C)$

The first check proves that A sent the commitment C with label L to B. The second check proves that A submitted the message key K with label L to the TTP. The third check ensures that C and K are uniquely linked by L. The final check ensures that M is in fact the result of decrypting C with K. If all four checks succeed, the arbiter will conclude that M did in fact originate from A.

In the second case, A must present (EOR_C, con_K, M, C, K, L) to the arbiter. In evaluating the evidence the arbiter must ensure:

1. $EOR_C = sig_B(f_s, A, L, C)$

2. $con_K = sig_{TTP}(f_6, A, B, L, K)$

3. $L = h(M, K)$

4. $M = dK(C)$

The first check proves that B received commitment C with label L from A. The second check proves B received, or is able to receive the message key K with label L from the TTP. The third check proves that C and K are uniquely linked by L. The final check ensures that M is in fact the result of decrypting C with K. If all four checks succeed, the arbiter will conclude that B did in fact receive M.

For the protocol to be practical, timing information needs to be considered. Consider the following scenario. After sending step two, B is committed to the protocol and must continually poll TTP for con_K until it is available. con_K will not become available until after A sends step three to TTP. After some period of time, B could give up waiting for con_K, to be available, and abandon the protocol – in doing so deleting C and EOO_C. A can later complete the protocol and receive con_K from TTP. This would arise in an unfair situation were A has EOR_C and con_K and can thus successfully claim that B has received M; despite the fact that B is unable to retrieve M. This problem can be overcome by having deadline T_{sub} that states when A must submit step three to TTP. TTP will reject A's submission of step three after T_{sub} has passed, thus B can safely abort the protocol run after the deadline. Also, it can be useful to have the ability to prove what time the message was sent and received. To prevent these times from being forged, their generation should be undertaken by a trusted third party. Zhou modifies the protocol from [90] in [85] to add two timing

1.	$A \rightarrow B$	:	f_1, B, L, C, EOO_C
2.	$B \rightarrow A$	:	f_2, A, L, T_{sub}, EOR_C
3.	$A \rightarrow TTP$	:	f_3, B, L, $enc_{TTP}(K)$, T_{sub}, sub_K
4.	$B \leftarrow TTP$	:	f_4, A, B, L, K, T_{con}, con_K
5.	$A \leftarrow TTP$	:	f_4, A, B, L, K, T_{con}, con_K

Figure 2.5: Extended Zhou Gollmann protocol with on-line TTP

features. (i) T_{sub}, this is the deadline by which A must submit step three to TTP. (ii) T_{con} the time that TTP makes con_K available to A and B and thus also the time that B is able to acquire M.

EOO_C	$sig_A(f_1, B, L, C)$
EOR_C	$sig_B(f_2, A, L, C, T_{sub})$
sub_K	$sig_A(f_5, B, L, K, T_{sub})$
con_K	$sig_{TTP}(f_6, A, B, L, K, T_{con})$

Table 2.3: Evidence for extended Zhou Gollmann protocol with on-line TTP

Table 2.3 shows the evidence generated in the extended protocol shown in Figure 2.5. Step 1 of the protocol is the same as step 1 in Figure 2.4. In step 2, B includes T_{sub} which indicates the deadline by which A must have submitted step 3 to *TTP*. If A disagrees with this deadline, A can abort the protocol run. In step 3, A submits T_{sub} to *TTP*. If A were to decide to modify T_{sub} it would be to make B abort the protocol prematurely. This could be done by passing T'_{sub}, where $T'_{sub} > T_{sub}$. A would wait until some time T where $T_{sub} < T < T'_{sub}$, before submitting step 3. B would abort the protocol as T_{sub} has passed, but A and TTP will continue as T'_{sub} has not passed. However, in this scenario, $T_{con} > T_{sub}$ which is not possible. T_{sub} is irrefutable as it is present in EOR_C and will need to be presented to the arbiter. Thus A must submit a correct T_{sub} to *TTP* for the protocol to be successful. Finally, TTP will generate a time T_{con} that con_k was available to A and B to retrieve in steps 4 and 5.

2.2.4 Off-line TTP-based protocols

Protocols employing an off-line TTP are usually seen as preferable as they place the least burden on the TTP. These protocols are often called *optimistic* protocols as the TTP is only involved if one or more participants misbehaves or in the case of a communication failure. Several such protocols are present in the literature [4, 5, 47, 53, 90, 89, 87, 88]. In 2003 Gurgens et al. [34, 35] conducted an intensive survey of the security of these protocols. According to their findings the claimed levels of fairness may not be guaranteed under some specific attacks. Protocols [90, 89] allow an attack due to their chosen protocol label composition. A modification to these protocol labels is proposed

that prevents this attack. The protocol in [53] contains a flaw that allows a dishonest receiver to access the message without issuing a receipt to the sender, thus breaking fairness. Furthermore, Gurgens et al also found flaws in [88, 87]. As a result, of discovering so many flawed off-line TTP based non-repudiation protocols, Gurgens et al. proposed a new protocol that avoided these security shortcomings [35].

There follows a description of the Zhou Gollmann off-line TTP based protocol [88]. This protocol is chosen for discussion as it is a commonly studied protocol and appears frequently in the literature.

2.2.4.1 Zhou Gollmann protocol with off-line TTP

In [89] a fair non-repudiation protocol with an offline TTP was proposed. This protocol was designed for transacting parties who only want to involve a TTP as a last recourse. As the TTP is offline, it is only involved in abnormal protocol runs. This protocol had a similar problem to [71] in that at a particular step, B must wait indefinitely for the protocol to complete. In [88] a *timely terminable* fair non-repudiation protocol utilising an offline TTP, was proposed. Zhou [85] defines *timely terminable* as follows:

> A fair non-repudiation protocol is *timely terminable* if either transacting party can unilaterally bring a transaction to completion without losing fairness.

In the protocol, M is split into two parts, C and K, where $M = dK(C)$. When required, the TTP acts as a lightweight notary and operates only on K. Typically K will be much smaller than M, thus reducing the load on the TTP.

Notation	Description
$C = eK(M)$	Commitment (cipher text) for message M.
$L = H(M, K)$	A unique label linking C and K.
$f_i(i = 1, 2, ...)$	Flags indicating the intended purpose of a signed message
$EOO_C = sig_A(f_1, B, L, C)$	Evidence of origin of C.
$EOR_C = sig_B(f_2, A, L, H(C), enc_{TTP}(K))$	Evidence of receipt of C.
$EOO_K = sig_A(f_3, B, L, K)$	Evidence of origin of K
$EOR_K = sig_B(f_4, A, L, K)$	Evidence of receipt of K
$sub_K = sig_A(f_5, B, L, K, H(C))$	Authenticator of K provided by A.
$con_K = sig_{TTP}(f_6, A, B, L, K)$	Evidence of confirmation of K issued by TTP.
$abort = sig_{TTP}(f_8, A, B, L)$	Evidence of abortion of a transaction issued by the TTP

Table 2.4: Zhou Gollmann protocol with off-line TTP - additional notation

Figure 2.4 shows additional notation required to describe this protocol. The protocol is split into three sub-protocols, *exchange*, *abort* and *resolve*. The protocol assumes that the communication channels between A and B are confidential if the two parties wish to exchange messages secretly. Also it is assumed that the communication channel between A and B is never permanently broken.

1	$A \rightarrow B$	:	$f_1, f_5, B, L, C, TTP, enc_{TTP}(K), EOO_C, sub_K$
			IF B gives up **THEN** quit **ELSE**
2	$B \rightarrow A$	:	f_2, A, L, EOR_C
			IF A gives up THEN *abort* **ELSE**
3	$A \rightarrow B$	:	f_3, B, L, K, EOO_K
			IF B gives up **THEN** *resolve* **ELSE**
4	$B \rightarrow A$	:	f_4, A, L, EOR_K
			IF A gives up **THEN** *resolve*

Figure 2.6: Zhou Gollmann protocol with off-line TTP - exchange protocol

Figure 2.6 shows the *exchange* sub-protocol. This protocol begins with A sending the commitment C and EOO_C to B. A also sends $eP_{TTP}(K)$ and sub_K to B. This will allow B to complete the protocol in a timely way by invoking the resolve subprotocol. Before sending EOR_C to A, B can quit the transaction without losing fairness, otherwise B must invoke the *resolve* subproto-

col. Similarly, A may quite the transaction by invoking the *abort* subprotocol, as long as K and EOO_K have not been sent to B. Otherwise, A must invoke the *resolve* subprotocol to complete the transaction.

On successful completion of the *exchange* subprotocol, B will have C and K and can thus obtain M. B will also have EOO_C and EOO_K and can thus prove the origin of M. On successful completion of the *exchange* subprotocol, A can prove receipt of C and K using EOR_C and EOR_K and can thus prove B's receipt of M.

1	$A \rightarrow TTP$	:	$f_7, B, L, sig_A(f_7, B, L)$
			IF resolved **THEN**
2	$A \leftarrow TTP$	:	$f_2, f_6, A, B, L, K, con_K, EOR_C$
			ELSE
3	$A \leftarrow TTP$	:	$f_8, A, B, L, abort$

Figure 2.7: Zhou Gollmann protocol with off-line TTP - abort sub-protocol

Figure 2.7 shows the *abort* subprotocol. Only A may invoke this protocol. On the protocol's invocation, TTP will check to see if the protocol run has previously been resolved. If the protocol run has been resolved TTP will do nothing. This is because the required information to complete the transaction was placed in a publicly accessible place when the *resolve* subprotocol was executed. If the protocol run has not been resolved, TTP will generate the *abort* token and place it in a publicly accessible place for A to retrieve. TTP will then set the status of the protocol run to *aborted*. A may then go to the publicly accessible directory to establish the outcome of the protocol run and to acquire the relevant evidence to prove its outcome.

1	$U \rightarrow TTP$	:	$f_2, f_5, A, B, L, enc_{TTP}(K), H(C), sub_K, EOR_C$
			IF aborted **THEN**
2	$U \leftarrow TTP$	:	$f_8, A, B, L, abort$
			ELSE
3	$U \leftarrow TTP$	:	$f_2, f_6, A, B, L, K, con_K, EOR_C$

Figure 2.8: Zhou Gollmann protocol with off-line TTP - resolve subprotocol

Figure 2.8 shows the *resolve* subprotocol. This protocol may be invoked by either A or B. In Figure 2.8 U refers to the initiating party of the *resolve* subprotocol. On the protocols invocation, TTP will check to see if the protocol run has previously been aborted. If the protocol run has been aborted, TTP will do nothing. This is because the required information to complete the transaction was placed in a publicly accessible place when the *abort* subprotocol was executed. If the protocol

run has not been *aborted*, TTP will undertake the following:

1. Use EOR_C to verify that B received $enc_{TTP}(K)$.

2. Decrypt $enc_{TTP}(K)$ to verify with sub_K that K was submitted by A.

3. Check that EOR_C is consistent with sub_K in terms of L and *H(C)*.

4. Generate evidence con_K

5. Place (f_2, f_6, A, B, L, K, con_K, EOR_C) in a publicly accessible place

6. set the status of the protocol run to resolved.

U may then go to the publicly accessible directory to establish the outcome of the protocol run and to acquire the relevant evidence to prove its outcome.

2.2.5 Transparent off-line TTP based protocols

When a transparent TTP is employed, the evidence generated from a protocol run is identical, regardless of whether the TTP was involved or not. This scheme has the benefit that it is impossible to tell whether the protocol needed the TTP's involvement for successful completion. This attribute is seen as favourable because the presence of TTP generated evidence could suggest the misbehaviour of a participant were in fact the TTP could have been involved due to communication failures.

In [57], Micali proposed a protocol for certified e-mail employing a transparent TTP. However, as stated in [81], this protocol has three weaknesses. (i) The protocol does not provide non-repudiation of origin (NRO) of the message. (ii) The protocol is inefficient for large messages as the whole message is encrypted using an asymmetric encryption algorithm. Micali states that the message could be encrypted under a symmetric key which is in turn encrypted using asymmetric encryption. However, it was found in [34, 35] that the conversion is not as trivial as Micali had hoped. (iii) There exists a flaw in the protocol that allows A to trick B into believing that a different TTP was used. To ensure fairness, B would need to contact each TTP individually to enquire if they were the actual TTP used. This represents a viable attack, because it would be infeasible to the expect B to have a list of every single TTP. This flaw is also present in a certified email protocol (utilising a transparent TTP) proposed by Imamoto in [40]. Another certified email protocol utilising a transparent TTP was proposed in 2002 by G. Ateniese [6]. This protocol has similar weaknesses to [57] in that it does not provide NRO and it utilises a rather inefficient signature scheme [7, 9]. In 2005 G. Wang proposed a protocol utilising a transparent TTP that addressed the problems found in the previous protocols. This protocol improves on the existing transparent TTP based protocols and also on other optimistic non-repudiation protocols. There follows a detailed description of the G. Wang [81] protocol. This protocol was chosen as it appears to offer the greatest feature set.

2.2.5.1 G Wang protocol

The following notation is used to describe the protocol and associated sub-protocols (in addition to that in Section 1.1.4).

Notation	Description
f_K, f_{EOO}, f_{EOR}, f_{AT}, f_{Rec}	publicly known *unique* flags that indicate distinct purposes of different protocol messages.
$L = h(A, B, TTP, h(C), h(K))$	unique label to identify a protocol run
$C = eK(msg)$	msg encrypted under K
$EK = enc_{TTP}(f_K, L, rn, K)$	secret key K, ciphered with (f_K, L, rn)
$EOO = sig_A(f_{EOO}, L, E, K)$	evidence of origin, showing that A sent a message msg to B, if both EOO and EK are valid
$EOR = sig_B(f_{EOR}, L, EK)$	evidence of receipt, showing that B received a message msg from A, if both EOR and EK are valid
$AT = sig_A(f_{AT}, L)$	abort token issued by A to cancel the protocol run indexed by label L
$Rec = sig_B(f_{Rec}, L, EK)$	recovery request from B to resolve the protocol run indexed by label L

Table 2.5: G Wang protocol - additional notation

The basic idea of the protocol is as follows. A begins by sending B (*C, EK, EOO*). The non-repudiation of origin (NRO) is defined by the concatenation of *EOO* with (*K,rn*). *EOO* can only be verified if *EK* can be verified and *EK* can only be verified with correct values for K and *rn*. B will only receive (*K,rn*) if a correct *EOR* is submitted to A or TTP.

The protocol is composed of a main *exchange* protocol and two exception handling sub-protocols — namely *abort* and *resolve*. The abort protocol allows A to cancel the protocol if (i) B does not respond, (ii) B does not respond correctly or in a timely way or (iii) The communication channel is interrupted. Similarly, the resolve protocol protects B against foul play from A and communication failures.

e1	$A \rightarrow B$	:	A, B, TTP, C, $h(K)$, EK, EOO
			IF B gives up **THEN** quit
e2	$B \rightarrow A$	:	EOR
			IF A gives up **THEN** run abort protocol
e3	$A \rightarrow B$	:	K, rn

Figure 2.9: G Wang - exchange protocol

Figure 2.9 shows the exchange protocol. The protocol begins by A selecting a session key K and a random number *rn*. The following are then computed: $C = E_K(M)$, *h(k)*, *h(C)*, $L = (A, B, TTP, h(C), h(K))$, $EK = enc_{TTP}(f_K, L, K, rn)$ and $EOO = sig_A(f_{EOO}, L, EK)$. A then sends protocol message e1 to B.

On receiving this message, B first checks the identities (*A,B,TTP*) and then stores L. *EOO* is then checked to ensure it represents a valid signature on (f_{EOO}, L, EK) by A. If any of these checks fail or B choses not to interact with A, B may simply quit the protocol without any liability. B can then confidently respond with the *EOR*. The EOR is constructed from (f_{EOR}, L, EK). If *EK* is correct, B will receive *(K,rn)* from either A or *TTP*. If *EK* is incorrect then so is *EOR* and A does not have a valid evidence of receipt. On receiving *EOR*, A checks that it represents B's signature on (f_{EOR}, L, EK). If this is true, A has full non-repudiation of receipt evidence, *NRR* = (*A,B,TTP,msg,K,rn,EOR*). As a result, A reveals the values (*K,rn*) to B. If A did not receive *EOR* in a timely way or *EOR* was invalid, A can run the abort sub-protocol to cancel the protocol run. If confidentiality of *msg* is required, (*K,rn*) should be encrypted with B's public key. This will prevent an eavesdropper from discovering the session key used to encrypt *msg* in the first protocol message.

On receiving (*K,rn*), B can check $EK \equiv enc_{TTP}(f_K, L, K)$. If this is true, B has a complete non-repudiation of origin *NRO = (A,B,TTP,msg,K,rn,EOO)*. However, if (*K,rn*) is incorrect or not received, B can complete the protocol by running the resolve sub-protocol.

a1	$A \rightarrow TTP$	:	A, B, TTP, $h(C)$, $h(K)$, $enc_{TTP}(AT)$
			IF state=recovered **THEN**
a2.1	$TTP \rightarrow A$	:	EOR
			IF state=aborted **THEN**
a2.2	$TTP \rightarrow A$	:	$L, confirm$
			ELSE (set state=aborted)
a2.3	$TTP \rightarrow A$	:	L, $confirm$
a2.4	$TTP \rightarrow B$	:	A, B, TTP, $h(C)$, $h(K)$, AT

Figure 2.10: G Wang - abort protocol

Figure 2.10 shows the abort sub-protocol. This protocol is executed by A if EOR is not received in a timely way or correctly from B. The protocol has the effect of canceling the protocol. A begins the protocol by computing *h(C)*, *h(K)*, $AT = sig_A(f_{AT}, L)$, and $enc_{TTP}(AT)$ and then sending protocol message a1 to TTP. On receiving the abort request, TTP decrypts $enc_{TTP}(AT)$ and then checks the resulting plain text represents A's signature on (f_{AT}, L). If this is true, the TTP checks in its database for a protocol run associated with L. If a run is found, TTP knows that it must have been aborted or resolved already. In which case the TTP retrieves the related items and forwards them to A. Otherwise, TTP records the following in its database (*L, state=aborted, AT, confirm*) and then informs A and B that the protocol has been aborted by sending messages a2.3 and a2.4. The composition of the *confirm* token is not discussed in [81]. Care must be taken when chosing the nature of this token in order to prevent a possible flaw. This is discussed in more detail at the end of this section.

Figure 2.11

r1	$B \rightarrow TTP$	:	*A*, *B*, TTP, $h(C)$, $h(K)$, *EOO*, *EOR*, *Rec*
			IF state=aborted **THEN**
r2.1	$TTP \rightarrow AB$	:	*AT*
			IF state=recovered **THEN**
r2.2	$TTP \rightarrow AB$	:	*K*, *rn*
			ELSE (set state=recovered)
r2.3	$TTP \rightarrow A$	:	*EOR*
r2.4	$TTP \rightarrow B$	:	*K*, *rn*

Figure 2.11: G Wang - resolve protocol

shows the resolve sub-protocol. This protocol is executed if B sends *EOR* to A but B does not receive *(K,rn)* correctly or in a timely way from A. The protocol allows B to get correct values for (*K,rn*). When TTP receives message r1, it first validates each of the signatures. If all the signatures are valid, a record is looked up in the database for a protocol run with label L. If an associated run is found, the protocol has already been aborted or resolved, in which case TTP simply forwards the related items to B. Otherwise, TTP checks the validity of *EK*. *EK* is valid if and only if, (i) $EK \equiv enc_{TTP}(f_K, L, K, rn)$ and (ii) *h(K)* is correct. If *EK* is found to be valid, TTP records (*L, state=recovered,Rec,EOR,(K,rn)*) in the database, and sends *EOR* to A and (*k,rn*) to B. Again, confidentiality of *msg* can be maintained by encrypting (*K,rn*) with B's public key.

As stated earlier, the composition of the confirm token must be chosen carefully. It should take the form of TTP's signature over a suitable flag and the label L. For example, $confirm = sig_{TTP}(f_{confirm}, L)$. This prevents an attack in which B can trick A into believing that the protocol

has been aborted when in fact the protocol has been resolved. The steps that form this attack follow:

1. *B* receives a correct message e1 from *A*.
2. *B* does not respond to *A* with message e2, but instead waits until *A* initiates the abort sub-protocol.
3. *B* intercepts and delays message a1 from *A* for a finite amount of time. *B* can only delay message a1's arrival at *TTP* as the communication channel between *A* and *TTP* is assumed to be resilient. That being that all messages inserted into the channel will arrive at the recipient after a finite but unknown period of time.
4. *B* spoofs message a2 and passes to *A*. *B* can only do this if *confirm* is composed in such a way that it can be forged by *B*. For example, if it was simply a string as could be inferred from [81].
5. *A* now thinks the protocol has aborted and so discards all information related to this run.
6. *B* then initiates the recovery protocol. This will succeed as *TTP* hasn't received *A*'s request to abort yet.
7. *B* then receives (K,rn) from *TTP* and thus has complete NRO.
8. *A* receives *EOR* for a protocol run that it is no longer aware of and, as a result, discards *EOR* as an unexpected message.
9. *B* will now stop delaying *A*'s abort request a1 and *TTP* will then receive a1.
10. In response to receiving a1, *TTP* will note that this protocol cannot be aborted as it has already been resolved. As a result, *TTP* will send *EOR* to A, which, again, *A* will disregard as an unexpected message.

This attack raises an unfair situation in which *B* has full *NRO* evidence and *A* has partial (only *EOR*) or no *NRR* evidence. This attack is only feasible if the *confirm* token is interpreted as just a string (which can be reasonably inferred from the literature [81]) and can be fixed by composing *confirm* from $sig_{TTP}(f_{confirm}, L)$.

2.2.6 Non-repudiation protocol comparisons

Table 2.6 is taken (and augmented) from [81], and shows a comparison of the deterministic TTP based protocols discussed in this thesis.

The first column states what type of TTP is used (TTP types are discussed in Section 1.1.3.3). The next two columns are concerned with timeliness and fairness. All of the protocols (apart from

Protocol	TTP	Timeliness	Fairness	Generic	# of Mess.	# of Oper.
CS [16]	Inline	No	Strong*	Yes	7	20
RCS [?]	Inline	Yes	Strong	Yes	6	14
ZS [84]	Online	No	Strong	Yes	5	13
ZG [90]	Online	Weak	Strong*	Yes	5	9
ZG [87]	Offline	Weak	Strong*	Yes	4	8
ZDB [87, 88]	Offline	Yes	Strong*	Yes	4	12
KMZ[47]	Offline	Yes	Strong*	Yes	4	12
GRV[35]	Offline	Yes	Strong	Yes	4	10
MK[53]	Offline+Transp	Yes	True*	No	4	12
GW[81]	Offline+Transp	Yes	True	Yes	3	6

Table 2.6: Comparison of TTP based non-repudiation protocols

Protocol	Reference
CS [16]	The Coffey Saihda protocol was described in Section 2.2.2.1
RCS[?]	The Robinson Cook Shrivastava protocol is presented later in this thesis (Section 3.2)
ZG [90]	The Zhou Gollmann protocol with an on-line was described in Section 2.2.3.1
ZDB [87, 88]	The Zhou Gollmann protocol with an off-line was described in Section 2.2.4.1
GW[81]	The G Wang protocol was described in Section 2.2.5.1

Table 2.7: References to protocol descriptions in this thesis

CS and ZH) provide mechanisms for unilateral termination. However, the ZG [90, 87] schemes only provide weak timeliness based on deadlines. The timeliness is said to be weak as it requires a participant to wait for some finite period of time (i.e. until the deadline is reached). Each of the protocols claim to be fair (types of fairness are discussed in Section 2.2); however, flaws have been found in a number of them. These protocols are marked with an asterisk. Each of these flaws have a potential fix; however, there is no evidence to suggest that the fixed versions are correct. It should be noted at the time of writing the GW [81] protocol is very new and, as stated by the author, has not yet been formally verified.

When comparing efficiency, the cost of communication (# of messages) and computation (# of operations) in the normal case are considered. It is assumed that abort and resolve protocols will happen very infrequently and, as such, their performance is of little interest. When comparing the computation costs, only asymmetric cryptography operations are considered (i.e. encryption with a public key and signing with a private key). Symmetric cryptography operations are ignored as they require much less computation. From the table, it can be seen that the G. Wang [81] protocol offers the best performance in terms of number of messages transfered and number of operations.

2.3 Non-repudiation middleware and applications

Various implementations exist for achieving non-repudiation. A discussion of each implementation follows.

2.3.1 CORBA Non-repudiation Service

The CORBA security service [32] specification describes the following services:

1. Identification and authentication
2. Authorisation and access control
3. Security auditing
4. Security of communication
5. Non-repudiation

The CORBA non-repudiation service is based on the ISO standard in [41]. The specification describes various services that should be available to an application requiring non-repudiation. These services provide: evidence generation and verification; evidence storage and retrieval; and a delivery authority. However, the orchestration and utilisation of these services is up to the application and there are no systematic mechanisms for transparently invoking non-repudiation services.

Early work by Wichert et al. [82] define a middleware mechanism for providing non-repudiable service invocation for the CORBA platform. The middleware makes use of the CORBA non-repudiation service for evidence generation, verification, storage and retrieval. The generated evidence consists of a signed copy of the invocation request and response. The implementation uses filters within the ORB-core transparently to intercept method invocations. Using filters it is possible to intercept method calls at four points:

- As the request leaves the client (Pre-request). Here the request is signed and the signature is appended. This forms the non-repudiation of origin of request evidence.
- As the request reaches the server (pre-dispatch). Here the non-repudiation of origin of request evidence is verified. If the verification fails, or the evidence does not exist, the invocation message is dropped.
- Before the response is dispatched to the client (post-dispatch). Here the response is signed and the signature is appended. This forms the non-repudiation of origin of response evidence.
- Before the client receives the response (post-request). Here the non-repudiation of origin of response evidence is verified. If the verification fails, or the evidence does not exist, the response message is dropped.

In CORBA, remote service invocation is undertaken over the IIOP protocol. IIOP is a binary protocol intended to be machine readable. Thus the transmitted request and response messages are not easily human readable and, as such, would not make good non-repudiation evidence. To solve this problem, the non-repudiation evidence is generated on an XML encoded version of the request and response. This has the benefit of being CORBA independent and also human readable.

This middleware only supports non-repudiation of origin evidence generation and, as such, the client cannot prove the server received the invocation request. Similarly, the server cannot prove the client received the invocation response.

2.3.2 TY*SecureWS

TY*SecureWS [50] is a project undertaken by Tong Yang Systems Corparation and consists of a middleware solution for securing inbound and outbound Web service messages. Using the middleware it is possible to specify which security aspects should be applied to outbound messages and similarly, which security aspects are required for inbound messages. Each of the security aspects defined in Section 1.1.1 are supported by the middleware. Usage of XKMS (described in Section 5.1.8) for certificate management is also supported.

The middleware supports the generation and verification of non-repudiation of origin evidence. Non-repudiation of origin is facilitated by signing outbound messages with the sender's private key. It can be specified that all inbound messages must be signed. If this is the case, the non-repudiation handler will check for a signature on the message. If the digital signature is invalid or non-existent, the message will be dropped.

The recipient of the message could choose to send back a signed acknowledgement message to form the non-repudiation of receipt. However, this would have to be done at the application level and cannot be enforced by the message's originator. Thus TY*SecureWS supports non-repudiation, but not *fair* non-repudiation.

2.3.3 DataPower XML Security Gateway and Verisign Trust Gateway

The DataPower XML Security Gateway [23] and the Verisign Trust Gateway [80] are both types of XML firewalls. They consist of a hardware device for intercepting inbound and outbound network traffic at the network boundary. Their purpose is to apply and verify WS-Security tokens embedded in SOAP messages. The devices are capable of applying policies to different types of traffic. For example, a particular policy could state that a particular Web service is only willing to receive requests if they have been encrypted and signed. The firewall will drop messages destined to this service that do not meet the requirements described in the policy.

The DataPower XML Security Gateway and the Verisign Trust Gateway both provide similar

functionality to TY*SecureWS [50] in that they can be used to sign outgoing messages and verify signatures on incoming messages. Like TY*SecureWS, there is no support for mandatory proof of receipt.

2.3.4 FIDES

The FIDES system [60] provides services, including TTP services, and an associated application for fair exchange of documents. Application clients submit documents to the FIDES system for fair exchange with partners who also have a FIDES client for verifying and receipting documents received. In effect, FIDES offers a standalone service for fair exchange. The system also supports pluggable protocols, so new fair exchange protocols can be inserted into the system with the minimum of disruption. This system is not implemented in middleware and, as a result, FIDES cannot easily provide non-repudiation to existing applications.

2.3.5 B2B Objects

In 2002, colleagues published a prototype for non-repudiable information sharing [19]. This prototype was called *B2B Objects* and allowed multiple organisations to have controlled access to a shared information state. The abstraction of a single shared state is presented to the organisation. However, in reality, each organisation would have their own copy. Each copy of the state is kept synchronised by the middleware. Updates to the shared state would only occur if all other organisations agreed. A two phase protocol was used to update the state. Firstly the updating party would propose a new state to every other participant. These participants would then decide whether they agree with the proposed new state and return the result. The second phase is used to communicate the result of the update request to each participant. If every participant agreed with the new state, the update would go ahead. In which case, each participant changes their copy of the state to that of the new proposed state. Thus each copy of the state remains synchronised. Requests to update the state and votes on the validity of the update are non-repudiable. Fair non-repudiation was deemed unnecessary as the organisations involved rely on future business. Thus the reputation of the organisations was seen as more valuable than any potential gain. B2B objects was extended in 2003 to support transactional updates to the shared state [18]. Here, several pieces of shared information could be updated in the context of an ACID transaction whilst still providing the non-repudiation guarantees of the earlier work. In 2004 we developed a component based implementation [?, ?] of B2B Objects. Here, the shared state was represented as an EJB Entity Bean and, as such, each organisation involved would have their own copy. Here the two phase protocol was used to regulate updates to the Entity Bean and to ensure synchronisation. The work also presented an implementation for non-repudiable invocation of Session Bean EJBs. This was similar to the work by Wichert et

al. [82]; however, non-repudiation of receipt was also provided. Again, fair non-repudiation was not provided. Furthermore, the location at which non-repudiation evidence was created was not suitable. The evidence was created on entry to the application server hosting the EJBs (both for information sharing and service invocation), but typically an organisation would not expose EJBs directly to other organisations. It would be more likely that a web service would present a set of coarse grained interactions externally which, in turn, would make calls to multiple EJBs. Thus, it it would be more sensible to provide non-repudiation at the organisation boundary (the Web service). As a result, the actual messages sent between organisations are what is signed (and thus comprise the non-repudiation evidence), rather than messages created by the implementation. A Web services implementation was published in 2005 [?, ?] which supported fair non-repudiation. This forms the basis of this thesis and will be discussed in chapters 3 and 5.

2.4 Automatic security protocol implementation

There have been various publications regarding the automatic implementation of security protocols. However, none (that we are aware of) purport to solve the problem for non-repudiation protocols. Despite this fact, it is still useful to consider their approaches as it may prove insightful when proposing an aproach for non-repudiation protocols.

Researchers at the University of Berkeley [66, 65, 75] have developed a process for automatically generating, verifying and implementing security protocols (not including non-repudiation protocols). The process begins with a specification of what the protocol needs to achieve and the maximum cost (such as number of messages or encryption operations) of execution. The automatic generation technique then uses a brute force technique to generate all possible permutations of protocols that fit within the cost metrics allowed. Pruning techniques are then used to remove the majority of these protocols. A smaller set of candidate protocols is then obtained. Starting with the *cheapest* protocol each one is modeled and then verified. This process of verification is automated by a tool called Athena [74]. If a suitable and correct protocol is found, the implementation phase creates a Java implementation. Unfortunately, it is unclear how the Java implementation is produced and what it comprises as there is no published work on this matter. It is also commented on by the authors that they are unsure that the process of automatically generating security protocols will scale up to protocols requiring more than three steps. As a result, it looks doubtful that the process could be used to create a fair non-repudiation protocol with four or five steps. Furthermore, there is no mention of support for sub-protocols — these are essential for providing timeliness in fair non-repudiation protocols.

Several pieces of work [25, 67, 79, 56], take a formal specification of a security protocol and then automatically generate Java code. This Java code uses the formal specification of the protocol

automatically to generate and verify protocol messages. The protocol is then implemented by creating a stub for each participant that sends, receives and verifies protocol messages in the right order. If the protocol completes, it is assumed that the intentions of the protocol (authenticating both parties, for example) have succeeded. Unfortunately, this aproach is not appropriate for non-repudiation protocols. Once the protocol run has completed successfully, it must remain possible to prove what has happened to an external party. This requires a large amount of infrastructure, such as logging and time stamping services. None of these implementations provide such a framework.

Work by Abdullah and Menascé [2] proposes a mechanism for automating the protocol implementation part of a specific protocol production process. This process is slightly different to the traditional protocol development cycle described in Section 1.2. Here the protocol designer produces an FSM and RFC (request for comments — a specification document that describes the protocol). Protocol developers then read and interpret these documents and then implement the protocol. Abdullah and Menascé automate the implementation phase by taking the FSM and represent it in an XML document. This XML document also contains descriptions of the protocol messages. The protocol implementation is created by using XSLT (a mechanism for transforming an XML document into some other document) to transform the XML FSM description into Java class files. This protocol production process does not allow for protocol verification. As a result, the input protocol could be incorrect. Furthermore, the XML language for describing the FSM is rather complicated and thus error prone. As with the other work in this area, there is no support for non-repudiation.

2.5 Business interactions

This sections describes mechanisms for representing and regulating business interactions.

2.5.1 RosettaNet PIPs

The RosettaNet Consortium [70] have developed a suite of business interaction patterns, called Partner Interface Protocols (PIPs) in RosettaNet terminology. Each PIP is designed for a very specific purpose, such as sending an invoice or purchase request. Each PIP forms a coarse grained interaction that occurs between two business partners. Each PIP is designed to achieve some very specific task. Applications can be constructed by composing multiple PIPs to achieve some higher level task.

There are currently 111 PIPs defined, each falling in to one of eight categories (or clusters in RosettaNet terminology). The messages involved in a PIP can be classified into two broad categories: business action messages and business signal messages.

- **Business actions** are messages with content that is of a business nature, such as a purchase order or a request for quote.

- **Business signals** acknowledge receipt of *business actions*. A *business signal* can be positive or negative. A positive *business signal* states that the corresponding *business action* was received and found to be a structurally and syntactically valid RosettaNet *business action* message. A negative *business signal* states that there was some problem with the *business action*, or an exception has occurred, which is preventing process of the *business action*.

Examples of two specific PIPs are given in Section 3.1.1.

Work by Khalaf [43] proposes a scheme for transferring some of the implementation burden to the PIP designer. As a result, end users of the PIP have a simplified process for implementation. The reduced implementation requirements may also encourage some smaller companies (that were previously unable) to use PIPs. In order to achieve this goal, the PIP designer produces extra documents that describe how the PIP should operate in a Web services environment. These documents are produced using the Business Process Execution Language for Web services (BPEL). BPEL is an XML workflow-based composition language for Web services. In order to create an executable PIP, the PIP user must follow a set of completion rules provided by the PIP designer. These completion rules are required as the BPEL documents are abstract and thus not tied to a particular execution framework — undertaking the completion steps provides an execution framework specific implementation.

IBM's WebSphere application server [39] and BEA's WebLogic application server [11] are examples of PIP execution frameworks. They both support non-repudiation, but not *fair* non-repudiation. Non-repudiation is achieved by signing outgoing messages and verifying incoming message — a process very similar to that used in TY*SecureWS (discussed in Section 2.3.2).

2.5.2 Contracts

When undertaking inter-organisational interactions it is important that these interactions are strictly controlled. In the paper-based world, business interactions are conducted under the control of contracts that are signed by each organisation. When moving towards electronic mechanisms for doing business, it is important that the interactions continue to be conducted under control of contracts. Thus business interactions requiring non-repudiation are likely to execute some interaction agreed upon and enforced by means of contracts. Such contracts will contain information regarding what interactions are legal and the sequence in which they may occur. Other information related more closely to non-repudiation would also be present. For example (if required) a particular trusted third party must be agreed upon. Furthermore, each participant must also agree the TTP's jurisdiction in case of dispute. Other non-repudiation clauses would include information such as how long the evidence remains valid. This would prevent the requirement to store evidence indefinitely.

Work by colleagues [58, 73] addresses the issue of taking a conventional paper-based contract and

converting into an electronic equivalent (X-Contract in their terminology). This process consists of taking the paper contract and representing it in some mathematically precise notation such that the result can be subjected to rigorous analysis. This analysis frees the contract of ambiguities that the original human oriented text may contain. Furthermore, an X-Contract has the benefit of being computer interpretable and, thus, a suitable run time infrastructure may be used to monitor the actual interaction described within the contract. The notation used to represent a contract is that of finite state machines (FSMs). It is this FSM that is used to ensure that the interaction obeys the contract. Essentially, any message exchange is validated by ensuring it enacts a valid transition in the X-contract (which is an FSM).

One such suitable infrastructure for exploiting X-Contracts for monitoring an interaction is B2B Objects (discussed in Section 2.3.5). Here an X-Contract is consulted when a state change request is received. This ensures that the state change is allowed according to the contract. Similarly, a service invocation can also be validated by the X-Contract. This ensures that invoking that particular service with the passed parameters does not break any clause of the contract.

2.6 Discussion

To achieve non-repudiation, it must be possible for either party involved to prove to an external party the other party's participation. In obtaining this, several issues must be addressed. It is important to be able to ascertain the time at which the non-repudiation evidence was generated. This prevents the signatory from revoking their certificate and claiming the evidence was generated after revocation. Several schemes have been proposed to prevent denial of responsibility of a signature. These schemes vary in accuracy and level of reliance on a trusted third party. Unfortunately, for high value interactions, the only way to prevent denial of signature responsibility is by using a trusted timestamping service. This method has the added benefit of providing the exact time at which the signature was notarised.

In order to enable a participant to prove another participant's participation, a full set of evidence must be available. If, for some reason, this evidence is unavailable, participation cannot be proved. A participant may not have access to evidence for several reasons. It may be lost due to failure, be buried among a large amount of other evidence or not even have been received. To prevent loss of evidence, participants must be fault tolerant and evidence must be stored in some failure resistant medium. Evidence may need to be kept for many years, by which time there may be a large amount of evidence to manage. Thus it is important that the evidence is stored in such a way that a particular piece can easily be retrieved. To ensure evidence is received, a fair non-repudiation protocol is operated. This ensures either all parties receive the expected evidence or none do.

Several fair non-repudiation protocols have been developed. Early protocols required a large

dependency on a TTP. As new protocols were developed, the reliance on a TTP was reduced until the TTP was only required for dispute resolution (an offline TTP). Some of the more advanced protocols ensured that the evidence produced was identical, regardless of whether the TTP was involved. However, the protocols utilising an offline TTP did not consider the issue of signature time stamping. As stated earlier, it is necessary for a TTP to time stamp signatures to prevent the signatory denying responsibility. As a result, communication with a TTP is required each time a signature is created. Therefore, even the protocols claiming to only need a TTP for dispute resolution still require a TTP for every protocol run.

Various frameworks for achieving non-repudiation are available, only one of which supports *fair* non-repudiation — the FIDES project. Although FIDES does support *fair* non-repudiation, it is only provided to a single previously developed application. A more desirable implementation would be developed in middleware, thus allowing arbitrary applications to benefit from the fair non-repudiation guarantees.

To ensure a protocol is correct, it is normal to undertake some validation process. This adds confidence that the protocol is free of flaws. However, implementing the verified protocol is an error-prone task which can greatly reduce the confidence attained from validation. This problem is common to all security protocols. As a result, several groups have proposed schemes for automating the implementation process. However, none of these processes support non-repudiation protocols. This is largely due to the large amount of associated infrastructure and the protocols' complexity. Other security protocols consist of one protocol whereas the more practical non-repudiation protocols consist of one main protocol and one or two exception handling sub-protocols.

Fair non-repudiation is rather expensive to achieve due to the heavy reliance on trusted third parties, evidence storage and the relatively complex protocols (compared to other security protocols). As a result, it would be expected that fair non-repudiation would only be used for high value business to business (B2B) interactions. Work by the Rosettanet Consortium presents a collection of interaction patterns (PIPs) designed for business to business transactions. PIPs can be composed in order to produce B2B applications. However, these PIPs need to be executed in a suitable framework. IBM and BEA produce two such frameworks. Although they both support non-repudiation, they do not support *fair* non-repudiation. Furthermore, high value transactions tend to undertake actions described in a contract. Work by colleagues show how a contract can be translated into a computer interpretable format. This allows the B2B interaction to be monitored to ensure it does not break contract.

From surveying current literature it can be seen that achieving non-repudiation is a demanding task. One which no implementation has satisfactorily achieved. Furthermore, the automatic implementation of non-repudiation protocols is a neglected area. One which has not been covered by the collection of research into automatic security protocol implementation.

Chapter 3

Non-Repudiation Protocol Hierarchy

This chapter presents a hierarchy of non-repudiation protocols. This hierarchy provides a multitude of messaging patterns for the application programmer. Examples of such messaging patterns are request/response and asynchronous messaging. Furthermore, the hierarchy allows configuration of the level of non-repudiation provided and also what messages are validated. Validation is intended to allow the recipient of the message to undertake some application specific checks on the received message. Results from this validation are returned to the originator of the message. The validation is application dependent and can be arbitrarily complex. For example, the validation could simply check that the message contains all the expected elements. However, the validation could do something more complex, such as checking against a business contract in force among the participants to ensure the message is valid and timely.

A modular approach is taken where a particular higher level messaging pattern is composed from one or more lower level messaging patterns. The organisation of the hierarchy is such that the non-repudiation protocols from the literature (see Section 2.2) are situated at the bottom. Higher level messaging patterns are composed from these protocols to create a richer set of non-repudiation protocols. Finally, at the top of the hierarchy, domain specific messaging patterns are situated that undertake very specific tasks, such as placing an order or sending an invoice. The modular approach is similar to that employed by Hiltunen and Schlichting [37, 38]. Here they modularise an RPC protocol by the types of properties it exhibits. As a result, an RPC protocol has a number of modules[1] and a specific implementation is chosen for each. The chosen implementation dictates what properties are exhibited by the protocol. There exists modules for: *failure detection*, *recovery*, *membership* and *message ordering*. For example, if *total ordering* was required, a *total ordering* implementation would be used for the *message ordering* module.

The non-repudiation protocol hierarchy allows higher level messaging patterns to be composed from one or more lower level messaging patterns. Furthermore, the composition of chosen protocols

[1]The number of modules is in fact extensible. As a result any number of extra modules may be developed to add extra properties to the RPC protocol.

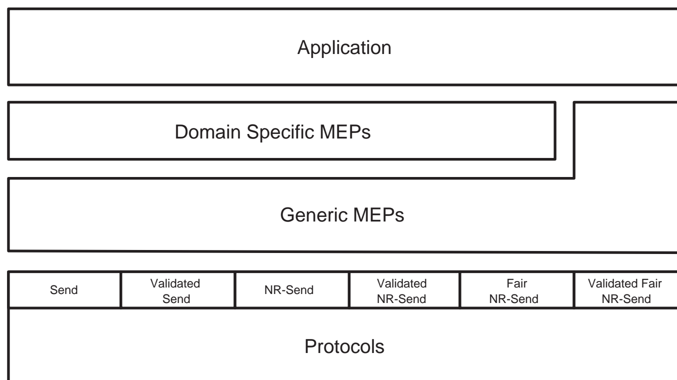


Figure 3.1: Non-repudiation protocol hierarchy

can be modified without disturbing the application. This allows non-repudiation and the validation requirements to be modified without having to modify the application. For example, the application may use a messaging pattern to send an invoice. This messaging pattern can be first composed from a fair non-repudiation protocol using an online TTP. At a later date, it may be decided that an offline TTP would be better for this interaction, in which case the chosen non-repudiation protocol can be changed without changing the application. Furthermore, the modular approach encourages extensibility. Consider a situation in which a new non-repudiation protocol appears in the literature: an implementation can be developed and plugged into the hierarchy without having to modify any of the existing messaging patterns. Higher level messaging patterns can also be developed and then plugged into the hierarchy.

This chapter describes each level of the hierarchy. A suite of initial messaging patterns are presented. These patterns span multiple levels of the hierarchy and create a rich and highly configurable set of messaging patterns to the application. In particular, a fair non-repudiation protocol is presented. This protocol provides fair non-repudiation guarantees whilst also providing validation of the application message. Here, a delivery-agent (inline TTP) is used to enforce fairness. The delivery-agent also provides extra services that can be exploited to enable lighter weight end users. This protocol is taken from work published in cooperation with Nick Cook and Santosh Shrivastava [**?**, **?**].

3.1 The non-repudiation protocol hierarchy

This section describes each level of the non-repudiation protocol hierarchy. The hierarchy is discussed in a top down fashion. The protocols at each level are described in turn and their requirements from the layer below are discussed.

The protocol hierarchy can be presented as a stack. This stack is shown in Figure 3.1.

The application is located at the top of the stack and can use the message exchange patterns (MEPs) provided by the *domain specific MEP* layer and the *generic MEP layer*. *Domain specific MEPs* are used for interactions with a very specific action (such as submitting a purchase order), *generic MEPs* are used for more general interactions (such as sending a message and waiting for a response). The domain specific MEP layer is located above the generic MEP layer. This is because domain specific MEPs are constructed as a composition of one or more generic MEPs. The generic MEP layer is situated above the protocol layer. This is because generic MEPs are constructed from a composition of one or more *message delivery patterns* (MDP). Each *MDP* is responsible for delivering a single message; the distinguishing feature of each *MDP* is whether the message is validated and what level of non-repudiation is provided. Six *MDPs* are provided by the protocol layer. These are used by the layer above to construct *generic MEPs*. The protocol layer contains implementations of several protocols, including the fair non-repudiation protocols present in the literature. These are mapped onto one or more of the six MDPs to provide a common interface for the *generic MEP* layer.

3.1.1 Domain specific message exchange patterns

Domain specific message exchange patterns (MEP) are designed for very specific purposes, such as sending an invoice or purchase request. Business protocols are one such type of domain specific MEP. A business protocol is a course grained interaction that occurs between two business partners. Each business protocol is designed to achieve some very specific task. Applications can be constructed by composing multiple business protocols to achieve some higher level task. The RosettaNet Consortium [70] define a set of business protocols, referred to as *Partner Interface Protocols* (PIPs). Each PIP defines the externally visible aspects of a business protocol.

There are currently 111 PIPs defined, each falling into one of eight categories (or clusters in RosettaNet terminology). The messages involved in a PIP can be classified into two broad categories, business action messages and business signal messages.

- **Business actions** are messages with content that is of a business nature, such as a purchase order or a request for quote.

- **Business signals** acknowledge receipt of *business actions*. A *business signal* can be positive or negative. A positive *business signal* states that the corresponding *business action* was received and found to be a structurally and syntactically valid RosettaNet *business action* message. A negative *business signal* states that there was some problem with the *business action*, or an exception has occurred, which is preventing process of the *business action*.

Their follows two examples of RosettaNet PIPs from the *Order Management* cluster.

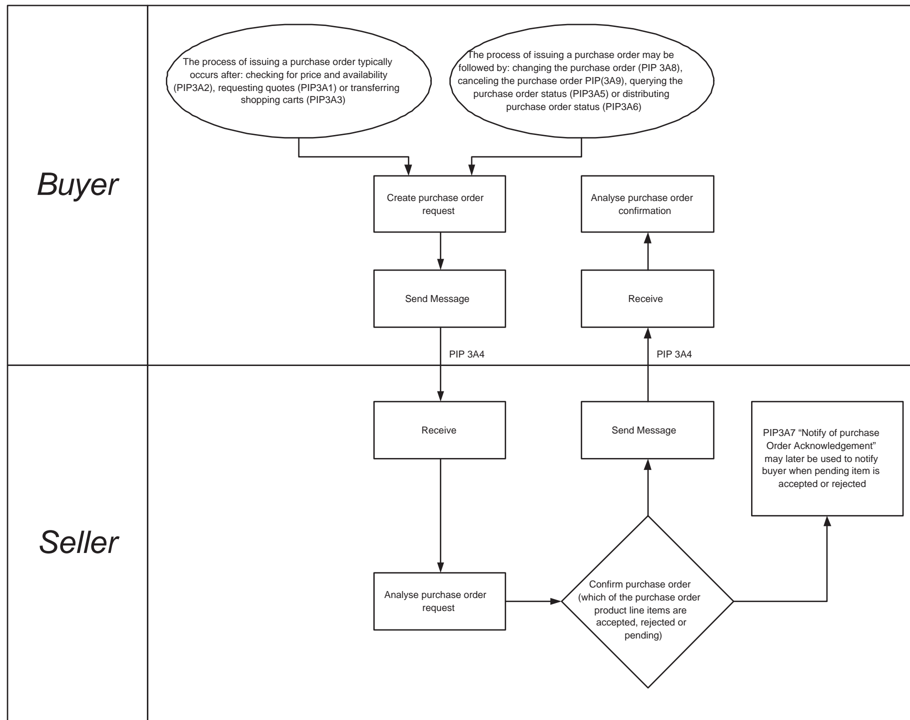


Figure 3.2: PIP3A4 Process Diagram

3.1.1.1 RosettaNet - request purchase order

PIP3A4 (Request Purchase Order) enables a buyer to issue a purchase order, and a seller to acknowledge if the order is accepted, rejected or pending.

Figure 3.2 shows a process diagram for this interaction. It begins with the buyer submitting a purchase order message to the seller. This message is a *business action*. The business action message is first verified and a suitable *business signal* is returned. The seller then checks for availability and sends a response message. The response message forms the second *business action*, which is acknowledged by the buyer with an appropriate *business signal*. Both *business actions* are PIP3A4 messages.

Alternatively, if the status of the purchase order is pending, the second *business action* is not sent and another PIP is invoked (PIP3A7).

Figure 3.3 shows the sequence of messages exchanged between the buyer and seller. If the purchase order is pending, only messages 1 and 2 are sent and PIP3A7 is invoked afterwards. This interaction is essentially one of validated request response, in which the response forms notification of whether the purchase order was accepted or rejected. The validation messages represent the *business signals*.

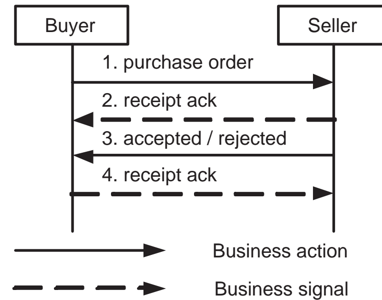


Figure 3.3: PIP3A4 Sequence Diagram

3.1.1.2 RosettaNet - notify of purchase order update

PIP3A7 (Notify of Purchase Order Update) enables a seller to accept a pending purchase order or notify the buyer of an updated purchase order.

Figure 3.4 shows a process diagram for this interaction. It begins with the seller analysing the purchase order. The order is accepted, rejected or a new order is proposed. If a new order is proposed, the buyer must respond by accepting or rejecting the changes to the purchase order (by invoking another PIP).

If the purchase order is accepted or rejected, a *business action* is sent to the buyer (which is acknowledged with a *business signal*). Should the purchase order be updated, the messages shown in Figure 3.3 are exchanged. This interaction is formed from one *business action* and one *business signal*.

3.1.1.3 Business protocol decomposition

All PIPs can be composed from a combination of two generic MEPs, validated single message and validated request/response. Figures 3.6 & 3.7 show, respectively, the general form of these two message exchange patterns. When composing a *domain specific MEP* from multiple *generic MEPs*, it is the responsibility of the *domain specific MEP* layer to add context to the composition of messages. This is necessary for associating each run of a *generic MEP* with a single *domain specific MEP*. PIP3A4 can be composed from a single *validated request/response* and PIP3A7 can be composed from a single *validated message*.

3.1.2 Generic MEPs

Generic MEPs are designed to undertake general tasks, such as send a message or participate in a request/response type interaction. Unlike domain specific MEPs, generic MEPs are not concerned with the contents of the application message; it is this attribute that makes them generic. Generic MEPs are used to compose domain specific MEPs and for building applications. Each generic MEP

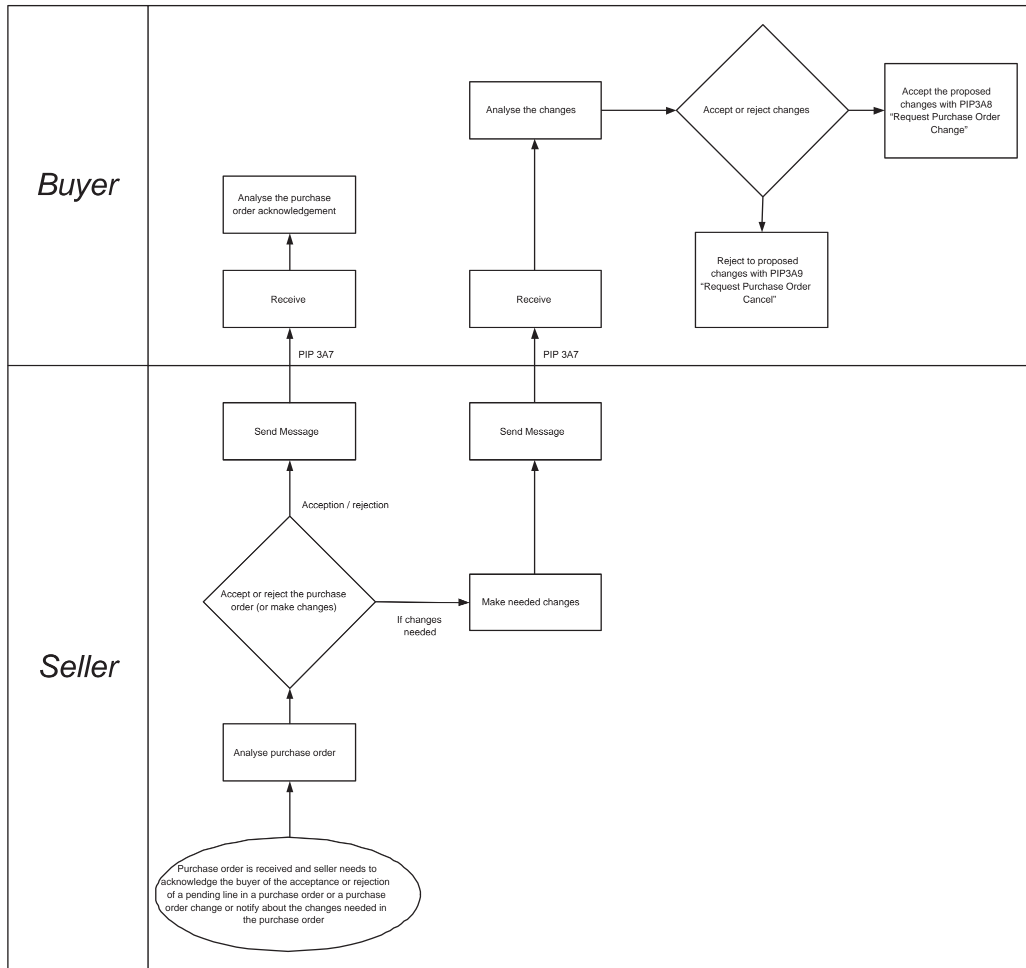


Figure 3.4: PIP3A7 Process Diagram

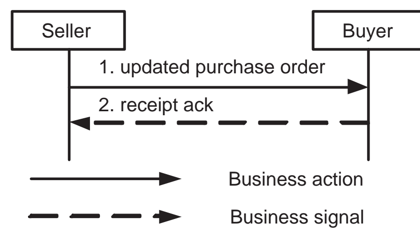


Figure 3.5: PIP3A7 Sequence Diagram

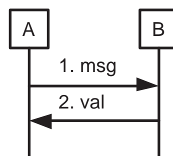


Figure 3.6: Validated Message

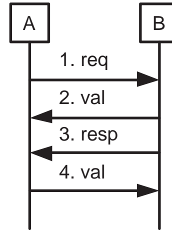


Figure 3.7: Validated Request/Response

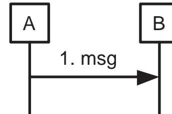


Figure 3.8: Single Message

is composed from six types of general protocols exposed at the protocol level.

This section is concerned with four generic MEPs. However, due to the extensibility of the hierarchy, it is possible to create any number of such protocols. Barros et. al present a large collection of MEPs [10], of which some are present in this section. Those not present could easily be implemented and added to this level of the hierarchy[2]. A description of these four generic MEPs follows.

- **Single Message**. This represents a single message sent from one participant to another (Figure 3.8).
- **Validated Message**. *Single message* with a validation acknowledgement returned by the recipient of the message (Figure 3.6).
- **Request/Response**. A request message with an associated response (Figure 3.9).
- **Validated Request/Response**. Similar to request/response, however the recipient of the request must validate the request. Similarly, the recipient of the response also validates the response. The results of the validation are returned to the other party (Figure 3.7).

Each of the four patterns are composed from one or two runs of a protocol, situated at the protocol level of the hierarchy. The protocol level offers two types of protocols, those which simply send a single message and those which send a single message and receive a validation response. The *Request/Response* pattern can be composed from two single message patterns. Similarly, the *Validated Request/Response* pattern can be composed from two single message with validation protocols. To

[2]It should be noted that those MEPs requiring multicast would need a multicast protocol present at the protocol level of the hierarchy. Non-repudiable multicast protocols are outside the scope of this thesis and have been suggested as further work.

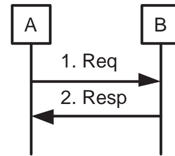


Figure 3.9: Request/Response

tie each run of a protocol to a single *generic MEP*, some common context identifier must be present in each message.

3.1.3 Protocol level

The Protocol level is responsible for abstracting from the actual protocol being executed. Many non-repudiation protocols (such as those discussed in Section 2.2) can be located at the protocol layer. A common interface is exposed which allows protocols to be changed without disrupting any generic MEPs that use them. The common interface presents six types of message delivery primitives (MDPs). Each message delivery primitive provides various non-repudiation guarantees and maps onto real protocols (such as those discussed in the literature) capable of fulfilling the promised guarantees. There follows a description of each message delivery protocol.

- **Send**. The *Send* MDP simply sends a message to the intended recipient. No non-repudiation guarantees are provided.

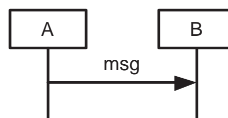


Figure 3.10: Send protocol

- **Validated Send**. The *Validated Send* MDP sends a message to the intended recipient and receives a validation (of the message) response in return. No non-repudiation guarantees are provided.

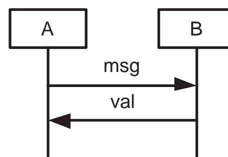


Figure 3.11: Validated Send protocol

- **NR-Send**. The *NR-Send* MDP is similar to the *Send* protocol, but with non-repudiation of origin.

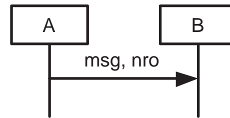


Figure 3.12: NR-Send protocol

- **Validated NR-Send**. The *Validated NR-Send* MDP is similar to the *Validated Send* protocol, but with non-repudiation of origin for the message and the validation.

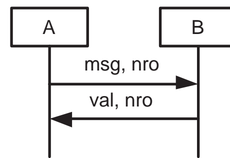


Figure 3.13: Validated NR-Send protocol

- **Fair NR-Send**. The *Fair NR-Send* MDP is similar to the *Send* protocol, but with fair exchange of non-repudiation of origin for non-repudiation of receipt.

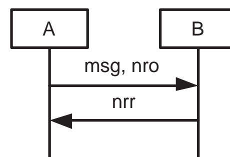


Figure 3.14: Fair NR-Send protocol

- **Validated Fair NR-Send**. The *Validated Fair NR-Send* MDP is similar to the *Validated Send* protocol, but with fair exchange of non-repudiation of origin for non-repudiation of receipt, on both the message and the validation.

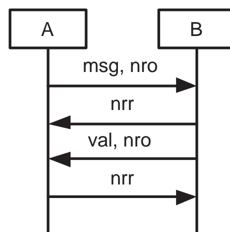


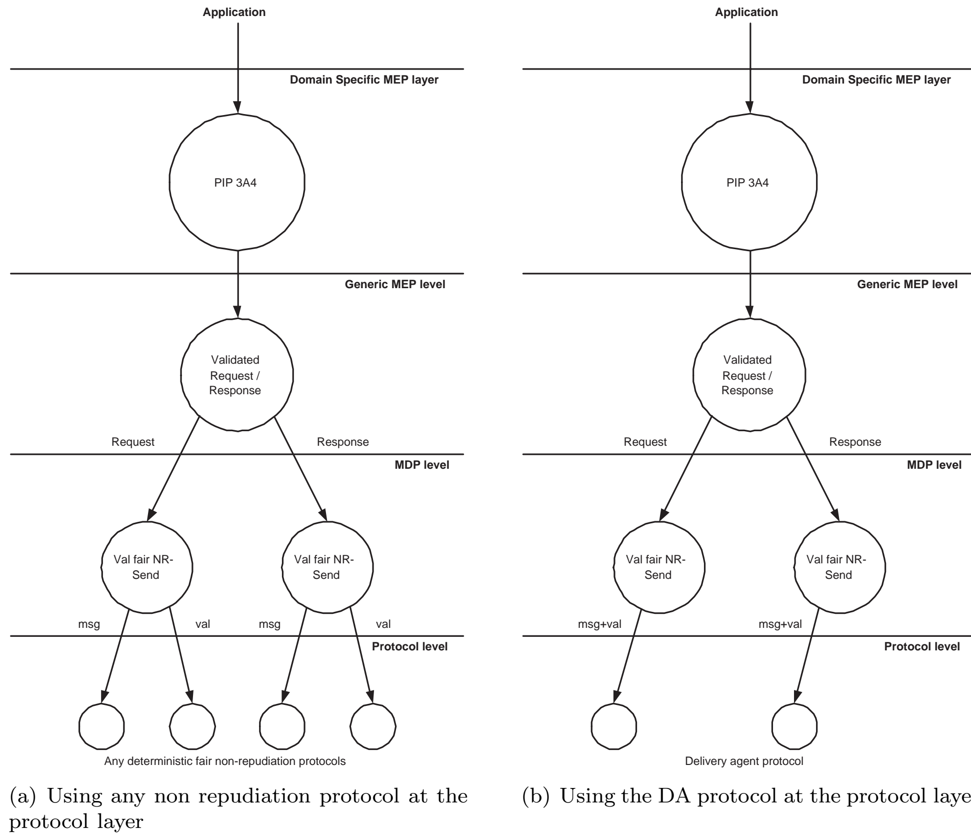
Figure 3.15: Validated Fair NR-Send protocol

Each *generic MEP* could be implemented by using numerous different MDPs, the choice made would be dependent on the non-repudiation guarantees required. For example the *send message* generic MEP could be implemented in the following ways:

- One run of the *Send* protocol if no non-repudiation is required.
- One run of the *NR-Send* protocol if the recipient requires non-repudiation of origin, but the sender does not require non-repudiation of receipt.
- One run of the *Fair NR-Send* protocol if the recipient requires non-repudiation of origin and the sender requires non-repudiation of receipt.

Each of the six MDPs are implemented using one or two (suitable) protocols (such as those discussed in Section 2.2). If the MDP does not require validation, one protocol is used. If the MDP does require validation, either one protocol supporting validation is used, or two protocols that do not support validation are used. In the latter case, the first instance of the protocol delivers the application message and the second instance delivers validation of the application message. Furthermore, the chosen protocol must provide the guarantees required by the MDP. For example the *fair NR-Send* MDP requires fair exchange, thus a fair non-repudiation protocol must be used.

Four of the MDPs have a single (trivial) implementation that directly reflects the messages exchanged in the above diagrams. More specifically, Send is implemented as shown in Figure 3.10, Validated Send is implemented as shown in Figure 3.11, NR-Send is implemented as shown in Figure 3.12 and Validated NR-Send is implemented as shown in Figure 3.13. Fair NR-Send and Validated Fair NR-Send need to ensure fairness. Thus a fair non-repudiation protocol must be used. Many such protocols exist (see Section 2.2). A novel extension to the Coffey and Saidha protocol (See section 2.2.2.1) is presented in the following section (Section 3.2). As this protocol supports validation, it can be used for both the *Fair NR-Send* MDP and the validated *Fair NR-Send* MDP.



(a) Using any non repudiation protocol at the protocol layer

(b) Using the DA protocol at the protocol layer

Figure 3.16: Two compositions for achieving non-repudiation of PIP3A4

3.1.4 Example protocol composition

This section shows a possible composition for rendering PIP3A4 non-repudiable. The example shows which protocol instances are present at each level of the protocol hierarchy.

Figure 3.16 (a) shows how non-repudiation of PIP3A4 can be achieved using any non-repudiation protocol, where as Figure 3.16 (b) shows how non-repudiation of PIP3A4 can be achieved using the Delivery Agent protocol (presented in Section 3.2). In both diagrams the *PIP3A4* implementation (located at the *Domain Specific MEP* level) maps onto a single *validated request/response* implementation (at the *Generic MEP* level). The *validated request / response* MEP uses two *validated fair NR-send* MDPs, one for the request and one for the response. The *validated fair NR-Send* MDP is mapped onto different non-repudiation protocols in each figure. In Figure 3.16 (a), the MDP is mapped onto any protocol that does not offer validation. As a result, a separate protocol instance is required for the message and the validation. In Figure 3.16 (b), the MDP is mapped onto two instances of the Delivery Agent protocol. As a result, each instance of the protocol handles both the message and the validation.

3.2 Delivery-agent based fair exchange

This section shows how a delivery agent (inline TTP) can be used to achieve the non-repudiation guarantees provided by the *Fair NR-Send* and *Validated Fair NR-Send* MDPs (Figures 3.14 & 3.15 respectively).

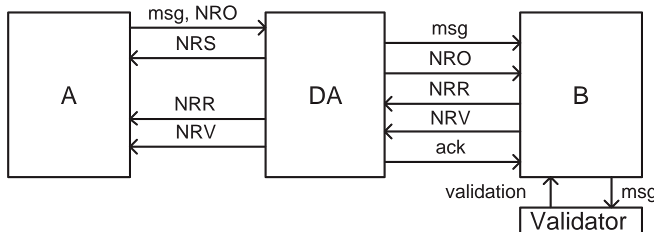


Figure 3.17: Executing a validated fair NR-send MDP through a delivery agent

Figure 3.17 shows the addition of this inline TTP to the interaction in Figure 3.15. The TTP acts as a delivery agent to *A* and *B* and will be referred to as *DA*. Four pieces of evidence are generated:

- **NRS**. Proof that the *DA* received *msg* and is able to continue the protocol;
- **NRO**. Proof that *msg* originated at *A*;
- **NRR.** Proof that *B* has received *msg*;
- **NRV.** Proof of validation regarding the outcome of *B*'s validation of *msg* (NRO of the validation).

It should be noted that the *NRV* maps directly to the second *NRO* in Figure 3.15. The distinction has only been made to ease the discussion. The *NRS* does not have a direct mapping to any evidence in the *Fair NR-Send* or *Validated Fair NR-Send protocols* and, as a result, the *NRS* is ignored by the MDP.

As shown in Figure 3.17, *A* starts an exchange by sending a message, with proof of origin, to *DA*. This is equivalent to the first message in Figure 3.15 with the *NRO* appended. *DA* exchanges *msg* and *NRO* for *NRR* with *B* (before application-level validation of *msg*). The *DA* provides *NRR* to *A* — equivalent to the second message in Figure 3.15. Subsequently, *B* performs application-level validation of *msg* (as in the third message of Figure 3.15) and provides *NRV* to *DA*. The *DA*, in turn, provides *NRV* to *A* and an acknowledgment to *B*. Note the exact sequence of message exchange will be dictated by the actual protocol used and should not be inferred from Figure 3.17.

It is the responsibility of the *DA* to ensure fairness and liveness for well-behaved parties in interactions that the *DA* supports. Further, the *DA*'s fairness and liveness guarantees hold for well-behaved parties in spite of any misbehaviour by any other party involved in an interaction (including misbehaviour by interceptors). Since the cooperation of misbehaving parties cannot be guaranteed, *in extremis* the *DA* will ensure that any disputes that arise can be resolved in favour of well-behaved parties.

There follow two variations of a fair non-repudiation protocol supported by the *protocol* Layer, providing the guarantees required by the *Fair NR-Send* and *Validated Fair NR-Send* protocols. First, an overview of the chosen protocols is presented with discussion of the motivation behind the choice. Then the protocol assumptions and notation are stated. The section concludes with a detailed description of each protocol. In addition to the main protocol, sub-protocols for abnormal termination are also presented. Since the delivery agent is trusted, these sub-protocols can be used to deliver fairness and liveness guarantees in the event of failure of the main protocol.

3.2.1 Protocol overview

Coffey and Saidha developed a fair non-repudiation protocol utilising an in-line TTP [16]. This was later improved by Zhou and Gollman in [92]. Two protocols are discussed, both of which are derived from the improved version. Both protocols include an extension to support the extra validation messages required by the *Validated Fair NR-Send* protocol. The first protocol is a further modification to support lightweight end users. The second is intended for use with a more lightweight delivery agent, as envisaged by Coffey and Saidha.

In the first protocol, the delivery agent is responsible for much of the evidence verification and for the long-term storage of evidence for audit. The end users are only required to verify evidence produced by the delivery agent. They only require long-term storage for the information necessary to link an interaction to the evidence held by the delivery agent (such as a protocol run identifier). This approach means the end users do not have to provide as much infrastructure support for fair exchange. For example, they only need access to the certificate and public key of the delivery agent for verification and not the information for all parties they will potentially interact with.

In the second protocol, the responsibilities for evidence verification and long-term storage are transferred to the end users. In this case, the delivery agent may discard information after completion of a protocol run. Furthermore, the lightweight delivery agent only signs the NRS as opposed to all evidence generated.

3.2.2 Assumptions

The standard perfect cryptography assumptions [72] are made:

1. Message digests are one-way, collision resistant.

2. It is computationally infeasible to predict the next bit of a secure pseudo-random sequence even with complete knowledge of the algorithmic or hardware generator and all of the previous bits in the sequence.

3. Digital signatures cannot be forged.

4. Encrypted data cannot be decrypted except with the appropriate decryption key.

The following assumptions are made with respect to well-behaved parties to a non-repudiable interaction:

1. Communication channels between well-behaved parties provide eventual message delivery (there is a bounded number of temporary network and computer related failures).

2. Each party has persistent storage for messages. More precisely, well-behaved parties will ensure that messages are available for as long as is necessary to meet their obligations to other parties. Longer term storage may be required for their own purposes.

3. Well-behaved parties only exchange messages that are well-constructed with respect to the protocol being executed. For example: messages exchanged are either tamper-resistant (encrypted), or tampering is detectable and well-behaved parties will cooperate to ensure a well-constructed message is eventually delivered.

To guarantee fairness, the same assumptions are made with respect to the DA as in existing fair exchange protocols, namely:

1. The DA is well-behaved.

2. The DA ensures protocol resolution in favour of well-behaved parties.

3.2.3 Notation

In the protocols, participant A wishes to send a business message, msg, to participant B. All communications between A and B take place through delivery agent DA. Table 3.1 provides the notation used for basic protocol elements. To simplify protocol descriptions, and without loss of generality, it is assumed that the signature scheme is recoverable. That is, if necessary, x (and any items that are concatenated to construct x) may be recovered from $sig_P\,(x)$[3]. To allow verification of rn as a protocol authenticator, it is also assumed that id contains $h\,(rn)$. Table 3.2 defines the non-repudiation evidence exchanged during a protocol run. DA associates a termination state with each exchange. The state is $SUCCEEDED$ if the exchange is successfully completed and $ABORTED$ if the exchange is cancelled.

[3] If the signature scheme is non-recoverable, then any necessary items are sent with the associated signature.

Notation	Description
rn	secure pseudo-random number
$h(x)$	secure digest of x
id	unique protocol run identifier
i, j	concatenation of items i and j
$P \rightarrow Q : m$	P sends m to Q
$sig_P(x)$	P's digital signature on x
$enc_P(x)$	encryption of x with P's public key
$VAL \mid INV$	signifies msg validity or invalidity
$NVAL$	signifies msg not validated

Table 3.1: Notation for protocol elements

Non-repudiation token	Description
$NRS_{DA} = sig_{DA}(id, A, B)$	DA's NRS of initial protocol message
$NRO_A = sig_A(id, A, B, h(msg))$	A's NRO of msg
$NRR_B = sig_B(id, A, B, h(msg))$	B's NRR of msg
$NRV_B = sig_B(id, VAL \mid INV)$	B's NRV of msg
$NRO_{DA} = sig_{DA}(id, A, B, msg)$	DA's NRO of msg for B
$NRR_{DA} = NRO_{DA}$	DA's NRR of msg for A
$NRV_{DA} = sig_{DA}(id, VAL \mid INV)$	DA's NRV of msg
$NRV'_{DA} = sig_{DA}(id, NVAL)$	DA's substitute NRV of msg

Table 3.2: Definition of non-repudiation tokens

3.2.4 Fair exchange for light-weight end users

This section first discusses normal execution to successful completion of the main protocol for light-weight end users. We then present abort and resolve sub-protocols for exception handling.

Normal protocol execution

Normal execution of the main protocol is shown below, followed by a commentary on each step.

$$
\begin{array}{llcll}
1 & A & \rightarrow & DA & : \; enc_{DA}\,(msg,\, rn,\, NRO_A) \\
2 & DA & \rightarrow & A & : \; NRS_{DA} \\
3 & DA & \rightarrow & B & : \; id,\, A,\, B,\, h\,(msg) \\
4 & B & \rightarrow & DA & : \; enc_{DA}\,(NRR_B) \\
5 & DA & \rightarrow & A & : \; NRR_{DA} \\
6 & DA & \rightarrow & B & : \; msg,\, NRO_{DA} \\
7 & B & \rightarrow & DA & : \; NRV_B \\
8 & DA & \rightarrow & B & : \; id,\, rn \\
9 & DA & \rightarrow & A & : \; NRV_{DA}
\end{array}
$$

Step 1: A sends a business message (msg), a secure pseudo-random number (rn) and its non-repudiation of origin token (NRO_A) to DA. At this step, if DA finds that the id included in NRO_A is not unique, an appropriate response will be generated to prompt A to restart the protocol with a newly generated id. Otherwise, the protocol will proceed to step 2. The rn provided by A is used in step 8 as the acknowledgement of receipt of *B's* NRV token. All items are encrypted to guarantee that B does not obtain the items before providing non-repudiation of receipt.

Step 2: DA provides proof of submission to A to signal willingness to proceed with protocol execution. This step may be executed in parallel with step 3.

Step 3: To enable B to construct NRR_B: DA sends the id, the participant identifiers A and B (recovered from NRO_A) and a digest of msg to B.

Step 4: B responds with NRR_B. It is safe for B to send the receipt to DA before obtaining msg because DA, as TTP, can and will provide msg in return. NRR_B is encrypted to guarantee that A can only obtain the receipt if the exchange runs to some form of successful completion.

Step 5: DA sends NRR_{DA} to A. This is DA's receipt for msg and assurance that it has received and verified NRR_B. This step may be executed in parallel with step 6.

Step 6: DA sends msg and associated NRO_{DA} to B.

Step 7: B performs application-level validation of msg. The outcome of this validation is signed along with id to form NRV_B that is sent to DA.

Step 8: rn, hitherto known only to A and DA, is sent to B as acknowledgement of receipt for NRV_B. This step may be executed in parallel with step 9.

Step 9: DA sends NRV_{DA} to A — non-repudiation of the outcome of validation of msg.

At the end of execution of the main protocol, A has acquired the following evidence: NRS_{DA}, NRR_{DA} and NRV_{DA} — non-repudiation of submission, receipt and validation of msg. In return, B has acquired: msg, NRO_{DA} and rn — the business message with non-repudiation of origin and acknowledgement of validation of msg. DA has the complete set of evidence, including: NRO_A, NRR_B and NRV_B.

Fairness is guaranteed because DA controls the release of the evidence to A and B. Furthermore, DA's signature on the evidence provided to A and B: (i) serves as a guarantee that DA has seen, verified and will store the evidence for future reference; and (ii) reduces the verification work of A and B to that of verifying the signature and associated credentials of a well-known TTP.

If B does not wish to perform application-level validation of msg, then the protocol can terminate at step 6. In this case, at step 4, B sends DA both NRR_B and a default NRV_B token that confirms the validity of msg. At step 5, DA sends A both NRR_{DA} and NRV_{DA}. At step 6, DA sends rn to B with msg and NRO_{DA}.

On successful completion of the main protocol, DA sets termination state to $SUCCEEDED$.

Exception handling

In exceptional circumstances A or B may request that DA terminate the main protocol before completion. Such requests typically occur because A or B is concerned about the liveness of protocol execution (whether, as a result, of the non-cooperation of a participant or extraneous factors such as network delays). There are two types of request:

abort: where the requesting party wishes to terminate the protocol as if no exchange had taken place. That is, neither A nor B receive any useful information about the exchange.

resolve: where the requesting party seeks DA's assistance in securing normal termination. That is, all expected items (or their equivalent) are available to well-behaved parties.

These requests are, in effect, the statement of a preference for how the exchange should complete. Irrespective of the type of request, it is the responsibility of DA to ensure that fairness guarantees hold for all honest parties. Depending on the progress of the main protocol and whether the exchange termination state has already been set, DA must determine whether the exchange should terminate

in $ABORTED$ state (no exchange has taken place) or $SUCCEEDED$ state (exchange has taken place). An exchange can terminate in $SUCCEEDED$ state if and only if: (i) A is entitled to NRS_{DA}, NRR_{DA} and NRV_{DA} (or an equivalent substitution); **and** (ii) B is entitled to msg, rn and NRO_{DA}.

DA is empowered to issue the substitute non-repudiation of validation, NRV'_{DA}, in place of NRV_{DA}. NRV'_{DA} is DA's signed confirmation that B has not validated msg. Once DA has produced NRV'_{DA} no validation of msg by B will be accepted. NRV'_{DA} is equivalent to invalidation of msg with the supplementary information that B did not cooperate in the decision. At first sight, this places A at a disadvantage, since B can receive msg and simply decide not to cooperate in its validation. However: (i) in any case, B may autonomously decide that a message is not valid (and such invalidation may be subject to *extra-protocol* dispute resolution); and (ii) A obtains evidence of B's lack of participation in validation. Thus, in terms of evidence exchanged, the substitution of NRV'_{DA} for NRV_{DA} is fair.

From the above, we observe that fairness is guaranteed to both A and B if:

1. the main protocol completes normally; or

2. B chooses not to engage in the main protocol by not responding to step 3 (up to and including step 3, A has only received NRS_{DA} and B has no useful information about msg); or

3. the exchange is aborted when the main protocol has progressed no further than step 4 (at step 4, B sends NRR_B to DA but the protocol can still be aborted because A does not have NRR_{DA} and B is yet to receive msg or NRO_{DA}); or

4. the exchange is completed successfully after execution of step 4 (at step 4, DA has all the information necessary to complete the exchange; after execution of step 5, DA must guarantee that all expected items are available to both A and B).

The pivotal point in the main protocol is step 4. Before step 4, DA can only respond to either type of termination request by aborting the exchange. Upon execution of step 4, DA has rn, msg, NRO_A and NRR_B but is yet to complete the release of information to either A or B. Thus, at this point, they can satisfy whichever type of termination request they receive first. Once DA releases critical information in step 5, they must respond to a termination request by successfully resolving the exchange.

A request from an end user, $U \in \{A, B\}$, to DA to abort an exchange results in execution of the following abort sub-protocol:

1 $U \rightarrow DA : sig_U(sid, ABORT, id)$

if $ABORTED$

or (not $SUCCEEDED$ and $lastStep < 5$) then:

2.1 $DA \rightarrow U : sig_{DA}(sid, ABORTED, id)$

else if $lastStep < 7$ then:

2.2 $DA \rightarrow U : SUCC_{DA}, res_U, NRV'_{DA}$

else:

2.3 $DA \rightarrow U : SUCC_{DA}, res_U, NRV_{DA}$

where:

$$
\begin{aligned}
sid &= \text{unique sub-protocol identifier} \\
SUCC_{DA} &= sig_{DA}(sid, SUCCEEDED) \\
res_A &= NRS_{DA}, NRR_{DA} \\
res_B &= rn, msg, NRO_{DA}
\end{aligned}
$$

In step 1, U submits a signed request to abort the exchange identified by id. DA then checks the state of the exchange. If termination state is not already $SUCCEEDED$ and the main protocol has not progressed beyond step 4 ($lastStep < 5$), then DA sets termination state to $ABORTED$. DA provides U with non-repudiation of aborted exchange in step 2.1. If the exchange cannot be aborted, DA checks whether the main protocol has progressed beyond step 6. If not, then step 2.2 above is executed to complete a successful exchange with substitute NRV'_{DA}. Otherwise step 2.3 is executed to complete successful exchange of the evidence that would have been provided during normal execution. As shown above, res_U is the resolution evidence provided to A or B, as appropriate. If either step 2.2 or 2.3 is executed, DA sets termination state to $SUCCEEDED$.

The corresponding resolve sub-protocol is:

1 $U \rightarrow DA : sig_U(sid, RESOLVE, id)$

if $ABORTED$ or $lastStep < 4$ then:

2.1 $DA \rightarrow U : sig_{DA}(sid, ABORTED, id)$

else if $lastStep < 7$ then:

2.2 $DA \rightarrow U : SUCC_{DA}, res_U, NRV'_{DA}$

else

2.3 $DA \rightarrow U : SUCC_{DA}, res_U, NRV_{DA}$

Apart from the signed request that initiates the resolve sub-protocol, the only significant difference to the abort sub-protocol is that execution of step 2.1 is triggered if either termination state is $ABORTED$ or the main protocol has not progressed beyond step 3.

Once the termination state of an exchange has been set (whether after execution of the main protocol or one of the above sub-protocols), *DA* will forever respond in the same way to any subsequent request to abort or resolve the identified exchange (with appropriate abort token or resolution evidence). *DA* also responds in the same way to any subsequent message of the main protocol. That is, once the termination state has been set, *DA* suspends the main protocol at $lastStep$.

Termination may also be triggered by the *a priori* indication of deadlines for the acknowledgements provided in the main protocol (*NRS, NRR* and *NRV).* In this case, in step 1 of the main protocol *A* can indicate deadline(s) for delivery that they wish to be observed (on a best effort basis). During protocol execution, *DA* determines locally whether a delivery deadline is achievable. If not, *DA* will pro-actively terminate the exchange and issue appropriate abort or resolve tokens to *A* and *B* depending on the state of the main protocol at the time of termination.

3.2.5 Fair exchange with light-weight delivery agent

Normal protocol execution

The protocol for fair exchange with light-weight *DA* is:

1	A	$\rightarrow$	DA	:	$enc_{DA}\,(msg,\, rn,\, NRO_A)$
2	DA	$\rightarrow$	A	:	NRS_{DA}
3	DA	$\rightarrow$	B	:	$id,\, A,\, B,\, h\,(msg)$
4	B	$\rightarrow$	DA	:	$enc_{DA}\,(NRR_B)$
5	DA	$\rightarrow$	A	:	NRR_B
6	DA	$\rightarrow$	B	:	$msg,\, NRO_A$
7	B	$\rightarrow$	DA	:	NRV_B
8	DA	$\rightarrow$	B	:	$id,\, rn$
9	DA	$\rightarrow$	A	:	NRV_B

This protocol is closer to the Coffey-Saidha protocol with the addition of steps 7 to 9 for NRV of msg. The difference between this protocol and the light-weight end user protocol is that *A* and *B* are now responsible for verification of each other's evidence and for its long-term storage. Thus, in steps 5, 6 and 9, *DA* relays the tokens provided by *A* and *B* rather than generating new signed tokens.

Exception handling

The abort and resolve sub-protocols are basically as defined in Section 3.2.4 except that the resolution evidence provided in steps 2.2 and 2.3 is now:

$$\begin{aligned} res_A &= NRS_{DA}, NRR_B \\ res_B &= rn, msg, NRO_A \end{aligned}$$

For successful termination before step 7, the *DA* provides NRV'_{DA} (as described in Section 3.2.4). For successful termination after step 6, *DA* provides NRV_B (as opposed to NRV_{DA}). The abort token is identical to that defined in Section 3.2.4.

3.2.6 TTP message validation

TTP can host B's message validators if during the execution of the non-repudiation protocol TTP receives a plain text version of the message. When this is the case, *DA* can verify *msg* on receiving step 1. If the message is valid, the protocol continues, otherwise *DA* responds to A with an appropriate *NRV* and the protocol ends.

3.3 Summary

This chapter presented a non-repudiation protocol hierarchy which allows for a large variety of application requirements. These requirements are fulfilled by providing a multitude of messaging patterns coupled with varying levels of non-repudiation and validation guarantees. Furthermore, these messaging patterns are (to some extent) agnostic to the choice of non-repudiation protocol. As a result, the underlying non-repudiation protocol can be modified without disrupting the application.

To facilitate configurable service-based non-repudiation, a delivery agent based protocol was presented that provides fair non-repudiation of application message and validation of the application message. It was shown how this protocol could be modified to facilitate lightweight end users or lightweight delivery agent. Furthermore, it was shown how exception handling sub-protocols can be used to ensure fairness is achieved in the presence of unexpected behaviour.

Chapter 4

Protocol Representation

This chapter develops a notation for specifying a class of TTP based fair non-repudiation protocols. The chapter then shows how an executable protocol can be derived automatically from such a specification. Due to high level of automation, confidence that the protocol being executed faithfully represents the intended protocol specification is greatly increased. Furthermore, the tedium (and possibility of bug introduction) associated with protocol development is also removed.

The chapter begins by discussing a class of supported protocols. This includes a definition of the properties required for a protocol to be added to this class. The chapter then presents a means for representing this class of protocols as a series of finite state machines (FSMs). The actions and events used in these FSMs largely comprise sending and receiving protocol messages (respectively). The chapter concludes with a rigorous method for automatic generation and verification of protocol messages.

In this chapter three participants will be discussed. The participant who initiates the non-repudiation protocol will be referred to as A. The participant with whom A wishes to exchange with is referred to as B. The trusted third party used in the execution is referred to as TTP.

When discussing non-repudiation protocols, three types of sub-protocol are discussed. The main protocol is referred to as *exchange*. The exception handling sub-protocols are referred to as *abort* and *resolve*.

4.1 Supported non-repudiation protocols

The protocol support described in this chapter is intended for a subset of non-repudiation protocols. Protocols in this subset must conform in the following ways.

1. **Structure**. The protocol must comprise one main protocol and zero to two sub-protocols. The *exchange* protocol must facilitate normal execution of the protocol. If present, the exception handling sub-protocols must be used to request a trusted third party to bring the main protocol to completion. These sub-protocols should move the protocol to a consistent state in which

no party is disadvantaged. The two possible sub-protocols are namely *abort* and *resolve*. The *abort* sub-protocol attempts to move the protocol to an aborted state in which no party has any evidence and B does not have the application message. The *resolve* sub-protocol attempts to move the protocol to a resolved state in which both A and B have required evidence and B has the application message.

2. **Representation**. The protocol must be representable using the notation described in Section 4.3. The semantics for each of the described tokens must also match the semantics intended by the protocol author.

From the protocol comparison in Section 2.2.6, nine out of the ten protocols are supported; these are [16, 90, 87, 88, 47, 35, 81, **?**]. The Zhang and Shi protocol [84] and a derivation [91] of the Zhou Gollmann [90] protocol are both unsupported as they contain timing information in the protocol. The Zhang and Shi protocol uses timing information to provide deadline by which a certain part of the protocol must complete — this prevents the protocol responder from needing to wait an arbitrarily long time for the protocol to complete. The Zhou Gollmann protocol uses timing for this same reason and also to allow the TTP to add a time-stamp to the evidence (this states at what time the message was available to the protocol responder). The issue of supporting timing and deadlines has been left as further work and is discussed in Section 6.4.1.

4.2 Finite state machine representation

The progress of a participant through the various stages of a non-repudiation protocol can naturally be expressed as a finite state machine. Here, transitions between states occur when a protocol message is received. Transitions also occur when an exception is detected, such as the perceived non-cooperation of another participant. Each transition could trigger an action. This action could be either to send another protocol message or begin exception handling. Each participant has a finite state machine for the exchange protocol and one for exception handling.

In order to create the FSMs, the subset of non-repudiation protocols was taken and each one carefully inspected. Due to their similarity, it was possible to construct a set of general FSMs that could be used to represent any protocol in that classification. A similar process could be applied to other security protocols.

4.2.1 Exchange protocol FSMs

During the execution of a non-repudiation protocol, each participant will operate a finite state machine that facilitates normal movement through the exchange protocol and also the initiation of exception handling. There follows a description of the exchange protocol FSM for A, B and TTP.

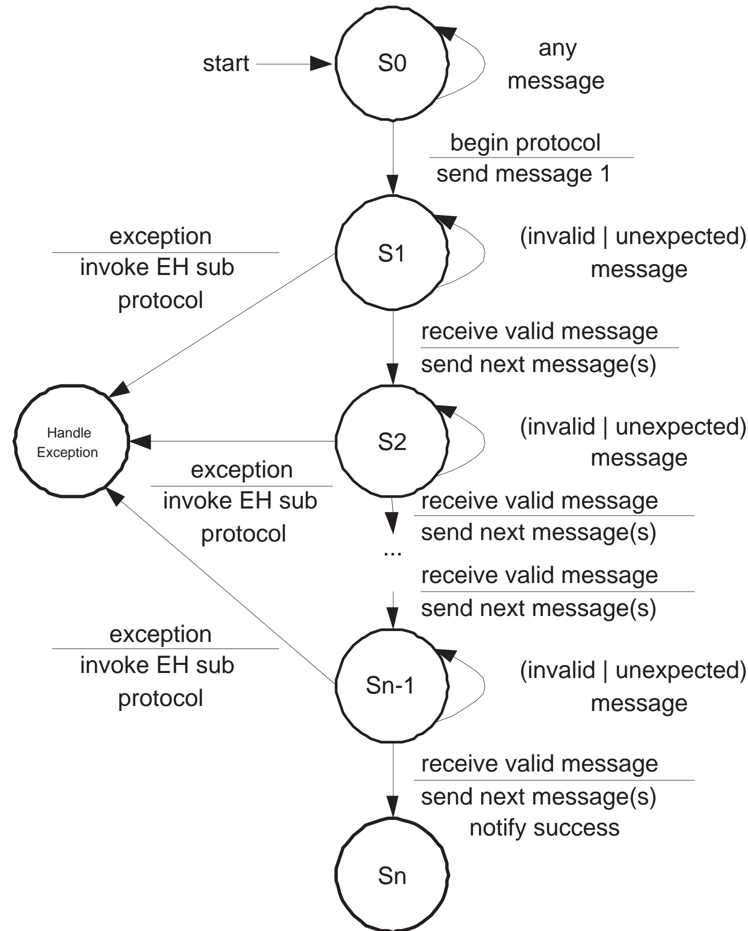


Figure 4.1: A's Exchange Protocol FSM

4.2.1.1 A's exchange protocol FSM

Figure 4.1 shows the FSM of the exchange protocol as executed by A. A begins in the initial state, S0. In this state, all received messages are ignored and thus result in A remaining in state S0. On occurrence of the *begin protocol* event, a transition is made to state S1. During this transition, an action is invoked, this action is to send the first protocol message to the intended recipient.

From state S1 A may do the following:

1. **Normal progress**. A will move from state *Sm* to *Sm+1* (where, $0 < m < n$) when a valid and expected protocol message is received. A send message action occurs if A's next involvement in the protocol is to send a protocol message. Numerous protocol messages could be sent until the participant's next involvement is to receive a protocol message or until the protocol has completed.

2. **No progress**. A will remain in the same state if an unexpected or invalid message is received.

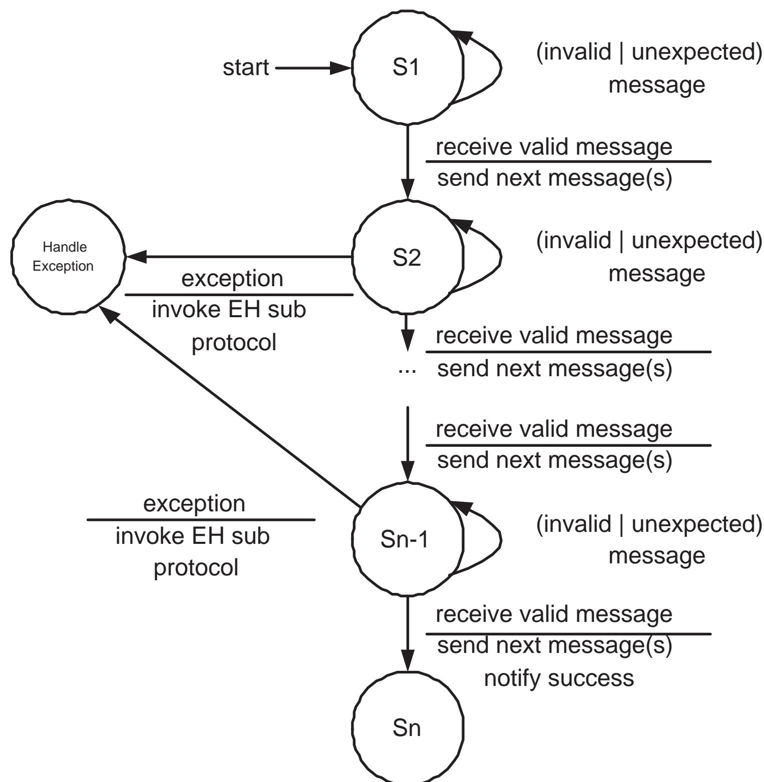


Figure 4.2: B's Exchange Protocol FSM

3. **Transition to *HandleException***. This will occur on detection of an un-handleable internal exception or in the event of a timeout. Timeouts occur when a particular protocol message is not received by a given deadline. The action on this transition is used to invoke an exception handling sub-protocol.

The above options are available for all states *Sm*, where, $0 < \mathrm{m} < \mathrm{n}$.

States *Sn* and *Handle Exception* are end states. State Sn represents a successful completion of the protocol where as the *Handle Exception* end state may represent a successful or aborted protocol run. The FSM for the exception handling sub-protocol is discussed in Section 4.2.2.1.

4.2.1.2 B and TTP's exchange protocol FSM

Figure 4.2 Shows *B* and *TTP's* exchange protocol FSM. This FSM is similar to that shown in Figure 4.1, with the following differences:

- There is no state *S0*. This is because *B* and *TTP* may not initiate the exchange protocol.
- The state S1 is now the start state. This is because *B* and *TTP* begin by waiting for the first valid and relevant protocol message.

- The state *S1* does not detect an exception. This is because no exception can occur before the protocol run has begun (from point of view of B or TTP).

- The transition from state *S1* to *S2* can only occur on receipt of an expected and valid protocol message. Neither B nor TTP may send a protocol message until they have received their first protocol message.

In the case of an offline TTP, an exchange protocol FSM would not exist. This is because offline TTPs are only required to execute the exception handling sub-protocols. Furthermore, in a protocol with no exception handling sub-protocols, the TTP would not have an exception handling FSM.

4.2.2 Exception handling FSMs

In exceptional circumstances, a participant may request termination before (its perception of) the exchange protocol is completed. This would typically occur if the participant is concerned about the timeliness of the protocol execution. This could be a result of non-cooperation of another participant or excessive network delays.

If an exception is detected during the execution of the exchange protocol, the exception handling FSM is executed. Some non-repudiation protocols have sub-protocols for bringing the main protocol to a consistent completion. Other's do not support exception handling. A consistent completion is one in which all participants have the same view regarding the status of the protocol. These views consist of:

- **Aborted**. The protocol terminates as if no exchange has taken place. That is, no participant (excluding trusted third parties) receives any useful information about the exchange.

- **Resolved**. All well behaved participants experience normal termination. That is, all expected items are available to well behaved participants.

When participating in a non-repudiation protocol without exception handling sub-protocols, there is a risk of breaking fairness. On detection of an exception, the participant is forced to simply quit the protocol, regardless of its progress.

4.2.2.1 A and B's exception handling FSM

Figure 4.3 shows the finite state machine for exception handling at end user participants, A and B. The state machine starts in the *Handle Exception* state which is arrived at when the main state machine detects an exception. From this initial state, there are three options available for handling the exception.

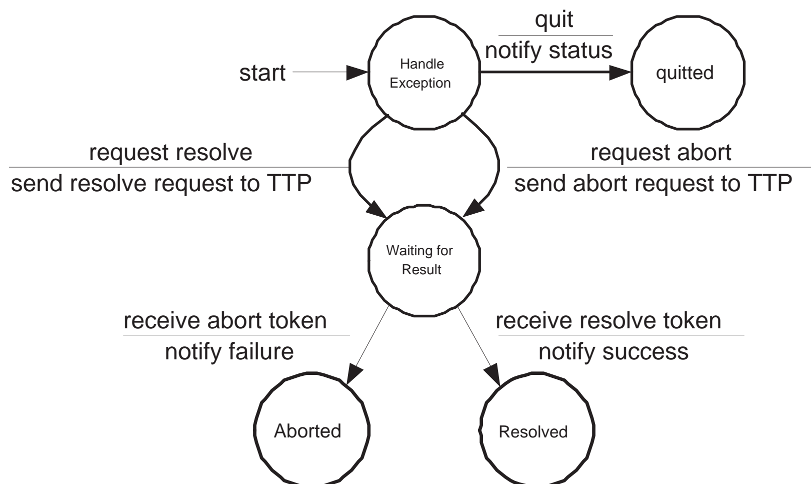


Figure 4.3: Generic exception handling FSM

1. **Quit.** This event is raised if the exchange protocol has no exception handling sub-protocol or if the participant has not released any information that could be deemed useful to another party, such as non-repudiation evidence.

2. **Request abort.** This event is raised if the exchange protocol has an associated sub-protocol capable of bringing the exchange protocol to an aborted state.

3. **Request resolve.** This event is raised if the exchange protocol has an associated sub-protocol capable of bringing the exchange protocol to a resolved state.

When both abort and resolve exception handling sub-protocols are available, a choice can be made on the preferable outcome of the exchange protocol. This choice is only a preference and is not guaranteed to happen in practice. This occurs because the exchange protocol may not be in a position that allows fair completion to the preferred state. For example a TTP cannot resolve a protocol if it does not have enough tokens. Similarly a protocol cannot be aborted if one participant (other than the TTP) has received useful information.

4.2.2.2 TTP's exception handling FSM

Should an exception handling sub-protocol exist, it is the responsibility of the trusted third party to execute it at the request of end user participants. It could also be the case that the TTP detects an exception and initiates the exception handling sub-protocol itself.

Figure 4.4 shows the FSM for the exception handling sub-protocol from the TTP's point of view. The FSM shown is for an exception handling sub-protocol that supports abort and resolve requests; although a particular sub-protocol may only support one of these requests. If the sub-protocol does

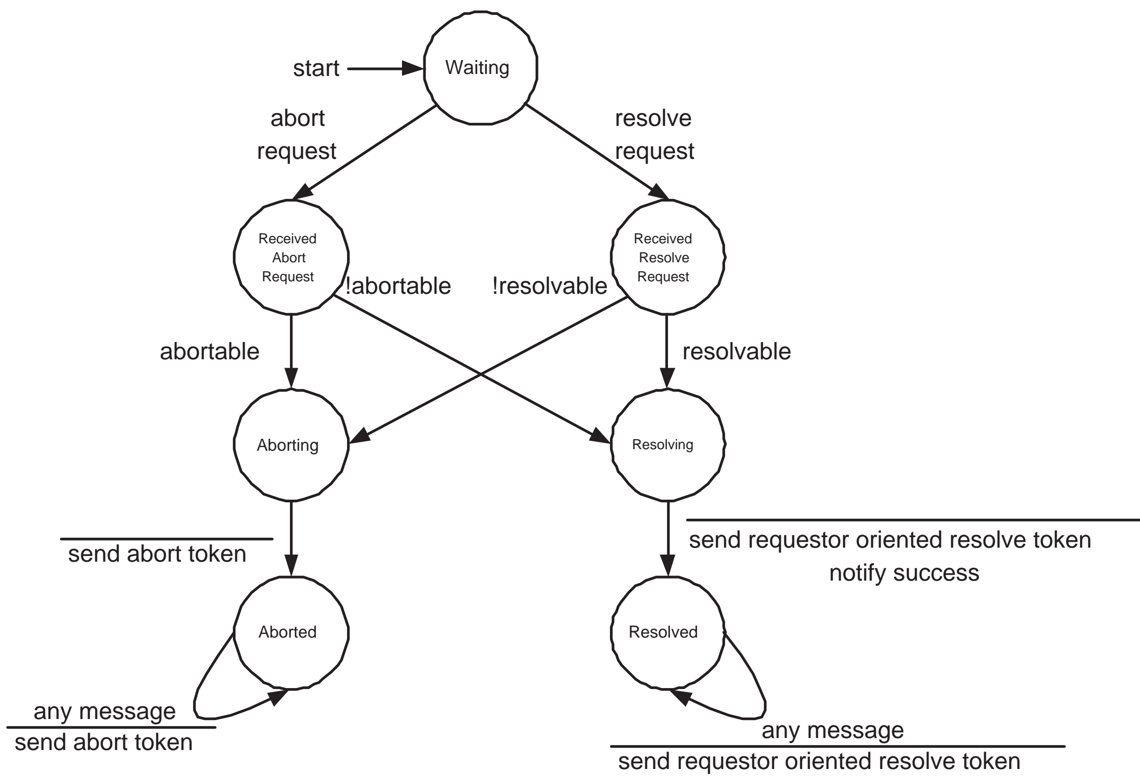


Figure 4.4: TTP Exception Handling FSM

not support resolve requests, the FSM will be similar, but with no 'Received Resolve Request' state (and attached transition arrows). Similarly, if the sub-protocol does not support abort requests, there will be no 'Received Abort Request' state.

The TTP knows in each state of the exchange protocol whether it can be aborted or resolved. It is possible that in some states of the exchange protocol, that it could be both aborted or resolved. This can occur when the TTP has all knowledge required to complete the protocol and no end user participant has received any useful information, such as non-repudiation evidence. On receiving a request to abort the main protocol, the TTP will check to see if the exchange protocol is abort-able. If this is the case, the TTP will mark the protocol as aborted and indicate this by responding to the exception handling request with an abort token. If the exchange protocol is not abort-able, the TTP will mark the protocol as resolved and then respond to the exception handling request with requester specific resolve tokens. Similarly, when the TTP receives a request to resolve the protocol, the exchange protocol will be resolved if possible and aborted if not.

Once the exchange protocol has been marked as aborted or resolved, all protocol messages received by the TTP will receive the final status of the protocol as a response.

4.3 Non-repudiation protocol notation

The transition through a finite state machine is mostly dependent on the sending and receiving of valid protocol messages. In this section, a rigorous mechanism for creating and verifying valid protocol messages is presented. In order to undertake this process, the non-repudiation protocol must be canonicalised. This is done by expressing the protocol in some specific notation. This notation is described in Section 4.3.2.

Given a protocol expressed in the notation, the section goes on to describe how a protocol message can be generated and verified. This is done by exploiting a particular property of non-repudiation protocols. This property being that non-repudiation protocols are composed of a careful and gradual release of information. Once in possession of a full set of correct information, a participant holds all required evidence to the interaction. The release of information is guarded by prior receipt of correct protocol messages.

4.3.1 Assumptions

The following assumptions are made when automatically creating and verifying protocol messages.

- The protocol messages can be described using the following notation.
- The protocol messages can be generated using the following protocol message generation techniques.

- The protocol messages can be verified using the following protocol message verification techniques.
- Where necessary, all participants have a set of initial knowledge that contains the participant's own private key and the public key of all other participants. It is assumed that participants will add to this knowledge as they receive new tokens.
- The initiator of the protocol knows the identity of all other participants and the contents of the application level message.
- Each participant has a common agreement on what cryptographic algorithms to use.

Typically, all protocol messages are sent (or *pushed*) from one participant to another. However, some protocols operate a feature where a participant is required to *pull* a protocol message from another participant. This feature is used slightly to reduce the requirements of trusted third parties. To simplify the process, all *pull* messages must be replaced with *pushes*.

4.3.2 Notation for describing protocols

To allow a non-repudiation protocol to be machine interpretable, it must be specified using a specific notation. This notation allows the specification of common terms and methods commonly used in non-repudiation protocols. Furthermore each token could be represented using a combination of other tokens, recursively.

Message

The message is the payload of any non-repudiation protocol and is the subject of subsequent evidence. The message is represented using the following notation:

```
msg
```

The message is a primitive term and is present in the initial knowledge of A.

Validation

Validation tokens state that a particular token must be validated. This validation mechanism can be used to integrate message validation into a protocol. A validation token can be expressed using the following notation:

```
val(x)
```

Where x can be any concatenation of tokens. x will usually be the message, *msg*.

Flags

Flags are used to give a human readable label to a particular protocol message. They typically describe the purpose of the protocol message. A flag is represented using the following notation:

```
f(x)
```

Where x is a piece of free text used for the description. Multiple flags can be present in a single protocol message. Some special flags are also present; these are interpreted by the system interpreting the protocol notation. These special flags are as follows:

`f(ABORT)`. Represents a request to abort the current protocol run.

`f(RESOLVE)`. Represents a request to resolve the current protocol run.

`f(ABORTED)`. States that the current protocol run has been aborted.

`f(SUCCEEDED)`. States that the current protocol run has succeeded.

Participants

Each participant in the protocol must be identified. Only the identifiers used in the section of the protocol describing the message flow can be used. For example, “A”, “B” and “DA” could be three participant identifiers.

Nonce

Several nonces may be used in a protocol and, as a result, need to be uniquely identifiable. A nonce is a unique number. All parties must ensure that they have never seen this nonce before.

The following notation is used to represent a particular nonce:

```
ni
```

Here “i” represents a number unique to each instance of a particular nonce. For example, a protocol containing two separate nonces would use the notation n1 and n2 to define each instance. The shorthand 'n' may be used if only one nonce is present.

Random number

Several random numbers may be used in a protocol and, as a result, need to be uniquely identifiable. A random number should not be predictable. The following notation is used to represent a particular random number:

```
rni
```

Here 'i' represents a number unique to each instance of a particular random number. For example, a protocol containing two separate random numbers would use the notation rn1 and rn2 to define each instance. The shorthand 'rn' may be used if only one random number is present.

Symmetric key

Similarly to a nonce, several symmetric keys may be used in a protocol and, as a result, need to be uniquely identifiable. The following notation is used to represent a particular symmetric key:

```
ki
```

Here 'i' represents a number unique to each instance of a particular symmetric key. For example, a protocol containing two separate symmetric keys would use the notation k1 and k2 to define each instance. The shorthand 'k' may be used if only one symmetric key is present.

Digest

A digest over a concatenation of tokens can be described using the following notation:

```
h(t1,t2...,tn)
```

Here 't1' represents the first token and 'tn' represents the nth token. For example, a digest over two participants A and B along with the message msg could be represented as follows:

```
h(A,B,msg)
```

Run identifiers

A run identifier has the same semantics as a digest, but is named differently. In addition to it being a digest, it also conveys some extra information that states it can be used to uniquely identify a protocol run. By expressing a run identifier in the notation, the middleware knows how to identify protocol runs. If no run identifier is present, the middleware will provide a default. Many protocols have unique labels that are used as run identifiers — this notation allows the middleware to utilise them.

A run identifier created from a concatenation of tokens can be described using the following notation:

```
runID(t1,t2...,tn)
```

Here 't1' represents the first token and 'tn' represents the nth token. For example, a run identifier created from two participants A and B along with the message msg could be represented as follows:

```
runID(A,B,msg)
```

The actual value of the run identifier will be a digest of the concatenation of each enclosing term.

Asymmetric encryption

Asymmetric encryption, or more specifically public key encryption is represented with the following notation:

```
eP_x(t1,t2...,tn)
```

Here 'x' represents the participant whose public key is used for encrypting the concatenation of the tokens t1 to tn.

Symmetric encryption

Symmetric encryption is represented using the following notation:

```
ekn(t1, t2...,tn)
```

Here 'kn' represents the particular symmetric key used to encrypt the concatenation of the tokens t1 to tn. 'kn' should be present elsewhere in the protocol as an individual token (unless part of a participant's initial knowledge).

Signature

A digital signature is represented using the following notation:

```
sS_x(t1, t2...,tn)
```

Here 'x' represents the identity of the participant who's private key is used to sign the concatenation of the tokens t1 to tn.

Control flow notation

Control flow notation can be used to make a decision on what message to send next. The control flow takes the following form:

```
                IF <expr1> THEN
n.1 X -> Y : message1
                ELSE IF <expr2> THEN
n.2 X -> Y : message2
                ELSE
n.3 X -> Y : message3
```

If participant X evaluates expr1 to true, participant X sends message1 to participant Y. Otherwise, if participant X evaluates expr2 to true, participant X sends message2 to Y. Otherwise participant X sends message3 to Y.

Expressions use the logical operators (==, !=, &&, ||, <, >). Also, the tokens 'AND' and 'OR' may be used in place of '&&' and '||' respectively. Some variables may be queried as part of an expression, these comprise:

- **lastStep**. This is the last step of the exchange protocol that the participant evaluating the expression successfully undertook.
- **status**. This variable reports the status of the protocol. status can take one of three values, (ABORTED, SUCCEEDED, INCOMPLETE).

The protocol step numbers use a decimal point to indicate that each step is sub step of a given step n.

4.3.2.1 Example

There follows a representation of the Delivery Agent protocol (Described in Section 3.2) represented using this notation. In the notation, several variables have been used to represent run identifiers and non-repudiation evidence. These can be replaced inline with the following definitions:

```
ID      = runID(A, B, DA, rn)
NROa    = sS_A(ID, A, B, h(msg))
NRSda   = sS_DA(ID, A, B)
NRRb    = sS_B(ID, A, B, h(msg))
NRVb    = aS_B(ID, val(msg))
NROda   = sS_DA(ID, A, B, msg)
NRRda   = NROttp
NRVda   = sS_ttp(ID, val(msg))
NRNVda  = sS_ttp(ID, val(msg))
```

There follows the exchange protocol of the Delivery Agent protocol (Described in Section 3.2).

```
1 A -> DA : eP_TTP(ID, A, B, rn, msg, NROa)
2 DA -> A : ID, A, B, NRSda
3 DA -> B : ID, A, B, h(msg)
4 B -> DA : ID, A, B, eP_TTP(NRRb)
5 DA -> A : ID, A, B, NRRda
6 DA -> B : ID, A, B, msg, NROda
7 B -> DA : ID, A, B, val(msg), NRVb
8 DA -> B : ID, A, B, rn
9 DA -> A : ID, A, B, val(msg), NRVda
```

The remainder of the Delivery Agent protocol notation is presented in Section A.1.

4.3.3 Protocol message creation

A protocol message can be generated providing each of the composing tokens is available in the participant's knowledge. Some tokens will be available in the participant's initial knowledge, such as the participant's private key, other participants' public keys and maybe some symmetric keys. Other knowledge is acquired as the protocol progresses. Non-primitive tokens can be generated if and only if all the composed tokens are in the participant's knowledge and any other tokens required to undertake the operation are also in the participant's knowledge. It could be envisaged that a participant may try to construct a message without having enough knowledge. However, it is assumed that the protocol has been subjected to some verification and thus a participant will not attempt to send a message unless all prior messages received are correct. These assumptions ensure that all protocol message creation attempts will succeed.

It should be noted that tokens are only created if their value is not already in the participant's knowledge. Otherwise, the value is simply taken from the knowledge and re-used. There follows a description of how each token is created. Some tokens have to be taken from the knowledge as the participant does not currently know the tokens used to create it.

Message

The message is a primitive token that will be present in A's initial knowledge. Other participants can only generate protocol messages containing this token if they have received it in a prior protocol message.

Validation

Validation tokens have one of three values, *valid*, *invalid* and *not validated*. When creating a validation token it must be evaluated to one of these values. This validation is delegated to the system interpreting the notation in much the same way that a digest token is evaluated by the system to create a digest value.

Flags

All participants can create any flag.

Participants

The participant tokens are present in A's prior knowledge, thus the initiator is always able to use them. Other participants will receive these tokens when they first participate in the protocol. It is likely that these tokens will be passed in each protocol step.

Nonce

If generating a message in which a nonce is specified for the first time (with respect to the whole protocol run), it should be created. This involves checking for uniqueness. If the nonce has prior specification (with respect to the whole protocol run), it should be known and thus can be placed in the protocol message.

Random number

If generating a message in which a random number is specified for the first time (with respect to the whole protocol run), it should be created. This involves checking for uniqueness. If the random number has prior specification (with respect to the whole protocol run), it should be known and thus can be placed in the protocol message.

Symmetric key

A symmetric key is either in a participant's initial knowledge, or is received in an earlier protocol message.

Digest

A digest of a concatenation of tokens can be created iff the whole concatenation of tokens is known.

Run identifier

A run identifier based on a concatenation of tokens can be created iff the whole concatenation of tokens is known.

Asymmetric encryption

A concatenation of tokens may be encrypted iff the whole concatenation of tokens is known. It is assumed that the public key of the specified participant will be known.

Symmetric encryption

A concatenation of tokens may be encrypted using a symmetric key iff the whole concatenation of tokens is known and the specified symmetric key is also known.

Signature

A concatenation of tokens may be signed iff the whole concatenation of tokens is known and the specified secret key is that of the participant generating the message.

4.3.4 Protocol message verification

On receiving a protocol message, the recipient participant must verify it to be correct. This involves verifying each token that composes the message. Primitive tokens (ones that are not composed from other tokens) can always be verified. Whereas non-primitive tokens can only be verified if the tokens used to represent them are also verifiable. Thus verification of each non-primitive token must be deferred until, but no later than, all the composed tokens are available.

Message

The message itself cannot be verified as only the initiator knows what it should contain. However, it is likely that the knowledge of this message will allow other non-primitive tokens to be verified. For example, a digest of the message may have been sent in a prior step. Receiving the message will allow the digest of the message to be verified.

Validation

Validation tokens do not need verifying. It is typical that they would be enclosed in a signature token, in which case the signature token is verified.

Flags

None of the flags need validating. It is typical that they would be enclosed in a signature token, in which case the signature token is verified.

Participants

The first message of the protocol will state which participants are to be present in the execution of the protocol. Subsequent steps of the protocol can be verified by ensuring the participant identifiers are used where intended.

Nonce

On first sight of a particular nonce, it should be checked for uniqueness and then stored for subsequent use. When receiving a message supposed to contain a nonce, that has already been seen, it should be checked for equality against the nonce received earlier.

Random number

On first sight of a particular random number, it should be stored for subsequent use. When receiving a message containing a random number that has already been seen, it should be checked for equality against the random number received earlier.

Symmetric key

Symmetric keys should be stored such that any tokens encrypted under that particular key can be decrypted.

Digest

The digest value can only be verified if the participant is in possession of the tokens subject to the digesting algorithm. Otherwise the verification must be deferred until the required tokens have been received.

Run identifier

The run identifier value can only be verified if the participant is in possession of the tokens subject to the digesting algorithm used to create the run identifier. Otherwise the verification must be deferred until the required tokens have been received.

Asymmetric encryption

If the token was encrypted under the recipient's public key, then the token may be decrypted and the contents stored. Otherwise the token is stored in its encrypted form for later use, maybe to be forwarded on to another participant.

Symmetric encryption

Tokens encrypted under a symmetric key can be decrypted, provided that the corresponding symmetric key is available. Again, it may be necessary to wait for the symmetric key to arrive before the contents can be acquired. Similarly to tokens encrypted under an asymmetric key, the sole purpose could be to forward the cipher text to another participant.

Signature

Tokens signed using a digital signature may be verified iff the signed tokens are available. It is assumed that the public key of the signatory is available.

4.4 Example creation and verification steps

This section shows the steps undertaken by *A* and *DA* when creating and verifying the first two protocol messages of the delivery agent protocol (Described in Section 3.2).

Protocol message 1 - created by A

To create protocol message 1, *A* begins by inspecting the notation:

```
ID      = runID(A, B, DA, rn)
NROa    = sS_A(ID, A, B, h(msg))
1 A -> DA : eP_DA(ID, A, B, rn, msg, NROa)
```

The following steps are undertaken by *A* to create the protocol message:

1. **Generate *ID***. *ID* represents a run identifier token, this token is simply a digest over the enclosed tokens. *A* can construct the run identifier as all the enclosed tokens are known or can be generated. *A, B* and *DA* are participant tokens and are in *A*'s initial knowledge. *rn* represents a random number and appears for the first time. *A* creates a value for *rn* by generating a new random number.

2. **Generate *NROa***. *NROa* represents *A*'s signature over a concatenation of tokens. The signature can be created because *A* has access to his own private key and all the containing tokens are known. *ID* has just been created, *A* and *B* are participant tokens, and are thus in *A*'s initial knowledge. Finally, the digest over the message can be constructed as the message is also in *A*'s initial knowledge.

3. **Generate the protocol message**. The protocol message comprises of several tokens encrypted with *DA*'s public key. *DA*'s public key will be in *A*'s initial knowledge and all the encrypted tokens are now known by *A*.

The following steps are undertaken by *DA* to verify the protocol message:

1. **Decryption**. Firstly the protocol message must be decrypted. This can be done as the protocol message was encrypted with DA's public key and DA holds the corresponding private key.

2. **Verify *NROa***. The signature can be verified as *A*'s public key is in *DA's* initial knowledge and *ID*, *A*, *B* and *h(msg)* are all present in the now decrypted protocol message. Furthermore, *h(msg)* can also be verified as *msg* is also now known.

3. **verify *ID***. *ID* is a digest over four tokens, all present in the decrypted protocol message, so it can be verified.

Protocol message 2 - created by B

To create protocol message 2, *DA* begins by inspecting the notation:

```
ID      = runID(A, B, DA, rn)
NRSda   = sS_DA(ID, A, B)
2 DA -> A : ID, A, B, NRSda
```

The following steps are undertaken by *DA* to create the protocol message:

1. **Generate *NRSda***. *ID*, *A* and *B* are all present in the protocol message so DA can sign their concatenation to generate NRSda.

2. **Generate the protocol message**. *ID*, *A* and *B* can be taken from protocol message 1 and *NRSda* was generated in the previous step. Thus *DA* can generate the full protocol message.

The following steps are undertaken by *A* to verify the protocol message:

1. **Verify previously seen tokens**. Those tokens (*ID*, *A*, *B*) proposed by *A* in protocol message 1 can be checked for equality. Leaving just *NRSda* requiring verification.

2. **Verify *NRSda***. *A* knows the values for *A*, *B*, *DA* and *rn* (they were used by *A* in protocol message 1). Thus the signature over these tokens can be verified (*A* also knows *DA's* public key).

The remaining protocol messages can be generated and verified in a similar fashion.

4.5 Summary

In this chapter it was shown how a class of TTP based fair non-repudiation protocols could be modeled in order to facilitate a rigorous implementation. Correct progress through a protocol was managed using a collection of FSMs. State transitions in the FSM where undertaken on receipt of a valid protocol message. Such a transition would then trigger an action to send zero or more valid protocol messages. The chapter presented a rigorous means for generating and verifying protocol messages. This process relied on the protocol specification being canonicalised. Canonicalisation was achieved by representing the protocol using some specific notation. The process of representing the FSM, creating protocol messages and verifying protocol messages is automated from the canonicalised protocol specification. This increases confidence that the running protocol is faithful to the protocol's specification.

Chapter 5

Middleware Implementation

This chapter presents a Web services based implementation (WS-NRExchange) of a framework for executing non-repudiation protocols. The middleware implements the messaging stack described in Chapter 3. Firstly an overview of Web services and the standards used to support WS-NRExchange is presented. Section 5.2 describes a high-level view of the WS-NRExchange architecture. This section presents the middleware architecture and the interaction with existing Web service standards and services. Section 5.3 describes the form of protocol messages exchanged during the the execution of a non-repudiation protocol and how they are verified and generated using the principles presented in Chapter 4. Section 5.4 shows the internal implementation of the middleware and describes how protocol messages are handled. Furthermore, the implementation of each layer of the messaging stack is presented (presented in Section 3.1). The chapter concludes by showing how the finite state machines from Section 4.2 can be used as a basis for rigorous implementation of a non-repudiation protocol. Example protocol messages are present in Appendix A.

The middleware described in this chapter is based on work published in cooperation with Nick Cook and Santosh Shrivastava [**?**].

5.1 Web services and supporting standards

The implementation of WS-NRExchange is based on Web services. A Web service is a service that can be described, published, located and invoked over the Web. Web services are based on open standards and are designed to be platform-neutral. Web services are an instance of a service oriented architecture (SOA). The following is a definition of SOA by IBM:

> SOA is an approach to build distributed systems that deliver application functionality as services to end user applications or to build other services.

A service is a package of functions that do not depend on the context or state of other services. Each service can publish its functionality, while other services are capable of discovering and binding

dynamically to this functionality. Web services adopt the SOA concept using an XML-based message layer (called SOAP [33]) and can use any transport layer such as HTTP and SMTP. The main significance of Web services is that it has been embraced by a large proportion of the industry.

5.1.1 SOAP

The SOAP specification [33] describes a message format encoded in XML. The purpose of a SOAP message is primarily for inter-service communication and is the messaging format used by Web services. A SOAP message consists of an envelope and zero or more attachments. The envelope is split into two sections, a header and a body. The body contains the message destined for the intended recipient. The header contains information required by intermediaries who may need to process the SOAP message in some way. For example, routing information may be placed in the header and processed by various routing devices as the SOAP message travels to its intended recipient.

```
<soap:envelope>
  <soap:header>
    <!-- information intended for intermediaries -->
  </soap:header>
  <soap:body>
    <!-- message intended for ultimate recipient -->
  </soap:body>
</soap:envelope>
```

Figure 5.1: SOAP Envelope

As a SOAP envelope is encoded in XML, it is understandable by any XML aware system; thus SOAP is a good communication mechanism for heterogeneous systems.

5.1.2 WSDL

Web service description language (WSDL) is a language for describing how to interface with a Web service. To interact with a Web service, a consuming application must know the services interface, including how to structure content and which transport protocol to use. WSDL explicitly describes this interface in a standardized, machine-readable format. Figure 5.2 shows a very simplified WSDL document for a purchase order service. This service exhibits a single operation for receiving a purchase order message. The service responds with an acknowledgement message. Towards the top of the WSDL document the purchaseOrder and acknowledgement messages are defined using a *message* element. This message element describes the required structure of the message. In this example, both message elements state that the body of their message must contain a String and nothing else. This type could also comprise a more complex structure. The portType element states what operations can be called on the service. In this example, there is an operation for raising a

```
<definitions>
  <message name="purchaseOrder">
    <part name="body" element="xs:String"/>
  </message>
  <message name="acknowledgement">
    <part name="body" element="xs:String"/>
  </message>
  <portType name="purchaseOrderService">
    <operation name="raisePurchaseOrder">
      <input message="purchaseOrder"/>
      <output message="acknowledgement"/>
    </operation>
  </portType>
</definitions>
```

Figure 5.2: Example simplified WSDL document

purchase order (*raisePurchaseOrder* operation). This operation has an *input* and *output* element. The *input* element states what type of message is required to invoke the service. The *output* element states what message type is returned from the service. In this example the service is invoked with a *purchaseOrder* message and an *acknowledgement* message is returned.

5.1.3 XML Signature

XML Signature [27] specifies a mechanism for creating an XML representation of a digital signature.

```
<PurchaseOrder id=’’po1’’>
  <ItemID>15683</ItemID>
  <Quantity>2</Quantity>
  <CreditCardNum>1234-5678-9012-1314</CreditCardNum>
</PurchaseOrder>
```

Figure 5.3: XML Purchase Order

Figure 5.3 shows an XML document that represents a purchase order. Figure 5.4 shows the purchase order signed with XML Signature.

```
<PurchaseOrderDocument>
  <Signature>
    <SignedInfo>
      <Reference "URI=#po1" >
        <DigestMethod Algorithm="..." />
        <DigestValue> ... </DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>...</SignatureValue>
    <KeyInfo>...</KeyInfo>
  </Signature>
  <PurchaseOrder id="po1">
    <ItemID>15683</ItemID>
    <Quantity>2</Quantity>
    <CreditCardNum>1234-5678-9012-1314</CreditCardNum>
  </PurchaseOrder>
</PurchaseOrderDocument>
```

Figure 5.4: XML Signature Example

The example is split into two parts. The signature and the information being signed. The *Signature* block contains the XML signature which is signing the PurchaseOrder block. The signature contains a SignedInfo block – This is what is actually signed. This block contains a reference to the purchase order and a digest of the purchase order. The digest algorithm is also specified. The digest ties the purchase order to the *SignedInfo* block. The *SignedInfo* block is digitally signed and the signature placed in the *SignatureValue* block. The KeyInfo element contains information regarding which key to use when verifying the signature. This may be the key itself or some reference to a key.

Verification of the signature occurs in two phases. Firstly the signature value needs to be checked, using the key described in the *KeyInfo* Block. This ensures it represents a valid signature on the *SignedInfo* block. If this is valid the second phase occurs. This phase is responsible for ensuring that the referenced data has not been modified. This entails passing the target of the reference through the specified digest algorithm and ensuring it matches the contents of the *DigestValue* block.

The example uses a detached signature. This is because the reference points to something outside the Signature block. This reference can be any URL and thus may point to information outside the root of the XML document. Other types of XML signatures are Enveloped and enveloping signatures. An enveloped signature is placed inside the XML block to be signed. An enveloping signature contains the signed block within the Signature block.

The example shows the key aspects of XML Signature. The reader is encouraged to refer to [27] for the definitive usage.

5.1.4 XML Encryption

XML Encryption provides a means for encrypting certain portions of an XML document.

```
<PurchaseOrder>
  <ItemID>15683</ItemID>
  <Quantity>2</Quantity>
  <CreditCardNumber>
    <EncryptedData>
      <EncryptionMethod Algorithm=''...''/>
      <CipherData>
        <CipherValue>...</CipherValue>
      </CipherData>
      <KeyInfo>...</KeyInfo>
    </EncryptedData>
  </CreditCardNumber>
</PurchaseOrder>
```

Figure 5.5: XML purchase order with encrypted credit card number

Figure 5.5 shows the the purchase order from Figure 5.3 with an encrypted credit card number. This is done by replacing the value from the *CreditCardNumber* block with an EncryptedData block. The EncryptedData block contains several tags, one CipherData block for holding the cipher text and several meta tags to describe how to undertake decryption. The *EncryptionMethod* tag states which encryption algorithm was used and the *KeyInfo* block describes what key should be used for decryption.

The example shows the key aspects of XML encryption. The reader is encouraged to refer to [26] for the definitive usage.

5.1.5 WS-Security

WS-Security [59] is a specification for creating self-protecting SOAP documents. The fundamental components of the specification and those of most interest to this thesis are XML Signature and XML Encryption. XML signature and XML encryption are simply XML-based standards. They are completely agnostic as to the rest of the XML document and thus are unaware of the SOAP standard. WS-Security provides a SOAP specific way of using XML Signature and XML Encryption. It describes where the various parts of the two standards should be placed and also adds some extra stipulations regarding their use.

```
<Soap:Envelope>
  <Soap:Header>
    ...
  </Soap:Header>
  <Soap:Body>
    <PurchaseOrder>
      <UnitID>15683</UnitID>
      <Quantity>2</Quantity>
      <CreditCardNumber>1234-5678-9012-1314</CreditCardNumber>
    </PurchaseOrder>
  </Soap:Body>
</Soap:Envelope>
```

Figure 5.6: Purchase Order SOAP Message

Figure 5.6 shows the purchase order from Figure 5.3 placed in the body of a SOAP message.

```
<Soap:Envelope>
  <Soap:Header>
    <wsse:Security>
      <!-- XML Signature -->
      <ds:Signature>
        <ds:SignedInfo>
          <ds:Reference URI=''#po1'' >
            <ds:DigestMethod Algorithm=''...''/>
            <ds:DigestValue> .. </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue> .. </ds:SignatureValue>
        <ds:KeyInfo>..</ds:KeyInfo>
      </ds:Signature>
      <!-- XML Encryption -->
      <xenc:ReferenceList>
        <xenc:DataReference URI=''#CCNum'' />
      </xenc:ReferenceList>
      ...
    </wsse:Security>
  </Soap:Header>
  <Soap:Body>
    <PurchaseOrder id=''po1''>
      <ID>15683</ID>
      <Quantity>2</Quantity>
      <CreditCardNumber>
        <xenc:EncryptedData Id=''CCNum''>
          <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
          </xenc:CipherData>
        </xenc:EncryptedData>
      </CreditCardNumber>
    </PurchaseOrder>
  </S:Body>
</S:Envelope>
```

Figure 5.7: WS-Security Protected Purchase Order SOAP Message

The example in Figure 5.7 shows XML Signature and XML Encryption applied to the SOAP message to sign the *PurchaseOrder* element and encrypt the *CreditCardNumber* block. WS-Security dictates that the entire XML Signature must be placed within the SOAP header within a wsse:Security

element. The wsse:Security block contains all the WS-Security specific elements.

The credit card number cipher-text still replaces the plain text version. However, meta data, such as information regarding which key to use when decrypting, is delegated to the header and placed within the *wsse:Security* block. Also inside this block is a ReferenceList. This refers to all the encrypted elements in the SOAP document. Their is no way of stating whether the signature was created before or after the credit card number was encrypted. However, good practice dictates that signatures should be taken on plain text. This is because, given some signature over some ciphered text, it cannot be proved that the signatory was able to view the corresponding plain text.

Other WS-Security information can be put in the wsse:security block. However, this has been left out for the sake of brevity. The interested reader should refer to [59].

5.1.6 WS-Reliability & WS-Reliable Messaging

WSReliability [21] is a recent OASIS standard where as WS-Reliable Messaging [68] is an industry-proposed standard by IBM, Microsoft et al, for reliable messaging. Both specify message content, exchange patterns and persistent storage to which each party must adhere to achieve various forms of reliable message delivery.

5.1.7 WS-Addressing

WS-Addressing [20] provides transport-neutral mechanisms to address Web services and messages. Web service end points and message identifiers can be specified. The former allows intermediary message processors to inspect and modify the target end point. The latter allows multiple messages to be associated with a context.

5.1.8 XML Key Management Service (XKMS)

WS-Security operates best using public key cryptography; this is because it does not suffer from the key distribution problems of shared key cryptography. However, public key cryptography requires a supporting infrastructure for managing key distribution, certification and life cycle management. This has been managed in the past by PKI (Public Key Infrastructure); however, this technology has proved hard to implement and maintain. The advent of Web services have meant that this technology can be optionally outsourced to a (trusted) specialist provider. XKMS is a W3C submission [36] that provides a means for implementing such services.

5.1.9 Digital Signature Service (DSS)

DSS is an OASIS proposed standard for digital signature services for signature creation and validation, and (secure) trusted time stamping. The service can be used by a trusted third party to

implemented a time stamping service. The service can also be used, within an organisation, to apply a corporate signature, rather than an individual's signature.

5.2 WS-NRExchange architecture

This section describes the WS-NRExchange architecture. The architecture is based on Web service technologies; this section describes how each of the technologies is used and how WS-NRExchange utilises various services.

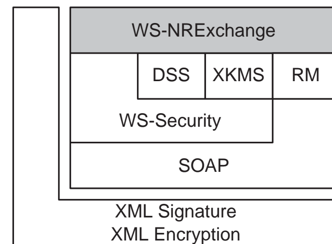


Figure 5.8: WS-NRExchange and Web service standards

To place the following discussion in context, Figure 5.8 shows how various XML and Web service standards support WS-NRExchange.

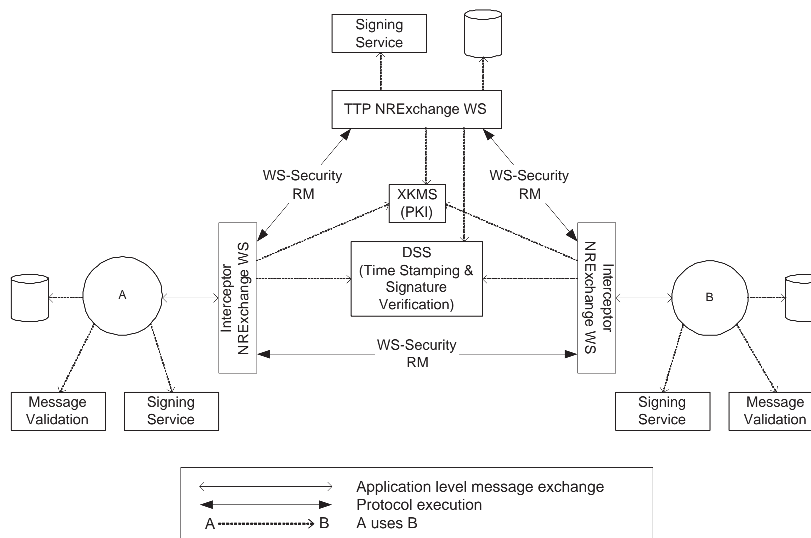


Figure 5.9: WS-NRExchange architecture

Figure 5.9 shows the interactions between the various components and services that make up the implementation. *TTP*, *A* and *B* each provide an NRExchange Web service that manages their

participation in non-repudiation protocols. Each Web service exposes the same interface for protocol execution. At A and B this service is deployed as an interceptor that mediates Web service interactions that require fair non-repudiation. This interceptor may be co-located with the local application that uses it or may, for example, be part of a corporate firewall service. The end-users, A and B, may themselves be Web services or Web service clients or both. The NRExchange services access additional local services for signing evidence, message persistence and application-level validation. The signing service is required to apply signatures to the parts of messages that have not already been signed[1]. This service may be an implementation of DSS or some other mechanism for obtaining private keys to apply signatures as defined by WS-Security. Persistence is required to meet fault tolerance requirements and also for audit. The NRExchange services also accesses trusted timestamping services and public key management services (DSS and XKMS services provided by third parties). For protocols that use an inline TTP, trusted timestamps may optionally be applied by the DA Web service.

As described in Section 5.3.1, a WSDL interface has been defined for the interaction between NRExchange services. The messages exchanged comply with the WS-Security specification. The services are written to the DSS and XKMS specifications for access to trusted timestamping, signature verification, public key life-cycle management etc.

The NRExchange Web service also provides a local interface to allow registration of application-specific listeners for message validation and other events. A message validation listener may trigger arbitrarily complex validation of a business message. If no validation listener is registered, then the NRExchange service assumes that a message is valid with respect to business contract. Registration of event listeners allows notification of protocol-related events. For example, an application can register to receive notification of zero or more of the acknowledgements generated by the protocols.

5.3 Protocol messages

This section describes the interface to to the WS-NRExchange service and the structure of the protocol messages exchanged during the execution of a non-repudiation protocol. The method for generating protocol messages is also presented. Example protocol messages can be found in Section A.3.

5.3.1 WS-NRExchange interface

Figure 5.10 shows an extract of the WSDL of an NRExchange Web service showing the operations that are exposed to other NRExchange services for protocol execution.

[1] It is possible, for example, that the message body or documents attached to a message have been signed at the application level, in which case the NRExchange service does not need to countersign.

```
<portType name="NRExchange">
 <operation name="processMessage">
  <input message="ProtocolMessage"/>
 </operation>
 <operation name="abort">
  <input message="ProtocolStateMessage"/>
 </operation>
 <operation name="resolve">
  <input message="ProtocolStateMessage"/>
 </operation>
 <operation name="setProtocolState">
  <input message="ProtocolStateMessage"/>
 </operation>
</portType>
```

Figure 5.10: Extract of NRExchange WSDL

Participants in a non-repudiation protocol use the `processMessage` operation to exchange protocol messages with each other. The sender provides a protocol message for the receiver to process according to a specified non-repudiation protocol. Message elements are defined in a related XML Schema and are sufficiently general and extensible to allow all protocol execution between NRExchange services to be conducted using the `processMessage` operation.

The `abort` and `resolve` operations are for pro-active termination (see Section 3.2.4). These operations are typically used by a *TTP* to inform another participant service that an identified protocol run has been aborted or resolved with the given protocol state. Invocation of these operations may result in execution of a new non-repudiation protocol using the protocol execution operations — assuming both end users still want to undertake the original interaction.

The SOAP binding for the NRExchange service specifies two types of message:

1. Protocol messages that are exchanged during execution of an exchange protocol or of an exception handling sub-protocol using `processMessage` and, optionally, `processRequest`; and

2. Protocol state (housekeeping) messages inform participants of the outcome of an exception handling sub-protocol.

Both types of message use a WS-Security header to carry security tokens such as: signatures over evidence; timestamps; credential and key information; security context and access control information.

5.3.2 Protocol message format

As shown in Figure 5.11, protocol messages must have a NRExchangeProtocol header. This is an extensible container for non-repudiation protocol data items that are defined in the NRExchange XML schema. The NRExchange schema specifies that any NRExchangeProtocol header must have protocol name, runId and messageNumber attributes. The protocol name is a URI that serves to

```
<soapenv:Envelope>
   <soapenv:Header>
     <wsse:Security />
     <nrex:NRExchangeProtocol name runId messageNumber
         purpose? runSequence? Id?>
       <nrex:Acknowledgement />*
       <nrex:AcknowledgementsRequired />?
       (<nrex:DigestReference Id?>
         <nrex:Reference />+
         <ds:Transforms />?
         <ds:DigestMethod />
         <ds:DigestValue />
       </nrex:DigestReference>)*
       <nrex:Participant role? nrexchangeURI? Id?/>2+
       <nrex:RandomNumber />*
       <nrex:RelatedRun />*
       (<nrex:RunIdGenerator baseURI? runId Id?>
         <nrex:DigestReference />?
       </nrex:RunIdGenerator>)?
     </nrex:NRExchangeProtocol>
   </soapenv:Header>
   <soapenv:Body />?
</soapenv:Envelope>
Key:
? = 0 or 1 occurrences
+ = one or more
n+ = n or more
* = 0 or n occurrences
```

Figure 5.11: General form of ProtocolMessage

uniquely identify the protocol, or sub-protocol, being executed and may also provide access to protocol documentation including schema that specialise the NRExchange schema. The runId is a unique identifier that is normally generated from some base URI and a random digest (a digest of a secure pseudo-random number and other associated input). The inputs to runId generation can be specified using a RunIdGenerator element. The messageNumber is a positive, non-zero double value that corresponds to the step of the protocol being executed. Depending on the protocol being executed or the step of the protocol, the following optional items may be included in the NRExchangeProtocol header: the purpose of the protocol message (ABORT, RESOLVE etc.); the participants in the protocol; the runIds of any related protocol or sub-protocol runs; random numbers; digest values; encrypted elements and message validation information. The message body, if present, contains the application data originally intended to be conveyed from sender to receiver as the body of the business message. The full protocol message schema can be found in Setion A.4.

Housekeeping messages do not have an NRExchangeProtocol header. They carry protocol state or state request information intended for another NRExchange service in the message body. The general form of a protocol state message is shown in Figure 5.12. In addition to identifying the protocol and run to which a message relates, a ProtocolState element may include information such as: the protocol status; if terminated, the termination state; and the message numbers of any messages seen by the recipient. Protocol messages may be provided as attachments to a protocol

```
<soapenv:Envelope>
 <soapenv:Header>
  <wsse:Security />
 </soapenv:Header>
 <soapenv:Body>
  <nrex:ProtocolState name runId />
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 5.12: General form of ProtocolStateMessage

state message.

5.3.3 Protocol message creation

A protocol message SOAP envelope is created by inspecting each of the tokens of the message in the protocol notation. Each of the tokens is taken individually and a transformation is applied to generate the corresponding XML. There follows a description of the transformations undertaken for each of the protocol tokens specified in Section 4.3.2.

Message

The message is not transformed, this is because a message token in WS-NRExchange is a SOAP envelope. The message becomes the basis of the protocol message. WS-Security and NRExchange-Protocol headers are added to the SOAP header of the message. Thus the original message can be derived by removing the NRExchangeProtocol header and any additional WS-Security elements.

Validation

Message validation tokens are represented using an Acknowledgement element. The following protocol header shows the transformation applied to the three validation results (*valid*, *invalid* and *not validated*).

```
<nrex:NRExchangeProtocol ...>
  <nrex:Acknowledgement type=''nrex.ack.VALIDATOR'' isValid=''(true|false) runID=''...'' >
    <nrex:ValidationInfo>
      <!-- if not valid state so here -->
    </nrex:ValidationInfo>
  </nrex:Acknowledgement>
</nrex:NRExchangeProtocol>
```

For the validation result *valid*, the *isValid* attribute is set to “true”. For the validation results *invalid* and *not validated*, the *isValid* attribute is set to “false”. For the token *not validated*, the value of the *ValidationInfo* element is set to “Not Validated”.

Participants

Every participant taking part in the protocol must be defined in every protocol message. Thus for the exchange protocol, participant blocks are present for A, B and the TTP, whereas in the exception handling sub-protocols there will only be two participant elements, one for the TTP and one for the exception handling protocol initiator. There follows the general form of the *Participant* element, where participant A is referred to as *SENDER* and B is referred to as *RECEIVER*.

```
<nrex:Participant role=’’nrex.role.(SENDER|RECEIVER|TTP)’’>
  //ID of (SENDER|RECEIVER|TTP)
</nrex:Participant>
```

Nonce

The following protocol notation:

```
n1
```

Is transformed into the following XML:

```
<nrex:Nonce id=’n1’ >
  <!-- nonce value -->
</nrex:Randomnumber>
```

This element is placed in the *NRExchangeProtocol* header. The id refers to the token name in the protocol notation and the actual nonce value is set to the element’s value.

Random number

The following protocol notation:

```
rn1
```

Is transformed into the following XML:

```
<nrex:RandomNumber id=’rn1’ >
  <!-- random number value -->
</nrex:Randomnumber>
```

This element is placed in the *NRExchangeProtocol* header. The id refers to the token name in the protocol notation and the actual random number is set to the element’s value.

Symmetric key

The following protocol notation:

```
kn
```

Is transformed into the following XML:

```
<wsse:Security>
  <wsse:Embedded>
    <wsse:BinarySecurityToken
                ValueType='...'
                EncodingType='...'
                Id='kn' >
          ...
    </wsse:BinarySecurityToken>
  </wsse:Embedded>
</wsse:Security>
```

The symmetric key is placed in the WS-security header as an embedded security token. The key itself is encoded and becomes the value of the *wsse:BinarySecurityToken* element. The type of encoding used is specified in the *EncodingType* attribute and the type of symmetric key is described by the *ValueType* attribute. The key name as stated in the protocol description is set to the *Id* attribute.

Digest

A digest is transformed into a *nrex:Digest* element which is placed in the root of the *nrex:NRExchangeProtocol* header.
The following protocol notation:

```
h(x,y)
```

Is transformed into the following XML:

```
<nrex:Digest Id=''h_x_y''>
  <nrex:Reference runId=''...'' messageNumber=''...'' URI=''...'' Id=''h_x'' type=''...''>
    <ds:Transforms>...</ds:Transforms>
  </nrex:Reference>
  <nrex:Reference runId=''...'' messageNumber=''...'' URI=''...'' Id=''h_y'' type=''...''>
    <ds:Transforms>...</ds:Transforms>
  </nrex:Reference>
  <ds:Digestmethod algorithm=''...''/>
  <ds:DigestValue>...</ds:DigestValue>
</nrex:Digest>
```

Here, the *nrex:Digest* element contains a series of *nrex:Reference* elements which point to the location of the digested tokens, x and y. The *nrex:Reference* element gives the protocol run (*runId* attribute), protocol message number (*messageNumber* attribute), and location in that protocol message (*URI* attribute) of the digested token. Each *nrex:Reference* element contains a series of zero or more *ds:transforms* elements. These are used to state what transformations have been applied to the elements before the digest was applied. For example, removing white space characters. The *ds:DigestMethod* element states what algorithm was used to create the digest. This would typically be specified in the configuration of the middleware. The *nrex:DigestValue* element contains

the digested value. This digested value is achieved by undertaking several steps. Firstly, each of the elements referenced by the *nrex:Reference* are obtained and their transforms are applied. The transformed elements are then concatenated and passed to the digest algorithm, which produces the digest value.

This mechanism for creating a digest ensures that a single digest of the concatenation of tokens x and y is produced, as opposed to creating individual digests for x and y, which will break security in some protocols. Also, it should be noted that the *nrex:Reference* may reference an element in a protocol message that has not yet been received. In which case, verification of the digest must be defferred.

Run identifier

The run identifier token from the protocol notation describes how the run identifier value is to be generated. The protocol initiator must include this value in the first protocol message. Thus the protocol initiator must be capable of generating the run identifier. The value of the run identifier is placed in the *runId* attribute of the *nrex:NRExchangeProtocol* header and is then present in every protocol message.

The *nrex:RunIdGenerator* element is present in the *nrex:NRExchangeProtocol* header of all protocol messages that have a *runId* token in the associated protocol notation. There follows an example of a *nrex:RunIdGenerator*:

```
<nrex:RunIdGenerator runId='...'>
  <nrex:Digest>
    ...
  </nrex:Digest>
</nrex:RunIdGenerator>
```

The *nrex:RunIdGenerator* is simply a wrapper around a *nrex:Digest* which states to the middleware that the value of the digest is used to create the run identifier.

Encryption

Encryption of tokens is handled differently depending on the presence of other tokens. If the encryption token is solitary, the whole SOAP envelope is encrypted. If other (plain text) tokens exist in the protocol message, only the tokens enclosed in the encryption token are encrypted.

When encrypting the whole SOAP envelope, the following procedure is followed:

1. **Encrypt the body**. The contents of the body is encrypted and replaced by an *EncryptedData* element.

2. **Encrypt the headers**. All headers are concatenated and the result encrypted. The headers are replaced by an EncryptedData element. If a WS-Security header is present, that is also encrypted.

3. **Insert WS-Security header**. A new WS-Security header is added with references to the encrypted header and body.

The following example shows the transformation of protocol message 1 from the Delivery Agent protocols described in Section 3.2.
Protocol description:

```
1 A -> DA : eP_DA(msg, rn1, NROa)
```

The resultant XML is as follows:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <xenc:ReferenceList>
        <xenc:DataReference URI=''#encHeaders''/>
        <xenc:DataReference URI=''#encBody''/>
      </xenc:ReferenceList>
    </wsse:Security>
    <xenc:EncryptedData Id='encHeaders'>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Header>
  <soap:Body>
    <xenc:EncryptedData Id='encBody'>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```

When decrypting the SOAP envelope above, the following procedure is followed:

1. **Decrypt the body**. The CipherValue of the body is decrypted and the corresponding *EncryptedData* element is replaced with the result.

2. **Decrypt the headers**. The CipherValue of the headers is decrypted and the corresponding *EncryptedData* element is replaced with the result.

3. **Discard the Security header**. The security header used in the encrypted SOAP envelope is discarded.

When the encryption element is not solitary in the protocol message, only the tokens encapsulated by the encryption token are encrypted. Each token is first transformed into XML and placed in the

appropriate place. The tokens are then grouped by common token parent. Each group is encrypted individually and the resultant EncryptedData element replaces the clear text XML. A *DataReference* element is added to the WS-Security header for each *EncryptedData* element. The following example shows the transformation of a protocol message containing encrypted and plain text tokens.
Protocol description:

```
A -> B : eP_B(rn1, rn2), rn3
```

Resultant XML

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <xenc:ReferenceList>
        <xenc:DataReference URI=’’#eP_B_rn1_rn2’’/>
      </xenc:ReferenceList>
    </wsse:Security>

    </nrex:NRExchangeProtocol>
      </xenc:EncryptedData>
      <xenc:EncryptedData Id=’eP_B_rn1_rn2’>
        <xenc:CipherData>
          <xenc:CipherValue>...</xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedData>
      <nrex:RandomNumber id=’’rn3’’>...</nrex:RandomNumber>
    </nrex:NRExchangeProtocol>
  </soap:Header>
  <soap:Body/>
</soap:Envelope>
```

In the above SOAP envelope the three random numbers (rn1,rn2,rn3) have been placed in the *NRExchangeProtocol* header. The two random numbers (*rn1*,*rn2*) requiring encryption have been encrypted and replaced by *EncryptedData* element. The two random numbers (*rn1*,*rn2*) have a common token parent, and so have been concatenated and placed in a single *EncryptedData* element.

The key required to decrypt the cipher value is inferred from the *Id* attribute which states what protocol token the *EncryptedData* element represents.

Signature

A signature is transformed into an XML signature and is placed within the WS-security header. There are two types of signatures (i) signatures over the application message and (ii) signatures over other protocol tokens. In case (i) MSG is signed. MSG is a SOAP envelope and will comprise a body and zero or more headers. For example, the protocol notation:

```
sS_A(MSG), MSG
```

Results in the following XML:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:Reference URI='#header1'>
            <Transforms>...</Transforms>
            <DigestMethod Algorithm='...'/>
            <DigestValue>...</DigestValue>
          </ds:Reference>
          <ds:Reference URI='#header2'>
            <Transforms>...</Transforms>
            <DigestMethod Algorithm='...'/>
            <DigestValue>...</DigestValue>
          </ds:Reference>
          <ds:Reference URI='#body'>
            <Transforms>...</Transforms>
            <DigestMethod Algorithm='...'/>
            <DigestValue>...</DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
      </ds:Signature>
    </wsse:Security>
    <header1 id='header1'/>
    <header2 id='header2'/>
  </soap:header>
  <soap:Body id='body'/>
</soap:Envelope>
```

Here, additional XML is added to MSG. The additional XML comprises a WS-Security header with a digital signature. The signature references each of the headers and the body.

Case (ii) caters for the signature of other protocol tokens, these are all placed in the *NRExchange-Protocol* header. The *NRExchangeProtocol* header contains other information that adds context to the protocol tokens, thus it is required that the whole *NRExchangeProtocol* header is signed. For example, the following protocol notation:

```
sS_A(rn1), rn1, rn2
```

Results in the following XML:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:Reference URI='#nrex_header'>
            <Transforms>...</Transforms>
            <DigestMethod Algorithm='...'/>
            <DigestValue>...</DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>...</ds:SignatureValue>
      </ds:Signature>
    </wsse:Security>
    <nrex:NRExchangeProtocol id='nrex_header' ... >
      <nrex:RandomNumber id='rn1' ...>
      <nrex:RandomNumber id='rn2' ...>
    </nrex:NRExchangeProtocol>
  </soap:header>
```

```
  <soap:Body/>
</soap:Envelope>
```

Here the whole NRExchangeProtocol header is referenced and signed. As a result, both rn1 and rn2 are signed, despite the protocol notation stating that only rn1 should be signed. Signing extra tokens is not of concern as it creates extra evidence and does not affect the tokens that were intended to be signed. Furthermore, it simplifies the signature process and ensures all contextual information is also signed.

In the above examples the signed tokens are also present in the same SOAP envelope as their corresponding signature. This is not a requirement; it is possible that the Signature element may reference elements from other protocol messages. When this is the case, their is still a Signature element in the WS-Security header; however, the notation described in Section **??** is used to reference an element in another protocol message.

Flags

The flags, *f(ABORT)*, *f(RESOLVE)*, *f(ABORTED)* and *f(SUCCEEDED)* are placed in the purpose attribute of the *NRExchangeProtocol* header as follows:

```
<nrex:NRExchangeProtocol
  purpose='(ABORT|RESOLVE|ABORTED|RESOLVED)'
  ...>
```

Proactive termination messages

Exception handling sub-protocols occur between the TTP and one other party that was present in the corresponding exchange protocol. At the end of the protocol these two parties have a common view of the outcome. The third party, who was involved in the exchange protocol, is unaware that an exception handling sub-protocol has occurred. Thus it is common for the TTP to pro-actively notify the third party of the outcome. This is done by the TTP sending the third party a protocol state message. The protocol state message appears as follows:

```
<soap:Envelope>
  <soap:Header>
    <wsse:Security ... />
    <nrex:NRExchangeProtocol ... />
  </soap:header>
  <soap:Body>
    <nrex:ProtocolState name=''...'' runid=''...''>
      <nrex:Status>nrex.status.TERMINATED</nrex:Status>
      <nrex:TerminationStatus>nrex.termination.(ABORTED|SUCCEEDED)</nrex.TerminationStatus
    </nrex:ProtocolState>
  </soap:Body>
</soap:Envelope>
```

The status is set to 'TERMINATED' because the protocol has completed. The completion status of the protocol ('ABORTED' or 'SUCCEEDED'), is specified in the *TerminationStatus* element. The

NRExchangeProtocol header will contain any tokens required by the third party to ensure a fair outcome.

If the TTP does not proactively notify the third party, the third party will timeout and eventually invoke an exception handling protocol in which the final outcome of the protocol run will be discovered.

5.3.4 Protocol message verification

This section describes how to obtain and verify protocol tokens from a protocol message. XPath queries are used to select specific values from the SOAP envelope. XPath queries provide a means for selecting a set of elements, attributes and values from an XML document according to some criteria. The basic notation allows the selection of elements using a directory structure like notation. Here the '/' character used to delimit each level of the hierarchy. Further filtering can be applied based on values of attributes.

Message

The application message does not receive any transformation as it is initially a SOAP envelope. However, when transforming it into a protocol message, a NRExchangeProtocol header is added and possibly some security tokens into the WS-Security header. These need to be removed to obtain the original application message. The following XPath query returns all (there should only be one) NRExchangeProtocol headers.

```
//nrex:NRExchangeProtocol
```

The '//' states that any element fulfilling the forthcoming query should be returned. Therefore this query will return all elements named 'nrex:NRExchangeProtocol' within the SOAP envelope.

Participants

The following query is used to extract a particular participant element form the protocol message.

```
//nrex:NRExchangeProtocol/nrex:Participant[@role='nrex.role.RECEIVER']/text()
```

The value of the role attribute is set dependent on which participant element is to be received. This query can be run three times, once for each type of participant. Here all 'nrex:NRExchangeProtocol' headers are selected with a 'nrex:Participant' child. A further filter is applied with the "[@role=-'nrex.role.RECEIVER']" notation. This states that only participant elements containing an attribute called 'role' with value 'nrex.role.RECEIVER' should be returned. The '/text()' notation states that the text value of the element should be returned, rather than the element itself.

Nonce

The following syntax returns the nonce's value.

```
//nrex:NRExchangeProtocol/nrex:Nonce[@id='n1']/text()
```

Here the '[@id='n1'] ' syntax ensures that the nonce returned is that with a specific id attribute value. The id attribute is used to associate the value of the nonce with a particular nonce specified in the protocol description. In this example, the nonce returned is the one associated with the token 'n1' from the protocol description.

Random number

The following syntax returns the random number's value.

```
//nrex:NRExchangeProtocol/nrex:RandomNumber[@id='rn1']/text()
```

Here the '[@id='rn1'] ' syntax ensures that the random number returned is that with a specific id attribute value. The id attribute is used to associate the value of the random number with a particular random number specified in the protocol description. In this example, the random number returned is the one associated with the token 'rn1' from the protocol description.

Symmetric key

Several pieces of information need to be obtained when acquiring a symmetric key. Each piece of information is with respect to a particular instance of a symmetric key. This is because the protocol message could contain several symmetric keys. Each symmetric key has its own 'BinarySecurity-Token' element, the id attribute is used to associate it with a particular symmetric key from the protocol description. Each XPath query specifies the value of the id attribute to ensure the relevant values are returned.

The following XPath query returns the value of the actual symmetric key.

```
//wsse:Security/wsse:Embedded/wsse:BinarySecurityToken[@id='kn']/text()
```

The following XPath query returns the type of the key.

```
//wsse:Security/wsse:Embedded/wsse:BinarySecurityToken[@id='kn']/@ValueType
```

The following XPath query returns the encoding type. Due to the symmetric key being originally binary, it must be encoded before placing it into a SOAP envelope.

```
//wsse:Security/wsse:Embedded/wsse:BinarySecurityToken[@id='kn']/@EncodingType
```

Digest

Several pieces of information need to be obtained when acquiring a digest. The *id* attribute of the *nrex:Digest* element is used to specify which *nrex:Digest* element is of interest. The *id* attribute is the same as that of the digest token in the protocol description (with parenthesis replaced with underscores and the final underscore removed).

The following XPath query returns all the *nrex:Reference* elements from the *nrex:Digest* with the required id.

```
//nrex:NRExchangeProtocol/nrex:Digest[@id=''...'']/nrex:Reference
```

Each *nrex:Reference* element should then be handled individually to obtain and transform the referenced element. The referenced element could be in a different protocol message of a different protocol run. Thus, the run identifier, message number and location in the protocol message must be found.

The following XPath query should be applied to the *nrex:Reference* element to obtain the value of the *runId* attribute.

```
//nrex:Reference/@runId
```

The following XPath query returns a message number from the *nrex:Reference* element:

```
//nrex:Reference/@messageNumber
```

The following XPath query returns the *URI* value the *nrex:Reference* element:

```
//nrex:Reference/@URI
```

The following XPath query returns a collection of values for each *ds:Transforms* element.

```
//nrex:Reference/ds:Transforms/text()
```

Once each of the referenced elements have been obtained and transformed they are concatenated and then passed to a digesting algorithm. The chosen digest algorithm is obtained using the following XPath query:

```
//nrex:NRExchangeProtocol/nrex:Digest[@id=''...'']/ds:DigestMethod/text()
```

The result of applying the digest is then compared for equality with the claimed digest value. The claimed value of the digest is obtained using the following XPath query:

```
//nrex:NRExchangeProtocol/nrex:Digest[@id=''...'']/ds:DigestValue/text()
```

Run identifier

An *nrex:RunIdGenerator* is a wrapper on an *nrex:Digest* and, as a result, is validated in the same fashion as an *nrex:Digest* element. The only difference is that the XPath queries applied to the protocol message need to be modified to take into account the wrapping of the *nrex:RunIdGenerator* element. For example, the following XPath query is used to obtain all the *nrex:Refernce* elements from the enclosed *nrex:Digest* element:

```
//nrex:NRExchangeProtocol/nrex:RunIdGenerator/nrex:Digest/nrex:Reference
```

Encryption

To decrypt the elements of a SOAP envelope, a list of encrypted elements must first be obtained. This is done with the following XPath query.

```
//wsse:Security/wsse:ReferenceList/wsse:DataReference/@URI
```

The next step is to decrypt each of these elements. The following XPath query returns a particular *EncryptedData* block, given an *Id* found by the previous query.

```
//xenc:EncryptedData[@Id='...']/xenc:CipherData/xenc:CipherValue/text()
```

This value is decrypted and the result replaces the *EncryptedData* block. The recipient of the protocol message knows which elements can be decrypted as the protocol notation states which key was used to encrypt each token.

Signature

A signature consists of one or more digests encrypted by the signatory. To verify a signature element each of the enclosed digests must first be verified. On successful verification, the signature value is obtained and decrypted using the signatory's public key. The decrypted value is then compared to the value of the *SignedInfo* block.

The following XPath query returns the signature value:

```
//wsse:Security/ds:Signature/ds:SignatureValue/text()
```

The following XPath query returns the signedInfo block:

```
//wsse:Security/ds:Signature/ds:SignedInfo
```

Flags

Flags are passed in the purpose attribute of the NRExchangeProtocol header. The following XPath query returns the value of the purpose element:

```
//nrex:NRExchangeProtocol/@purpose
```

Validation

To establish whether an application message was valid, the following XPath query is used:

```
//nrex:NRExchangeProtocol/nrex:Acknowledgement[@type='nrex.ack.VALIDATOR']/@isValid
```

If an application message is found to be invalid, it could be due to it not actually have been validated. If this is the case, the text of the *ValidationInfo* element would be set to 'Not Validated'. The following XPath query returns the text value of the *ValidationInfo* element:

```
//nrex:NRExchangeProtocol/nrex:Acknowledgement[@type='nrex.ack.VALIDATOR']/nrex:ValidationInfo/text()
```

Pro-active termination messages

On receiving a protocol status message, the protocol can be discovered to be terminated by running the following XPath query.

```
//nrex:ProtocolState/nrex:status/text()
```

If the protocol has been terminated, the outcome of the termination is discovered by running the following XPath query.

```
//nrex:ProtocolState/nrex:TerminationStatus/text()
```

5.4 WS-NRExchange service implementation

This section describes the implementation of the WS-NRExchange service. Each participant is required to host this service before participating in a non-repudiation protocol. The WS-NRExchange service is a Web service that is hosted from a Web services enabled web server. The implementation is based on the non-repudiable messaging stack presented in Section 3.1. Each layer communicates via messages. A particular protocol is run by exchanging a series of messages with the layer bellow. It is assumed that all nodes are fault tolerant and that all messages will arrive in a finite but unknown amount of time.

5.4.1 Domain specific MEP layer

The domain specific MEP layer is implemented by mapping the messages of a particular domain specific MEP onto one or more runs of a generic MEP protocol. If two message exchange pattern instances are used, an identifier must be present in both runs, such that it is apparent from any non-repudiation evidence that both runs form a single domain specific MEP. This identifier is also used by the generic MEP layer to provide cross protocol run context.

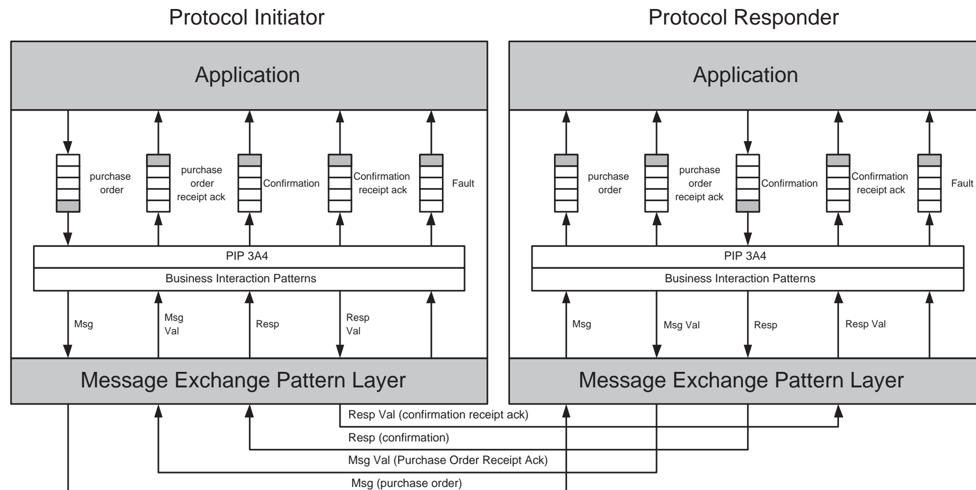


Figure 5.13: domain specific MEP layer implementation

Figure 5.13 shows the implementation of the *domain specific MEP* layer from the point of view of both the protocol initiator and the protocol responder. The domain specific MEP layer shown in the diagram supports a single domain specific MEP (PIP3A4 presented in Section 3.1.1.1). In practice other domain specific MEPs would also be supported. This particular implementation of PIP3A4 exposes four queues for the protocol initiator and four for the protocol responder. It should be noted that the implementation of this PIP undertakes validation of the *purchase order* and the *confirmation* messages. After undertaking the validation, the PIP implementation notifies the application of the result and also passes the result to the MEP layer (such that it can be passed to the other participant). A description of each queue follows:

Protocol Initiator

- **Outgoing 'purchase order' queue.** The protocol initiator application begins the PIP3A4 protocol by depositing the purchase order onto this queue.
- **Incoming 'purchase order receipt ack' queue.** The protocol responder's validation of the purchase order is returned to the application via this queue.
- **Incoming 'confirmation' queue.** The protocol responder's decision to accept, reject or pend the purchase order is returned to the application via this queue.
- **Incoming 'confirmation receipt ack' queue.** The PIP3A4 implementation undertakes validation of the confirmation message. The result is placed on this queue such that the application can discover the result.

Protocol Responder

- **Incoming 'purchase order' queue.** The protocol responder receives the purchase order from this queue.
- **Incoming 'purchase order receipt ack' queue.** The PIP3A4 implementation undertakes validation of the purchase order message. The result is placed on this queue such that the application can discover the result.
- **Outgoing 'confirmation' queue.** The protocol responder application decides whether to accept, reject or pend the purchase order and submits an appropriate PIP3A4 a confirmation message to this queue.
- **Incoming 'confirmation receipt ack queue.** The protocol initiator's validation of the confirmation message is returned to the application via this queue.

The implementation of PIP3A4 then maps each of these messages onto a single run of *validated request/response* provided by the generic MEP layer. Figure 5.14 shows this mapping.

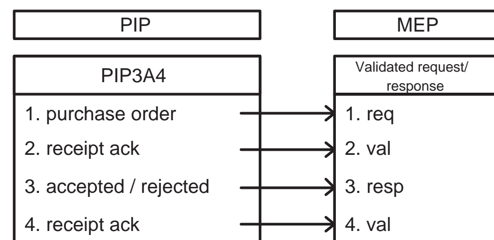


Figure 5.14: PIP3A4 to message exchange pattern mapping

As stated earlier, other domain specific MEPs would be hosted by the business interaction layer. These would be implemented using potentially different generic MEPs and would expose different queues as an interface to the application. However, the implementation would be similar to that described for PIP3A4. For example, PIP3A7 (presented in Section 3.1.1.2) would expose two sets of queues, two to the protocol initiator and the other two to the protocol responder. The first set would allow the protocol initiator to submit an updated purchase order (and validation). The second set would allow the protocol responder to receive the updated purchase order (and validation). This protocol would be implemented as a *validated single message* generic MEP.

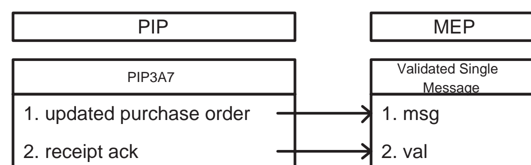


Figure 5.15: PIP3A7 to message exchange pattern mapping

Figure 5.15 shows the mapping from the PIP3A7 messages to the messages sent in the *validated single message* generic MEP.

5.4.2 Generic MEP layer

The generic MEP layer (as described in Section 3.1.2) provides four types of messaging patterns. Each generic MEP is realised by a mapping to one or two instances of a message delivery primitive (MDP) at the protocol layer (presented in Section 3.1.3). If two protocol MDP instances are used, an identifier must be present in both runs, such that it is apparent from any non-repudiation evidence that both runs form a single generic MEP. This identifier is also used by the generic MEP layer to provide context over both runs of the MDP.

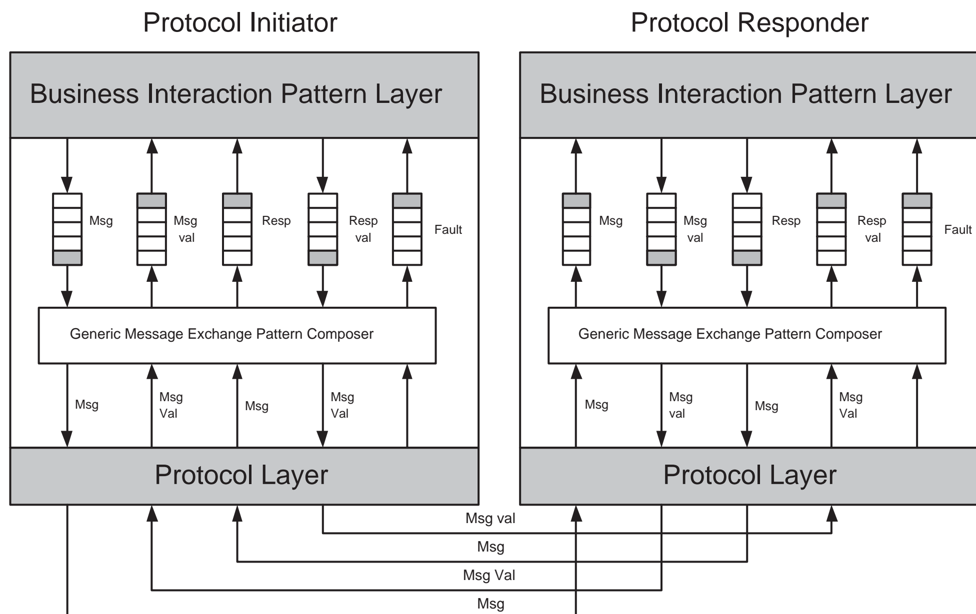


Figure 5.16: Message exchange pattern layer implementation

Figure 5.16 shows the implementation of the *generic MEP* layer from the points of view of both the protocol initiator and protocol responder. Here the generic MEP composer is responsible for managing the mappings between generic MEP and the underlying MDP run(s). The generic MEP layer exposes eight queues to the business exchange pattern layer. Four are specific to the protocol initiator and the other four are specific to the protocol responder.

Protocol Initiator:

- **Outgoing 'Msg' queue.** The protocol initiator deposits the first message of the protocol run onto this queue. The chosen generic MEP is also stated here.

- **Incoming 'Msg val' queue.** If a generic MEP supporting validation is chosen, the protocol responder's validation of the first message returns on this queue.

- **Incoming 'Resp' queue.** If a generic MEP supporting a response is chosen, the responder's response to the first message returns on this queue.

- **Outgoing 'Resp val' queue.** If a generic MEP supporting validation of response is chosen, the initiator's validation of the response message is deposited on this queue.

Protocol Responder:

- **Incoming 'Msg' queue.** The protocol responder receives the first message of the protocol run from this queue.

- **Outgoing 'Msg val' queue.** If a generic MEP supporting validation of the first message is chosen, the responder's validation of the first message is deposited on this queue.

- **Outgoing 'Resp' queue.** If a generic MEP supporting a response is chosen, the responder's response to the first message is deposited on this queue.

- **Incoming 'Resp val' queue.** If a generic MEP supporting validation of response is chosen, the initiator's validation of the response returns on this queue.

The generic MEP composer holds multiple mappings from each generic MEP onto each MDP offered by the protocol layer. The following tables show the mappings for achieving fair non-repudiation for each of the generic MEPs. Various other mappings exist for achieving different levels of non-repudiation — these have been left out for the sake of brevity.

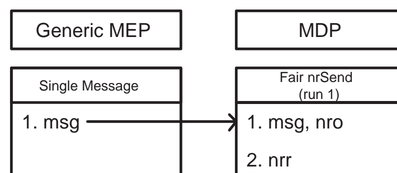


Figure 5.17: Single message to fair non-repudiation protocol mapping

Figure 5.17 shows how *Single Message* generic MEP is mapped to a single run of *fair nrSend*.

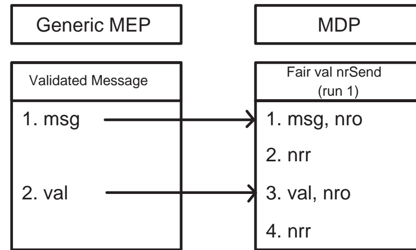


Figure 5.18: Validated single message to fair non-repudiation protocol mapping

Figure 5.18 shows how *Validated Message* generic MEP is mapped to a single run of *fair Val nrSend*.

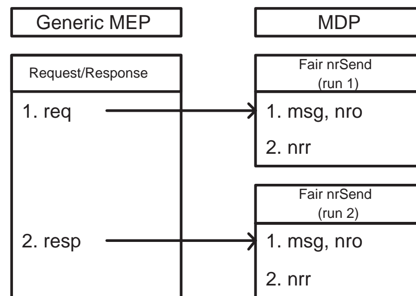


Figure 5.19: Request/response to fair non-repudiation protocol mapping

Figure 5.19 shows how *Request/Response* generic MEP is mapped to two runs of *fair nrSend*.

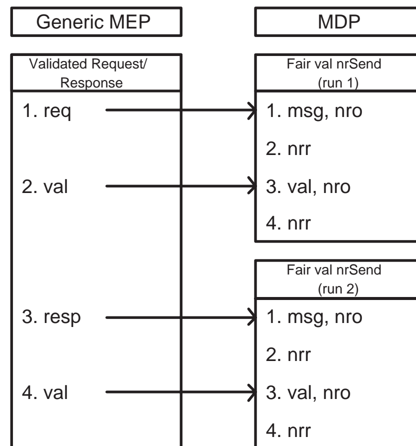


Figure 5.20: Validated Request/response to fair non-repudiation protocol mapping

Figure 5.20 shows how *Validated Request/Response* generic MEP is mapped onto two runs of *fair Val nrSend*.

5.4.3 Protocol layer

Figure 5.21 shows the internal design of the protocol layer from the perspective of both the protocol initiator and the protocol responder.

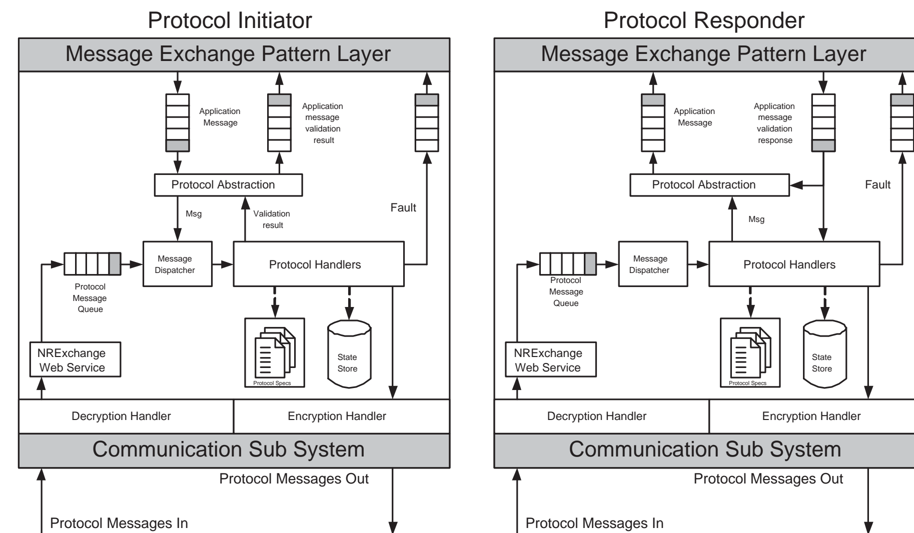


Figure 5.21: NRExchange Service Internals

A protocol run is initiated when the protocol layer receives an application message on the *Application msg* queue. In addition to the application message, some meta information is also passed. This meta information states which message delivery primitive (and protocol) to run and which participants to involve. If a message delivery primitive offering validation is required, but the chosen protocol does not support validation, two runs of the protocol must be run. The second protocol run is used to obtain validation of the message from the protocol responder. If two protocol runs are required, a common context identifier must be inserted into the application message and the validation message. The protocol abstraction module then initiates the protocol run by making a request to the message dispatcher. The message dispatcher is responsible for taking protocol begin requests (and protocol messages) and passing them to the appropriate protocol handler. One protocol handler is present for each supported protocol. On receiving a protocol begin request, the protocol handler would typically log the application message and then send the first protocol message to the appropriate participant (the first protocol message would also be logged).

The protocol layer exposes a Web service entry point to accept protocol messages from other participants (for protocols that are currently in progress). On receiving a protocol message, it is first intercepted by the *Decryption Handler*. It is the responsibility of this handler to decrypt as much of the message as possible and pass it on to the *NRExchange Web service*. The *NRExchange Web service* immediately puts the message on the protocol message queue and then listens for another

message. No processing or inspection of the message is done here. The entry point is intentionally lightweight to facilitate higher availability. The message dispatcher removes messages from the queue one at a time. The message dispatcher is responsible for undertaking some rudimentary verification. This involves checking that the message is part of a protocol supported by the NRExchange service and that the run identifier is for a valid protocol run. If this verification is unsuccessful, the message is simply dropped. On successful verification, a protocol handler is looked up for the current protocol type.

As stated earlier, each supported protocol must have an associated protocol handler. The protocol handler has a single method called *proccessMessage* that takes a protocol message and undertakes some appropriate processing. For the Web services implementation, this protocol message is a SOAP envelope. The protocol layer is agnostic to the implementation of the protocol handler. It is the responsibility of the protocol implementor to design an appropriate mechanism for handling protocol messages. A rigorous protocol handler implementation is presented in Section 5.5. It is the responsibility of the protocol handler to verify the protocol message and then act upon it. Each protocol handler has access to protocol specifications and a state store. The protocol handler notifies the protocol abstraction module when it obtains an application message (as a protocol responder) or validation of an application message (as a protocol initiator).

The protocol specifications can (optionally) be used by a protocol handler to discover what a particular protocol message should contain. The protocol specifications are used by the protocol handler presented in Section 5.5 for automatic message generation and verification. Whenever a correct protocol message is sent or received, it must be placed in the state store. The state store is a persistent collection of protocol messages which can be used to establish the state of a current protocol run. This is done by observing the sequence of messages previously exchanged. The state store is also responsible for maintaining the non-repudiation evidence.

When the protocol abstraction module receives an application message from the protocol handler, it places it on the outgoing *Application Message* queue. If the chosen message delivery primitive requires validation, the consumer of the application message should validate the message and return the result on the incoming *Application message validation response* queue. If the chosen protocol does not support validation, the protocol abstraction module will consume this validation response and initiate a second protocol run to return the validation response to the originator of the application message.

5.5 Finite state machine based protocol handler

This section describes one pattern for implementing a protocol handler. The pattern uses the techniques described in sections 5.3.3 & 5.3.4 for generating and verifying protocol messages. The

FSMs presented in Section 4.2 are used to decide on the correct course of action, in response to the receipt of valid protocol messages. In using these techniques a rigorous implementation is produced.

The protocol handler is invoked when a new protocol message is received, for the associated protocol. Protocol message handling begins with a series of verification steps. Should any of these steps fail, the protocol message is dropped. After protocol message verification, some housekeeping steps are undertaken and finally the finite state machine is notified with an event. There follows a more detailed discussion of each step.

1. **Signature verification**. Signature verification would either be done autonomously or by delegation to the DSS.
2. **Token verification**. The new message is inspected such that the protocol message type is known. Each token is then verified using the verification principels presented in Section 5.3.4.
3. **Expand knowledge**. The protocol message is inspected for new tokens. The values of these tokens are placed in a table and used when verifying and generating future protocol messages.
4. **Message Persistence**. The protocol message is persisted and forms part of the non-repudiation evidence.
5. **FSM transition**. The protocol message is now known to be valid and a state transition can be made. This is done by inspecting the protocol message number and raising an event on the finite state machine implementation.

The finite state machine implementation will expose several event triggers. These events triggers allow the protocol handler to inform the finite state machine when an event has occurred. There are two types of events:

- **Receive a particular protocol message**. This event will cause the finite state machine to move to another state (and undertake an action), or remain in the same state. The latter will be the case if an unexpected protocol message is received.
- **Detect an exception.** This will cause the finite state machine to move to an exception handling state. An action associated with this transition is run to invoke the exception handling sub-protocol.

If the finite state machine receives an expected message (one that enacts an event causing transition to a different state), the state will change and some action will be invoked. Three types of actions exist:

- **Send one or more protocol messages**. This action occurs if the participants next involvement in the protocol is to send a protocol message. Numerous protocol messages could be

sent until the participant's next involvement is to receive a message or until the protocol has completed.

- **Invoke exception handling sub-protocol**. This action occurs when the *detect exception* event occurs (providing the FSM is in a state supporting exception handling). This action begins a new protocol with its own protocol handler.
- **Notify success**. This action notifies the protocol layer that the protocol has completed successfully.
- **Notify failure**. This action notifies the protocol layer that the protocol failed to complete successfully.

5.5.1 Protocol handler implementation

The protocol handler is implemented as a short method that undertakes the five steps discussed at the beginning of this section. The pseudo code for the protocol handler follows:

```
handleMessage(protocolMsg)
{
  if (!verifySignatures(protocolMsg))
  {
    //drop message
   return;
  }
  if (!verifyTokens(protocolMsg))
  {
    //drop message
   return;
  }
  expandKnowledge(protocolMsg);
  persist(newMsg);
  fsm.receiveM<number>();
}
```

The FSM implementation exposes a method for each event. It is one of these event methods that is called at the end of the method. The actual event will depend on the protocol step that the protocol message represents. For example, if a protocol message representing step 4 is received, the event *receiveM4* is raised on the FSM implementation by calling the *receiveM4* method on the *fsm* object.

5.5.2 FSM implementation

Several automated FSM implementation generation tools exist [61, 1]. Nunni FSM Generator is one such tool. The tool takes a FSM transition table and generates source code. The source code

State	Event	New State	Action
S0	recv_M1	S1	send_M2_M3
S1	recv_M4	S2	send_M5_M6
S2	recv_M7	S3	send_M8_M9
S3	exception	HE	inv_EH_Protocol
S4	exception	HE	inv_EH_Protocol

Table 5.1: State transition table for Delivery Agent protocol - participant DA

exposes methods for triggering events. On triggering an event, the current state is inspected and a state transition is made (providing one exists for that particular event). On triggering an event an action may be invoked. A callback method is created for each event. These callback methods are left black, and must be filled in by the developer.

Table 5.1 shows the state transition table for the delivery agent's FSM for the Delivery Agent exchange protocol. The table has a row for each state transition. The first column shows the current state. The second column represents the event needed to trigger the transition. The third column holds the name of the new state to move to. The fourth column defines the action to invoke as part of the transition.

```
public interface ProtocolFSM
{
  //Events
  public void recv_M1();
  public void recv_M4();
  public void recv_M7();

  //Actions
  public void send_M2_M3();
  public void send_M5_M6();
  public void send_M8_M9();
}
```

The above code shows the interface to the FSM implementation produced by the Nunni FSM Generator tool. The three event methods are called to trigger events on the FSM. The three action methods are called when the FSM triggers the associated action. In this example, the *send_ M2_ M3* method would generate a protocol message for step 2 of the Delivery Agent exchange protocol and then send it to the intended participant. The techniques described in Section 5.3.3 are used to generate this message. This action also sends a step 3 protocol message in the same way.

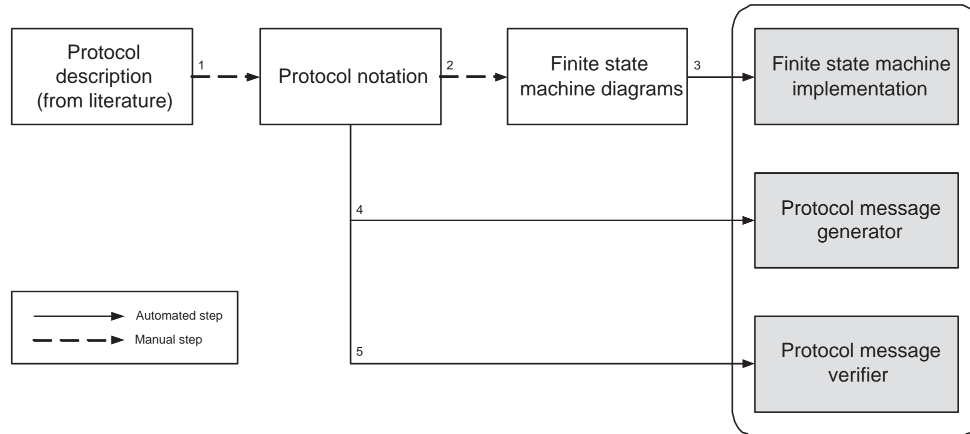


Figure 5.22: Protocol implementation process

5.6 Evaluation

This Section evaluates the solution provided by the thesis. It describes, (i) limitations in the protocol implementation process; (ii) configuration of the WS-NRExchange framework; and (iii) how confidence in the correctness of the WS-NRExchange framework can be achieved.

5.6.1 Automation

Figure 5.22 shows the process of taking a protocol description from the literature and creating an implementation, suitable for execution within the WS-NRExchange middleware. The dashed arrows represent a manual step and the solid arrows represent an automated step. Each step is as follows:

1. **Protocol description → Protocol notation**. This step is manual. Here the protocol description, from the literature, is inspected and re-written in the notation described in Section 4.3.

2. **Protocol notation → Finite state machine diagrams**. This step is manual. Here the protocol notation, is inspected and a set of FSMs are produced. The generic FSMs presented in Section 4.2 should be used as a template.

3. **Finite state machine diagrams → Finite state machine implementation**. This step is automated. The diagram is first converted into a state transition table. The state transition table is then passed to a tool for creating Java implementations of FSMs. This process is described in Section 5.5.

4. **Protocol notation → Protocol message generator**. This step is automated. Routines exist for constructing a protocol message from the protocol notation. This process takes each

token and represents it as a fragment of XML. Each XML fragment is then placed inside a SOAP based protocol message. This process is described in Section 5.3.3.

5. **Protocol notation → Protocol message verifier**. This step is automated. Queries exist for obtaining values of protocol tokens from a SOAP based protocol message. The protocol notation is used to establish which tokens are expected. Then particular queries are used to obtain the values of each token in turn. This process is described in Section 5.3.4.

5.6.2 Configuration

Configuration of the WS-NRExchange framework has not been addressed in this thesis.

- **Security Tokens.** Configuration of security token implementations is required. This is used to state what implementation must be used for each of the tokens used in a protocol's notation. For example, to state what encryption and signature schemes should be used. This is also required for protocols that use special signature schemes.
- **Protocol Hierarchy**. Configuration of protocol mappings is required. This would be used to specify how each of the generic MDPs and domain specific MDPs are composed.
- **Service Location**. A configuration mechanism is required to state whether services (such as audit and signature) are in house or third party. In the case of third party services the configuration should state their location and also give the required information needed to authenticate with the service.

5.6.3 Confidence in implementation

This Section describes two mechanisms that can be used to increase confidence in the WS-NRExchange framework. They do not however, prove that the framework is correct.

Work by Kleiner and Roscoe [44] takes security protocols that have been designed for Web services and reverse engineers some formal notation from them. The protocols targeted where never designed in any formal notation. Instead they where designed using sequence diagrams and plain English. The reverse engineering process involves observing a run of the protocol and obtaining each protocol message. These protocol messages are parsed for recognised tokens such that a protocol specification (similar to that used in the literature) can be obtained. The intention is to then verify the protocol specification. However, the authors found some protocols that were so obviously flawed that visual inspection was all that was needed to discover their flaws.

Round trip verification could be achieved using this process. Real protocol messages would be observed from a protocol developed and run under the WS-NRExchange framework. These protocol messages would then be reverse engineered using the process presented in [44]. The resulting protocol

specification could then be compared to the original protocol specification. If they appear analogous, we can be confident that the implementation process and execution framework is correct.

A. Gordon et. al. [31] provide a tool for inspecting single WS-Security protected SOAP messages and checking them for known flaws. This process can be used to check some number of protocol messages produced by the WS-NRExchange framework. The confidence in the correctness of the produced protocol messages will be proportional to the number and variation of the protocol messages inspected. This process takes each SOAP message in isolation, so per protocol run inspection cannot be achieved.

5.7 Summary

This chapter described the implementation of a framework for executing non-repudiation protocols. A high level view showed how the various services used by the framework fitted together. The framework utilises many current Web services standards; these were discussed. It was shown how a protocol message could be represented as a SOAP envelope and how, given a protocol described in a specific notation (described in Section 4.3), the protocol messages could be automatically generated and verified.

In Section 5.4 the implementation of the WS-NRExchange Web Service was presented. This service implemented each level of the non-repudiation hierarchy presented in Chapter 3. The chapter concluded by presenting a rigorous pattern for developing protocol handlers. This protocol handler implemented the FSMs presented in Section 4.2.

Example protocol messages and the protocol message schema can be found in Appendix A.

Chapter 6

Summary and Future Work

This thesis has developed a framework for executing non-repudiation protocols. The framework facilitated a flexible mechanism for creating a wide selection on non-repudiable message exchange patterns. It was shown how a non-repudiation protocol could be represented using some formal notation. This notation was used to derive finite state machines that model each participants' execution of the non-repudiation protocol. It was also shown how protocol messages could be automatically created and verified from the notation.

A concrete implementation of the framework was presented. This framework also showed how the finite state machines could be implemented and how protocol messages could be constructed and verified.

6.1 Non-repudiable business protocols (Chapter 3)

This chapter presented a non-repudiation protocol hierarchy which allows for a large variety of application requirements. These requirements are fulfilled by providing a multitude of messaging patterns coupled with varying levels of non-repudiation and validation guarantees. Furthermore, these messaging patterns are (to some extent) agnostic to the choice of non-repudiation protocol. As a result, the underlying non-repudiation protocol can be modified without disrupting the application.

To facilitate configurable service-based non-repudiation, a delivery agent based protocol was presented that provides fair non-repudiation of application message and validation of the application message. It was shown how this protocol could be modified to facilitate lightweight end users or lightweight delivery agents. Furthermore, it was shown how exception handling sub-protocols can be used to ensure fairness is achieved in the presence of unexpected behaviour.

6.2 Protocol representation (Chapter 4)

In this chapter it was shown how a class of TTP-based fair non-repudiation protocols could be modeled in order to facilitate a rigorous implementation. Correct progress through a protocol was managed using a collection of FSMs. State transitions in the FSM were undertaken on receipt of a valid protocol message. Such a transition would then trigger an action to send zero or more valid protocol messages. The chapter presented a rigorous means for generating and verifying protocol messages. This process relied on the protocol specification being canonicalised. Canonicalisation was achieved by representing the protocol using some specific notation. The process of representing the FSM, creating protocol messages and verifying protocol messages is automated from the canonicalised protocol specification. This increases confidence that the running protocol is faithful to the protocol's specification.

6.3 Middleware implementation (Chapter 5)

This chapter described the implementation of a framework for executing non-repudiation protocols. A high level view showed how the various services used by the framework fitted together. The framework utilises many current Web services standards; these were discussed. It was shown how a protocol message could be represented as a SOAP envelope and how, given a protocol described in a specific notation (described in Section 4.3), the protocol messages could be automatically generated and verified.

In Section 5.4 the implementation of the WS-NRExchange Web Service was presented. This service implemented each level of the non-repudiation hierarchy presented in Chapter 3. The chapter concluded by presenting a rigorous pattern for developing protocol handlers. This protocol handler implemented the FSMs presented in Section 4.2.

6.4 Future work

There are several directions of future work that could be taken from this thesis. These are detailed in the following sections.

6.4.1 Deadlines and Timing

Deadlines allow a participant to specify how long they are willing to wait for a response and are used to ensure timeliness (i.e. preventing a participant waiting indefinitely). Timeliness can be achieved simply by having a participant wait until the deadlines are reached and then quit the interaction. However, quitting during a fair non-repudiation protocol may jeopardise fairness. As a

result, exception handling protocols must be executed.

To ensure timeliness, it must be possible to specify deadlines for receipt of each message received during domain specific MEPs, generic MEPs and MDPs. The protocol level would raise an exception to the currently running protocol's FSM. This would enact the exception handling FSM, which would bring the protocol to completion.

Two non-repudiation protocols [84, 91] use time tokens as a mechanism for ensuring timeliness and for adding timing information to evidence. It is not clear how a general handling of time tokens can be achieved. As a result, the protocol representation techniques presented in Chapter 4 do not support time tokens. However, if time tokens were supported, they could possibly be used to set common deadlines between participants.

6.4.2 Fault tolerance

In order to maintain fairness, well behaved participants to a non-repudiation protocol must be capable of executing the protocol correctly. If a participant were to fail during a protocol's execution, fairness may not be maintained. It could be possible that the failure causes loss of non-repudiation evidence. It can't be assumed that other participants will hand over the evidence for a second time, especially if the failure has given them an advantage. Thus, non-repudiation services should be fault tolerant to prevent loss of fairness. The implementation of the NRExchange service is not fault tolerant. Rendering the service fault tolerant would require substantial work which would distract from the ideas presented in this thesis.

The requirement for fault tolerance in non-repudiation is discussed further in [51, 29].

6.4.3 Reliable messaging

Many fair non-repudiation protocols are capable of handling loss of protocol messages. However, in doing so they execute a separate sub-protocol which enacts the help of a TTP. This is quite an expensive means for handling lost messages. By introducing a reliable messaging scheme, it can be guaranteed that protocol messages are received exactly once or at least once. As a result, the TTP does not need to be contacted in case of communication failures. WS Reliable Messaging and WS-Reliability both provide guaranteed message delivery of SOAP messages. Implementing reliable messaging at a lower level prevents the non-repudiation service from needing to worry about message reliability and can simply assume that all messages will reach their destination. However, this approach is rather inefficient. The NRExchange service duplicates a lot of effort present in reliable messaging implementations. Thus it may be a good idea to implement reliable messaging at the same level as the NRExchange service.

6.4.4 DA Protocol verification

The protocol representation presented in Section 4 assumes a correct non-repudiation protocol specification. The Coffey and Saidha protocol [16] that formed the basis of the DA protocol (Section 3.2) has been formally verified in [17]. The changes Zhou and Gollmann propose were published in [92] but the resultant protocol was not verified. Furthermore, the changes made in [?] and in this thesis have also not been formally verified. Formal verification of the DA protocol has been left as further work.

The lack of formal verification does not affect the validity of this thesis. The ideas proposed in this thesis ensure a faithful implementation of the designed protocol. Thus if this design is incorrect, the implementation will also be incorrect. However, the process of taking the design and producing an implementation remains correct; which is the contribution of this thesis.

6.4.5 Multi-party non-repudiation

The interactions presented in this thesis are only concerned with two party non-repudiation. Here one party wishes to send a message to one other participant. Multi-party non-repudiation is a generalisation of two party non-repudiation. Here one participant wishes to send a message to one or more other participants. Fair multi-party non-repudiation protocols ensure fairness between the message originator and each recipient. To ensure fairness, either (i) a particular recipient acquires the message and non-repudiation of origin, and the message originator acquires non-repudiation of receipt for that participant, OR (ii) that particular recipient does not receive the message or non-repudiation of origin and the originator does not receive non-repudiation of receipt from that particular recipient. Some multi-party fair non-repudiation protocols ensure that either all recipients receive the message (and non-repudiation of origin) or none of them do. Whereas others allow a sub-set of recipients to acquire the message (and non-repudiation of origin). Fair multi-party non-repudiation is a new research area and there are few publications on this subject [63, 64, 45, 48, 52].

Fair multi-party non-repudiation protocols have a different interface to two party fair non-repudiation protocols. The first difference is that the recipient is replaced with a set of recipients. The second difference is that a set of non-repudiation of receipts is obtained (as opposed to a single receipt), one for each recipient that successfully completed the fair exchange. As a result, none of the MDPs currently present at the protocol level of the non-repudiation protocol hierarchy, support multi-party non-repudiation. It should be possible to develop multi-party versions of all the MDPs, should they be required.

To develop a multi-party fair NR-Send MDP, the following needs to be undertaken:

- **Protocol implementation**. A fair multi-party non-repudiation protocol must be developed. This implementation would be located at the protocol level of the non-repudiation protocol

hierarchy.

- **New MDP**. A new MDP must be presented by the protocol level. This MDP would take a single message and a set or recipients; it would return a set of receipts.

- **New generic MEP**. A new generic MEP must be presented by the generic MEP level. This generic MEP would take a message and set of recipients; it would return a set of acknowledgements. This set would state which participants received the message. This generic MEP is required as the application does not interface directly to the protocol level.

The protocol representation techniques presented in Chapter 4 do not support fair multi-party non-repudiation protocols. However, multi-party non-repudiation protocols are simply a generalisation of two party non-repudiation protocols. Therefore, it should be possible to extend the ideas in this chapter to support multi-party variants. However, the research into fair multi-party non-repudiation protocols is fairly recent and, as a result, there are not currently enough protocols present to establish a suitably general scheme for representing them as finite state machines.

6.4.6 NR Information sharing

Work by colleagues described in Section 2.3.5 presents a scheme for achieving controlled access to shared information. This service can be implemented as part of the non-repudiation protocol hierarchy. The bulk of the NR-Information Sharing service would be implemented as a domain specific MEP. This pattern would allow the application to read and write to the shared information. When writing to the shared information state, a two phase commit protocol is executed. This protocol would be implemented as a generic MEP. The generic MEP first proposes a new information state to every other participant. Each participant then validates the request and returns the result. The MEP collates the responses and then notifies each participant of the result. This MEP can be composed of several validated NR-Send MDPs to propose the new state to each participant. The validation part of the MDP is used to return the validation decision regarding the new proposed information state. The notification of the result can be sent as either a send MDP or an NR-Send MDP, depending on whether non-repudiation is required. NR-Information Sharing does not require fair non-repudiation, so the simple non-repudiation MDPs are used. However, if fairness was required at a later date, the validated fair NR-Send and fair NR-Send protocols could replace the validated Send MDP and Send MDP (respectively) without disrupting the generic MDP, domain specific MDP or the application. Furthermore, the generic MEP could be implemented using a multi-party validated NR-Send MDP followed by a single multi-party Send MDP (should multi-party protocols have been developed).

6.4.7 Support for other protocol classifications

Fair non-repudiation protocols are members of the broad classification of fair exchange protocols. Future work could see the rigorous implementation techniques used to implement other security protocols. In order to achieve this a set of FSM templates would need to be specified for each supported security protocol. Furthermore, a new set of message exchange patterns would need to be added to the hierarchy, to enable the application to utilise the new protocols.

Bibliography

[1] Smc — state machine compiler. *http://smc.sourceforge.net, 2005.*

[2] I. Abdullah and D. Menascé. Protocol specification and automatic implementation using xml and cbse. *In the Proc. of Int. Conf. on Communications, Internet and Information Tech. (CIIT2003), Scottsdale, AZ, 17-19 Nov. 2003.*

[3] S. G. Akl. Digital signatures: a tutorial survey. *Computer, 16(2):15-24, February 1983.*

[4] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. *Proc. 4th ACM Conf. on Computer and Communications Security, Zurich, Switzerland.*

[5] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Areas in Communications, 18 (4): 591-606, 2000.*

[6] G. Ateniese and C. Nita-Rotaru. Stateless-receipt certified e-mail system based on verifiable encryption. *CT-RSA'02, LNCS 2271, pp. 182-199. Springer-Verlag, 2002.*

[7] Giuseppe Ateniese. Efficient Verifiable Encryption (and Fair Exchange) of Digital Signatures. In *Proc. ACM Conf. on Comp. and Comm. Security (CCS)*, Singapore, 1999.

[8] A. Bahreman and J. D. Tyger. Certified electronic mail. *Proc. Internet Society Symposium on Network and Distributed System Security, San Diego, California.*

[9] F. Bao, R. H. Deng, and W. Mao. Efficient and practical fair exchange protocols with off-line ttp. *Proc. 1998 IEEE Symposium on Security and Privacy, Oakland, California.*

[10] A. Barros, M. Dumas, and A. T. Hofstede. Service interaction patterns: Towards a reference framework for service-based business process interconnection. *Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.*

[11] BEA. Weblogic. *http://www.bea.com, 2005.*

[12] M. Blum. How to exchange (secret) keys. *ACM Transactions on Computer Systems 1 (2) (1983) 175-193, previously published in ACM STOC 83 proceedings, pages 440-447.*

[13] K. S. Booth. Authentication of signatures using public key encryption. *Communications of the ACM, 24(11):772-774, November 1981.*

[14] E. F. Brickell, D. Chaum, I. B. Damgard, and J. van de Graaf. gradual and verifiabe release of a secret. *Lecture Notes in Computer Science 293, Advances in Cryptology: Proc. Crypto '87, Santa Barbara, California.*

[15] R. Cleve. Controlled gradual disclosure schemes for random bits and their applications. *Lecture Notes in Computer Science 435, Advances in Cryptology: Proc. Crypto '89, Santa Barbara, California.*

[16] T. Coffey and P. Saidha. Non-repudiation with mandatory proof of receipt. *Computer Communications Review*, 26(1), 1996.

[17] T. Coffey, P. Saidha, and P. Burrows. Analysing the security of a non-repudiation communication protocol with mandatory proof of receipt. *In Proc. of the 1st Int. Symposium on Information and Communication Technologies, pp. 351-356, Dublin, Ireland, 2003.*

[18] Nick Cook, Santosh Shrivastava, and Stuart Wheater. Middleware support for non-repudiable transactional information sharing between enterpises.

[19] Nick Cook, Santosh Shrivastava, and Stuart Wheater. Distributed Object Middleware to Support Dependable Information Sharing between Organisations. In *Proc. IEEE Int. Conf. on Dependable Syst. and Networks (DSN)*, Washington DC, USA, 2002.

[20] D. Box, E. Christensen et al. *Web Services Addressing (WS-Addressing).* W3C Member Submission, http://www.w3.org/Submission/ws-addressing/, August 2004.

[21] D. Bunting, J. Durand et al. *Web Services Reliable Messaging TC WS-Reliability 1.1.* OASIS Committee Working Draft, http://www.oasis-open.org/committees/wsrm/, August 2004.

[22] I. B. Damgard. Practical and provably secure release of a secret and exchange of signatures. *Lecture Notes in Computer Science 765, Advances in Cryptology: Proc Eurocrypt'93, Lofthus, Norway.*

[23] DataPower. *XML Security Gateway.* http://www.datapower.com/products/xs40.html, 2004.

[24] R. DeMillo and M. Merritt. Protocols for data security. *Computer, 16(2):39-50, February 1983.*

[25] X. Didelot. Cosp-j: A compiler for security protocols. *MSC Thesis, Oxford University Computing Laboratory, 2003.*

[26] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon. *XML Encryption Syntax and Processing.* W3C Recommendation, http://www.w3.org/TR/xmlenc-core/, 2002.

[27] D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing.* W3C Recommendation, http://www.oasis-open.org/committees/wss/, 2004.

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

[29] P.D. Ezhilchelvan and S.K. Shrivastava. Systematic Development of a Family of Fair Exchange Protocols. In *Proc. 17th IFIP WG 11.3 Working Conf. on Database and Applications Security*, Colorado, USA, 2003.

[30] O. Goldreich, S. Micali, and R. Rivest. A fair protocol for signing contracts. *IEEE Transaction on Information Theory 36 (1) (1990) 40-46.*

[31] A. D. Gordon, K. Bhargavan, C. Fournet, and G. O'Shea. An advisor for web services security policies. *2005 ACM Workshop on Secure Web Services (SWS 2005), Fairfax, VA, USA. Pages 1-9. ACM Press, 2005.*

[32] OMG (Object Management Group). Corba security specification. *http://www.omg.org, 1998.*

[33] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework.* W3C, W3C Recommendation, http://www.w3.org/TR/soap12-part1/, June 2003.

[34] S. Gurgens and C. Rudolph. Security analysis of (un-) fair non-repudiation protocols. *In: Formal Aspects of Security (FASes'02), LNCS 2629, pp. 97-114, Springer-Verlag, 2003.*

[35] S. Gurgens, C. Rudolph, and H. Vogt. On the security of fair non-repudiation protocols. *In: Information Security Conference (ISC'03), LNCS 2851, pp. 193-207. Springer-Verlag, 2003.*

[36] P. Hallam-Baker. *XML Key Management Specification (XKMS 2.0).* W3C, http://www.w3.org/TR/xkms2/, April 2004.

[37] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. *Proc. of the 12th IEEE Symposium on Reliable Distributed System, pages 105-114, October 1993.*

[38] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group rpc service. *Proc. of the 15th International Conference on Distributed Computing Systems, pages 288-295, May 1995.*

[39] IBM. Websphere. *http://www.ibm.com/software/websphere/, 2005.*

[40] K. Imamoto and K.Sakurai. A certified e-mail system with receiver's selective usage of delivery authority. *Indocrypt'02, LNCS 2551, pp. 326-338. Springer-Verlag, 2002.*

[41] ISO. ISO13888-1. *Information Technology - Security Techniques - Non-repudiation - part 1: General*, 1997.

[42] G. Itkis and L. Reyzin. Sibir: Signer-base intrusion-resilient signtures. *LNCS 2442, Advances in Cryptology: Proc. of Crypto'02, Santa Barbara, California, August 2002.*

[43] R. Khalaf. From rosettanet pips to bpel processes: A three level approach for business protocols. *In Proc. of Third International Conference on Business Process Management, Nancy, September 2005.*

[44] E. Kleiner and A. Roscoe. Web services security: a preliminary study using casper and fdr. *In Proc. of Automated Reasoning for Security Protocol Analysis (ARSPA 04), Cork, Ireland.*

[45] S. Kremer and O. Markowitch. A multi-party non-repudiation protocol. *Proc. of 15th IFIP International Information Security Conference, pp. 271–280, Beijing, China, August 2000.*

[46] S. Kremer and O. Markowitch. Selective receipt in certified e-mail. *Advances in Cryptology: Proc. of Indocrypt 2001, LNCS. Springer-Verlag, December 2001.*

[47] S. Kremer, O. Markowitch, and J. Zhou. An Intensive Survey of Fair Non-repudiation Protocols. *Computer Communications*, 25:1601–1621, 2002.

[48] Steve Kremer and Olivier Markowitch. Fair multi-party non-repudiation protocols. *International Journal on Information Security*, 1(4):223–235, July 2003.

[49] Steve Kremer, Olivier Markowitch, and Jianying Zhou. An intensive survey of fair non-repudiation protocols.

[50] S Lee, O Kwon, J Lee, C Oh, and S Ko. TY*SecureWS: An Integrated Web Service Security Solution Based on Java. *in Proc E-Commerce and Web Technologies: 4th International Conference, EC-Web, Springer LNCS 2738*, pages 186–195, 2003.

[51] Peng Liu, Peng Ning, and Sushil Jajodia. Avoiding Loss of Fairness Owing to Process Crashes in Fair Data Exchange Protocols. In *Proc. IEEE Int. Conf. on Dependable Syst. and Networks (DSN)*, New York, USA, 2000.

[52] O. Markowitch and S. Kremer. A multi-party optimistic non-repudiation protocol. *LNCS 2015, Proc. of 3rd Int. Conference on Information Security and Cryptology, pp. 109–122, Seoul, Korea, December 2000.*

[53] O. Markowitch and S. Kremer. An optimistic non-repudiation protocol with transparent trusted third party. *In: Information Security Conference (ISC'01), LNCS 2200, pp. 363-378. Springer-Verlag, 2001.*

[54] O. Markowitch and Y. Roggeman. Probabilistic non-repudiation without trusted third party. *2nd Conf. Security in Communication Networks '99, Amalfi, Italy, 1999.*

[55] Olivier Markowitch, Dieter Gollmann, and Steve Kremer. On Fairness in Exchange Protocols. In *Proc. 5th Int. Conf. on Information Security and Cryptology (ISISC 2002)*, Springer LNCS 2587, 2002.

[56] L. Mengual, N. Barcia, E. Jiménez, and E. Menasalvas. Automatic implementation system of security protocols based on formal description techniques. *In Journal of 7th IEEE International Symposium on Computers and Communications (ISCC'02) p. 355.*

[57] S. Micali. Simple and fast optimistic protocols for fair electronic exchange. *Proc. of 22nd annual ACM Symp. on Principles of Distributed Computing (PODC'03), pp. 12-19. ACM Press, 2003.*

[58] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne. Contract Representation for Run-time Monitoring and Enforcement. In *Proc. IEEE Int. Conf. on E-Commerce (CEC)*, pages 103–110, Newport Beach, USA, 2003.

[59] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security 1.0 (WS-Security 2004).* W3C Recommendation, http://www.w3.org/TR/xmldsig-core/, 2002.

[60] A Nenadic, N Zhang, and S Barton. FIDES - A Middleware E-Commerce Security Solution. *in Proc 3rd European Conf on Inf Warfare and Security (ECIW 2004)*, 2004.

[61] NunniSoft. Nunnifsm. *http://nunnifsmgen.nunnisoft.ch, 2005.*

[62] T. Okamoto and K. Ohta. How to simultaneously exchange secrets by general assumptions. *Proc. 2nd ACM Conf. on Computer and Communications Security, Fairfax, Virginia.*

[63] J. Onieva, J. Zhou, J. Lopez, and M Carbonell. Agent-mediated non-repudiation protocols. *In the Journal of Electronic Commerce Research and Applications, 3(2):152–162, Summer 2004.*

[64] Jose A. Onieva, Jianying Zhou, Javier Lopez, and Mildrey Carbonell. Agent-mediated non-repudiation protocols. *Electronic Commerce Research and Applications*, 3:152–162, 2004.

[65] A. Perrig and D. Song. A first step towards the automatic generation of security protocols. *In Proc. of Int. Workshop on Cryptographic Techniques and E-Commerce (CrypTEC), City University of Hong Kong Press, 1999.*

[66] A. Perrig and D. Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. *In Proc. of 13th IEEE Computer Security Foundations Workshop, 2000.*

[67] D. Pozza, R. Sisto, and L. Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. *In 18th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA'04) Volume 1 p. 400.*

[68] R. Bilorusets, A. Bosworth et al. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging).* BEA, Microsoft, IBM and TIBCO Software, http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-reliablemessaging.asp, March 2004.

[69] M. O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences 27 (2) (1983) 256-267.*

[70] Rosettanet. *http://www.rosettanet.org, 2004.*

[71] B. Schneier and J. Riordan. A certified e-mail protocol. *Proc. Fourteenth Annual Computer Security Applications Conference, Pheonix, AZ, Dec. 1998.*

[72] Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, 2nd edition, 1996.

[73] E. Solaiman, C. Molina-Jimenez, and S. Shrivastava. Model checking correctness properties of electronic contracts. *In Proc. of the International Conference on Service Oriented Computing (ICSOC03)Trento, Italy, Lecture Notes in Computer Science Volume 2910 pp. 303-318, Springer, December 2003.*

[74] D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *In Journal of Computer Security, 9(1,2):47–74, 2001.*

[75] D. Song, A. Perrig, and D. Phan. Agvi — automatic generation, verification and imlementation of security protocols. *In Proc. of 13th Conference on Computer Aided Verification (CAV), Vol. 1504 of LNCS, Springer-Verlag, 2001.*

[76] P. Syverson. Weakly secret bit commitment: Applications to lotteries and fair exchange. *Proc. 1998 IEEE Computer Security Foundations Workshop (CSFW11), 1998.*

[77] T. Tedrick. Fair exchange of secrets. *G. R. Blakely, D. C. Chaum (Eds.), Advances in Cryptology: Proc. Crypto 84, Vol. 196 of Lecture Notes in Computer Science, Springer-Verlag, 1985, pp. 434-438.*

[78] T. Tedrick. How to exchange half a bit. *D Chaum (Ed.), Advances in Cryptology: Proc. Crypto 83, Plenum Press, New York and London, 1984, 1983, pp. 147-151.*

[79] B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. *In Proc. IFIP World Computer Congress - CSES 2004 2nd International Workshop on Certification and Security in Inter-Organizational E-Services, Toulouse, France.*

[80] Verisign. *Simplifying Application and Web Services Security: Verisign Trust Gateway.* Verisign White Paper, http://www.xmltrustcenter.org/index.htm, 2004.

[81] G. Wang. Generic fair non-repudiation protocols with transparent off-line ttp. *Proc. of 4th International Workshop for Applied PKI, Singapore, September 2005.*

[82] Michael Wichert, David B. Ingham, and Steve J. Caughey. Non-repudiation evidence generation for CORBA using XML. In *ACSAC*, pages 320–, 1999.

[83] N. Zhang and Q. Shi. Acheiving non-repudiation of receipt. *The Computer Journal.*

[84] N. Zhang and Q. Shi. Achieving non-repudiation of receipt. *The Computer Journal 39 (10) (1996) 844-853.*

[85] J Zhou. *Non-repudiation in Electronic Commerce.* Artech House, 2001.

[86] J. Zhou, F. Bao, and R. Deng. Validating Digital signatures without TTP's Time-stamping and Certificate Revocation. In *Proc. 2003 Inf. Security Conf.*, Springer LNCS 2851, Bristol, UK, 2003.

[87] J. Zhou, R. Deng, and F. Bao. Evolution of fair non-repudiation with ttp. *In: Information Security and Privacy (ACISP'99), LNCS 1587, pp. 258-269. Springer-Verlag, 1999.*

[88] J. Zhou, R. Deng, and F. Bao. Some remarks on a fair exchange protocol. *In: Public Key Cryptography (PKC'00), LNCS 1751, pp. 46-57. Springer-Verlag, 2000.*

[89] J. Zhou and D. Gollmann. An efficient non-repudiation protocol. *Proc. 10th IEEE Computer Security Foundations Workshop, Rockport, Massachusetts.*

[90] J. Zhou and D. Gollmann. A fair non-repudiation protocol. *Proc. 1996 IEEE Symposium on Security and Privacy, Oakland, California.*

[91] J. Zhou and D. Gollmann. Observations on non-repudiation. *Lecture Notes in Computer Science 1163, Proceedings of Asiacrypt'96, pages 133–144, Kyongju, Korea, November 1996.*

[92] J. Zhou and D. Gollmann. Evidence and non-repudiation. *J. Network and Comp. Applications*, 20(3):267–281, 1997.

[93] J. Zhou and K. Y. Lam. A secure pay-per-view scheme for web based video service. *LNCS 1560, Proc. of 1999 International Workshop on Practice and Theory in Public-Key Cryptography, pp. 315-326, Kamakura, Japan, March 1999*.

[94] J. Zhou and K.Y. Lam. Securing Digital Signatures for Non-repudiation. *Computer Communications*, 22:710–716, 1999.

Note: References with the asterisk [*] symbol, are own publications

Appendix A

Examples

This appendix uses the Delivery Agent protocol (Described in Section 3.2) for the basis of an example of what actual protocol notation and FSMs would look like. Two example protocol messages and the protocol message schema is also present.

A.1 Protocol notation

This section presents the Delivery Agent protocol (Described in Section 3.2) represent in the notation described in Section 4.3. In the notation several variables have been used to represent run identifiers and non-repudiation evidence. These can be replaced inline with the following definitions:

```
ID      = runID(A, B, TTP, rn)
ID1     = runID(A, TTP, rn1)
ID2     = runID(B, TTP, rn2)
NROa    = sS_A(ID, A, B, h(msg))
NRSttp  = sS_TTP(ID, A, B)
NRRb    = sS_B(ID, A, B, h(msg))
NRVb    = aS_B(ID, val(msg))
NROttp  = sS_TTP(ID, A, B, msg)
NRRttp  = NROttp
NRVttp  = sS_TTP(ID, val(msg))
NRNVttp = sS_TTP(ID, val(msg))
```

A.1.1 Exchange protocol

```
1 A -> TTP : eP_TTP(ID, A, B, TTP, rn, msg, NROa)
2 TTP -> A : ID, A, B, NRSttp
3 TTP -> B : ID, A, B, h(msg)
```

```
4 B -> TTP : ID, A, B, eP_TTP(NRRb)
5 TTP -> A : ID, A, B, NRRttp
6 TTP -> B : ID, A, B, msg, NROttp
7 B -> TTP : ID, A, B, val(msg), NRVb
8 TTP -> B : ID, A, B, rn
9 TTP -> A : ID, A, B, val(msg), NRVttp
```

A.1.2 Exception handling protocols

A.1.2.1 Protocol initiator's (A) abort protocol

```
1   A -> TTP : ID1, A, TTP, f(ABORT), ID, sS_A(ID1, f(ABORT), ID)
               IF status != SUCCEEDED
               AND lastStep < 5 THEN
2.1 TTP -> A : ID1, A, TTP, f(ABORTED), ID, sS_TTP(ID1, f(ABORTED), ID)
               ELSE IF lastStep < 7 THEN
2.2 TTP -> A : ID1, A, TTP, f(SUCCEEDED), sS_TTP(ID1, f(SUCCEEDED)),
                                              NRSttp, NRRttp, NRNVttp
               ELSE
2.3 TTP -> A : ID1, A, TTP, f(SUCCEEDED), sS_TTP(ID1, f(SUCCEEDED)),
                                              NRSttp, NRRttp, NRVttp
```

A.1.2.2 Protocol responder's (B) abort protocol

```
1   B -> TTP : ID2, B, TTP, f(ABORT), ID, sS_B(ID2, f(ABORT), ID)
               IF status != SUCCEEDED
               AND lastStep < 5 THEN
2.1 TTP -> B : ID2, B, TTP, f(ABORTED), ID, sS_TTP(ID2, f(ABORTED), ID)
               ELSE IF lastStep < 7 THEN
2.2 TTP -> B : ID2, B, TTP, f(SUCCEEDED), ID, sS_TTP(ID2, f(SUCCEEDED)),
                                              rn1, msg, NROttp, NRNVttp
               ELSE
2.3 TTP -> B : ID2, B, TTP, f(SUCCEEDED), ID, sS_TTP(ID2, f(SUCCEEDED)),
                                              rn1, msg, NROttp, NRVttp
```

A.1.2.3 Protocol initiator's (A) resolve protocol

```
1   A -> TTP : ID1, A, TTP, f(RESOLVE), ID, sS_A(ID1, f(RESOLVE), ID)
               IF status == ABORTED
```

```
                AND lastStep < 4 THEN
2.1 TTP -> A : ID1, A, TTP, f(ABORTED), ID, sS_TTP(ID1, f(ABORTED), ID)
                ELSE IF lastStep < 7 THEN
2.2 TTP -> A : ID1, A, TTP, f(SUCCEEDED), ID, sS_TTP(ID1, f(SUCCEEDED)),
                                              NRSttp, NRRttp, NRNVttp
                ELSE
2.3 TTP -> A : ID1, A, TTP, f(SUCCEEDED), ID, sS_TTP(ID1, f(SUCCEEDED)),
                                              NRSttp, NRRttp, NRVttp
```

A.1.2.4 Protocol responder's (B) resolve protocol

```
1   B -> TTP : ID2, B, TTP, f(RESOLVE), ID, sS_B(ID2, f(RESOLVE), ID)
                IF status == ABORTED
                AND lastStep < 4 THEN
2.1 TTP -> B : ID2, B, TTP, f(ABORTED), ID, sS_TTP(ID2, f(ABORTED), ID)
                ELSE IF lastStep < 7 THEN
2.2 TTP -> B : ID2, B, TTP, f(SUCCEEDED), ID, sS_TTP(ID2, f(SUCCEEDED)),
                                              rn, msg, NROttp, NRNVttp
                ELSE
2.3 TTP -> B : ID2, B, TTP, f(SUCCEEDED), ID, sS_TTP(ID2, f(SUCCEEDED)),
                                              rn, msg, NROttp, NRVttp
```

A.2 Finite state machines

This section presents the finite state machine representation of the Delivery Agent protocol (Described in Section 3.2).

A.2.1 Exchange protocol FSMs

A.2.1.1 Protocol initiator (A)

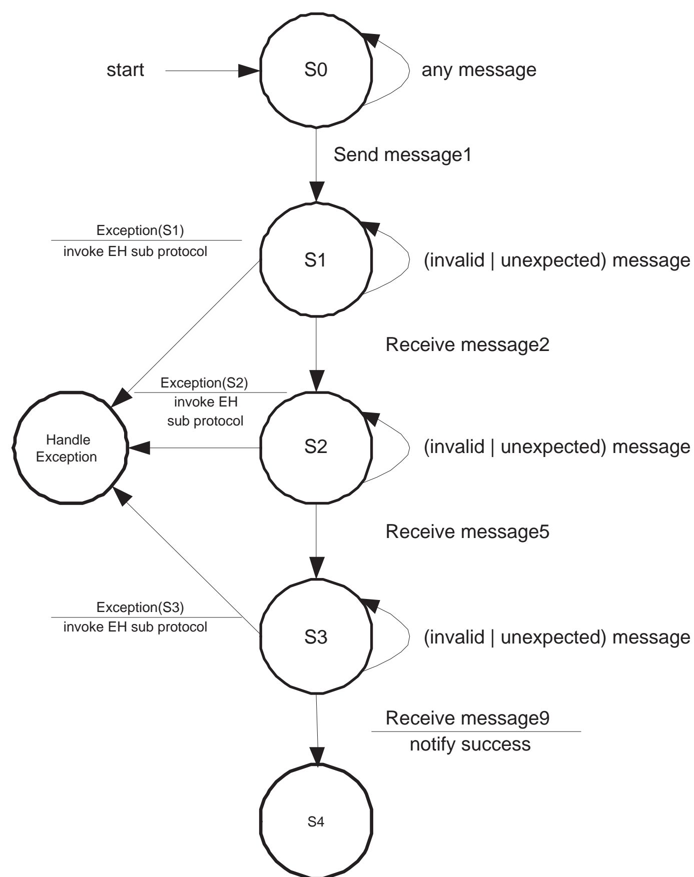


Figure A.1: Exchange protocol FSM for initiator of DA protocol

A.2.1.2 Protocol responder (B)

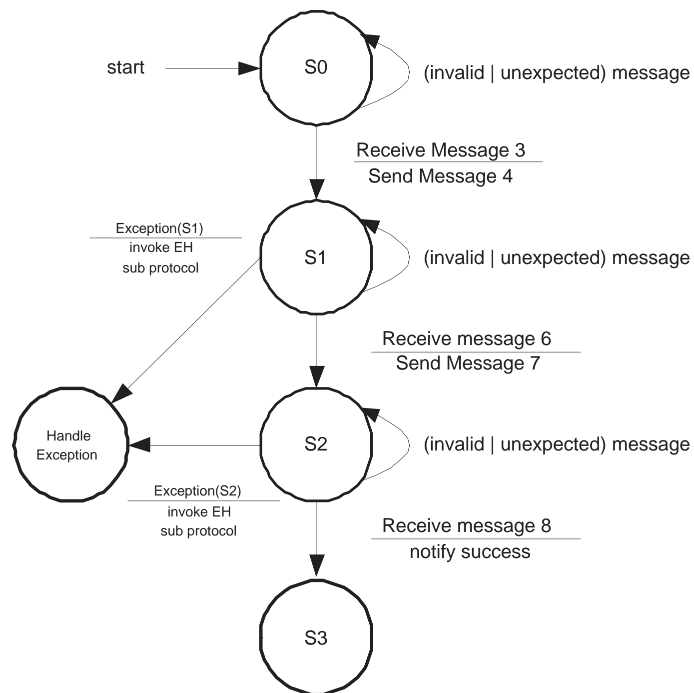


Figure A.2: Exchange protocol FSM for responder of DA protocol

A.2.1.3 Trusted third party (DA)

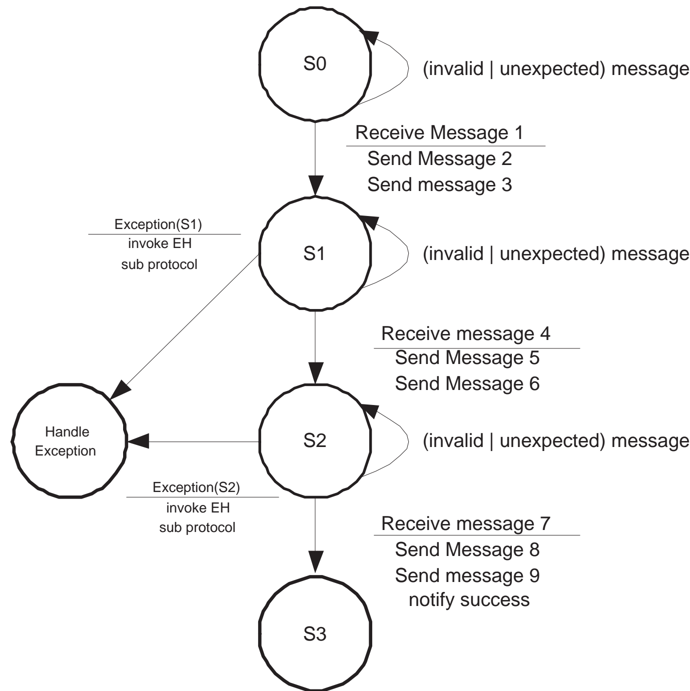


Figure A.3: Exchange protocol FSM for TTP of DA protocol

A.2.2 Exception handling protocol FSMs

A.2.2.1 Protocol initiator and responder (A and B)

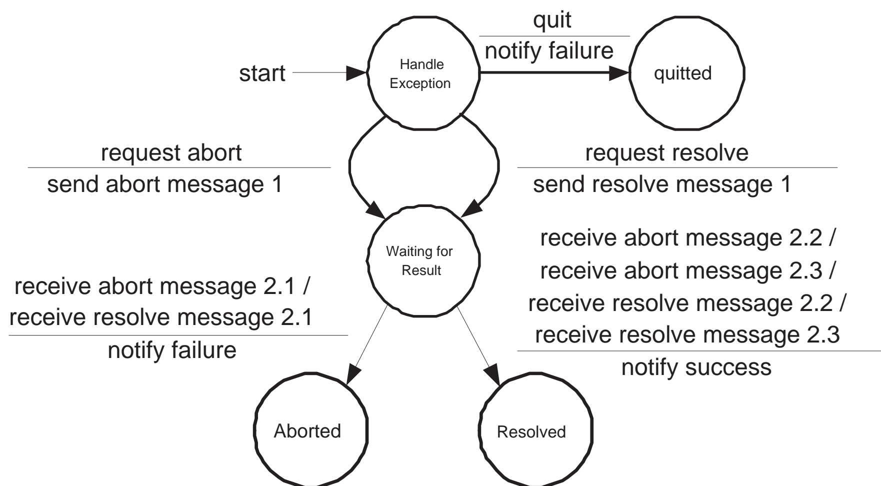


Figure A.4: Exchange protocol FSM for initiator of DA protocol

A.2.2.2 Trusted third party (DA)

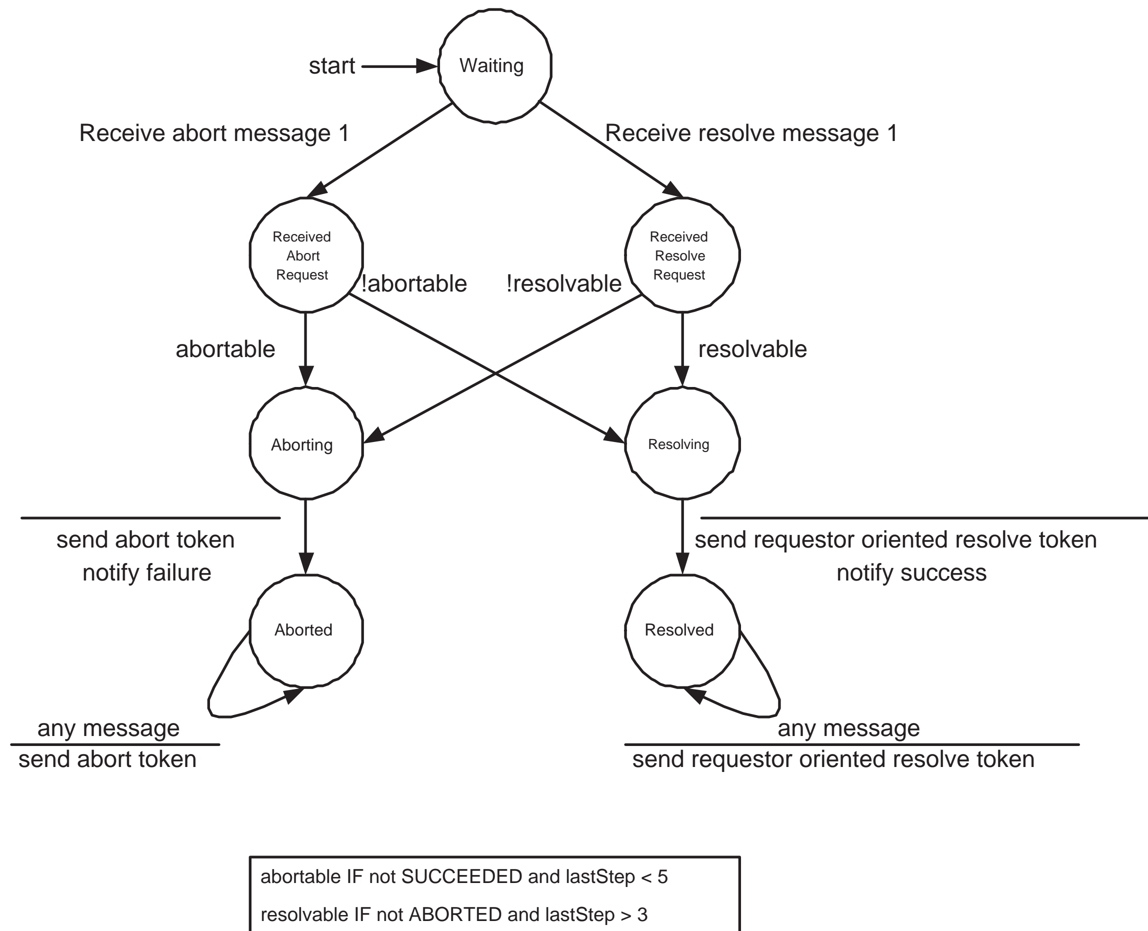


Figure A.5: Exchange protocol FSM for TTP of DA protocol

A.3 Soap protocol messages

This section uses a purchase order as a simple example of a business message. The discussion shows a subset of business message content and the most significant annotations to the message for non-repudiation protocol execution. Timestamps are omitted and so is encryption.

The following SOAP message represents a purchase order in an application that does not use NRExchange services.

```
<SOAP:Envelope>
  <SOAP:Header>
    ...
  </SOAP:Header>
  <SOAP:Body>
```

```
      <po:PurchaseOrder>
        <Quantity>1000000</Quantity>
        <itemID>234233</itemID>
      </po:PurchaseOrder>
     ...
    </SOAP:Body>
  </SOAP:Envelope>
```

The above message is deliberately kept simple. In practice, business messages may be quite complex and include numerous header and body elements along with mixed media attachments and references to external information. Given the NRExchange infrastructure described in Section 5.2, it is possible to ensure that fair exchange can be applied to messages regardless of their content, attachments or references.

A.3.1 Exchange protocol message 1

The following XML shows message 1 of the Delivery agent protocol transformed into a SOAP envelope, where the application message is the purchase order.

```
<soap:Envelope>
 <soap:Header>
 <!-- WS-Security header -->
  <wsse:Security soap:actor='...'  soap:mustUnderstand='1'>
   <!-- sender certificate -->
   <wsse:BinaryToken ValueType='wsse:X509v3'>
    <!-- token value -->
   </wsse:BinaryToken>
   <ds:Signature>
    <ds:SignedInfo>
     <!-- reference to nrex_header -->
     <ds:Reference URI='#nrex_header'>
      <ds:Transforms>
       <!-- any transforms -->
      </ds:Transforms>
     <ds:DigestMethod Algorithm='...'/>
      <ds:DigestValue>FtaewlewrldsfRGrt8/...</ds:DigestValue>
     </ds:Reference>
     <!-- reference to body -->
     <ds:Reference URI='#message_body'>
      <ds:Transforms>...</ds:Transforms>
      <ds:DigestMethod Algorithm='...'/>
      <ds:DigestValue>thjribUhkhgtPl...</ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>lsPiZkngCb4uDi/BsPear...</ds:SignatureValue>
```

```
   </ds:Signature>
   <!-- other security tokens -->
  </wsse:Security>
  <!-- WS-NRExchange header -->
  <nrex:NRExchangeProtocol
        xmlns:nrex='http://www.cs.ncl.ac.uk/ws/nrex/v1'
        name='http://www.cs.ncl.ac.uk/ws/nrex/protocols/coffsaid-lwuser/main'
        runId='Gfr56Rt4..'
        messageNumber='1.0'
        soap:actor='...'  soap:mustUnderstand='1' id='nrex_header'>
   <nrex:Participant role='nrex.role.TTP'>
    http://www.ttp.com/
   </nrex:Participant>
   <nrex:Participant role='nrex.role.SENDER'>
    http://www.purchaser.com/
   </nrex:Participant>
   <nrex:Participant role='nrex.role.RECEIVER'>
    http://www.supplier.com/
   </nrex:Participant>
   <nrex:RandomNumber id=''rn1''>
     salfoeiur838rjlfdsk...
   </nrex:RandomNumber>
   <nrex:RunIdGenerator runId='Gfr56Rt4..'>
    <nrex:DigestReference runID=''Gfr56Rt4..''  messageNumber=''1.0''>
     <ds:Reference URI=''#rn1''>
      <ds:DigestMethod>...</ds:DigestMethod>
      <ds:DigestValue>...</ds:DigestValue>
     </ds:Reference>
    </nrex:DigestReference>
   </nrex:RunIdGenerator>
   </nrex:NRExchangeProtocol>
   <!-- other soap header elements -->
  </soap:Header>
 <soap:Body Id='message_body'>
  <po:PurchaseOrder>
   <Quantity>1000000</Quantity>
   <itemID>234233</itemID>
  </po:PurchaseOrder>
 </soap:Body>
</soap:Envelope>
```

A.3.2 Exchange protcol message 2

```
<soap:Envelope>
 <soap:Header>
```

```
    <!-- WS-Security header -->
    <wsse:Security soap:actor='...'  soap:mustUnderstand='1'>
     <!-- sender certificate -->
     <wsse:BinaryToken ValueType='wsse:X509v3'>
      <!-- token value -->
     </wsse:BinaryToken>
     <!-- signature on nrexchange header -->
     <ds:Signature>
      <ds:SignedInfo>
       <!-- xml signature info -->
        <ds:Reference URI='#nrex_header'>
        <ds:Transforms>
         <!-- any transforms -->
        </ds:Transforms>
        <ds:DigestMethod Algorithm='AlgorithmName'/>
        <ds:DigestValue>8j8jkj8TUpnmMM...</ds:DigestValue>
       </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>lb4uDiear...</ds:SignatureValue>
     </ds:Signature>
     <!-- other security tokens -->
    </wsse:Security>
    <!-- WS-NRExchange header -->
    <nrex:NRExchangeProtocol
          xmlns:nrex='http://www.cs.ncl.ac.uk/ws/nrex/v1'
          name='http://www.cs.ncl.ac.uk/ws/nrex/protocols/coffsaid-lwuser/main'
          runId='Gfr56Rt4..'
          messageNumber='2.0'
          soap:actor='...'  soap:mustUnderstand='1' id='nrex_header'>
     <nrex:Participant role='nrex.role.TTP'>
      http://www.ttp.com/
     </nrex:Participant>
     <nrex:Participant role='nrex.role.SENDER'>
      http://www.purchaser.com/
     </nrex:Participant>
     <nrex:Participant role='nrex.role.RECEIVER'>
      http://www.supplier.com/
     </nrex:Participant>
    </nrex:NRExchangeProtocol>
   <!-- other soap header elements -->
  </soap:Header>
  <soap:Body/>
</soap:Envelope>
```

A.4 Protocol message schema

There follows the NRExchange protocol message schema:

```
<?xml version='1.0' encoding='UTF-8'?>
<xsd:schema xmlns:ds='http://www.w3.org/2000/09/xmldsig#'
                  xmlns:nrex='http//www.cs.ncl.ac.uk/ws/nrex/v1'
                  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
                  attributeFormDefault='unqualified'
                  elementFormDefault='qualified'
                  targetNamespace='http//www.cs.ncl.ac.uk/ws/nrex/v1'>
  <xsd:import namespace='http://www.w3.org/2000/09/xmldsig#'
              schemaLocation='http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd'/>
  <xsd:element name='Acknowledgement' type='nrex:AcknowledgementType'/>
  <xsd:element name='AcknowledgementsRequired'
              type='nrex:AcknowledgementsRequiredType'/>
  <xsd:element name='DigestReference' type='nrex:DigestReferenceType'/>
  <xsd:element name='LogMetaInf' type='nrex:LogMetaInfType'/>
  <xsd:element name='NRExchangeLogEntry' type='nrex:NRExchangeLogEntryType'/>
  <xsd:element name='NRExchangeProtocol' type='nrex:NRExchangeProtocolType'/>
  <xsd:element name='Participant' type='nrex:ParticipantType'/>
  <xsd:element name='ProtocolState' type='nrex:ProtocolStateType'/>
  <xsd:element name='ProtocolStateRequest' type='nrex:ProtocolStateRequestType'/>
  <xsd:element name='RandomNumber' type='nrex:RandomNumberType'/>
  <xsd:element name='ReceiptsRequired' type='nrex:ReceiptsRequiredType'/>
  <xsd:element name='Reference' type='nrex:ReferenceType'/>
  <xsd:element name='RelatedRun' type='nrex:RelatedRunType'/>
  <xsd:element name='RunIdGenerator' type='nrex:RunIdGeneratorType'/>
  <!-- enums -->
  <xsd:simpleType name='AckType'>
    <xsd:restriction base='xsd:string'>
      <xsd:enumeration value='nrex.ack.TTP'/>
      <xsd:enumeration value='nrex.ack.RECEIVER'/>
      <xsd:enumeration value='nrex.ack.VALIDITOR'/>
      <xsd:enumeration value='nrex.ack.OTHER'/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name='RoleType'>
    <xsd:restriction base='xsd:string'>
      <xsd:enumeration value='nrex.role.RECEIVER'/>
      <xsd:enumeration value='nrex.role.SENDER'/>
      <xsd:enumeration value='nrex.role.TSA'/>
      <xsd:enumeration value='nrex.role.TTP'/>
      <xsd:enumeration value='nrex.role.OTHER'/>
    </xsd:restriction>
  </xsd:simpleType>
```

```
<xsd:simpleType name='PurposeType'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='nrex.purpose.NRO'/>
    <xsd:enumeration value='nrex.purpose.NRR'/>
    <xsd:enumeration value='nrex.purpose.NRS'/>
    <xsd:enumeration value='nrex.purpose.NRV'/>
    <xsd:enumeration value='nrex.purpose.OTHER'/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name='StatusType'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='nrex.status.ACTIVE'/>
    <xsd:enumeration value='nrex.status.TERMINATED'/>
    <xsd:enumeration value='nrex.status.OTHER'/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name='TerminationStatusType'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='nrex.termination.ABORTED'/>
    <xsd:enumeration value='nrex.termination.SUCCEEDED'/>
    <xsd:enumeration value='nrex.termination.OTHER'/>
  </xsd:restriction>
</xsd:simpleType>
<!-- protocol components -->
<!-- to specify the acknowledgements that are required -->
<xsd:complexType name='AcknowledgementsRequiredType'>
  <xsd:sequence>
    <xsd:element name='AckSpecification' type='nrex:AckSpecificationType'
            maxOccurs='unbounded'/>
  </xsd:sequence>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- acknowledgement to convey ack type information and validation status -->
<xsd:complexType name='AcknowledgementType'>
  <xsd:sequence>
    <xsd:element minOccurs='0' name='ValidationInfo' type='xsd:string'/>
    <xsd:any maxOccurs='unbounded' minOccurs='0' namespace='##other'
         processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='type' type='nrex:AckType'/>
  <xsd:attribute name='messageReference' type='nrex:MessageNumberType'/>
  <xsd:attribute name='runIdReference' type='nrex:RunIdType'/>
  <xsd:attribute name='isValid' type='xsd:boolean'/>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
```

```
<!-- specification of an acknowledgement including optional deadline for its receipt -->
<xsd:complexType name='AckSpecificationType'>
  <xsd:sequence>
    <xsd:any namespace='##other' minOccurs='0' maxOccurs='unbounded'
             processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='type' type='nrex:AckType' use='required'/>
  <xsd:attribute name='expires' type='xsd:dateTime'/>
</xsd:complexType>
<!-- reference type for forwared/backward reference to messages/elements
    within messages or URIs relative to messages, optionally including digest
   information -->
<xsd:complexType name='ReferenceType'>
  <xsd:sequence>
    <xsd:element ref='ds:Transforms' minOccurs='0'/>
    <xsd:element ref='ds:DigestMethod' minOccurs='0'/>
    <xsd:element ref='ds:DigestValue' minOccurs='0'/>
  </xsd:sequence>
  <xsd:attribute name='protocolName' type='xsd:anyURI'/>
  <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
  <xsd:attribute name='messageNumber' type='nrex:MessageNumberType'
               use='required'/>
  <xsd:attribute name='URI' type='anyURI'/>
  <xsd:attribute name='Type' type='anyURI'/>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- collection of references used to create a single digest value where the
    references themselves may contain a digest value -->
<xsd:complexType name='DigestReferenceType'>
  <xsd:sequence>
    <xsd:element ref='nrex:Reference' maxOccurs='unbounded'/>
    <xsd:element ref='ds:Transforms' minOccurs='0'/>
    <xsd:element ref='ds:DigestMethod'/>
    <xsd:element ref='ds:DigestValue'/>
  </xsd:sequence>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- list of messages that are available at a participant and who they are
    available to -->
<xsd:complexType name='MessageAvailabilityType'>
  <xsd:sequence>
    <xsd:element name='Messages' type='nrex:MessageNumberList'/>
    <xsd:element name='AvailableTo' type='nrex:ParticipantType'/>
  </xsd:sequence>
</xsd:complexType>
```

```
<xsd:simpleType name='MessageNumberList'>
  <xsd:list itemType='nrex:MessageNumberType'/>
</xsd:simpleType>
<xsd:simpleType name='MessageNumberType'>
  <xsd:restriction base='xsd:double'>
    <xsd:minExclusive value='0.0'/>
  </xsd:restriction>
</xsd:simpleType>
<!-- a log entry element for annotating a protocol with meta-information
    when logging -->
<xsd:complexType name='LogMetaInfType'>
  <xsd:sequence>
    <xsd:element maxOccurs='unbounded' minOccurs='0' ref='nrex:Participant'/>
    <xsd:element minOccurs='0' name='Annotation' type='xsd:string'/>
    <xsd:any namespace='##other' processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='logDate' type='xsd:dateTime'/>
  <xsd:attribute name='logTime' type='xsd:unsignedLong'/>
  <xsd:attribute name='protocolName' type='xsd:anyURI'/>
  <xsd:attribute name='protocolRunId' type='nrex:RunIdType'/>
  <xsd:attribute name='protocolMessageNumber' type='nrex:MessageNumberType'/>
  <xsd:anyAttribute namespace='##other' processContents='lax'/>
</xsd:complexType>
<xsd:complexType name='NRExchangeLogEntryType'>
  <xsd:sequence>
    <xsd:element minOccurs='0' ref='nrex:LogMetaInf'/>
    <xsd:any namespace='##other' processContents='lax'/>
  </xsd:sequence>
</xsd:complexType>
<!-- the top-level nrexchange protocol header type -->
<xsd:complexType name='NRExchangeProtocolType'>
  <xsd:sequence>
    <xsd:element ref='nrex:Acknowledgement' minOccurs='0'
                maxOccurs='unbounded'/>
    <xsd:element ref='nrex:AcknowledgementsRequired' minOccurs='0'/>
    <xsd:element ref='nrex:DigestReference' minOccurs='0'
                maxOccurs='unbounded'/>
    <xsd:element ref='nrex:Participant' minOccurs='2'
                maxOccurs='unbounded'/>
    <xsd:element ref='nrex:ProtocolState' minOccurs='0'/>
    <xsd:element ref='nrex:ProtocolStateRequest' minOccurs='0'/>
    <xsd:element ref='nrex:RandomNumber' minOccurs='0'
                maxOccurs='unbounded'/>
    <xsd:element ref='nrex:ReceiptsRequired' minOccurs='0'/>
    <xsd:element ref='nrex:Reference' minOccurs='0' maxOccurs='unbounded'/>
```

```
    <xsd:element ref='nrex:RelatedRun' minOccurs='0'
                 maxOccurs='unbounded'/>
    <xsd:element ref='nrex:RunIdGenerator' minOccurs='0'/>
    <xsd:any namespace='##other' processContents='lax' minOccurs='0'
                 maxOccurs='unbounded'/>
  </xsd:sequence>
  <xsd:attribute name='name' type='xsd:anyURI' use='required'/>
  <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
  <xsd:attribute name='messageNumber' type='nrex:MessageNumberType' use='required'/>
  <xsd:attribute name='runSequence' type='xsd:unsignedLong'/>
  <xsd:attribute name='purpose' type='nrex:PurposeType'/>
  <xsd:attribute name='Id' type='xsd:ID'/>
  <xsd:anyAttribute namespace='##any' processContents='lax'/>
</xsd:complexType>
<xsd:complexType name='ParticipantType'>
<xsd:simpleContent>
  <xsd:extension base='xsd:anyURI'>
    <xsd:attribute name='authenticationRef' type='xsd:anyURI'/>
    <xsd:attribute name='messageOwner' type='xsd:boolean'/>
    <xsd:attribute name='role' type='nrex:RoleType'/>
    <xsd:attribute name='nrexchangeURI' type='xsd:anyURI'/>
    <xsd:attribute name='Id' type='xsd:ID'/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>
<!-- a request for the status of a protocol as perceived by recipient of
    request -->
<xsd:complexType name='ProtocolStateRequestType'>
  <xsd:sequence>
    <xsd:element name='LastMessageSeen' type='nrex:MessageNumberType'/>
    <xsd:element name='AttachMessages' type='nrex:MessageNumberList'
                 minOccurs='0' />
    <xsd:any namespace='##other' minOccurs='0' maxOccurs='unbounded'
                 processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='name' type='xsd:anyURI' use='required'/>
  <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<xsd:complexType name='ProtocolStateType'>
  <xsd:sequence>
    <xsd:element name='Status' type='nrex:StatusType'/>
    <xsd:element name='LastMessage' type='nrex:MessageNumberType'
                 minOccurs='0'/>
  <xsd:element name='TerminationStatus'
```

```
              type='nrex:TerminationStatusType' minOccurs='0'/>
  <xsd:element name='TerminationOriginator' type='nrex:ParticipantType'
              minOccurs='0'/>
  <xsd:element name='MessageAvailability'
              type='nrex:MessageAvailabilityType' minOccurs='0'
              maxOccurs='unbounded'/>
  <xsd:element name='AttachedMessages' type='nrex:MessageNumberList'
              minOccurs='0'/>
  <xsd:any namespace='##other' minOccurs='0' maxOccurs='unbounded'
              processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='name' type='xsd:anyURI' use='required'/>
  <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- referencable base64 encoded random number -->
<xsd:complexType name='RandomNumberType'>
  <xsd:sequence>
    <xsd:element minOccurs='1' name='Value' type='xsd:base64Binary'/>
  </xsd:sequence>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- specification of what should be receipted during protocol execution,
  may, for example, identify external information that must be
  receipted -->
<xsd:complexType name='ReceiptSpecificationType'>
  <xsd:sequence>
    <xsd:any maxOccurs='unbounded' minOccurs='0' namespace='##other' processContents='lax'/>
  </xsd:sequence>
  <xsd:attribute name='URI' type='xsd:anyURI'/>
</xsd:complexType>
<!-- list of receipt specifications -->
<xsd:complexType name='ReceiptsRequiredType'>
  <xsd:sequence>
    <xsd:element maxOccurs='unbounded' name='ReceiptSpecification' type='nrex:ReceiptSpecificationType'/>
  </xsd:sequence>
  <xsd:attribute name='Id' type='xsd:ID'/>
</xsd:complexType>
<!-- specification of a protocol run that is related to current run in some
    way, along with optional status information -->
<xsd:complexType name='RelatedRunType'>
  <xsd:sequence>
    <xsd:element maxOccurs='unbounded' minOccurs='0' ref='nrex:Participant'/>
    <xsd:element minOccurs='0' name='Status' type='nrex:StatusType'/>
    <xsd:element minOccurs='0' name='LastMessage' type='nrex:MessageNumberType'/>
```

```
      <xsd:element minOccurs='0' name='TerminationStatus' type='nrex:TerminationStatusType'/>
      <xsd:element minOccurs='0' name='TerminationOriginator' type='nrex:ParticipantType'/>
      <xsd:element minOccurs='0' name='AttachedMessages' type='nrex:MessageNumberList'/>
      <xsd:any maxOccurs='unbounded' minOccurs='0' namespace='##other' processContents='lax'/>
    </xsd:sequence>
    <xsd:attribute name='protocolName' type='xsd:anyURI'/>
    <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
  </xsd:complexType>
  <!-- run id generator to specify how a the given run id was generated from
      a set of 0 or more digest references -->
  <xsd:complexType name='RunIdGeneratorType'>
    <xsd:sequence>
      <xsd:element ref='nrex:DigestReference' minOccurs='0'/>
    </xsd:sequence>
    <xsd:attribute name='baseURI' type='xsd:anyURI'/>
    <xsd:attribute name='runId' type='nrex:RunIdType' use='required'/>
    <xsd:attribute name='Id' type='xsd:ID'/>
  </xsd:complexType>
  <xsd:simpleType name='RunIdType'>
    <xsd:restriction base='xsd:anyURI'/>
  </xsd:simpleType>
</xsd:schema>
```