Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance
for Internet Applications

First Specification of APIs and Protocols for the MAFTIA Middleware

Nuno Ferreira Neves and Paulo Veríssimo (editors)
University of Lisboa

MAFTIA deliverable D24

Public document

September 9, 2001

Research Report RZ 3365, IBM Research, Zurich Research Laboratory
Technical Report CS-TR-738, University of Newcastle upon Tyne
Technical Report DI/FCUL TR-01-6, University of Lisboa

Editors

Nuno Ferreira Neves, *University of Lisboa (P)*
Paulo Veríssimo, *University of Lisboa (P)*

Contributors

Jim Armstrong, *University of Newcastle upon Tyne (UK)*
Christian Cachin, *IBM Zurich Research Lab (CH)*
Miguel Correia, *University of Lisboa (P)*
Alexandre Costa, *University of Lisboa (P)*
Hugo Miranda, *University of Lisboa (P)*
Nuno Ferreira Neves, *University of Lisboa (P)*
Nuno Miguel Neves, *University of Lisboa (P)*
Jonathan A. Poritz, *IBM Zurich Research Lab (CH)*
Brian Randell, *University of Newcastle upon Tyne (UK)*
Luis Rodrigues, *University of Lisboa (P)*
Robert Stroud, *University of Newcastle upon Tyne (UK)*
Paulo Veríssimo, *University of Lisboa (P)*
Michael Waidner, *IBM Zurich Research Lab (CH)*
Ian Welch, *University of Newcastle upon Tyne (UK)*

Contents

Table of Contents v

List of Figures ix

1 Introduction 1

2 Runtime Support Services and APIs 4

2.1 Appia 4

2.1.1 Overview 4

2.1.2 Appia Concepts 5

2.1.3 API Description by Class 8

2.2 Trusted Timely Computing Base 24

2.2.1 The TTCB Interface 25

2.2.2 The TTCB Services 26

2.2.3 Security Related Services 27

2.2.4 Time Related Services 34

2.2.5 Other Services 37

3 Middleware Services and APIs 40

3.1 Multipoint Network 40

3.1.1 Internet Protocol 40

3.1.2 IP Multicast 41

3.1.3 IPSec 42

3.1.4 Internet Control Message Protocol 44

3.1.5 Simple Network Management Protocol 44

3.2 Communication Services 47

3.2.1 Services using Static Groups 48

3.2.2 Services using Dynamic Groups and related to Group Membership . 55

3.3 Activity Services 58

3.3.1 Overview of the Transactional Support 58

3.3.2 Transaction Service Architecture 59

3.3.3 System model 61

3.3.4 Related Approaches 62

3.3.5 Use of the Transaction Service 64

4 Middleware Protocols 71

4.1 Multipoint Network 71

4.1.1 Internet Protocol 71

4.1.2 IP Multicast 71

4.1.3 IPSec 72

4.1.4 Internet Control Message Protocol 72

4.1.5 Simple Network Management Protocol 73

4.2 Communications Support 74

4.2.1 Architecture Description 74

4.2.2 Protocols under Static Groups 75

4.2.3 Protocols under Dynamic Groups 77

4.3 Activity Services 78

4.3.1 Assumptions 79

4.3.2 Atomic commitment and abort protocols 81

4.3.3 Recovery protocols 82

4.3.4 Distributed locking protocol 83

4.3.5 Resource manager replication protocol 85

5 Conclusion 86

List of Figures

1.1 Architecture of a MAFTIA Host 2

2.1 Relation between sessions, layers, channels and QoS's 6

2.2 Appia main UML 9

2.3 Appia predefined events UML 10

2.4 Appia framework exceptions UML 11

2.5 TTCB Services 27

2.6 Common TTCB API parameters 27

2.7 Consensus service API: sequence of function calls 29

2.8 Local authentication service API protocol 31

2.9 Distributed authentication protocol: simple case with just two entities, A and B 32

2.10 Distributed authentication with symmetric authentication: sequence of function calls 33

3.1 High-level View of Transaction Service Architecture 60

3.2 Overview of Use of Transaction Service 65

4.1 Class Overview 75

4.2 Protocol Functional Dependencies 77

4.3 Transactional Support Service Groups 80

Abstract

This document describes the first specification of the APIs and Protocols for the MAFTIA Middleware. The architecture of the middleware subsystem has been described in a previous document, where the several modules and services were introduced: Activity Services; Communication Services; Network Abstraction; Trusted and Untrusted Components. The purpose of the present document is to make concrete the functionality of the middleware components, by defining their application programming interfaces, and describing the protocols implementing the above-mentioned functionality.

1 Introduction

This document describes the first specification of the APIs and protocols for the MAFTIA middleware. The architecture of the middleware subsystem has been presented in a previous deliverable (D23), where the various system models, modules and services were introduced. This document is the first of two deliverables that will be related with the definition of the APIs and protocols. Therefore, it focuses mainly on the specification of the external interfaces of the modules, which will allow all partners to have a common basis of work contributing towards a coherent result. Many protocols are currently under development, and as such the protocols chapter is mostly devoted to the underlying principles, and whenever protocols are described, they are necessarily done in a concise way. The information provided by this document will expanded in the next deliverable (D9 - Complete specification of APIs and protocols for the MAFTIA middleware), due one year from now.

Figure 1.1 represents the architecture of a MAFTIA host, in which the dependence relations between modules are depicted by the orientation of the ("depends-on") arrows. The figure also represents the main runtime environments that will support the architecture, and other components in general that might want to call their interfaces, namely the Appia protocol kernel, the Trusted Timely Computing Base (TTCB), and of course the Operating System (OS).

This deliverable is organized in two main parts, one that introduces the services and APIs and another that presents the protocols. The services and APIs are explained using a bottom-up approach, it starts by describing the runtime environments and then the middleware modules.

Appia is the protocol kernel that is going to be employed by the MAFTIA middleware. In terms of protocol design, a protocol kernel provides the tools that allow a designer to compose stacks of protocols (or modules) according to the needs of the architecture. In run-time the protocol kernel supports the exchange of data (messages) and control information between layers and provides a number of auxiliary services such as timer management and memory management for message buffers.

The TTCB, like the Operating System (OS), is a component that might have its services called by every other module. Unlike the OS, the TTCB can be a fairly modest and simple component of the system, with properties well-defined and preserved by construction. The TTCB exhibits a fail-controlled behavior, i.e., it fails only by crashing, even in the presence malicious faults, and regardless of the characteristics of the applications using its services. It can be seen as an assistant for parts of the execution of the protocols and applications, since it provides a small set of trusted services related to time and security.

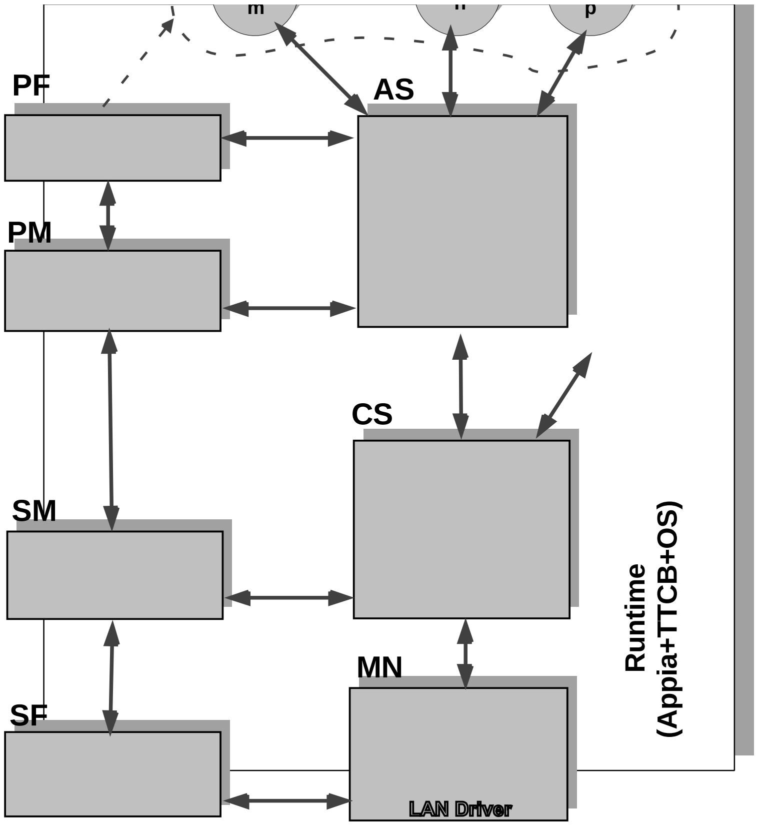

Figure 1.1: Architecture of a MAFTIA Host

The lowest module of the middleware architecture is the Multipoint Network, MN, created on top of the physical infrastructure. The MN hides the particulars of the underlying network to which a given site is directly attached, and is as thin as the intrinsic properties of the latter allow. The main services that it provides are multipoint addressing and best-effort message delivery, basic secure channels and message envelopes. Its API is composed by the standard interfaces to well-known protocols, such as TCP/IP and SNMP, that might be used by the modules above it.

Communication Support, CS, is the core module related to data interchange among sites. It depends on the information given by the Site Membership, SM, module about the composition of the groups, and on the MN to access the network. The CS module implements cryptographic group communication primitives, such as Byzantine agreement or message broadcast, and other core services. The group communication primitives are provided with several reliability and ordering guarantees, such as causal or atomic, and are defined in terms of several programming models: asynchronous, timed with support from the TTCB, or asynchronous with assistance from the TTCB. From all the combinations of guarantees and models that are possible, only a subset are going to be offered.

The Activity Support module, AS, implements building blocks that assist the development of specific classes of applications, such as distributed transactional support and replication management. Its main purpose is to offer top-level interfaces that will make the access to CS protocols and interfaces easier, and provide functionality that will simplify the construction of those applications. This deliverable describes essentially the transactional support APIs and services. It presents a typical transactional architecture and introduces its most important components, such as the resource managers and transaction managers, and then defines their interfaces.

The Site Failure Detector module, SF, as its name indicates, determines which sites have failed. This module can use the services of the TTCB to offer a reliable failure detection, since the TTCB is synchronous and secure. If a TTCB is not present, the SF module might not provide completely correct information because it depends on the conditions of the network, which has uncertain synchrony and might be prone to attacks. Site Membership, SM, keeps and updates the information related to the set of sites that are registered and in the current view (currently trusted sites). It depends on the information given by the SF module. The Participant Failure Detector module, PF, assesses the liveness and correctness of all local participants, based on information provided by the operating system.

In this deliverable, we do not present the *internal* interface of the modules, which will have to be specified in terms of Appia events and other constructs. This interface is currently being developed, and its definition mostly important only to the partners involved in the production of a middleware prototype (deliverable D25 - Running lab prototype of MAFTIA middleware, due six months from now).

2 Runtime Support Services and APIs

2.1 Appia

This section describes the API of Appia. Appia is a layered communication framework implemented in Java, providing extended configuration and programming possibilities. The conceptual model behind Appia is described in several papers [26, 25]. More information about the framework and the latest updates can be retrieved at the Appia website [2].

Appia is a general purpose framework that is being used in several research projects. Although Appia is not a fully developed component solely for MAFTIA, we decided to include this section in the deliverable mainly for two reasons: first, for completeness, since most of the development of MAFTIA middleware will be done within this framework, it is important to understand the capabilities and support provided by Appia; second, because MAFTIA influenced the development of Appia. When the MAFTIA project endorsed Appia, it was still in an early stage of design, and measures were taken in order that an adequate effort was put into it so that the deadlines demanded by MAFTIA were met, and more importantly, that its structure and API would meet the needs of the MAFTIA project.

This section presents the class signatures and details their usage and function.

2.1.1 Overview

Networked inter-process communication requires that several distinguishable properties be combined in order to provide the derived service.

Some networking standards, detailing the provided properties, have been developed and are now widely used. This is the case of the Internet Protocols such as IP, TCP, UDP [34, 35, 32] and the OSI model [53]. Most of them assume a layering model, having each protocol piled over another. Each protocol relies on the documented properties of the protocols below to provide his service to the layers above. Transmission Control Protocol (TCP), for instance, relies on routing capabilities of IP to ensure that the sent packets will be delivered to the correct destination. As IP does not ensure FIFO ordering, TCP provides this property.

Each combination of layers (protocols) on the stack provide a different set of properties[1] and can be considered as the service provided by the stack. The properties resulting from each combination and the protocols used in the stack are used interchangeably in this document and referred as the *Quality of Service* (QoS).

Appia is a layered communication support framework. Its mission is to define a standard interface to be respected by all layers and facilitate communication among them. Appia is protocol independent. That is, the framework layers any protocol as long as it respects the predefined interface, making no provisions to validate the final composition result.[2]

These services can be found in several previous works. For a comparison see [26].

2.1.2 Appia Concepts

This section briefly describes the concepts and terminology used in Appia.

Static and dynamic concepts Appia presents a clear distinction between the declaration of something (either a protocol or a stack) and its implementation.

A **Layer** is defined by the properties a protocol requires and those it will provide. A **Session** is a running instance of a protocol. A Session is always created on behalf of a layer and its state is independent from other instances.

A **QoS** is a static description of an ordered set of protocols. A **Channel** is a dynamic instantiation of a QoS. Protocol instances (sessions) communicate using channel infrastructure.

All these concepts are illustrated in Figure 2.1 and Table 2.1.

As they are static, layers do not exchange information between themselves. Instead, they declare the communication interface of their dynamic instantiations, the sessions. Communication between sessions and with the channel is made using **Events**. Appia provides a predefined set of events, each with a different meaning but programmers are encouraged to extend this set to detail protocol specific occurrences. Starting from the session which generated it, events flow through the stack in a predefined direction. The information contained in any particular event extends a basic set of fields that all events must contain.

[1] Different orderings of protocols can also provide different sets of properties.

[2] In fact, Appia provides a limited form of stack validation.

Reusability Reusability in Appia is based on inheritance. Since most of the protocols depend (at least weakly) on the service provided by others, upgrading some may produce incompatibilities. Appia uses inheritance to make the upgrades transparent. When a new version of a protocol is released, it is expected that the generated events will have richer information than the previous version. Assuming that none of the previously provided information format is changed, protocols may simply create new events extending previous ones. This way, protocol backward compatibility is assured.

Optimization Inheritance is also used to improve protocol performance. Timer events, for instance, are generated by protocols (as requests) and handled by the channel. Any session is free to extend the standard timer events, allowing it to add information that otherwise would have to be kept in the session state. A reliable delivery protocol for instance may include the message to be retransmitted in the timeout request event. When the timeout occurs, the session simply peeks the message from the event and resends it.

Event processing time is reduced by preventing protocol instances from handling unwanted events. Each protocol registers his interest in receiving event classes. Events of classes not declared are not delivered to the corresponding sessions.

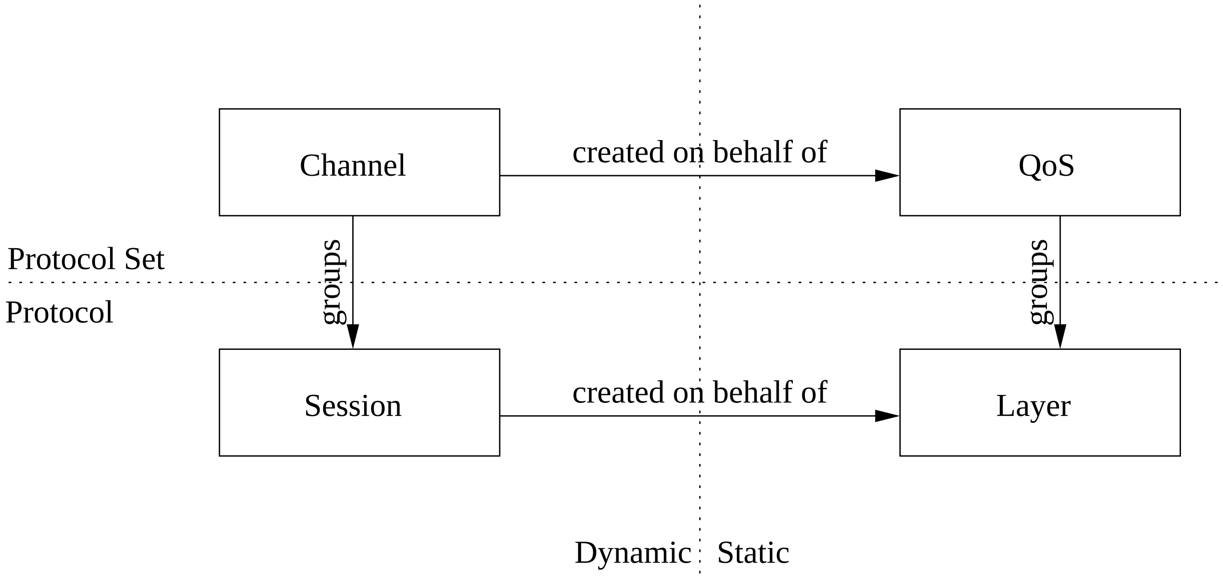

Figure 2.1: Relation between sessions, layers, channels and QoS's

Protocol definition Each protocol is defined by two different classes: one extending the Layer class and the other extending the Session class. By convention, the former is usually

Concept	**Behavior**	**#**	**Description**
Layer	Static	1	The static description of a protocol. The properties it requires and provides.
Session	Dynamic	n	Execution instance of a protocol. Keeps the protocol state and provides the properties described in the corresponding layer.
QoS	Static	1	An ordered set of layers. Describes the properties that a running instance of that combination of protocols would have.
Channel	Dynamic	n	An ordered set of sessions, modeled by one QoS. The entity providing the set of properties specified in the QoS.

# The expected number of instances per protocol/protocol set in an Appia process.

Table 2.1: Relation between static and dynamic concepts in Appia

named *Protocol*Layer and the later *Protocol*Session having *Protocol* to be the name of the protocol.

The *Protocol*Layer class is the one participating in QoS definitions. Its purpose is to export the event sets and to create instances of the *Protocol*Session class.

The *Protocol*Session class is the one participating in channels and executing protocol instances. It has two main goals: to cooperate in channel definitions and to handle and generate events, providing the properties expected from the protocol.

Relation between sessions and channels In Appia, a session (i.e. a running instance of a protocol) may participate in several channels simultaneously even if they have different QoS's. This means that a single protocol instance can participate in multiple protocol combinations.

This is one of the innovative aspects of Appia and offers a new perspective on the way different kinds of data are related. For instance, by having only one single FIFO session on two channels, one with an appropriate QoS for video transmission and another for audio, the receiver imposes the sending order of messages across the two media without any additional programming effort.

Whether sessions deal transparently with multiple channels or not is implementation and protocol dependent. On event reception, sessions are free to query the event's channel. Events can be forwarded without sessions knowing the channel being used.

Implementation classes There are eleven classes that are relevant for layer and application implementation in Appia: QoS, Channel, ChannelCursor, Layer, Session, Event, Message, MsgWalk, MsgBuffer, Direction and Appia. Figures 2.2, 2.3 and 2.4 present the UML models of the framework. The remaining classes of Appia are not presented as they do not provide relevant features to protocol and application development.

Notation Methods and classes are presented using usual object-oriented languages notation.

Classes always have an upper-case first character while methods are identified by a lower-case first character. The remaining characters will be lower case except when a new word is started.

The existence of optional arguments is signaled by the presentation of different methods with the same name. This document presents only the interface relevant for the user. By default, presented methods and attributes have no access restritions and are qualified as public.

2.1.3 API Description by Class

This section describes the interface of the classes that are relevant for protocol and application development. Class descriptions are ordered from the more generic to the more specific concept, attempting to avoid forward references.

Each class interface is introduced by a description of the role the class plays in the framework and how it is normally used by protocol and application programmers.

Class QoS A Quality Of Service is a set of properties, each independently provided by one protocol.

QoS missions are to glue protocols (presented as layers), attempt to validate the resulting composition and define the interaction rules between the protocols. At QoS definition time, layers declare the events their sessions are interested to receive. Using this knowledge, QoS builds for each event class an “event path”, including only the layers that are interested in receiving it. The information extracted can then be used to efficiently create channels.

Class QoS defines the Qualities of Services that will be available to the application. From the programmer’s point of view, a QoS instance is simply an array of layers.

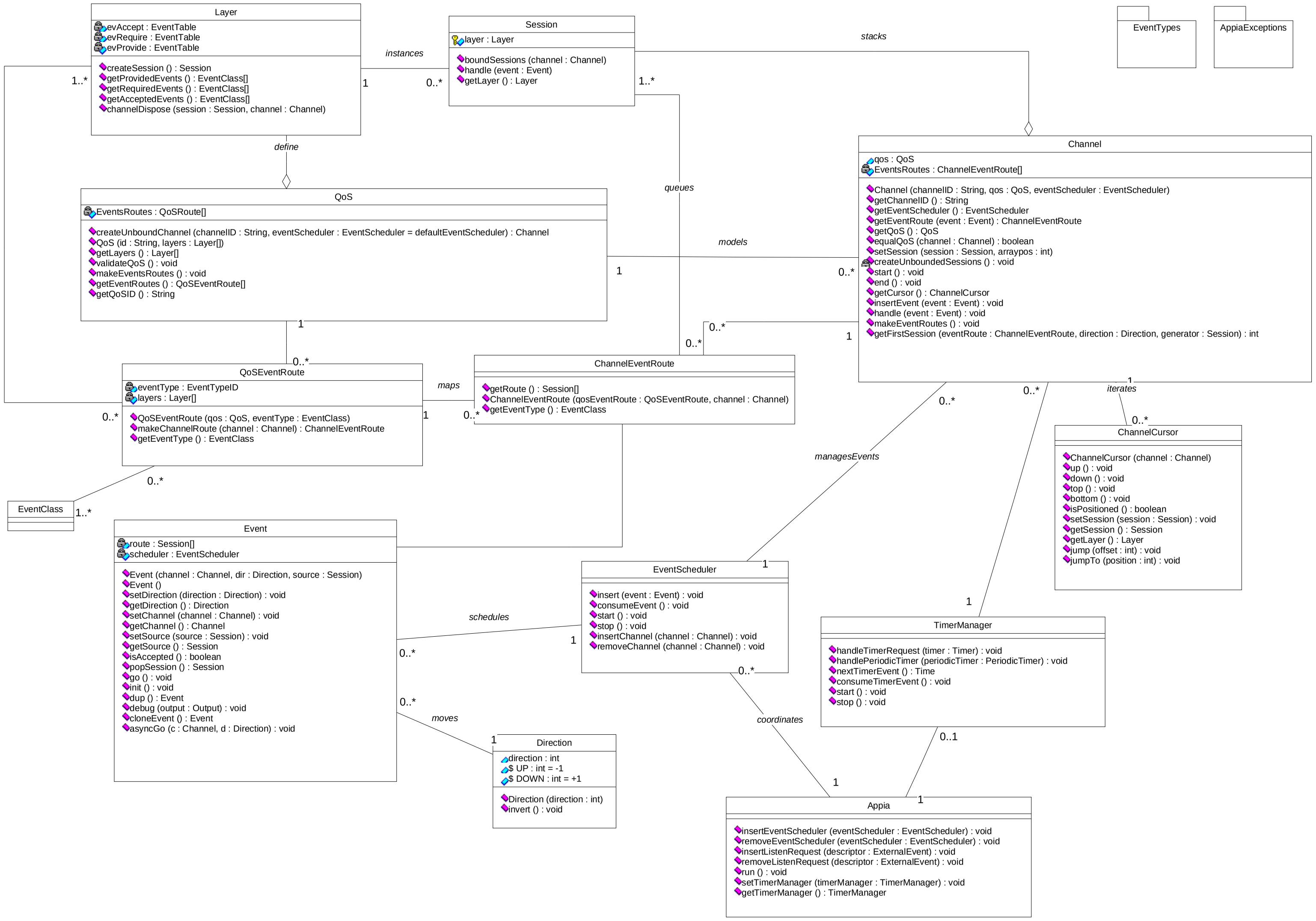
Layer
evAccept : EventTable
evRequire : EventTable
evProvide : EventTable
createSession () : Session
getProvidedEvents () : EventClass[]
getRequiredEvents () : EventClass[]
getAcceptedEvents () : EventClass[]
channelDispose (session : Session, channel : Channel)
instances
1
0..*
1..*
Session
layer : Layer
boundSessions (channel : Channel)
handle (event : Event)
getLayer () : Layer
1..*
stacks
EventTypes
AppiaExceptions
define
QoS
EventsRoutes : QoSRoute[]
createUnboundChannel (channelID : String, eventScheduler : EventScheduler = defaultEventScheduler) : Channel
QoS (id : String, layers : Layer[])
getLayers () : Layer[]
validateQoS () : void
makeEventsRoutes () : void
getEventRoutes () : QoSEventRoute[]
getQoSID () : String
queues
models
1
0..*
Channel
qos : QoS
EventsRoutes : ChannelEventRoute[]
Channel (channelID : String, qos : QoS, eventScheduler : EventScheduler)
getChannelID () : String
getEventScheduler () : EventScheduler
getEventRoute (event : Event) : ChannelEventRoute
getQoS () : QoS
equalQoS (channel : Channel) : boolean
setSession (session : Session, arraypos : int)
createUnboundedSessions () : void
start () : void
end () : void
getCursor () : ChannelCursor
insertEvent (event : Event) : void
handle (event : Event) : void
makeEventRoutes () : void
getFirstSession (eventRoute : ChannelEventRoute, direction : Direction, generator : Session) : int
1
0..*
0..*
1
QoSEventRoute
eventType : EventTypeID
layers : Layer[]
QoSEventRoute (qos : QoS, eventType : EventClass)
makeChannelRoute (channel : Channel) : ChannelEventRoute
getEventType () : EventClass
0..*
maps
1
0..*
ChannelEventRoute
getRoute () : Session[]
ChannelEventRoute (qosEventRoute : QoSEventRoute, channel : Channel)
getEventType () : EventClass
0..*
0..*
1
iterates
0..*
ChannelCursor
ChannelCursor (channel : Channel)
up () : void
down () : void
top () : void
bottom () : void
isPositioned () : boolean
setSession (session : Session) : void
getSession () : Session
getLayer () : Layer
jump (offset : int) : void
jumpTo (position : int) : void
0..*
EventClass
1..*
managesEvents
Event
route : Session[]
scheduler : EventScheduler
Event (channel : Channel, dir : Direction, source : Session)
Event ()
setDirection (direction : Direction) : void
getDirection () : Direction
setChannel (channel : Channel) : void
getChannel () : Channel
setSource (source : Session) : void
getSource () : Session
isAccepted () : boolean
popSession () : Session
go () : void
init () : void
dup () : Event
debug (output : Output) : void
cloneEvent () : Event
asyncGo (c : Channel, d : Direction) : void
1
EventScheduler
insert (event : Event) : void
consumeEvent () : void
start () : void
stop () : void
insertChannel (channel : Channel) : void
removeChannel (channel : Channel) : void
1
schedules
0..*
1
0..*
1
TimerManager
handleTimerRequest (timer : Timer) : void
handlePeriodicTimer (periodicTimer : PeriodicTimer) : void
nextTimerEvent () : Time
consumeTimerEvent () : void
start () : void
stop () : void
0..*
moves
1
Direction
direction : int
$ UP : int = -1
$ DOWN : int = +1
Direction (direction : int)
invert () : void
coordinates
0..1
1
1
Appia
insertEventScheduler (eventScheduler : EventScheduler) : void
removeEventScheduler (eventScheduler : EventScheduler) : void
insertListenRequest (descriptor : ExternalEvent) : void
removeListenRequest (descriptor : ExternalEvent) : void
run () : void
setTimerManager (timerManager : TimerManager) : void
getTimerManager () : TimerManager

Figure 2.2: Appia main UML

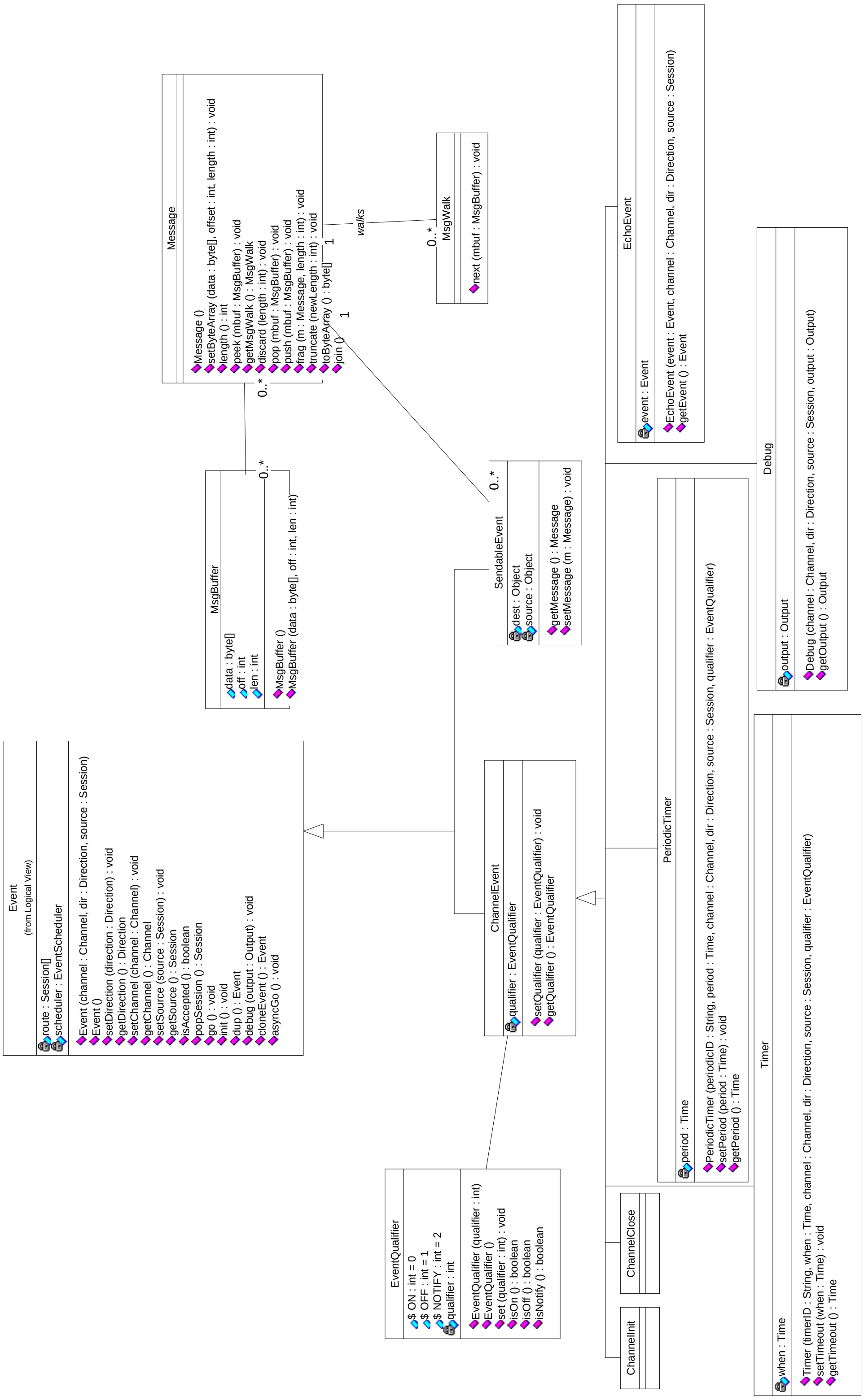
Message
Message ()
setByteArray (data : byte[], offset : int, length : int) : void
length () : int
peek (mbuf : MsgBuffer) : void
getMsgWalk () : MsgWalk
discard (length : int) : void
pop (mbuf : MsgBuffer) : void
push (mbuf : MsgBuffer) : void
frag (m : Message, length : int) : void
truncate (newLength : int) : void
toByteArray () : byte[]
join ()
walks
1
0..*
MsgWalk
next (mbuf : MsgBuffer) : void
0..*
1
0..*
MsgBuffer
data : byte[]
off : int
len : int
MsgBuffer ()
MsgBuffer (data : byte[], off : int, len : int)
0..*
SendableEvent
dest : Object
source : Object
getMessage () : Message
setMessage (m : Message) : void
EchoEvent
event : Event
EchoEvent (event : Event, channel : Channel, dir : Direction, source : Session)
getEvent () : Event
Debug
output : Output
Debug (channel : Channel, dir : Direction, source : Session, output : Output)
getOutput () : Output
Event
(from Logical View)
route : Session[]
scheduler : EventScheduler
Event (channel : Channel, dir : Direction, source : Session)
Event ()
setDirection (direction : Direction) : void
getDirection () : Direction
setChannel (channel : Channel) : void
getChannel () : Channel
setSource (source : Session) : void
getSource () : Session
isAccepted () : boolean
popSession () : Session
go () : void
init () : void
dup () : Event
debug (output : Output) : void
cloneEvent () : Event
asyncGo () : void
ChannelEvent
qualifier : EventQualifier
setQualifier (qualifier : EventQualifier) : void
getQualifier () : EventQualifier
PeriodicTimer
period : Time
PeriodicTimer (periodicID : String, period : Time, channel : Channel, dir : Direction, source : Session, qualifier : EventQualifier)
setPeriod (period : Time) : void
getPeriod () : Time
EventQualifier
$ ON : int = 0
$ OFF : int = 1
$ NOTIFY : int = 2
qualifier : int
EventQualifier (qualifier : int)
EventQualifier ()
set (qualifier : int) : void
isOn () : boolean
isOff () : boolean
isNotify () : boolean
ChannelClose
ChannelInit
Timer
when : Time
Timer (timerID : String, when : Time, channel : Channel, dir : Direction, source : Session, qualifier : EventQualifier)
setTimeout (when : Time) : void
getTimeout () : Time

Figure 2.3: Appia predefined events UML

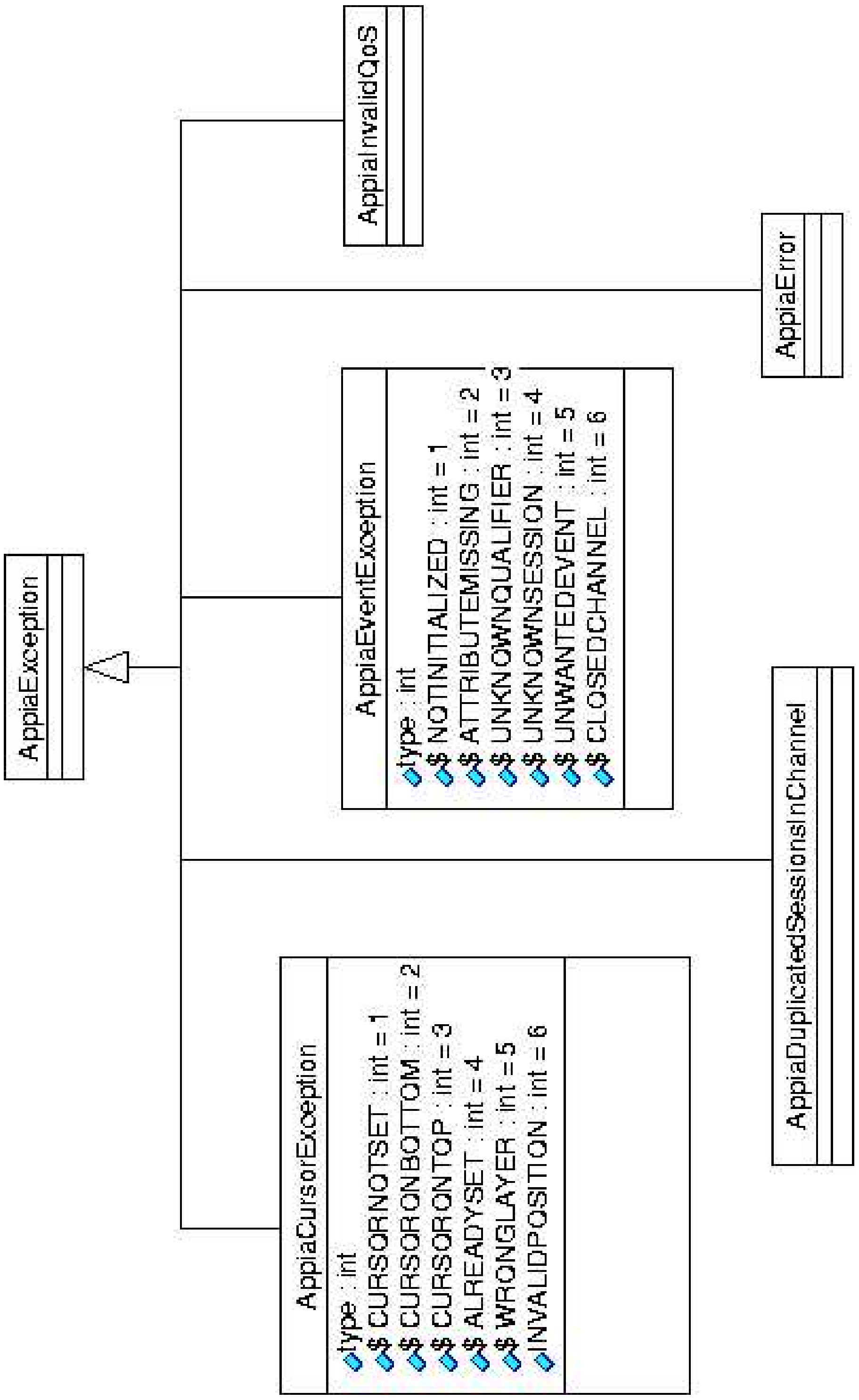
AppiaException
AppiaInvalidQoS
AppiaError
AppiaEventException
type : int
$ NOTINITIALIZED : int = 1
$ ATTRIBUTEMISSING : int = 2
$ UNKNOWNQUALIFIER : int = 3
$ UNKNOWNSESSION : int = 4
$ UNWANTEDEVENT : int = 5
$ CLOSEDCHANNEL : int = 6
AppiaDuplicatedSessionsInChannel
AppiaCursorException
type : int
$ CURSORNOTSET : int = 1
$ CURSORONBOTTOM : int = 2
$ CURSORONTOP : int = 3
$ ALREADYSET : int = 4
$ WRONGLAYER : int = 5
INVALIDPOSITION : int = 6

Figure 2.4: Appia framework exceptions UML

One optional argument of the createUnboundChannel method is the EventScheduler. Appia configuration options allow programmers to define event scheduling policies by redefining this class. The default implementation of the EventScheduler class is single threaded and puts all events in a FIFO queue. The internals of the EventScheduler class lie outside the scope of this document and can be found elsewhere [24].

```
class QoS {
    QoS(String qosID, Layer[] layers) throws AppiaInvalidQoS;
    Channel createUnboundChannel(String channelID, EventScheduler
        eventScheduler);
    Channel createUnboundChannel(String channelID);
    Layer[] getLayers();
}
```

Class Channel Channels are instantiations of QoS's. Channels glue sessions the same way QoS's glue layers. A Channel is created on behalf of a QoS type. When a channel is created, it inherits the knowledge captured from the layers in that QoS, improving performance.

On channel creation, event paths are exported from the QoS. The channel maps the layers on the QoS event paths into the bound session to route events.

Channels also provide the background run-time environment for session execution. They are responsible, for instance, for providing timers. The ChannelEvent sub-class of events is dedicated to these operations.

Channel definition Upon creation, a channel is as an array of "typed empty slots". Each of these slots must be filled with a session of the layer specified in the QoS for that position. Sessions can be bound to the slots explicitly (by the user) or implicitly by other sessions (automatic binding). By default, new sessions will be bound to the remaining slots.

Using explicit binding it is possible to associate specific sessions to specific channels. These sessions may either be already in use by other channels or may be intentionally created for the new channel. Explicit binding enables the user to have fine control over the channel configuration.

Using automatic binding it is possible to delegate to already bound sessions the task of specifying the remaining sessions for the channel. Typically, a mixture of explicit and automatic binding is used.

Both explicit and automatic binding are performed on a ChannelCursor object, which can be obtained from the Channel. Explicit binding must be performed prior to calling the start method of the channel. One of the tasks of this method is to ensure that every slot is fulfilled with a valid object. The first step performed by start is to invite sessions explicitly bounded to perform automatic binding by calling their boundSessions method. For those slots not explicitly or automatically bounded, the start method requests the creation of a new session from the corresponding layer.

Channel initialization and termination A channel is instructed to start and stop by its methods start and end. Besides the operations concerning session instantiation performed by start, both methods introduce an event in the channel. The Start event is supposed to be the first to flow in a channel. Protocols should be aware that events created in response to handling a Start event must be inserted after invoking the go method on the Start event. Although this requirement is not mandatory and does not produce inconsistencies in Appia, other protocols may rely on this property.

The end method introduces a Stop event in the channel. Sessions receiving the Stop event may not introduce more events in the channel but must be prepared to receive others. Received events may be propagated.

A stopped channel may latter be restarted by again calling the start method. However, for temporary suspensions, protocols should consider using EchoEvents to obtain the same feature.

```
class Channel {
    String getChannelID();
    QoS getQoS();
    boolean equalQoS(Channel channel);
    void start();
    void end();
    ChannelCursor getCursor();
}
```

Class ChannelCursor Channel cursors are helpers for channel session specification. The class provides a set of methods for iterating over the channel stack, retrieving references to already defined sessions and setting sessions for the yet empty slots. Methods of this class raise AppiaCursorException exceptions when a invalid operation is done.

Initially, the cursor is not positioned over any position on the channel. The initial position must be defined by either the top or bottom methods. Scrolling below the bottommost position of the channel or above the uppermost will also set the cursor to the not positioned state.

```
class ChannelCursor {

    void top();
    void bottom();
    void jumpTo(int position) throws AppiaCursorException;
    void down() throws AppiaCursorException;
    void up() throws AppiaCursorException;
    void jump(int offset) throws AppiaCursorException;
    boolean isPositioned();
    Session getSession() throws AppiaCursorException;
    void setSession(Session session) throws AppiaCursorException;
    Layer getLayer() throws AppiaCursorException;
}
```

Class Layer Layers are the static representatives of micro-protocols. They describe the behavior of micro-protocols. Layers are used in QoS definitions to reserve a specific position for a session implementing the protocol and to declare the needed, accepted and generated events, respectively, via the evRequire, evAccept and evProvide attributes.

Layers are responsible for instantiating sessions (in response to calls to the method createSession) and are notified by the channel whenever one session is dismissed by a channel (by calls to the method channelDispose).

```
class Layer {

    Class[] evProvide;
    Class[] evRequire;
    Class[] evAccept;
    Session createSession();
    void channelDispose(Session session, Channel channel);
}
```

Class Session A session is the dynamic part of a micro-protocol. Sessions maintain the state of a micro-protocol instance and provide the code necessary for its execution. Channels provide the connection between the different sessions of a stack. A session keeps a relation of "one-to-many" with channels: one single session can be part of multiple channels. A session is defined as *channel-aware* if its algorithm recognizes and acts differently upon reception of events flowing from different channels. Many of the protocols that can be found in existing stacks are channel-unaware. When a channel is being defined, sessions already bound to the channel may be invited to bind other sessions. The invitation is made by a call to the boundSessions method.

Sessions communicate with their environment by events. Reception of events is made by the handle method. A session can learn the channel that is delivering an event to it by querying the channel attribute of the Event.

```
class Session {
    protected Layer layer;
    Layer getLayer();
    void boundSessions(Channel channel);
    void handle(Event event);
}
```

Class Direction Class Direction is an implementation support class of Appia. It qualifies an event stating the direction it is flowing. The direction attribute accepts two values UP and DOWN defined as static constants.

```
class Direction {
    int direction;
    static final int UP=1;
    static final int DOWN=2;
    Direction(int direction);
    void invert();
}
```

Class Event Sessions use events to communicate with the surrounding environment. This class contains the attributes necessary for the event routing. In Appia, events can be freely defined by the protocol programmers as along as all descend from the main Event class. Programmers should be aware that sub-classing should be done as deeply as possible on the sub-classing tree, improving event sharing and compatibility among different micro-protocols.

The Event class has three attributes that must be defined prior to the event insertion in a channel. For each, a pair of set and get methods is defined. The attributes are:

direction Stating the direction of the movement (up or down).

channel Stating the channel where the event will flow.

source Stating the session that generated the event. This attribute is important to determine the event route.

The attributes can be defined either by the constructor or by the individual *set* methods. When methods are used, the method init must be invoked after all attributes are defined and prior to the invocation of the go method.

The cloneEvent method uses the Java clone operation of the Object class. Redefinition of this method should always start by invoking the same method on the parent classes.

Concurrency control Appia is not thread-safe in the sense that consistency is not ensured if protocols insert events in the channel while not owning the Appia main thread. However, a thread-safe event method, with a particular semantics, is provided.

The asyncGo method should be called only when an event is inserted asynchronously (i.e. concurrently with the Appia main thread) in the channel. If the direction defined at the event is UP, asyncGo will place the event at the bottom of the channel. Otherwise, the event will be placed at the top of the channel. The event will then present the same behavior as any other, respecting the FIFO order while crossing the channel and only visiting the sessions of the protocols that declared it in the accepted set. Events inserted in a channel using the asyncGo method should not be initialized either by the constructor or by the init method.

Asynchronous events are particular useful for protocols using their own thread to execute, like those receiving information from outside the channel. Examples of such protocols are those listening to a socket to retrieve incoming messages. When an incoming network message arrives, the session can use these events to request the synchronous delivery of the event through the Appia main thread.

Note: Protocol programmers should be aware that the asynchronous insertion of events in the channel must be handled with particular care since it subverts the usual event behavior. Events inserted assynchronously travel directly to the end of the stack, prior to being inserted. This does not respect possible causal dependencies between events. Furthermore, programmers should be aware that the use of asynchronous events may subvert the ordering of the stack. Consider the example of the previous paragraph. If some protocol is below the protocol receiving messages from the network, it should not be presented with incoming network messages, that are expected to be sent toward the top of the stack.

This problem may occur if the event type used for the asynchronous event is the one used for sending the message to the stack.

```
class Event {

    Event(Channel channel,Direction dir,Session source) throws
        AppiaEventException;
    Event();
    void init() throws AppiaEventException;
    void setDirection(Direction dir);
    Direction getDirection();
    void setChannel(Channel channel);
    Channel getChannel();
    void setSource(Session source);
    Session getSource();
    void go() throws AppiaEventException;
    void asyncGo(Channel c, Direction d) throws AppiaEventException;
    Event cloneEvent() throws CloneNotSupportedException;
}
```

Class EventQualifier The event qualifier class differentiates channel events with one of three types: ON, OFF and NOTIFY. The precise interpretation of these values will depend on the qualified event type. However, a common usage pattern is defined:

ON is used for setting requests or starting a mode or operation. OFF is intended for abortion of requests or mode cancellation. NOTIFY is used for notifications of occurrences.

```
class EventQualifier {

    static final int ON=0;
    static final int OFF=1;
    static final int NOTIFY=2;
    EventQualifier(int qualifier) throws AppiaEventException;
    EventQualifier();
    void set(int qualifier) throws AppiaEventException;
    boolean isSet();
    boolean isCancel();
    boolean isNotify();
}
```

Class ChannelEvent The ChannelEvent class is the topmost class grouping all channel related events. That is, all events provided by the channel or containing requests for services provided by the channel. This class, descendant of the main Event class, includes the attribute qualifier of type EventQualifier, allowing the channel to determine the type of operation to be performed. Instances of the ChannelEvent class are never created. Its subclasses are used to detail the requested or provided operation.

```
class ChannelEvent extends Event {
    void setQualifier(EventQualifier qualifier);
    EventQualifier getQualifier();
}
```

Class EchoEvent EchoEvent events are event carriers. When a EchoEvent reaches one of the sides of the channel, the event passed to the constructor is extracted and inserted in the channel in the opposite direction. No copies are realized: the inserted object instance is the same that was given to the EchoEvent.

EchoEvents allow protocols, for example, to perform composition introspection, like learning the available maximum PDU size, or to perform requests to other protocols like temporarily suspending the channel activity.

The carried event will be initialized prior to being inserted in the channel. The main Event class attributes will be set as if the event has been launched by the channel. The protocol launching this event should declare itself as the provider of the event.

```
class EchoEvent extends ChannelEvent {
    EchoEvent(Event event, Channel channel, Direction dir, Session source);
    Event getEvent();
}
```

Classes Timer and PeriodicTimer Appia offers periodic and aperiodic timer notification services. To request a aperiodic timer, sessions should send a Timer event to the channel. The direction the event flows and the EventQualifier attribute of the event distinguish requests from notifications. Table 2.2 presents the expected combinations. The attributes declared by a Timer extend those available in the ChannelEvent with a String and the time in milliseconds that the notification should occur. When issuing a timer request, the EventQualifier attribute must be set to ON.

Programmers are encouraged to extend the basic Timer class. This will impact

Operation	**Direction**	**Qualifier**
Request	DOWN	ON
Cancellation	DOWN	OFF
Notification	UP	NOTIFY

Table 2.2: Expected combinations of Directions and Qualifiers on Timers operations in an Appia execution

performance at two different levels. If the event type declared in the provided and accepted events for the protocol matches the newly defined event type, notifications requested by other protocols will not consume wasteful resources of this protocol. On the other hand, the new class may encompass any information required by the protocol to handle the timeout. This improves protocol execution time. When the timeout is delivered to the application, the same object instance is delivered to the protocol. The qualifier attribute will be set to NOTIFY and the direction attribute will have a value inverse to the one defined at timer request.

Cancellation of a timer is requested by the protocol issuing a new timer event with the same timer ID and an OFF qualifier. Note that event cancellation can not be ensured by Appia: the notification event may already be inserted in the channel when the cancellation reaches the bottom of the channel.

```
class Timer extends ChannelEvent {

    Timer(String timerID, long when, Channel channel, Direction dir,
        Session source, EventQualifier qualifier) throws AppiaException;
    void setTimeout(long when);
    long getTimeout();
}
```

Periodic timers differ from aperiodic timers by accepting a time interval instead of an absolute local clock time. The semantics associated with PeriodicTimer events is that a notification is due every “period” milliseconds. Appia only ensures that no more events will be raised than periods expire.

The object delivered upon timer expiration will be a copy of the original object. The copy is performed using the cloneEvent method. Specialization can also be used to redefine this method in order to provide a different semantics from that initially defined which is to perform a deep copy of all attributes except the timerID (which has its reference copied). If redefined, cloneEvent should start by calling its parent cloneEvent method. After issuing a request to cancel a periodic timer, a undefined number of notifications, those already inserted in the channel, can be received.

```
class PeriodicTimer extends ChannelEvent {
    PeriodicTimer(String timerID, long period, Channel channel, Direction
        dir, Session source, EventQualifier qualifier) throws AppiaException;
    void setPeriod(long period);
    Time getPeriod();
}
```

Note: Appia provides weak time delivery guarantees for notification as this may compromise the event FIFO ordering within the channel. The only guarantee provided is that notifications will be raised by the timer manager *after* the requested timeout period has expired.

Class SendableEvent SendableEvents are one of the branches of the event tree defined by Appia. The semantics expected to be applied by protocols regarding SendableEvents is that those are the events to be sent to the network. Non SendableEvents are supposed to be local to the channel that created them.

SendableEvents extend the basic event class with three attributes: source, dest and message. These attributes are of type java.lang.Object. Their instantiation type is supposed to be agreed by the protocols composing the channel and can even change while the event crosses the stack. It is expected that most of the layers use them transparently relying only in equality operations. It is therefore advised that value based comparison operations should be defined for the chosen class.

When retrieving SendableEvents (or any of its subclasses) from the network, protocols are expected to satisfy at least the following conditions on the event inserted in the receiving endpoint:

- All attributes of the Event class should be correctly filled; The creator of the event is the session that retrieved the event from the network and will insert it in the channel;
- Source and dest attributes are equal to the ones received by the session that sent the event to the network;
- The message attribute has the same sequence of bytes received by the session that sent the event to the network;
- The event type should be the same;

Note that besides the event type, no special requirements are imposed for sending subclasses of SendableEvents. In particular, attributes not inherited from SendableEvent are not expected to be passed to the remote endpoint. This is the behavior of the current protocols that interface the network, namely UDPSimple, TCPSimple and TCPComplete.

Messages are set and retrieved by two specific operators. Class Message is defined later in this document.

```
class SendableEvent extends Event {
    public Object dest;
    public Object source;
    SendableEvent();
    SendableEvent(Channel channel, Direction dir, Session source) throws
        AppiaEventException;
    SendableEvent(Message msg);
    SendableEvent(Channel channel, Direction dir, Session source, Message
        msg) throws AppiaEventException;
    Message getMessage();
    void setMessage(Message m);
}
```

Class Message The class Message provides an encapsulation of an array of bytes with methods providing efficient operations for adding and removing headers and trailers. The class was conceived as the principal method for inter-channel communication.[3] Message provides an interface for sessions to push and pop headers of byte arrays. Message interface is mainly imported from the x-Kernel [28]. The use of message was devised assuming that the layer responsible for sending messages to the network has weak serialization capabilities. Although this is not the case in some programming languages (for instance, Java), using this facility may raise some additional problems:

Over serialization Java default serialization procedures places the object and all its references in a stream. In Appia, an event contains references to the channel he is running and the event scheduler being used which in turn contain references for all sessions and for all events currently on the channel. Serializing an event in a straightforward way will transfer a huge amount of irrelevant information.

Language independence Java serializes objects in a language specific byte array. Compatibility between channels coded in different languages would be strongly compro-

[3] *Inter-channel* communication is defined as the means by which channels on different processes exchange information. This is the opposite of *intra-channel* communication, ideally performed by event attributes.

mised.

The class has only an empty constructor. To initialize a message instance with an array of bytes, one should call setByteArray, specifying the first position in the source array and the number of bytes to be copied. Most of the remaining methods take a MsgBuffer as an argument.[4] All push, peek and pop operations (which respectively add, query and extract an header) are called with the len attribute of MsgBuffer defined. The remaining values are ignored and overlaped by the method execution. When the call returns, the off attribute points to the first position in the data buffer where the header is stored or can be retrieved.

The sequence of actions performed to push an header is:

1. Prepare a MsgBuffer object with the lenght of the header;
2. Invoke the push method;
3. Copy the header to the data array, starting at the position indicated by offset;[5]

Popping an header requires the same sequence of actions to be performed, retrieving the data in step 3.

Note: The byte array presented to the protocol will tipically be larger than the required lenght. Most of the time, the remaining positions will have headers of other protocols in the channel. Appia makes no provisions to ensure that protocols act accordingly to this specification.

Iterating over an entire message (for checksumming or encryption) is made with the MsgWalk class.

[4] The goal of the MsgBuffer is to avoid memory copies. This class is described later in this document.

[5] The only restriction is that the header must be defined prior to calling the go method on the event owning the message, so, to avoid memory copies, the header can be constructed directly in the buffer.

```
class Message {
    Message();
    void setByteArray(byte[] data, int offset, int length);
    int length();
    void peek(MsgBuffer mbuf);
    void discard(int length);
    void push(MsgBuffer mbuf);
    void pop(MsgBuffer mbuf);
    void truncate(int newLength);
    void frag(Message m,int length);
    void join(Message m);
    MsgWalk getMsgWalk();
    byte[] toByteArray();
}
```

Class MsgBuffer The MsgBuffer class is used as an helper class to operations over messages. The goal of this class is to improve performance by avoiding message copies.

The MsgBuffer class is used to pass arguments to and receive arguments from methods of the Message class. The fields are used with the following meaning:

data An array of bytes retrieved from or to be included in the message;

off The first position in the array data containing information relevant to the operation. Respecting the usual array representation, the first position of an array has offset 0;

len The number of bytes of the array data relevant to the operation;

Array data positions not between **off** and **off+len-1** are reserved and can not be used.

Instances of this class always have the same usage pattern: the user fills the len attribute of one instance and invokes the method passing the instance as an argument. When the method returns, the data, off and len attributes will be appropriately filled. In peek, pop and next (from the MsgWalk class) the array contains the data retrieved from the message. In push the array contains the space to be filled with the headers by the session.

```
class MsgBuffer {
    byte[] data;
    int off;
    int len;
    MsgBuffer();
    MsgBuffer(byte[] data, int off, int len);
}
```

Class MsgWalk MsgWalk objects are iterators over messages. This class is intended to be used by protocols operating on the entire message buffer such has checksum or cipher protocols. The array returned by the next method can be used for reading and writing but no data can be appended or deleted from the message.

```
class MsgWalk {
    void next(MsgBuffer mbuf)
}
```

Objects Message As an extension to the default behavior, Appia provides the ObjectsMessage class that enriches Message with the methods to push and pop serializable objects. Since ObjectsMessage extends Message, the interface is transparent to protocols using the parent class. One ObjectsMessage object supports the interleaved use of both types of headers.

Note: For performance reasons, the ObjectsMessage objects keep only a reference to a pushed object. If the protocol also keeps a reference to the object, it will be able to change it. However, it is not possible to know if the object has already been serialized, in which case, changes would not affect the sent message.

2.2 Trusted Timely Computing Base

The Trusted Timely Computing Base (TTCB) was defined in MAFTIA deliverables D1 and D23. This section defines the TTCB services and their interface. We consider that the communication between entities and the TTCB is done with function calls, i.e., the TTCB API is simply a set of functions. This is generic enough and does not put special constraints on the TTCB implementation. At the end of the section we describe some additional APIs that do not correspond to services: distribution of TTCB public keys, delivery of entity ids, etc.

In this section we use the word *entity* to mean anything that calls the TTCB: a process, a thread, Appia, or a software module of some kind.

2.2.1 The TTCB Interface

This section discusses two basic assumptions about the TTCB interface:

- Malicious entities cannot interfere with the TTCB operation through its interface.
- Any entity can obtain correct information at the TTCB interface.

Therefore, correct entities can correctly use the services of the TTCB, despite the action of malicious entities. On the other hand, both correct and malicious entities can use the TTCB, but what a malicious entity does with correct TTCB information is irrelevant at this stage.

The second assumption above is handled using two *approaches*: (1) means are given to the entities to communicate securely with the TTCB; and (2) services are defined in order to provide useful results despite the lack of security and timeliness of the entities that use them.

Security of entity-TTCB communication, approach (1), can be ensured using cryptographic techniques. An entity (either in a host with a local TTCB or not) can decide to secure its communication with the TTCB using a *secure channel* or *secure envelopes* (signed service outputs). A secure channel is obtained calling the *local authentication service* (section 2.2.3.2) that establishes a shared key between the entity and the TTCB. That key can subsequently be used to encrypt or cryptographically checksum the messages they exchange, thus assuring the desired combination of communication authenticity, confidentiality and integrity.

The TTCB outputs can be encapsulated in secure envelopes in order to guarantee their integrity and authenticity. The TTCB is the appropriate place in a host to put *long-term keys* since it is the only component that we assume to be completely secure. So, every *local TTCB* has an asymmetric key pair. Its private key is held securely inside the local TTCB, while the public key is distributed to entities that may want to use it. Outputs of the TTCB may then be signed with the local TTCB's private key.

Approach (2) means that the services themselves have to be defined in order to be used by insecure and untimely entities. Most services that explore the security of the TTCB require only secure entity-TTCB communication in order to give useful results. This is the case of the *random number generation service*. Others, such as the *consensus service*, give a correct result if, e.g., the majority of the entities involved is correct, a common assumption for applications running in insecure environments.

For timeliness, an example can illustrate this approach. In general, a timestamp given by the TTCB is not particulary useful since the delay between its generation and the moment when the entity reads it is unknown. However, if an entity calls the TTCB before and after executing an operation, the TTCB can calculate an upper bound on the time taken to execute it (*duration measurement service*), and give feedback to the calling entity on how well it did with regard to time. This mechanism is inspired by the Timely Computing Base work, and explained with detail in [48].

Most TTCB functions have two versions, one that returns unsigned results and another that return them signed. The later has a `-S` in the end of the name, e.g.:

```
timestamp getTimestamp-S()
```

2.2.2 The TTCB Services

The TTCB services can be roughly divided in *security-related services* and *time-related services*. The security-related services were selected considering the TTCB design principles [47] and a set of informal criteria:

- The services should be the minimal set that assists in a useful manner the implementation of building blocks for trusted distributed systems.
- Services should give useful results to entities running in an insecure environment.
- Services should be *implementable* within the TTCB design principles. E.g., they should not be too complex to be verifiable.

The TTCB services are summarized in Figure 2.5. The following subsections describe the TTCB services and their APIs. Figure 2.6 shows the meaning of some common API parameters.

Security related services	
Trusted block consensus	Achieves consensus on a small, fixed size, block of data (e.g., 64 or 128 bits).
Local authentication	Used for an entity to authenticate the TTCB and establish a secure channel with it.
Distributed authentication	Used for two or more entities residing in different hosts to authenticate themselves mutually through the TTCB. A shared key is established.
Trusted random number generation	Generates trustworthy random numbers, that are essential for building cryptographic primitives such as authentication protocols.
Time services	
Trusted timely execution	Executes operations securely and within a certain interval of time.
Trusted duration measurement	Measures the duration of the execution of an operation.
Trusted timing failure detection	Checks if an operation is executed in an interval of time; if not, a function can be executed to handle the failure (e.g., to perform a fail-safe shutdown).
Trusted absolute timestamping	Provides globally meaningful timestamps, implying all TTCB internal clocks to be synchronized to an external standard reference (e.g. UTC).

Figure 2.5: TTCB Services

Parameter	Description
eid	an entity identification
elist	a list of entities identified by their eids
tag	an unique id for an execution of a service (given by the TTCB)
initiator	the eid of the service initiator
value	a value given to or returned by a service
encrypt	encryption algorithm
key	key established in a service

Figure 2.6: Common TTCB API parameters

2.2.3 Security Related Services

2.2.3.1 Trusted Block Consensus Service

The *trusted block consensus service* (or consensus service for short) achieves consensus between a set of distributed entities on a “small” fixed size block of data. Consensus is a classical distributed systems problem than can be informally stated as: how can a set of distributed entities agree unanimously on a value related to the values that each of them proposed?

This service was selected for the TTCB for several reasons: it can be useful to perform simple but crucial decision steps in more complex payload protocols; inside the TTCB it can be reliable, secure and timely due to the TTCB properties; since the TTCB is synchronous it can be solved deterministically, on the contrary to what happens in asynchronous systems (FLP impossibility result [14]); it can be lightweight since it achieves consensus on a small amount of data.

Consensus is defined in terms of two primitives, $propose(v)$ used by an entity to propose the value v, and $decide(v)$, used by an entity to decide v [16]. The TTCB consensus is specified by the following properties seen at its interface:

TBC1 Termination. Every correct entity eventually decides some value.

TBC2 Integrity. Every correct entity decides at most one value, and if it decides v then some entity must have proposed v.

TBC3 Agreement. If a correct entity decides v, then all correct entities eventually decide v.

TBC4 Validity. If all entities that propose a value, propose v, then all correct entities eventually decide v.

The definition of consensus does not specify *which* of the values proposed is chosen. The service is parameterizable in order to choose one of the following values:

- The most voted by the entities.
- The most voted by the local TTCBs. Every local TTCB votes with the value more voted locally by the entities.
- The entity proposed the same value as the majority of entities? This can be useful to test, say, a shared protocol variable that must be equal in all correct entities. Entities that proposed a value different from the majority receive just an error code. The others receive a list of the entities that voted the majority value.

The consensus service is timely at the TTCB interface, i.e., not considering the time taken for the TTCB to communicate with the entities. Formally:

TBC5 Timeliness. There is an instant t_o and a known constant Δ such that no correct (not crashed) local TTCB decides after $t_o + \Delta$.

The instant t_o is the last moment for the phase of proposals to end and the consensus protocol to start to run inside the TTCB.

The motivation for the consensus service is to supply multi-entity fault-tolerant protocols with an opportunity to synchronize at specific points in a reliable and timely manner. Despite the fact that some entities may be corrupt and try to disturb the operation of the protocol, they are prevented from: attacking the timeliness assumptions; sending disturbing and/or contradicting (Byzantine) information to different parties. Why is this so? Because the TTCB mediates this synchronization. As such, the API is based on the idea that entities propose a value and later call a function to receive the result.

		Action	Description
1	Entities	propose	The entities propose their values
2	Entities	decide	The entities get the value decided

Figure 2.7: Consensus service API: sequence of function calls

Figure 2.7 shows the sequence of functions that each entity calls and below we specify the function calls. Every function returns an error code that we omit for simplicity. Most functions receive as input the entity eid. These eids are generated by the TTCB and are unique.

```
tag propose(eid, elist, tstart, decision, value);
results decide(eid, tag);
```

To propose a value an entity calls `propose`. The parameter `tstart` is an absolute timestamp that indicates the last moment for the consensus to start, in the case that not all entities effectively propose a value. The `decision` parameter is used to indicate how the result should be calculated (most voted by the entities, by the local TTCBs,…). To get the result of the consensus the entities call `decide`. The format of this result depends on the type of decision requested.

All entities have to know beforehand what `tstart` to use, which should be the list of entities involved (`elist`), and how to decide (`decision`). This seems to be a reasonable assumption. In fact, this problem is similar to the problem of initiation of a clock synchronization round, and has been solved in that context (see for instance [43, 50]). A malicious entity could try to corrupt the result of the service proposing a different list of entities, a different `tstart` or even a different `decision` parameter. This is not possible because the TTCB decides that a set of entities want to engage in the same run of the consensus precisely because they have given the same list of entities, the same `tstart`, and the same `decision` parameter.

The Timeliness parameter t_o in property TBC5 is given by: $t_o = tstart$.

2.2.3.2 Local Authentication Service and Secure Channels

This service is used by an entity to authenticate a local TTCB and obtain a shared key to interact with it. This key can subsequently be used to guarantee the authenticity, integrity and/or confidentiality of their communication, thus establishing a *secure channel* between them.

It is especially useful for several reasons: for the entity to authenticate the local TTCB, to be sure that it is talking with it (the TTCB does not need to authenticate the entity since it is supposed to provide its services to any entity that requires them, correct or malicious); to secure the entity-TTCB communication (confidentiality and/or integrity). This service has several degrees of importance depending on the way an entity communicates with the TTCB. It is specially useful if the entity does not have a local TTCB in its host and is communicating with another host's local TTCB.

The established channel remains secure as long as the key is not disclosed. During the establishment of the channel the entity gives its eid to the TTCB.

In general entities will use a secure channel to communicate with the local TTCB. However, we also envisage channels with address- or topology-based authentication and secure communication with the TTCB ensured by the system architecture. These may not need the local authentication service.

The protocol (sequence of function calls) to establish the session key has to be an *authenticated key establishment protocol* with local TTCB authentication. The protocol has to have the following properties [23]:

SK1 Implicit Key Authentication. The entity and the TTCB know that no other entity has the session key.

SK2 Key Confirmation. Both the entity and the TTCB know that the other has the session key.

SK3 Authentication. The entity has to authenticate the TTCB.

SK4 Trusted Against Known-Key Attacks. Compromise of past session keys does *not* allow either (1) a passive adversary to compromise future session keys, or (2) impersonation by an active adversary[6].

The session key has to be changed (*rekeyed*) regularly since the probability of a

[6] A passive adversary "attempts to defeat a cryptographic technique by simply recording data and thereafter analyzing it (e.g., in key establishment, to determine the session key). An active attack involves an adversary who modifies or injects messages." [23]

passive attacker disclosing the key increases with time. This danger increases also with the amount of data exchanged, encrypted with the key. The time for rekey has to be a balance between the potential threat against the secure channel and the strength of the encryption algorithm. This last parameter should be balanced with the performance needed of communication.

A simple protocol with properties SK1 through SK3 can be implemented by a single function call. We assume that the entity can reliably obtain the local TTCB public key and that entities do not repeat prior challenges. Below we show the function call with the parameters represented as `in` and `out` for simplicity. Figure 2.8 shows the corresponding protocol.

```
out localAuthentication(eid, in);
```

When the entity calls the local TTCB it sends it the new key and a challenge, both encrypted with the TTCB public key. This information can be decrypted only with the corresponding private key, known only to the local TTCB. The local TTCB decrypts it and sends back the challenge signed with the private key, proving that it knows the private key and has received the shared key.

		Action	**Description**
1	P → T	$\langle E_{u_t}(K_{pt}, X_p)\rangle$	The entity sends the TTCB the new key K_{pt} and a challenge X_p, both encrypted with the local TTCB public key
2	T → P	$\langle S_{r_t}(X_p)\rangle$	The TTCB sends the entity the challenge signed with its private key

Figure 2.8: Local authentication service API protocol

The protocol verifies SK1 since only the local TTCB has the private key that can decrypt `in`. SK2 is trivial. SK3 is also verified since the entity gives the TTCB a value that only the TTCB can decrypt, and the TTCB gives back the decrypted challenge, showing that it was able to do the decryption. Property SK4 depends on the key generation method and encryption algorithm. We assume that the entity or the TTCB generate keys that verify that property.

2.2.3.3 Distributed Authentication Service

This service mutually authenticates a distributed set of entities and establishes a shared key between them. In other words, the TTCB checks the *identification* of a set of entities on behalf of each other. Each entity is identified by its eid, an asymmetric key pair and, optionally, by the ID of the local TTCB that it calls. The shared key established can be used to enforce the confidentiality and integrity of the communication between a set of entities.

The justification for putting this service in the TTCB is that the authentication between several distributed entities is crucial for distributed protocols and it can easily be made secure and timely inside the TTCB. Another reason is that such a service involves consensus so it would not be deterministic if executed in an asynchronous system, on the contrary to what happens in the TTCB.

Every entity P is individually authenticated in the same way: P gives its identification to the TTCB; one or more of the other entities give also the TTCB the identification of P; the TTCB compares both and, if they match, gives P the shared key. This basic protocol is shown in Figure 2.9, for two entities. The protocol assumes that entities communicate with the TTCB using secure channels.

		Action	**Description**
1	T → A	$\langle X_{ta} \rangle$	The TTCB sends entity A a challenge
2	T → B	$\langle X_{tb} \rangle$	The TTCB sends entity B a challenge
3	A → T	$\langle A, S_{r_a}(X_{ta}), B, K_{u_b} \rangle$	A sends the TTCB its eid, the challenge signed with its private key, entity B's eid and public key. Optionally, it could also send B's local TTCB ID.
4	B → T	$\langle B, S_{r_b}(X_{tb}), A, K_{u_a} \rangle$	B sends the TTCB its eid, the challenge signed with its private key, entity A's eid and public key. Optionally, it could also send A's local TTCB ID.
5	T	Challenges well signed?	The TTCB uses the public key of A given by B to check if A correctly signed the challenge. It does the same for B. If the entities gave the local TTCB IDs that comparison is also performed. If all these checks hold, the authentication was correct.
6	T → A	$\langle K \rangle$	The TTCB sends A the shared key K if the authentication was correct. Otherwise, an error is returned.
7	T → B	$\langle K \rangle$	The TTCB sends B the shared key K if the authentication was correct. Otherwise, an error is returned.

Figure 2.9: Distributed authentication protocol: simple case with just two entities, A and B

The service has two APIs:

- Symmetric authentication API. All entities authenticate all others. All entities provide the TTCB the *identification* of *all* others and the TTCB uses that information to do the authentication.

- Asymmetric authentication API. The initiator authenticates all other entities but the others authenticate only the initiator. Therefore, the trust that an entity can put on the entities with the shared key depends on the trust it has on the initiator.

The basic operation of the service considering the *symmetric authentication API* and several entities is the following:

1. The initiator asks the TTCB a challenge and the others block waiting also for a challenge.
2. Every entity calls the service on its local TTCB and gives it two things: (1) its own identification (eid and signed challenge, see Figure 2.9); and (2) the identification it has for the other entities (eid, public key and, optionally, local TTCB id).
3. The TTCB collects all information and decides which is the correct identification for every entity. This decision is made through the (internal) consensus protocol that elects the most voted pairs public-key/local-TTCB-id for every entity.
4. For every entity, the TTCB compares the identification it gave for itself and for the others with the identifications that were determined to be the correct (Figure 2.9). If all match, the entity is accepted and receives the shared key. Otherwise it is rejected.

		Action	**Description**
1	Entities	waitDistSymAuth	The entities block on the TTCB waiting for the service to start
2	Initiator	startDistSymAuth	The initiator starts the service
3	All	distSymAuth	The initiator and the entities get the result of the system execution: the shared key if they are authenticated or an error

Figure 2.10: Distributed authentication with symmetric authentication: sequence of function calls

The sequence of function calls is given in Figures 2.10. The function calls are:

```
tag waitDistSymAuth(eid, time_block, challenge, initiator, elist, tstart,
      encrypt);
tag startDistSymAuth(eid, elist, tstart, encrypt, challenge);
key distSymAuth(eid, tag, s_challenge, edata, elist);
```

The *initiator* starts the service asking the TTCB a challenge, calling function startDistSymAuth. It gives the TTCB the eids of the entities that will be involved, `elist`. The tstart is the latest instant for all entities (including the initiator itself) to give the signed challenge to the TTCB. Other parameters have the usual meanings (see Figure 2.6).

Entities other than the initiator wait for an authentication request blocking on function waitDistSymAuth for time_block units of time.

When startDistSymAuth and waitDistSymAuth return, respectively the initiator and the other entities all have their challenge. Next they have to sign it with their private key and give it to the TTCB calling distSymAuth. This function is also used by all to give the TTCB the identification of the others, edata. This parameter is a table with three columns: eid, public key and, optionally, local TTCB id. The correctly authenticated entities receive back the shared key.

The *asymmetric authentication API* is similar to this one. The difference is that each entities other than the initiator has to give the TTCB the identification of the initiator and its own.

2.2.3.4 Trusted Random Number Generation Service

The *trusted random number generation service* gives trustworthy uniformly distributed random numbers. These numbers are basic for building cryptographic primitives such as authentication protocols. If the numbers are not really random, those algorithms can be vulnerable.

The interface of the service has only one function that returns a random number:

```
number getRandom();
```

2.2.4 Time Related Services

2.2.4.1 Trusted Absolute Timestamping Service

Every local TTCB has an internal clock that is synchronized to the other local TTCB clocks. This is achieved with a clock synchronization protocol inside the TTCB. The *trusted absolute timestamping service* gives timestamps that, since clocks are synchronized, are meaningful to all local TTCBs. The precision of the timestamps is limited by the precision of the clock synchronization protocol. The interface of the service is:

```
timestamp getTimestamp();
```

When an application running on the payload part of the system asks for a timestamp, it receives it some time after it was generated by the TTCB. This delay is variable, depending mostly on the time taken by the operating system scheduler to give CPU time to the application, on the time the application takes to read the timestamp, and on potential attacks against time. However, a timestamp can still be useful since, e.g., the difference between two timestamps is an upper bound on the real duration of the time interval between them.

2.2.4.2 Trusted Duration Measurement Service

This services measures the time taken to execute a function. The service verifies the following property:

TDM1 Duration measurement. Given any two events occurring in any two nodes at instants t_s and t_e, the TTCB is able to measure the duration between those two events with a known bounded error [11].

The service is used calling the functions:

```
tag startMeasurement(start_ev);
duration stopMeasurement(tag, end_ev);
```

The parameters `start_ev` and `end_ev` are timestamps that indicate respectively the time of the beginning and end of the operation to measure. `duration` is the value measured for the duration of the operation. `start_ev` has to be obtained prior to the execution of the service calling the timestamping service.

2.2.4.3 Trusted Timely Execution Service

This service allows an application to execute (sporadically) a function with a strict timeliness and/or a high degree of security. The function is executed inside the TTCB before a deadline (eager execution) and/or after a liveline (deferred execution) [11]:

TTE1 Timely execution. Given any function `f` with an execution time bounded by a known constant $T_{X_{max}}$, and given a delay time lower-bounded by a known constant $T_{X_{min}} \geq 0$, for any execution of the function triggered at real time t_{start}, the TTCB does not start the execution of f within $T_{X_{min}}$ from t_{start}, and terminates f within $T_{X_{max}}$ from t_{start}.

The function f is executed between instants start_ev+delay and start_ev+t_exec:

```
end_ev exec(start_ev, delay, t_exec, f);
```

An issue to be studied later is what functions can be executed inside the TTCB. The TTCB can either offer a library of generic useful functions or let an entity upload functions. The later case requires an entity that ensures that the function is correct (that it will not attack the TTCB or create a vulnerability) and calculates the worst-case execution time (WCET) for the function. When an entity uses the *trusted timely execution service* to request the execution of a function, the WCET is used to make a schedulability analysis, that assesses if the TTCB has resources to execute it. In case the TTCB does not have resources, an error is returned.

2.2.4.4 Trusted Timing Failure Detection Service

This service is used to detect if a timed action is executed before its deadline. The action is executed in the payload system and the TTCB simply verifies its timeliness. It is defined by the two properties [11]:

TTFD1 Timed strong completeness. Any timing failure is detected by the TTCB within a known interval from its occurrence.

TTFD2 Timed strong accuracy. Any timely action finishing no later than some known interval before its deadline is never wrongly detected as a timing failure by the TTCB.

The service has different APIs depending on the timed action being local or remote.

Local detection API Local timing failure detection is done calling the following two functions:

```
tag startLocal(start_ev, spec, handler);
faulty endLocal(tag, end_ev, duration);
```

The first function requests the TTCB to observe the timeliness of the execution of an operation. start_ev is the start instant and spec the expected duration. The handler is used to tell the TTCB the reaction to have if a failure is detected, in case that is needed. The handler has to specify a function in the same way as `f` in the *trusted timely execution service*. Examples of reactions are a fail safe shutdown or a crash of the host.

The second function disables the detection, i.e., it indicates the TTCB that the action terminated. The parameters returned indicate the termination instant (`end_ev`), the `duration` measured and if there was a timeliness failure or not (`faulty`).

Distributed detection API The basic idea of this interface is that a distributed action is initiated by the transmission of a message from a sender to a receiver. The way the API works is similar to the *local detection API*, i.e., an entity calls the TTCB telling that it is going to send a message (start a distributed action, startDistributed), sends the message, the receiver receives the message, executes the remote operation, and tells the TTCB that it is delivered delivDistributed). If the time to receive the message expires, the TTCB executes a function, in case that was requested. Messages can be multicast to several receivers:

```
tag startDistributed(start_ev, spec, mid, elist, handler);
deliv_ev delivDistributed(mid, tag);
list_info waitInfo(tag);
```

The parameter `mid` is a unique message id. The `handler` is executed by the local TTCB of the sender in case there is a timeliness failure. In delivDistributed the parameters indicates that a message was received. When that call is made, the TTCB checks if there was a timing failure and returns that information.

An entity, either the sender or a receiver, can get information relative to timing failures using the function waitInfo. The input parameter `tag` is optional since the entity may decide to wait for information of all or only one of the distributed actions it is involved in. The parameter `list_info` contains the delay to deliver and the indication about timeliness faults for every receiver.

2.2.5 Other Services

2.2.5.1 TTCB Public Keys API

This section does not describe a service but some functions of the API needed to give the local TTCB public keys that have to be reliably obtained by the entities. A conventional way to distributed such keys is to use a Public Key Infrastructure. An entity can get keys from there in a certificate signed with the PKI private key.

Another solution is for the entity to ask the key directly from a local TTCB using function getLocalPublicKey that simply returns the local TTCB public key (see below). There is no version of this function that returns a signed value, since the output would be signed with the local TTCB private key.

```
key getLocalPublicKey();
key getLocalPublicKeySigned(remote_local_TTCB_id);
key getRemotePublicKeySigned(local_TTCB_id);
id getLocalTTCBid();
```

Two other functions exist that return signed values, one from a remote local TTCB, the other from the local one: getLocalPublicKeySigned and getRemotePublicKeySigned. The first one returns this host local TTCB public key signed with the public key of the local TTCB with id `remote_local_TTCB_id`. The second one returns the key of the local TTCB with id `local_TTCB_id` signed with this host local TTCB private key.

The last function, getLocalTTCBid, can be used to get the local TTCB id.

2.2.5.2 Eid API

Entities eids are supposed to be generated by the TTCB and to be unique. They contain enough information for the TTCB to known which is the local TTCB that produced the eid. Since the entity is supposed to gets its eid directly from the TTCB, the eid is supposed to indicate the entity local TTCB and the TTCB can use it to check this information. The function that returns an eid is:

```
eid getEid();
```

Whenever an entity wants to start to use the TTCB services, it has to start by requesting the eid calling this function.

2.2.5.3 Local TTCB Failure API

A local TTCB can fail only by crashing. Its crash is equivalent to the crash of the host where it exists, i.e., whenever one crashes the also does the same. Therefore, the information of a local TTCB crash can be useful for applications to test if a host crashed. This API allows entities to do precisely that. The two functions are below. The first receives a local TTCB as input. The second receives an entity eid as input and checks if its host crashed:

```
out crashedLocalTTCBid(id);
out crashedEid(eid);
```

3 Middleware Services and APIs

3.1 Multipoint Network

At the network level, there are several services that can be used by higher layers of the architecture. These services form the basis for all the communication to and from each site. Although it is possible to include a vast number of services at this level, we will only focus on the ones we see as elementary, either because there is a specific need for them, or because they are already standard services in the Internet. These services are IP, IP Multicast, ICMP, IPSec and SNMP protocols.

3.1.1 Internet Protocol

The Internet Protocol (IP) [34] is designed to be used in interconnected systems of packet-switched computer communication networks. IP provides the means for transmitting blocks of data, called datagrams, from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. It also offers the service of fragmentation and reassembly of packets transparently. IP by itself can not be used directly by an application, and so, it has two other protocols built on top of it: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

UDP [32] makes available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. Applications can send messages to other programs with a minimum set of guarantees using UDP. The key characteristics of the protocol are: it is transaction oriented, the delivery of messages is not ensured, nor is the order of message arrival, and there might be duplication of messages.

TCP [35], on the other hand, is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. Applications can send messages using TCP, in a reliable way, to other programs on host computers attached to distinct but interconnected computer communication networks. TCP does not rely on the protocols below, but rather assumes that it can obtain a simple, potentially unreliable datagram service from the lower level protocols, such as IP.

The socket interface to TCP and UDP is well-known, and we include summary below just for completeness. For details, consult for instance the book by Stevens [45].

The functions that must be called to establish a TCP or a UDP connection are different, since the protocols have different purposes. In order to establish a UDP connection, the server must call the functions: `socket()`, `bind()` and then `recvfrom()`, which

Function	**Description**
socket(domain, type, protocol)	specifies a communication domain and semantics, and particular protocol
bind(sockfd, my_addr, addrlen)	assigns to the socket descriptor (`sockfd`) a local address (`my_addr`)
listen(sockfd, backlog)	specifies the willingness to accept incoming connections in a given socket descriptor (`sockfd`) and specifies a queue limit (`backlog`)
accept(sockfd, addr, addrlen)	removes the first connection request from the queue of pending connections, and creates a new connected socket
recvfrom(sockfd, buf, len, flags, from_addr, fromlen)	receives messages from a socket
sendto(sockfd, buf, len, flags, to_addr, tolen)	transmits a message to a socket
read(sockfd, buf, len)	attempts to read from socket descriptor
write(sockfd, buf, len)	attempts to write to a socket descriptor

Table 3.1: Summary of socket interface.

blocks until data is received by the client. The client, in order to connect, must follow the order: `socket()`, `sendto()`, which sends data to the server (thus unblocking it) and then `recvfrom()` to receive the reply. The server will then process the request and do a `sendto()` back to the client.

For a TCP connection, the order for the server is: `socket()`, `bind()`, `listen()` and `accept()`, which blocks until connection is established with the client. The client, on its side, must do: `socket()` and `connect()`. With this, the connection is established, and now client and server can exchange through the call of `write()` and `read()` functions.

3.1.2 IP Multicast

IP multicast gives the ability to transmit an IP datagram to a group of hosts, which are identified by a single IP destination address. The multicast datagram is delivered to all members of the group with the same guarantees given by the regular IP datagrams: it is not guaranteed to reach all members, it is not guaranteed to arrive intact to all members and it is not guaranteed to arrive in the same order to all members, relative to other datagrams.

The membership of a group is dynamic, meaning that hosts can join and leave the group at any time. Not only can a host belong to more than one group at a time, but it also does not need to be a member of a group to send datagrams to it.

A group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available to be dynamically assigned to temporary groups, which exist only as long as the group has members in it.

The API to send IP Multicast packets can be the same as the IP, in which an application sends the packets to the group address, rather than to an individual host. However, some extensions to the IP Module are desirable, so that IP recognizes IP group addresses when routing outgoing datagrams.

In order to receive IP Multicast packets, the API must be expanded so that there are two necessary functions: the `JoinHostGroup` and the `LeaveHostGroup`, which are self-explanatory. The IP Module must also be expanded to maintain a list of host group memberships associated with each network interface. An incoming datagram with one of these groups as destination is processed in exactly the same way as if it has the host as destination.

3.1.3 IPSec

Given its importance for the MAFTIA middleware, this section provides a brief overview of the current state of IPSec. IPSec is designed to offer enhanced security to IPv4 and IPv6 protocols, providing inter-operable, high quality, cryptographically-based security. It offers several services, such as access control, connectionless integrity, data origin authentication, protection against replays, confidentiality (through encryption) and limited traffic flow confidentiality. Since these services are offered at the IP layer, they can be used by any higher layer protocol, such as TCP, UDP, ICMP, etc.

IPSec also supports negotiation of IP compression [42], which is motivated by the observation that encryption used within IPSec prevents effective compression by lower protocol layers.

To achieve these objectives, IPSec uses cryptographic key management procedures and protocols and two traffic security protocols:

Authentication Header(AH) Providing connectionless integrity, data origin authentication, and an optional anti-replay service.

Encapsulation Security Payload (ESP) Maybe providing confidentiality (encryption), and limited traffic flow confidentiality. Optionally, it may also provide connectionless integrity, data origin authentication, and an anti-replay service.

Both AH and ESP are vehicles for access control, based on the distribution of cryptographic keys and the management of traffic flows relative to these security protocols. These protocols may be applied alone or in combination with each other to provide the desired set of security services in IPv4 or IPv6, and both support two different modes of operation:

transport mode In this mode, the protocols provide protection primarily for upper layer protocols.

tunnel mode In this mode, the protocols are applied to tunneled IP packets.

IPSec allows the user (or the system administrator) to control the granularity at which a security service is offered, allowing, for example, the creation of a single encrypted tunnel to carry all the traffic between two security gateways or a separate encrypted tunnel for each TCP connection between a pair of hosts communicating across these gateways.

3.1.3.1 Current IPSec API

The most used IPSec implementation for the Linux operating system is FreeS/WAN. This implementation does not have an API that can be used by applications to transmit secure data. Instead, it works at the operating system level, and can only be configured by the system administrator. A system administrator can define the policy for IPSec on a host basis, determining the ways by which a host can connect securely to another.

FreeS/WAN does not use DES for encryption, since this algorithm is no longer secure for many types of applications. Instead, it uses triple DES, and is also looking for some other strong protocols to be included in the software.

In the present protocol implementation, FreeS/WAN does not allow the specification of IPSec policies based on connection ports; it only focuses on machines. Therefore, one can not specify that between the same two machines (be them gateways or not), the http traffic uses IPSec, but mail traffic does not. Either both or neither use IPSec. It can, however, be assumed that this feature **will** be available, since it is part of the standard.

One interesting thing that FreeS/WAN implements, and which is not in the standard definition, is what they call Opportunistic Encryption. This means that any two FreeS/WAN gateways will be able to encrypt their traffic, even if they have no prior knowledge of each other, by using some public key scheme, possibly provided by the Domain Name Service (DNS). This will create a default policy for security in the Internet, if followed by all the implementors.

3.1.4 Internet Control Message Protocol

Although not needed at the moment, we believe it is worth noting the existence of the Internet Control Message Protocol (ICMP) protocol [33]. It will not be defined here, nor will we give an exact API for it, since it should follow the API for the IP protocol. We merely name the protocol, so it can later be used, if so desired.

3.1.5 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) is also relevant for MAFTIA since it might be important to the middleware operation, namely in the failure/intrusion detection management. SNMP is actually a set of standards for network management that include a protocol, a database structure and some data objects.

This protocol was adopted as a standard for network management in TCP/IP-based networks in 1989 [6, 7, 8]). To enhance the functionality of SNMP and allow the management of both local area networks (LAN) as well as the devices attached to them, a supplement for monitoring networks was issued and is known as Remote Network Monitoring (RMON). In 1993, an upgrade to SNMP, called SNMP version 2 (SNMPv2) was proposed and a revision of it was issued in 1995 [10]. SNMPv2 adds functional enhancements to SNMP and codes the use of SNMP on OSI-based networks. In 1995, there was also an extension to RMON called RMON2. This extension includes a specification to include monitoring of protocol traffic above the MAC level. In 1998, a new version of SNMP was released and is known as SNMPv3. This new version defines a security capability for SNMP and an architecture for future enhancements. SNMPv3 is intended to use the functionality of SNMPv2, but can also be used with SNMPv1. It also introduces a new concept to SNMP, a User Security Model [3].

SNMP has three commands to perform its function: the `Get`, `Set` and `Trap` commands. The `Get` command is used by the manager to query the sensor status; the `Set` command is used to change some value in the sensor Management Information Base (MIB);

and the `Trap` mechanism is what the sensor uses to alert the manager of some event.

The Agent Extensibility (AgentX) Protocol is intended to provide regular SNMP with a high extension degree in order to enable multi-vendor compatibility and interoperability [13]. This was done because of the growing number of MIB modules defined by vendors and/or IETF workgroups. As a result of this, as the RFC states, the "managed devices are frequently complex collections of manageable components that have been independently installed on a managed node. Each component provides instrumentation for the managed objects defined in the MIB module(s) it implements."

Because no standard existed, and there was a wide deployment of extensible SNMP agents, it was very difficult to create SNMP-managed applications. One vendor would have to support multiple sub-agent environments (APIs), to support different platforms. The specification of a standard protocol to govern the communication between the master agents and the sub-agents, allows multiple vendor products to communicate and inter-operate.

There are also some problems with a monolithic SNMP agent, that the AgentX protocol tries to solve:

- For example, having an agent for the host, another for the print service and another for the web server means that we must have three agents on the same machine. To do this, they must be running in different system ports, since each SNMP agent must have a distinct connection point in order to communicate, which becomes very difficult to manage.
- Likewise, changes in a MIB require that the agent must be recompiled in order to incorporate them. This means that the agent must be shut down, recompiled, and re-run again. In the meantime, the management console will mark it as dead, since it can not contact its agent.

By using AgentX, these two problems are "automatically" solved, because, in the first case, AgentX subagents are multiplexed in a single SNMP agent, therefore using only one communication port; in the latter, subagents are dynamically added and removed, and so, when a MIB is changed, we need only shutdown the subagent handling it, and start the new one, without stopping the master agent and without disturbing the rest of the MIB.

More information on SNMP can be found in [44].

3.1.5.1 Current SNMP APIs

The most used implementation of SNMP in Linux is the one from University of California at Davis (ucd-snmp). Using this implementation, making an SNMP agent is done using a simple API, which we present by giving a skeleton of an SNMP agent. This skeleton can be used both for an SNMP agent as well as for an AgentX subagent.

```
int agentx_subagent=1; /* change this if you're a master agent */

 snmp_enable_stderrlog();

 /* if we're an agentx subagent */
 if (agentx_subagent) {
     /* make us a agentx client. */
     ds_set_boolean(DS_APPLICATION_ID, DS_AGENT_ROLE, 1);
 }

 init_agent("yourappname");

 /* initialize your mib code here */

 init_my_mib_code();

 /* yourappname will be used to read yourappname.conf files. */
 init_snmp("yourappname");

 /* If we're going to be a snmp master agent */
 if (!agentx_subagent)
     /* open port 161 (UDP:snmp) */
     init_master_agent(161, NULL, NULL);

 /* your main loop here... */
 while(whatever) {
     /* if you use select(), see snmp_api(3) */
     /*     --- OR ---  */
     agent_check_and_process(0); /* 0 == don't block */
 }

 /* at shutdown time */
```

```
    snmp_shutdown("yourappname");
  }
```

3.2 Communication Services

In this section we concentrate on *group communications* since these services are of vital importance for the MAFTIA middleware as they enable the construction of *dependable* distributed applications. In essence, dependability and fault-tolerance can be achieved by replicating a database or other applications service across a heterogeneous collection of servers.

MAFTIA middleware considers the existence of *groups of participants* that are mapped onto *groups of sites* hierarchically, i.e., every participant belongs to a site and for every group of participants there must be a group of sites such that the local sites of the participants belong to the group of sites. When addressing group communication protocols in this section, we do sometimes not distinguish between participants and sites, and simply speak of a group of "parties."

We shall in this section describe the semantics and provide APIs for the MAFTIA middleware group communications primitives. It is important to note that there are noticeable differences in these semantics depending upon precisely which types of groups are postulated. Two important axes upon which groups are measured are those of *open/closed* and *static/dynamic* [49]. An *open* group model permits arbitrary hosts to send messages to the group, while in a *closed* model only hosts which are already members of the group may so communicate. In contrast, a *static* group is one whose membership does not change over time (or changes at a very long time scale, such as only upon manual reconfiguration), while *dynamic* groups allow hosts to apply for group membership or, conversely, to be excluded from the group automatically when certain trigger events occur (or fail to occur).

The MAFTIA middleware leaves it to the applications programmer to decide how to implement either open or closed groups. The MAFTIA group communications API is designed to facilitate coordinated actions *within* established groups; these actions may be initiated by external parties, in an application requiring open groups, or exclusively by group members themselves, in closed group settings.

The distinction between static and dynamic groups requires more profound differences in both the APIs and implementation, since in the later case provision must be made for the whole issue of hosts joining, leaving and being excluded from a group. There are also subtle questions which must be addressed relating to re-sharing of cryptographic secrets when the group changes, which is particularly difficult to handle robustly in an asynchronous setting.

Regardless of which group model is followed, there are certain general ideas which are used ubiquitously. These are the basic primitives of “broadcast” and “agreement,” and the higher-level concept of “service replication.” *Broadcast* is used when a message is to be sent to all members of the group. There are various flavors of broadcast depending upon what other requirements are imposed. One such simply ensures all honest (properly operating, non-corrupted) servers deliver the same set of messages, be they all messages broadcast by all honest parties (*reliable broadcast*) or perhaps fewer but still uniquely delivered (*consistent broadcast*) messages. Another important choice is whether to guarantee all honest parties will deliver their messages in the same order (*atomic broadcast*).

Agreement is simply when all group members must agree to some binary value or, more generally, to a valued in some larger domain. The binary case here is usually known as Byzantine Agreement in the literature and requires all honest parties to come to the same value, which value must be the same as that proposed by all honest parties if the proposal is unanimous. The multi-valued case is more complex (but more useful), requiring the accepted value to satisfy a certain, globally-known predicate.

We shall now give more complete descriptions of these primitives and their APIs under the MAFTIA middleware in the static and then dynamic group models.

The APIs presented here are somewhat simplified compared to the description of the Appia API above. The dynamic composition capabilities provided by Appia are not essential for describing the particular properties of our group communication protocols and are thus not modeled. Therefore we will usually just mention abstract *protocols* and concrete *protocol instances* when referring to the group communication primitives. In fact, these can be mapped to the concepts of Appia in a straightforward way: when a “protocol” is mentioned below, this corresponds to an Appia “layer”; it can be used as part of a larger protocol stack, or “QoS” in Appia (which is not modeled here). Similarly, a “protocol instance” in the following sections corresponds to a dynamic “session” in Appia and could be realized as such.

3.2.1 Services using Static Groups

When groups are static, we may imagine that cryptographic keys are distributed once and for all in an entirely secure fashion and protocols need only deal with a known list of parties, as well as a known bound on the admissible number of corrupted parties. Therefore we may take an arbitrary group as given, identified by a group identifier `groupID`, and proceed directly to the communications within groups. MAFTIA middleware provides three types of communications primitives in this context: decision, stream (also known as "channel"), and broadcast.

Decision. We begin with *decision protocols.* These all implement the basic concept of Byzantine agreement, whereby the group members all propose some value to the group, and this value is the decision produced by the protocol if the honest parties all make the same proposal; in any case, all honest parties which decide, decide for the same value. Traditionally, the values in question are merely Boolean, but we also have a use for decision protocols where the values are in a larger domain. The difficulty in multi-valued Byzantine agreement is how to ensure the validity of the decided value, which may lie in a set whose size is not known *a priori.* Our approach is to introduce the idea of an external validating predicate, which is known to all parties and can be used to check all proposals, and which must be satisfied by the decision value of the protocol.

Note that "decision" is an abstract protocol that provides no implementation of its functionality. It isolates the common features of all decision protocols that will be modeled as subclasses of decision; but no instances of "decision" can be created. These relations are orthogonal to the concepts used for implementing them in Appia, where all protocols correspond to Appia layers, and the instances of a protocol correspond to an Appia session.

MAFTIA middleware provides implementations for the following decision protocols:

binary Byzantine agreement – ABBA
validated binary Byzantine agreement – VABBA
validated multi-valued Byzantine agreement – VBA

Despite their differences, there is a common high-level API for all decision protocols which is very simple. Let us say that XYZ is such a protocol. Then a new instance (i.e., Appia session) of the protocol can be created by the command

```
Decision decision = new XYZ(protocolID, groupID);
```

where protocolID is a String, with application-specific meaning, naming the protocol instance, and groupID is another String representing the group. This newly-created instance is still in a quiescent state; the simplest way actually to use it is to launch the protocol, with a certain initial vote, and to block until it returns, as follows:

```
Negotiable answer = decision.negotiate(initialVote);
```

Here both the answer and the initialVote are objects of class which extends the

Negotiable marker interface in a way useful for the particular protocol, such as containing a boolean value for ABBA, a byte[] value, byte[] proof and a MultivaluedValidator for VBA, etc; see below for details.

There is also a way to use the protocol without blocking, as for example in the code fragment

```
decision.propose(initialVote); // returns immediately!
while(true) {
  if (decision.isDecided()) {
    answer = decision.finalDecision();
    break;
  }
  else
    do something else interesting
}
```

There are also methods for *asynchronous* invocation of decision protocols which deliver their results as asynchronous events.

The methods negotiate, propose, isDecided and finalDecision, with the above semantics and argument and return types, exist for all decision protocols (in fact automatically, as all such protocols extend the Decision class), with only the specific content of the Negotiable objects changing with the context. In particular:

1. ABBA needs a BinaryNegotiable, which is of the form

```
public class BinaryNegotiable implements Negotiable {
  boolean value;
}
```

2. VABBA requires a ValidatedBinaryNegotiable, of the form

```
public class ValidatedBinaryNegotiable extends BinaryNegotiable {
  byte[] proof;
  BinaryValidator validator;
}
public interface BinaryValidator {
  boolean isValid(boolean value, byte[] proof);
}
```

The application programmer must create a class which implements BinaryValidator, providing a body for the isValid(boolean value, byte[] proof) method that determines the validity of the ValidatedBinaryNegotiable's value given its proof, perhaps using other identifying data which would therefore also be in the new class definition. The field validator will then be a handle to an instance of this new class, and well-formed instances vbn of the ValidatedBinaryNegotiable class (such as valid propose values in a VABBA instance or outputs of VABBA.finalDecision()) will satisfy the consistency condition that

```
vbn.validator.isValid(vbn.value, vbn.proof) == true.
```

3. VBA uses a ValidatedMultivaluedNegotiable, of the form

   ```
   public class ValidatedMultivaluedNegotiable implements Negotiable {
      byte[] value;
      byte[] proof;
      MultivaluedValidator validator;
   }
   public interface MultivaluedValidator {
      boolean isValid(byte[] value, byte[] proof);
   }
   ```

   Similar remarks as above in point 2 hold here for the classes that the application programmer must define and for the consistency of well-formed instances of class ValidatedMultivaluedNegotiable.

The Class Agreement As noted above, the concept of a "decision protocol" is embodied in an abstract class which is extended by the concrete classes which realize the protocols ABBA, VABBA and VBA. The abstract parent class, called Agreement, carries all of the publicly-accessible methods and fields of the decision protocol, as follows:

```
abstract class Agreement {
   Agreement(String protocolID, String groupID);
   Negotiable negotiate(Negotiable initialVote);
   boolean isDecided();
   Negotiable finalDecision();
   void propose(Negotiable initialVote);
}
```

The semantics of Agreement's instance methods were described above, as were the details of the specific implementations of the Negotiable interface which are appropriate for the various concrete extensions of Agreement.

Stream. We move on to *stream protocols*, a class of broadcast protocols which provide a long-lived communications channel for the group upon which multiple messages can be delivered in sequence. (One may always think of streams as communication *channels*, but the name “stream” is used here in order not to conflict with the Appia channels of Section 2.1.) In fact, MAFTIA provides at least two such protocols: ABC, implementing *atomic broadcast* and SC-ABC, implementing *secure causal atomic broadcast.* Atomic broadcast instances, identified by their naming String protocolID and String groupID, simply implement a communication stream with that group members can transmit messages with the guarantee that all honest parties will receive the messages at the end of this stream in the same order. Secure causal atomic broadcast is likewise a stream with broadcasting and ordered reception. The difference is that it preserves input causality in the sense of Reiter and Birman [37], which is achieved by dealing with a given message entirely in an encrypted form up until its order is determined. This permits applications where the cleartext of a message must be kept out of the hands of the adversary – and his minions running corrupted servers in the group – until it is ordered and delivered.

The common features in stream protocols are implemented in a class Stream, whose instances are created by the command

```
Stream stream = new XYZ(protocolID, groupID);
```

where XYZ is either ABC or SC-ABC. Streams are generally long-lived, and in fact cannot be safely terminated at any time, since a party which shuts down some stream may be leaving honest parties which are behind in their processing of that stream unable to finish correctly. Nonetheless, an emergency shut-down procedure is available if necessary, by invoking

```
stream.done();
```

A byte[] payload may be broadcast on a given stream by the method

```
stream.send(payload);
```

Here payload will be the message itself for a ABC but will merely be the ciphertext corresponding to the desired message in the case of SC-ABC. In either case, when the stream was created for synchronous access, the delivered messages may be found by calling

```
byte[] message = stream.receive();
```

which will block if there is no message currently waiting to be received. To avoid blocking, the method isReady() may be called, as in the following code fragment:

```
while(true) {
   if (stream.isReady()) {
      message = stream.receive();
      break;
   }
   else
      do something else interesting
}
```

Once again, there is an alternate non-blocking approach via what we called asynchronous invocation.

While this is not necessary for most applications, even those which rely upon the encryption of payloads input into SC-ABC, we should mention that in the precise description of secure causal atomic broadcast, deliveries of ciphertexts always must precede the corresponding (threshold) decryption and delivery of the cleartext message. Therefore SCABC objects have an additional pair of methods receiveCiphertext() and isReadyCiphertext() which perform the same tasks for the ciphertexts as the previously-described methods do for cleartext messages.

The Class Stream The abstract parent class for "stream protocols" is Stream, which has the following public methods, whose functionality was already explained:

```
abstract class Stream {
   Stream(String protocolID, String groupID);
   void send(byte[] m);
   byte[] receive();
   void done();
   boolean isReady();
}
```

This gives the entire public API for ABC, but, as was described above, SC-ABC needs two further methods:

```
class SCABC extends Stream {
   byte[] receiveCiphertext();
   boolean isReadyCiphertext();
}
```

Broadcast. Finally, we must describe *sequenced broadcast protocols.* These are again broadcast protocols, but they are short-lived and provide agreement on a single broadcast message. A party can send a messages on a new instance of such a protocol, while other group members must know to listen for the message from that sending party and with a predetermined "sequence number." MAFTIA middleware provides two variants of this type of broadcast: *reliable broadcast* RBC, which requires that all honest parties deliver the same set of messages and that this set includes all messages sent by all honest parties, and *consistent broadcast* CBC, which guarantees the uniqueness of delivered messages but not that all honest parties actually deliver all messages.

Both protocols are extensions of the class Broadcast, and can be created by

```
Broadcast broadcast = new XYZ(protocolID, sender, seq, groupID);
```

where XYZ is either RBC or CBC, and sender and seq are ints.

A message can be sent via sequenced broadcast only if the identity of the sending host matches the sender value which was specified when the Broadcast object was made. Furthermore, it is assumed that the sender is a member of the group named by groupID (i.e., the groups here are closed). Reception of messages is, however, identical to reception in ABC: under synchronous invocation, the blocking call broadcast.receive() may be used, or broadcast.isReady() may be tested first.

A non-blocking interface to these protocols is again provided by methods for asynchronous invocation.

The Class Broadcast For completeness, we also give the public class skeleton of the abstract parent class for "broadcast protocols":

```
abstract class Broadcast {
   Broadcast(String protocolID, int sender, int seq, String groupID);
   int getSender();
   int getSeq();
   void send(byte[] m);
   byte[] receive();
   void done();
   isReady();
}
```

3.2.2 Services using Dynamic Groups and related to Group Membership

Dynamic groups allow for addition and removal of members during operation, and such group membership changes should be transparent to the applications using the group's services. Thus, the communication services above are not affected by membership changes, and the focus of this section is on the orthogonal question of how new members join the group and current members leave the group.

As mentioned before, MAFTIA middleware considers the existence of *groups of participants* that are mapped on *groups of sites*. This provides a level of clustering that is useful both to handle security issues and scalability. The membership of groups of sites is handled by the *Site Membership* module, while the membership of groups of participants is handled by the *Participant Membership* module. In the following subsections we describe the interface of each of these modules. All functions are non-blocking and their results are returned as events.

Several groups of participants can be mapped onto a single site group. These groups, called *lightweight groups*, are useful to improve performance. The API of the membership module considering lightweight groups can be the same as for "normal" (heavyweight) groups [38]. Therefore, the API we describe below can be used in both cases.

The membership of both participant and site groups at a given instant is called a *view*. The concept was defined in the context of the *virtual synchrony* group semantics. Informally, virtual synchrony provides membership information to participants (sites) in the form of views and guarantees that all participants (sites) that install two consecutive views deliver the same set of messages between these views. However, the architecture of MAFTIA middleware does not impose a specific semantics and views can be seen simply as the membership at a given moment. When views are used, they are broadcasted atomically to all group members (participants/sites) when they change, in a special system management atomic broadcast stream. Changes can be due to member joins, leaves and to failures (of participants/sites).

Site membership module. This module offers the following calls:

```
joinSiteGroup(siteGroupID);
leaveSiteGroup(siteGroupID);
registerSiteEvents(id, siteGroupID, mask);
view = getSiteGroupView(siteGroupID);
```

joinSiteGroup makes the site try to join the group with site group id `siteGroupID`. Other sites in that group can grant or deny the join. If the group does not exist a new one is created with a single site. The result of the operation is returned as an event.

leaveSiteGroup makes the site leave the group `siteGroupID`. Only processes with the uid of the process that requested the site to enter the group can call this function. An error code is returned if the group does not exist or if the process is not allowed to request the operation.

registerSiteEvents allows a process to register (and unregister) the events that it wants to receive. Events can be the results of group operations (join, leave) or view changes. Events are registered by a process or module identified by `id`, for the site group identified by `siteGroupID`. `mask` indicates witch events are supposed to be registered (and/or unregistered).

getSiteGroupView asynchronously returns a site group view. The group is identified by `siteGroupID`.

In general, site groups are not used directly by user level entities but by the Participant Membership module to implement participant groups.

Participant membership module. This module offers the following calls:

```
joinGroup(id, participantGroupID, credential);
leaveGroup(id, participantGroupID, credential);
excludeGroup(id, participantGroupID);
registerEvents(id, participantGroupID, mask);
view = getGroupView(participantGroupID);
```

joinGroup can be used by a participant identified by `id` to try to join the group with group id `participantGroupID`. Other participants in that group can grant or deny the join based on the `credential` shown by the participant. We do not define what is the credential at this point, but it can be something that shows the possession of a certain cryptographic secret. If the group does not exist a new one is created with a single participant (and a site group is also created). The result of the operation is returned as an event.

leaveGroup makes the participant leave the group `participantGroupID`. Only the participant can request itself to leave the group, so it has to give his credential again in this operation. An error code is returned. If the participant was the last of this site in the group, the site leaves also the corresponding site group.

excludeGroup removes from group `participantGroupID` the participant with identifier `id` (a malicious member might not cooperate to execute `leaveGroup`). It is required that all honest (i.e., uncorrupted) participants execute this call. How the group members trigger this function is left unspecified at the moment; typically it could be the effect of an atomically broadcast control message or the instructions could be transmitted through an external communications mechanism.

registerEvents allows a participant to register (and unregister) the events that it want to be informed of. Events can be group operations results (join, leave) or view changes. Events are registered by a participant identified by `id`, for the group identified by `participantGroupID`. `mask` indicates witch events are supposed to be registered and/or unregistered.

getGroupView asynchronously returns the view of the group `participantGroupID`.

Maintaining shared secrets. The calls of membership modules have no direct effect on the parameters of the secret keys shared by the members of a particular group. Recall that the group communication primitives have two parameters n, the number of parties, and t, the maximal number of corrupted parties. With dynamic groups, the parameter n remains constant only between the deliveries of two successive views. Thus, `joinGroup` changes the parameters to $n' = n + 1$ and $t' = t$, and `leaveGroup` as well as `removeGroup` change the parameters to $n' = n - 1$ and $t' = t$.

Moreover, every party that joins a group must receive its share(s) of the cryptographic key(s). In absence of an on-line trusted dealer who can supply the key material, this requires that the group members carry out a distributed protocol whenever a new member joins the group. Such protocols are implicit in the description of the membership interface above.

When the size of the group shrinks, it must be possible to lower t as well, in order to maintain the system invariant $n > 3t$, and when the group grows, it is desirable to increase also t for more resiliency. This is achieved by a call to reshare(groupID, dt)

where dt is an integer denoting the threshold change Δt, which can be positive and negative. Its effect is to change the previous group parameters n and t to the new values $n' = n$ and $t' = t + \Delta t$.

In order to guarantee the security of the shared secrets despite the removal of faulty and potentially corrupted parties, the share refresh operation triggers a refresh of all key shares, similar to the one in proactive threshold cryptosystems [5]. This renders knowledge of old key shares from the previous view useless for the current view and all future views, which is necessary for maintaining proper operation.

3.3 Activity Services

The Activity Services implement building blocks that assist the development of specific classes of applications. This section describes essentially the MAFTIA transactional support service. The transactional support service is intended to support both applications built using the MAFTIA middleware and other activity support services, for example it can be used to guarantee the atomicity of updates to a replicated authorisation server. From a user's point of the view the transaction support service appears to be a CORBA-style transaction service; this is because its intrusion-tolerance is a property of the implementation rather than of its interfaces.

In section 3.3.1 we provide an overview of transactions, in section 3.3.2 we introduce a high-level view of the transaction service architecture, in section 3.3.3 we introduce our system model and describe how a degree of intrusion-tolerance for the service is achieved, in section 3.3.4 we describe some related approaches and contrast them with the approach taken within this project, and in section 3.3.5 we describe how the transaction service is used.

3.3.1 Overview of the Transactional Support

Services provided over a wide area network such as the Internet involve communication and cooperation between many diverse organisations and their hosts. Faults that may be due to hardware failure, software failure or malicious agents can disrupt service delivery. Ideally service functions are *atomic* in their effect. Atomicity is an "all or nothing" property. If a function is atomic then in the event of failure all of the operations that make up the function will either have taken effect or not taken effect. A function is not atomic if in the event of failure the participants are left in an inconsistent state.

For example, imagine a user of a shopping service. When the user purchases items from the service then the items are dispatched for delivery and the user's bank account is debited for the appropriate amount. This purchase function should be atomic. In the event of a server failure or client failure then a situation should never arise where the items have been dispatched but the user's bank account is not debited, or vice-versa.

Transactions are a well-known technique for providing atomicity. A transaction is a set of operations that has the following properties.

Atomicity – the transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).

Consistency – transactions produce consistent results and preserve application specific invariants.

Isolation – intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.

Durability – the effects of a committed transaction are never lost (except by a catastrophic failure).

These ACID properties guarantee that a transaction supports the "all or nothing" property. A transaction that has terminated with a commit has all the changes made within it made durable. A transaction that has terminated with an abort has the changes undone.

Typically transaction support services can provide the "all or nothing" property in the face of software or hardware failure. As transactions progress all participants keep local durable logs that can be used to restart the transaction after a crash. Also replication can be used to provide high availability for the objects that are changed during the execution of a transaction. However, in the MAFTIA project we extend the definition of fault tolerance to include tolerance of malicious faults. To do this we apply the general MAFTIA architectural principle of distributing trust by replicating the servers implementing the transaction service and optionally the resource managers and resources.

3.3.2 Transaction Service Architecture

A high-level view of the transaction service architecture is shown in figure 3.1. The transaction service architecture is made up of clients, resource managers and transaction

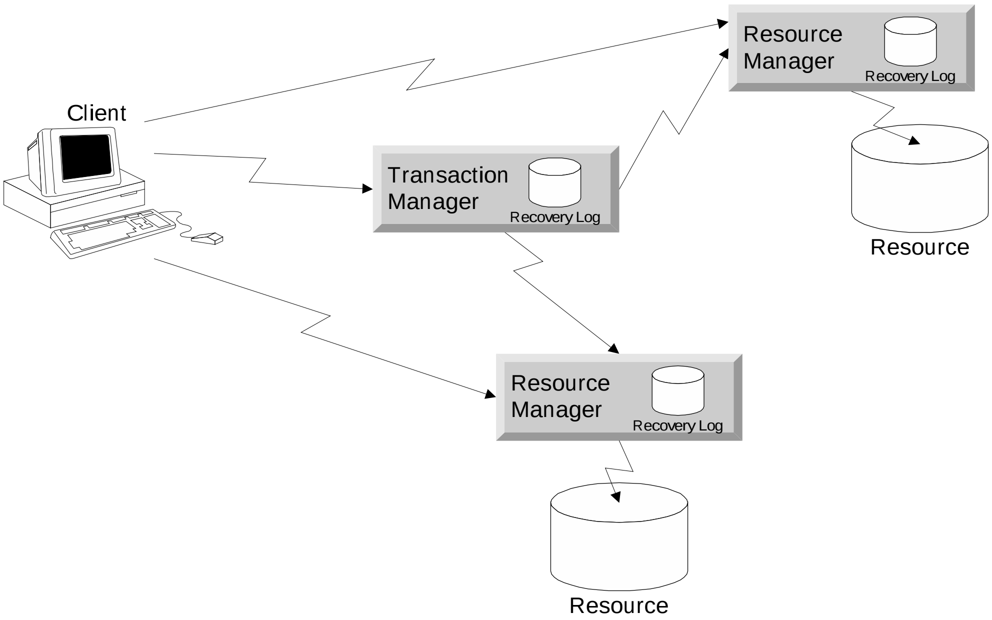

Figure 3.1: High-level View of Transaction Service Architecture

managers. In order to provide intrusion-tolerance these may be replicated. However, this is not visible from the viewpoint of a user of the service.

Clients. Clients use the transaction manager to begin and end transactions. Within the scope of a transaction the clients operate on resources via resource managers. As a slight extension to the typical transaction architecture, we allow multiple clients to participate in a transaction. A single client begins a transaction, and passes the transaction identifier to other clients so that they can cooperate within the transaction scope too. Individual clients can unilaterally force a transaction abort but all clients must unanimously agree to attempt a transaction commit.

Resource Managers. A resource manager is a wrapper for resources that allows the resource to participate in two phase commit and recovery protocols coordinated by a transaction manager. The resource may or may not be persistent. Persistent resources may use a persistency service or a database. Resource managers also manage the concurrent access by clients to resources. Concurrency control can be pessimistic or optimistic. Resource managers use a local recovery log to support recovery after a crash.

Transaction manager. The transaction manager is primarily a protocol engine. It implements the two phase commit protocol and recovery protocol. It also allows the creation of new transactions and the marking of transaction boundaries. In order to participate in transactions, resource managers are required to register themselves with the transaction manager. Transaction managers use a local durable recovery log to support recovery after a crash.

3.3.3 System model

A general approach taken in the MAFTIA project towards intrusion-tolerance is to use replication and voting for fault masking. We implement this approach using the intrusion-tolerant group communication protocols provided by the lower layers of the MAFTIA middleware. These group communication protocols can tolerate the byzantine behavior of a proportion of group members, and can tolerate timing faults.

We replicate transaction managers and resource managers/resources. These form server groups that are distributed across sites. Server groups are a set of n servers, of which up to t may fail in completely arbitrary ways. Requests are handled by all members of the service group and the majority result is returned to the user of the service. This means that as long as no more than t servers fail, the overall service remains trustworthy. To allow voting on results the servers are assumed to be deterministic. The value of t is determined using the *generalized adversary structures* introduced in [4]. This approach takes into account the diversity of the servers and attempts to find the minimum number with a common failure mode.

Interacting with the transaction managers and resource managers are an unknown number of possibly faulty clients. Clients are outside our control and can be implemented in any way. Therefore they can fail in arbitrary ways in the value domain. Currently we do not make clients intrusion-tolerant or the transaction service tolerant of misbehaving clients.

The replicated transaction managers, replicated resource managers, and clients are initialised by a trusted "dealer" with initial values such as the group identifiers necessary to use the system, public and private keys, etc. We assume that these initial values are stored securely. Depending on the construction of the site these keys may be protected by a security kernel or on a smartcard [1], or stored within a local trusted timely computing base (TTCB).

All parties in the system are connected by an asynchronous network and possibly a low capacity synchronous network. The timing model is an optimistic synchrony model. In [47] a system exhibiting optimistic synchrony is defined as a system that uses partially synchronous protocols where possible but falls back to probabilistic malicious-fault resilient asynchronous protocols if the system doesn't exhibit enough synchrony. The partially synchronous protocols rely on the presence of a global trusted timely computing base that is built out of local TTCB units and a trustworthy low capacity synchronous network linking the local TTCBs. The asynchronous network connecting the parties is assumed to be under the control of the adversary. An adversary may insert, delete, reorder or not deliver messages. The asynchronous protocols have been designed to tolerate such timing faults (short of a complete denial-of-service). On the other hand, the partially synchronous protocols do not need to tolerate timing faults as they rely on the use of the trusted synchronous network.

3.3.4 Related Approaches

The database community has explored the use of atomic broadcast protocols to support transactions for replicated databases (for a good review of various approaches see [51]). However, their focus has been on database replication for availability rather than intrusion tolerance as they assume only crash failures are possible. Furthermore, their models usually assume that transactions are executed locally on a member of a replica group and the effect of the transaction is replicated.

The dynamic terminating multicast proposed in [15] describes a primitive that can be used to build transactional and group communication systems. However, this paper only considers the implementation of the commit protocol and does not consider how the other ACID properties are guaranteed.

In [39] a unified framework is proposed based on extensions to a SEND-TO multicast primitive. The primitive is first defined with delivery permanence, i.e. the majority of members of a group deliver a message or no member of a group delivers a message. This is extended to support an all-or-nothing delivery property where instead of multicasting to a single group the multicast is to multiple groups. Finally a global total ordered delivery is imposed to ensure that every process delivers messages in the same order as every other message. This provides an elegant way to express some types of transactions using group communications. The transactional properties of atomicity, durability and serializability are guaranteed by the SEND-TO primitive as follows: essentially the *all-or-none* delivery property of the SEND-TO primitive equates to a transaction's *all or nothing atomicity*, the delivery permanence property of the SEND-TO equates with durability, and total ordering enforces serializability. However, [39] does not address isolation, subtransactions or recovery after failure. Also it is not clear how serializability is maintained if transactions are allowed to interleave operations.

For [22] the combined use of transactions and group communications is used to provide flexible support high availability applications. The approach is not to build transactions on top of group communications but to use transactions and group communications together to implement fault-tolerant replication. It is argued that transactions are best used to deal with all exceptions that cannot be masked (abort and retry). Group communication can be used to provide efficient support for binding service replication, fast switch over from a failed server, and to implement active replication. Transaction concurrency control is used to allow message ordering to be relaxed for some operations. This is achieved by extending the "exclusive write/shared read" policy to object groups. Clients are required to lock a majority of the members.

The GroupTransactions [31] model is a integrated model for transactions and group communications where transactional replicas can be groups of processes. In this model a single client interacts with transactional replicas. The replicas can be aware of each other and communicate using multicast. The model also supports subtransactions, multi-threaded transactions and considers failure atomicity. GroupTransactions is implemented as a library TransLib for Ada [17].

Our approach is to make use of standard group communication primitives, allow for heterogeneous resources, apply error compensation techniques to improve intrusion tolerance, to allow for multi-party (and potentially) multi-threaded transactions and to consider failure atomicity. This differentiates our work from approaches that make use of new or modified group communication primitives (for example, optimistic broadcast) [15, 39, 51]. Also, unlike other approaches, our focus is not on availability but on intrusion tolerance. This has resulted in us not being able to use techniques such as *passive* replication that are widely used by the database community. Passive replication is more efficient than active replication, and does not require deterministic replicas. However, the problem with adopting *passive* replication is its reliance on a leader-follower model. The updates occur at the leader and the followers are informed of the results. Whereas this adequate in a crash-fail fault model there are problems when the leader can fail (be corrupted) yet keep on functioning and sending corrupted updates to the leaders. By adopting an *active* replication approach we avoid this problem as there is no single point of failure and more that t members of the group must be corrupted before the group as a whole is compromised.

The approach that is closest to our is that of *GroupTransactions* [31] although our model of multi-party clients is different in that unlike their model our multiple parties are not created within the context of a group transaction (in their model the multiple parties are threads that are spawned by a single thread that begins a group transaction). We differ also in that their system model assumes a LAN and does not explicitly consider malicious faults. However, they do address nested transactions whereas we only implement a flat transaction model.

3.3.5 Use of the Transaction Service

In this section we describe how the transaction service is used. First we give an overview of the interactions involved in a transaction then discuss the use of resource managers, and transaction managers in more detail.

3.3.5.1 Overview

Figure 3.2 shows the main interactions between a client, transaction manager, resource manager and a resource. These interactions will be explained in more detail in following sections, the purpose of this figure is to illustrate at a high level the typical interactions between the components.

A client establishes a transaction by invoking the begin operation on the transaction manager. The client then makes its first invocation of an operation on a resource via the resource manager that wraps the resource, note that the transaction identifier is passed with the invocation. The resource manager uses the transaction identifier to determine if it is already involved in a transaction, if it is not then it invokes the registerResource operation on the transaction manager to register itself as participating in the transaction. The resource manager then delegates the operation to the resource it is wrapping. Although not shown we assume that the resource manager receives the result and returns it to the client. The client then invokes Another operation on the resource using the resource manager, again it is delegated to the resource for evaluation and the result returned to the client. When the client has finished invoking operations within the context of the transaction the client asks the transaction manager to attempt a commit by invoking the commit operation. The transaction manager then invokes the prepare operation all resource managers registered with it to determine if all resource managers can commit. If all resource managers return ok (not shown) to the transaction manager then the transaction manager asks all resource managers to commit by invoking their commit operation.

3.3.5.2 Resource Managers

Each resource manager implements all the operations that the wrapped resource implements, and in addition implements the ResourceManager interface that specifies the operations required for participation in transactions. We have based the design of this interface on the OPTIMA framework for open transactions [20].

When a resource manager is invoked in the context of a transaction, the resource

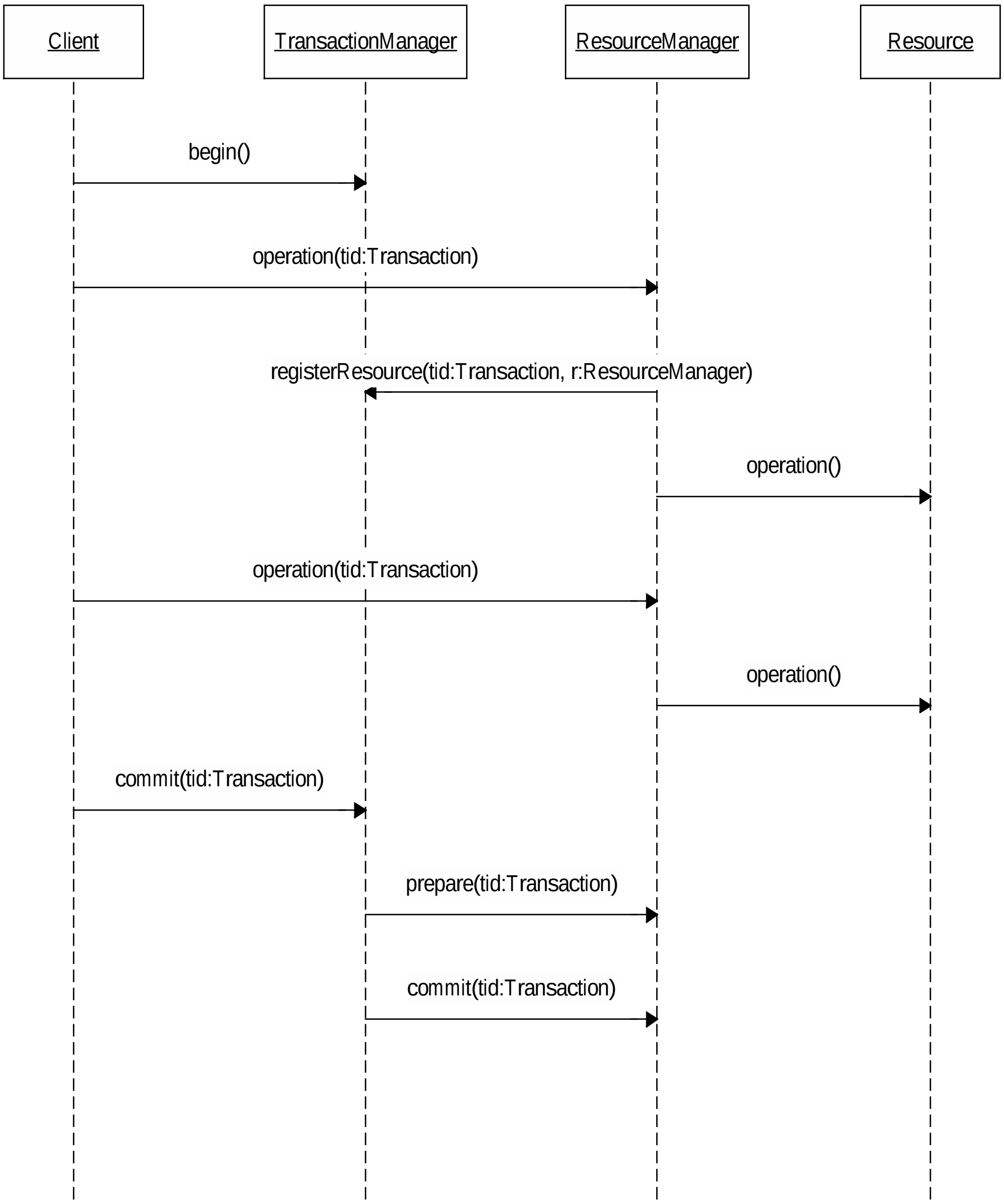

Figure 3.2: Overview of Use of Transaction Service

manager registers itself with the transaction manager using the transaction manager's register operation. The transaction manager keeps a list of resources participating within each transaction.

How does the resource manager determine the transactional context? The client must propagate the transaction identifier with every invocation of a resource manager operation. There are generally two ways to do this: implicit, or explicit propagation. With implicit propagation some hidden context is associated with the client threads and passed transparently whenever the client invokes a resource manager operation. The drawback with this approach is that it makes it difficult to program multi-party clients as the transaction identifier exposed and cannot be passed between clients. With explicit propagation the transaction identifier is exposed and client must add the transaction identifier as an argument to every invocation of a resource manager operation. Explicit propagation can be implemented by adding a transaction identifier argument each operation belonging to the resource's interface. For example, if a resource has a deposit operation with the signature:

```
public void deposit(float amt)
```

Then the resource manager implements the operation with an extra transaction identifier argument:

```
public void deposit(Transaction tid, float amt)
```

The resource manager operation prepare is called by a transaction manager to determine if the resource manager can commit or not. If the resource manager can commit changes (make them permanent) then the resource manager returns true, otherwise it returns false.

Once the transaction manager has taken into account the result from calling prepare on all resources registered as taking part in the transaction, it either calls commit or abort on each resource manager. If commit is called then the resource manager causes the resource to make permanent any changes made within the transactional context. If abort is called then the resource manager forces the resource to forget any changes made within the transactional context.

As resources may be used by multiple clients and may participate in multiple transactions it is necessary to use a concurrency control protocol to ensure that the *isolation* property is guaranteed. There are two main types of concurrency control schemes: pessimistic, and optimistic concurrency control.

Pessimistic. Locking is the most widely used pessimistic concurrency control protocol. A locking protocol requires that a client requests a lock for a resource before it accesses the resource. A lock manager tracks the locks held for resources and makes the decision whether to grant a new lock for a resource. Unless a deadlock arises the lock manager will grant all locks eventually. It will grant a lock immediately unless there is a conflict between the lock requested and the locks held for the resource. If there is a conflict then the request is blocked until it can be satisfied. Lock conflicts are determined by examining lock types and the transactional context of locks. Pessimistic locking schemes suffer from the problem of deadlock. Deadlock occurs when two concurrent transactions obtain locks on some of each other's required resources and neither transaction can progress until it has locks on all of its required resources. Deadlock can be resolved by using timeouts on lock possession or using a deadlock detection process.

Optimistic. Where conflicts between transactions is rare an optimistic concurrency control protocol can be used. An optimistic scheme allows a transaction to interact freely with resources but at upon attempting to commit the transaction is validated to determine if the isolation property would be violated if the transaction committed. If it would not be violated then the transaction is committed, otherwise the transaction is aborted. Unlike the locking protocol the optimistic concurrency control protocol does not suffer from deadlocks but is more complex to implement for a distributed system.

The ResourceManager interface supports the implementation of both pessimistic and optimistic concurrency protocols. A pessimistic lock based protocol (implemented as a lock manager) would implement the setLock and unLock methods and define validate to always returns true. An optimistic protocol would define the two lock methods as null, and the validate method would be defined to validate the transaction. There are two types of validation that could be implemented: forward validation, and backward validation [20]. Forward validation ensures that committing the transaction doesn't conflict with the activities of active transactions, if there is a conflict then the transaction is aborted. Backward validation checks that commitment isn't invalidated by a commit of a prior transaction. Initially we will only implement pessimistic concurrency control as this is the concurrency control scheme most widely used by practical systems.

If the resource being managed by the resource manager is a database then the concurrency control scheme supported by the database may be used by the resource manager. This may affect the granularity of the concurrency control. If the database was treated as a single object then the locking would be at the level of the entire database making for a very inefficient system. However, if the database has its own concurrency control scheme then concurrency control could be applied at the level of a single tuple. In this initial design, the transaction service supports concurrency control at the level of the resource.

Interface. A resource manager implements the ResourceManager interface and the application interface. For example,

```
public class ApplicationResourceMgr
   implements ResourceManager, Application {
}
```

The full ResourceManager interface is shown below:

```
public interface ResourceManager {
   // ask the object to commit
   public boolean prepare(Transaction tid);
   // force a commit - save changes
   public void commit(Transaction tid);
   // abort changes
   public void abort(Transaction tid);
   // request a lock on a resource
   public void setLock(Transaction tid, int lockType);
   // unlock a resource (not done by client but by transaction
   // manager)
   public boolean unLock(Transaction tid);
   // validate transaction, called by transaction manager
   public boolean validate(Transaction tid);
}
```

3.3.5.3 Transaction Manager

In our system the transaction managers are realized by a class implementing the TransactionManager interface. A transaction manager can support multiple transactions, each transaction has a unique transaction identifier (tid). A client starts a transaction by invoking the begin method of the transaction manager. The transaction manager generates a unique transaction identifier for the transaction. The tid is passed back to the client. This tid is used as a capability and it allows any client possessing it to participate in the transaction. Object signing is used as a technique to prevent the transaction identifier from being cloned and used by unauthorised clients. A client that wishes to take part in an active transaction sends the tid with an invocation of the join method of the transaction manager. A new transaction identifier which can only be used by that client is then returned by the transaction manager. Only authorised clients are allowed to join a particular transaction, and interact with resource managers also taking part in the transaction. A client ends a transaction by sending the tid with an invocation of either the commit method or abort method of the transaction manager. The current status of transaction can be queried by using getStatus. This is used for the recovery protocols.

An example is shown below:

```
// begin the transaction
Transaction myTid = TransactionManager.begin();

// transactional operations

// end the transaction
transMgr.commit(myTid);
```

Atomic commitment of transactions is achieved using a modified two phase commit protocol (2PC) [27]. We allow multiple clients initiate the protocol. We assume that all clients must agree on committing a transaction but that any client can force the abort of a transaction. This is similar in concept to the semantics of Coordinated Atomic Actions [52, 36] where participants must either agree on a normal or exceptional outcome, or abort the entire action. As our model of multi-party transactions does not currently consider exceptions then the agreement must be on whether to commit. If there is no agreement to commit then the transaction is aborted. Once a decision has been made the protocol executes in two phases:

- the transaction manager asks all resource managers participating in the transaction if they vote for commit or abort of the transaction.
- the transaction manager decides whether to commit or abort on the basis of the collected votes and informs each resource manager involved in the transaction.

In the standard protocol a transaction will only be committed if all the resource managers vote that they can commit; if any resource manager cannot commit then the transaction is aborted. As we replicate the resource managers this can be weakened to a majority vote.

As any of the participants (including the transaction manager) may fail, it is important that a log maintained in stable storage by resource managers and transaction managers is used to record decisions made during the protocol execution. This means that upon restart the transaction state can be recovered to a point where the transaction will either abort or commit. Even in the presence of failure transactions maintain their property of atomicity.

At the end of a transaction the transaction manager also has the responsibility of releasing any locks that have been acquired during the two phase locking protocol.

Interface. The full TransactionManager interface is shown below:

```
public interface TransactionManager {
   // create a new transaction
   public static Transaction begin();
   // join an active transaction
   public Transaction join(Transaction tid);
   // request a transaction commit
   public void commit(Transaction tid);
   // force a transaction abort
   public void abort(Transaction tid);
   // register a resource as being a member of the transaction
   public void registerResource(Transaction tid, ResourceManager r);
   // get current status of a transaction
   public int getStatus(Transaction tid);
}
```

4 Middleware Protocols

4.1 Multipoint Network

In this section, we will identify the protocols that compose the services defined section 3.1.

4.1.1 Internet Protocol

For IP, the two most used protocols are the UDP [32] and TCP [35], which are standard and widely used protocols. Therefore, we will not include a definition of these protocols here.

4.1.2 IP Multicast

In order for IP Multicast to fully work, hosts must support some kind of management for group membership. This is accomplished using the Internet Group Management Protocol (IGMP). This protocol is used by IP hosts to report their host group memberships to any immediately-neighboring multicast router.

The way the protocol works is by defining two types of IGMP messages: Host Membership Query (Queries) and Host Membership Report (Reports). Multicast routers send Queries to discover which host groups have members on their attached local networks. These Queries are sent to the special group address 224.0.0.1, which includes all hosts that implement IP multicast in the local network. The hosts respond by generating Reports, containing each host group to which they belong on the network interface at which the Query was received. In fact, to reduce the total number of Reports received and to avoid congestion, when a host receives a Query, it starts a report delay timer for each of its group memberships. Each timer is set to a different, randomly chosen value between 0 and 10 seconds. It only sends the Report when the timer expires, with the destination address of the host group being reported (with a time-to-live of 1), so that other members of the same group on the same interface can receive the Report as well. If a host receives a Report for a group to which it belongs, it stops its timer delay for its own Report for that group and does not generate the Report. So, in general, only one Report per host group is generated inside a local network. This works because multicast routers receive all IP multicast datagrams, and need not be addressed explicitly and also because the routers need not know which hosts belong to a group, they only need to know that there is at least one member of the group on a particular network.

4.1.3 IPSec

As explained above, IPSec uses two traffic security protocols: Authentication Header (AH) and Encapsulation Security Payload (ESP). A definition of these protocols is found in [18] and [19], respectively. These protocols, on their hand, use known protocols to perform the security functions they provide. In the AH, IPSec may use Keyed-Hashing for Message Authentication (HMAC) [21] with Message-Digest Algorithm (MD5) or with Secure Hash Algorithm (SHA-1) [29]. In ESP, IPSec may use also the two above algorithms for authentication or the Data Encryption Standard (DES) [30] in Cipher Block Chaining Mode (CBC), which is applicable to several encryption algorithms and for which a general description is given in [41], to perform encryption. Since encryption (confidentiality) and authentication are optional, the algorithm for authentication and for encryption may be "NULL", although they can not both be "NULL".

4.1.4 Internet Control Message Protocol

This protocol uses different types of messages to perform a set of actions. The following table presents the different ICMP messages, with its corresponding type, as stated in RFC 792 [33]:

Code	**Type**
0	Echo Reply
3	Destination Unreachable
4	Source Quench
5	Redirect
8	Echo
11	Time Exceeded
12	Parameter Problem
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply

Table 4.1: Summary of ICMP messages.

4.1.5 Simple Network Management Protocol

This section defines the AgentX Framework, which is, we believe, very important in extending existing SNMP agents. This definition is taken from [12].

Within the SNMP framework, a managed node contains a processing entity, called an agent, which has access to management information.

Within the AgentX framework, an agent is further defined to consist of

- a single processing entity called the master agent, which sends and receives SNMP protocol messages in an agent role (as specified by the SNMP version 1 and version 2 framework documents) but typically has little or no direct access to management information.
- 0 or more processing entities called subagents, which are "shielded" from the SNMP protocol messages processed by the master agent, but which have access to management information.

The master and subagent entities communicate via AgentX protocol messages, as specified in [12] . While some of the AgentX protocol messages appear similar in syntax and semantics to the SNMP, bear in mind that AgentX is not SNMP.

The internal operations of AgentX are invisible to an SNMP entity operating in a manager role. From a manager's point of view, an extensible agent behaves exactly as would a non-extensible (monolithic) agent that has access to the same management instrumentation.

This transparency to managers is a fundamental requirement of AgentX, and is what differentiates AgentX subagents from SNMP proxy agents.

4.1.5.1 AgentX Roles

An entity acting in a master agent role performs the following functions:

- Accepts AgentX session establishment requests from subagents.
- Accepts registration of MIB regions by subagents.
- Sends and accepts SNMP protocol messages on the agent's specified transport addresses.

- Implements the agent role Elements of Procedure specified for the administrative framework applicable to the SNMP protocol message, except where they specify performing management operations. (The application of MIB views, and the access control policy for the managed node, are implemented by the master agent.)
- Provides support for the MIB objects defined in RFC 1907 [9], and for any MIB objects relevant to any administrative framework it knows.
- Forwards notifications on behalf of subagents.

An entity acting in a subagent role performs the following functions:

- Initiates an AgentX session with the master agent.
- Registers MIB regions with the master agent.
- Instantiates managed objects.
- Binds OIDs within its registered MIB regions to actual variables.
- Performs management operations on variables.
- Initiates notifications.

4.2 Communications Support

In this section we shall give some indications of how the communications services described in section 3.2 are realized. There are two levels on which this bears examination, that of the over-all architecture of the MAFTIA group communications middleware – the various interactions between different protocol components as well as between MAFTIA protocols and other elements both above and below on the protocol stack – and the precise algorithmic details of the individual protocols themselves. The protocol algorithms have been given in some detail in other places [4], so here we shall concentrate instead on the large-scale architecture.

4.2.1 Architecture Description

The basic group communications primitives provided by MAFTIA middleware fall into the three classes of *agreement*, *stream broadcast* and *sequenced broadcast*. While the particular meaning and certainly the implementation of different protocols in the same

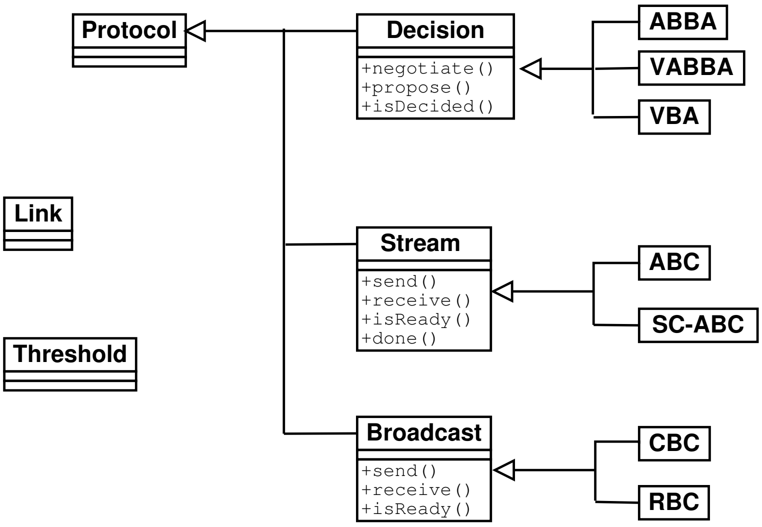

Figure 4.1: Class Overview

category are quite different, the API and general structure remains unchanged. For this reason each of these categories is represented by a class, called `Decision`, `Stream` and `Broadcast`, respectively. These classes are extended to form the particular implementations of all individual protocols, while in turn these all extend the class `Protocol` which carries basic information pertinent to all possible protocol types.

Additional infrastructure is provided by two other groups of classes, those responsible for cryptography and those related to basic system configuration and message handling. Of these we mention in particular only `Threshold`, which provides threshold signatures, encryption and coins, and `Link`, which is the access point for all protocol implementation to the multipoint network. Protocol layers communicate with each other and with `Link` by sending asynchronous events, which is also the mechanism behind the asynchronous invocation method allowed by the API.

An overview of these classes and their relationships is shown in Figure 4.1.

4.2.2 Protocols under Static Groups

We shall now provide some specific information about the implementation of the various protocols of the MAFTIA middleware in the static group model. In particular, we will very briefly summarize the algorithms underlying these protocols, in large part to point out the layering and composition which is used to build the complex protocols out of simpler ones. It shall quickly become clear that the simple picture in Figure 4.1 with individual protocols as terminal leaves in a simple tree, while literally true on the level of class inheritance, is quite deceptive in terms of protocol execution: in fact, most protocols call one or more of the others as subprotocols, and *all* protocols use the `Link` layer to handle external communications with the rest of the group.

1. In ABBA, each party begins by sending its initial vote to the whole group, then collecting sufficiently many such votes and determining what is the majority. Then the parties start going through rounds with a pre-vote stage, a main-vote stage, and then a decision or threshold coin-share generation stage. Each of these votes are justified by threshold signatures and each time sufficiently many votes or coin shares must be collected to assemble the full signature or coin. This is a very direct protocol, and while it does (potentially) quite a bit of multipoint communication, it does not call any subprotocols.

2. VABBA is a slight modification of ABBA whereby each party must keep track of validation data for it's own initial vote as well as for the opposing vote, if it should ever appear in some round's communication from another group member. Then whichever vote is selected at the end of the protocol run, a validation can be provided for it.

3. The basic idea of VBA is that every party uses CBC to propose its own value as a candidate value for the final result, and then one party whose proposal satisfies the validation predicate is selected as the final decision value in a sequence of VABBAs.

4. For CBC, the sender transmits the message to the whole group and then waits for sufficiently many parties to reply with a threshold signature share as a "witness" guaranteeing consistency. The shares are combined and the resulting signature is sent to the group; any party receiving this signature immediately delivers the message. Note that while there is extensive use of the `Link` here, no other protocols are called.

5. The sender in RBC first sends her message to the whole group; any party responds to the first such opening message by echoing the hash of the message to the group. Upon receiving sufficiently many echo messages, each party sends a ready message to the group, again with the hash of the message. A group member who receives sufficiently many ready messages may deliver the corresponding message if she has it, otherwise she will ask enough other parties for the message body to be she that some other honest party will have it and can respond. Again, this protocol has a high multiparty network communications complexity but depends upon no other subprotocols.

6. ABC proceeds by a series of global rounds in which the parties agree upon a set of messages to deliver at the end of each round. More precisely, in a given round every party signs the message it proposes and send this to the whole group. She then proposes a list of (sufficiently many) such messages for a run of VBA and the decided-upon list of messages is delivered in some order fixed *a priori*.

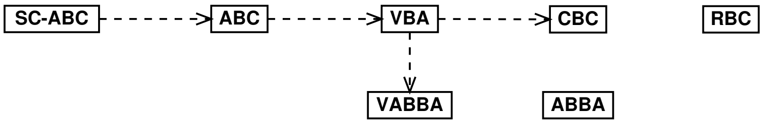

Figure 4.2: Protocol Functional Dependencies

7. In SC-ABC, an ABC channel is used to broadcast the ciphertexts. When a given ciphertext is atomically delivered for some party, she then sends via `Link` a request to the whole group to decrypt this message along with her share of threshold decryption. When sufficiently many such shares come in, she can combine them to get the message cleartext and deliver it.

These inter-protocol dependencies are depicted in Figure 4.2.

4.2.3 Protocols under Dynamic Groups

Dynamic groups are supported by the group membership modules mentioned above. The protocols are parameterized by an instance of a `View`, which contains all relevant parameters (such as the number and the identities of all members).

We need three basic protocols to maintain the secret keys within a dynamic group: Add, Remove, and Reshare. We give only a brief overview of these here; more details can be found in [46].

Protocol Add adds a new member to the group as a consequence of the `joinGroup` operation, and supplies the necessary secret keys. For the commonly used linear secret sharing schemes, this can be achieved by means of a distributed secure computation from which the new member learns its secret key.

Protocol Remove eliminates a member from the group as a consequence of the `leaveGroup` or `excludeGroup` operation. No communication between the members is needed to achieve this, but one must assume that all cryptographic keys are removed from the memory of the removed party.

Protocol Reshare basically involves replacing the degree-t sharing polynomial with a randomly chosen degree-t' polynomial that shares the same secret. This is a well-known primitive in synchronous threshold cryptography, but new protocols had to be developed for the asynchronous case [46].

4.3 Activity Services

This section describes at a high level the protocols used to implement the transactional support service.

The following protocols are outlined in this section:

- Atomic commitment and abort protocols
- Recovery protocols,
- Distributed locking protocol,
- Resource manager replication protocol.

The protocols make use of some communication primitives that abstract away from the issue of whether asynchronous or partially synchronous protocols are being used. There are primitives for atomic broadcast, secure causal broadcast, consensus, reliable broadcast and reliable send. Each of these is detailed below:

Atomic broadcast The atomic broadcast primitives are:

- a-broadcast(gid, message) initiates atomic broadcast of message to all group members of the group gid,
- a-deliver(message) is called when a message is delivered to a participant, this takes place after the message has been received and some computation has been performed to guarantee the total order atomic broadcast property.

Secure causal atomic broadcast The secure causal atomic broadcast primitives are:

- sc-broadcast(gid, message) initiates a secure causal atomic broadcast of message to all group members of the group gid,
- sc-deliver(message) is called when a message is delivered to a participant is delivered, this takes place after the message has been received and some computation has been performed to guarantee the secure causal total order atomic broadcast properties.

Consensus This service allows consensus on a proposed value amongst the members of the group *gid*, we assume that the consensus function is flexible and the decision algorithm can be adjusted. In the protocols described below, consensus is on the value proposed by the majority of honest participants.

The consensus primitives are:

- propose(gid, vote) launches consensus for the group gid with an initial vote (vote),
- decide(gid, decision) is called when the consensus ends and an agreed majority value (decision) has been decided upon.

Reliable send The reliable send primitives are:

- send(pid, message) send a message to a participant (pid).
- deliver(message) is called when the message is delivered to the participant.

Reliable broadcast The reliable broadcast primitives are:

- broadcast(gid, message) make a reliable broadcast of a message to a group gid of message (order is not guaranteed).
- deliver(message) is called when the message is delivered to the participant.

We assume that the group communication primitives make use of two types of group. There is a transaction manager replica group (transaction manager group) and possibly multiple resource manager replica groups (resource manager group). It is assumed that the transaction manager replica group is a static group that exists at system setup, and the resource manager groups are dynamic groups established when resource managers register themselves with the transaction managers. See figure 4.3 for an illustration. Here a client group is shown interacting with a group of transaction servers, and two groups of resource managers and resources. Note that the resource managers are the dark circle, and the resources are shown as white circles wrapped by a resource manager. Although it is possible to imagine a configuration where multiple resource managers manage a single resource we are initially assuming that there is always one resource manager to a resource and they are replicated as one component.

We assume that open groups are used which means clients do not need to belong to the transaction manager group to broadcast to them. Similarly the resource manager group is open and transaction managers can broadcast to them without having to be within their group. Closed groups could be supported but it would require the establishment of overlapping groups, for example a client and transaction manager replica group.

4.3.1 Assumptions

These protocols gain their intrusion tolerance from the use of replication and voting. If we assume that there are a minimum of $2t+1$ replicas then the protocols can tolerate up to t faulty replicas. Termination and liveness guarantees are provided by the underlying protocols. The underlying protocols also mask timing faults, and byzantine behavior where

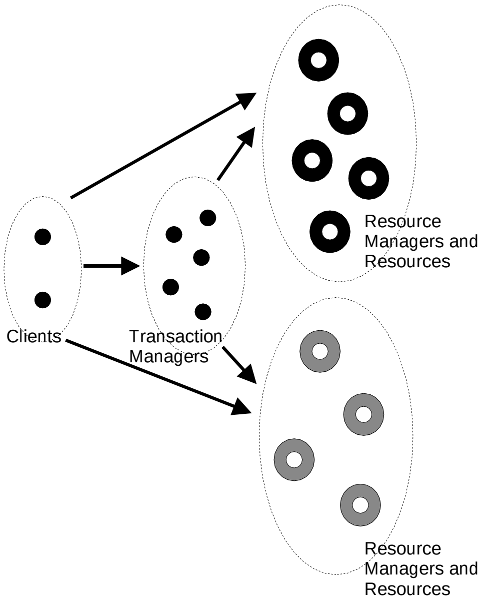

Figure 4.3: Transactional Support Service Groups

replicas send conflicting messages to each other. Where *serializability* is required we make use of secure causal broadcast which ensures that each replica receives messages in the same order that they were sent by the sender.

A common step in the protocols is to wait until $t + 1$ distinct replicas have sent identical messages and the deliver this message. As long as our assumption that no more than t replicas are faulty this is guaranteed to terminate. We define a function majority(primitive(message)) that uses the primitive to collect delivered messages and only delivers message when $t+1$ distinct replicas have sent identical messages. For example, majority(sc-delivery(message) will only deliver message when $t + 1$ distinct replicas have sent identical messages. We use this primitive to simplify the presentation of the protocols below.

The recovery protocols assume that failed replicas will eventually be re-initialised by some external agency, and that transaction managers and resource managers have secure durable recovery logs. The force-write(message) primitive is used to write message to the recovery log.

4.3.2 Atomic commitment and abort protocols

In this section we describe an intrusion tolerant two-phase atomic commitment protocol [27] and an intrusion tolerant abort protocol based upon the MAFTIA middleware atomic broadcast protocols. The protocol is blocking as a unanimous decision to commit is required from all clients participating in the transaction before commitment can take place.

The two-phase atomic commitment protocol (2PC) is described below:

Protocol *2pc-atomic-commit*:

1. Each client multicasts to the transaction managers using sc-broadcast(transaction manager group, commit),
2. each transaction manager (TM) waits for sc-delivery(message) where the message is commit from **every** client and then, if there is any disagreement then the abort protocol is invoked
3. if all clients have decided to commit then the TM performs sc-broadcast(resource manager group, prepare), thereby asking each resource manager to prepare,
4. each resource manager (RM) waits for majority(sc-delivery(message))and then,
5. each RM that is willing to commit force-writes a prepare log record, or if it only willing to abort force-writes an abort log record,
6. each RM performs sc-broadcast(transaction manager group, ok) if it has committed okay and otherwise performs sc-broadcast(transaction manager group, notok),
7. each TM waits for majority(sc-delivery(ok)) from each resource manager group, until ok has been received from each resource manager group,
8. if each RM group has decided ok then the TM decides on commit, if any RM group decided on notok then the TM decides on abort,
9. the TM force-writes either commit or abort to the log,
10. each TM multicasts its decision either to commit or abort to the resource manager groups using sc-broadcast(resource manager group, decision),
11. each RM waits for majority(sc-delivery(message)),

12. if the RM received commit then it force-writes a commit record, multicasts an acknowledgment to the transaction manager group using sc-broadcast(transaction manager group, ack) and commits the transaction,

13. if the RM has received abort then it force-writes an abort record, performs a sc-broadcast(transaction manager group, ack) and undoes any changes,

14. each TM waits for majority(sc-delivery(message)) from each resource manager replica group, if the message is ack then the transaction manager force-writes an end record.

The atomic abort protocol is described below. We assume that any client can force an abort of a transaction.

Protocol *atomic-abort*:

1. A client multicasts abort to the transaction manager group using sc-broadcast(transaction manager group, abort),

2. each transaction manager (TM) waits for sc-delivery(message), if the message is abort then,

3. each TM multicasts to the resource manager groups using sc-broadcast(resource manager group, abort),

4. each resource manager (RM) waits for majority(sc-delivery(message)),

5. if the message was abort then it force-writes an abort record, multicasts it result to the transaction manager group using sc-broadcast(transaction manager group, ack) and undoes any changes,

6. each TM waits for majority(sc-delivery(message)), if the message is ack from each resource manager replica group then TM force-writes an end record.

4.3.3 Recovery protocols

In the event of the failure of a site the aim is to still provide an atomic outcome for the transaction or *failure atomicity.* The recovery protocol attempts to do this by examining log messages maintained by different participants of the transaction service when they restart. There is a separate protocol for resource managers and transaction managers.

Protocol *resource-manager-recovery*:

1. If on restart, a resource manager (RM) finds itself in the prepare state then,
2. the RM multicasts to the transaction manager group using a-broadcast(transaction manager group, getStatus) to discover if the transaction has completed,
3. each transaction manager (TM) checks its logs to determine the result of the transaction and replies using send(client, result),
4. the TM waits until majority(deliver(message)) delivers a message,
5. if aborted have been received then the RM aborts, otherwise it commits any related local state.

Protocol *transaction-manager-recovery*:

1. If on restart, a transaction manager (TM) finds itself in either commit or abort state then it multicasts to the transaction manager group using a-broadcast(transaction manager group, getStatus) to discover if the transaction has completed,
2. each TM is delivered the getStatus message by a-delivery(message), checks its logs to determine the result of the transaction and sends back the result send(transaction manager group, result),
3. the TM waits until majority(deliver(message)) delivers the result,
4. if the result is aborted then the transaction was aborted, the TM writes an end message to its log,
5. if the message is committed then the transaction was committed, the TM writes an end message to its log,
6. if the message is committing or aborting messages are received then the TM completes the two-phase commit or abort protocols from the commit or aborting point.

This approach assumes that eventually a majority of transaction managers will recover and accordingly resource managers will commit or abort.

4.3.4 Distributed locking protocol

When locking resources the decision as to whether a resource manager will grant a lock or not depends on whether the locks are compatible. We assume a one writer, multiple reader model where the holding of a read lock does not prevent the granting of write lock but the holding of a write lock prevents the granting of another write lock by a client not participating in the transaction. We assume that clients within the same transactional context take care of concurrency between them using an application specific protocol.

The protocol must take into account the fact that resource manager replicas need to come to a majority agreement on the granting of a lock. To achieve this consensus is used.

The lock compatibility scheme implemented by the protocol is the commonly used lock compatibility scheme of *one writer and multiple readers*. Here only one write lock can be held for a resource, but multiple read locks can be held unless there is already a write lock held for the resource. Who owns the locks must also be taken into account. Where nested transactions are supported, multiple possession semantics can be defined that allow child transactions to gain locks that are compatible with their parent transaction. The current specification of the transaction service only allows flat transactions, so we only allow locks to be granted when the request is in the same transactional context as those locks that are already held.

The locks are acquired and released using a *two-phase locking* protocol which increases the visibility of resources but still ensures that the isolation property is guaranteed. In the two-phase locking protocol, lock acquisition for resources is attempted as needed but no lock can be released until all required locks have been acquired. In effect, this means the locks that have been acquired are only released when the transaction aborts or commits. Deadlock can occur if a lock request fails because the object is locked already by another transaction and the lock requester blocks in the hope that the lock may be released soon. In this protocol we use local timeouts to resolve the problem of deadlock. We also apply a timeout to the step where the protocol blocks until $t + 1$ messages have been received. If timeout occurs then the result is assumed to be notok.

Protocol *replica-locking*:

1. A client uses sc-broadcast(resource manager group, setLock(transaction id, lock type)) to multicast the lock request to the resource manager group,

2. Each resource manager (RM) has sc-delivery(message) invoked and begins consensus with the other members of the replica group,

3. each member proposes either ok or notok

4. once consensus has resulted in a decision each RM uses send(client,decision) to inform the client,

5. the client waits until majority(deliver(message)) delivers a result.

4.3.5 Resource manager replication protocol

Resource managers are replicated using an *active replication* group management policy. In active replication, all the functioning members of the group perform processing [40]. This requires that the replicas are deterministic and all client invocations are processed in the same order. We make use of the atomic broadcast protocol to ensure that client invocations are delivered in the same order to all honest replicas, and we make use of voting to determine the majority decision of all replicas that return a result. Note that when a request is send a sequential request id is encapsulated by the request, and when the reply is returned it encapsulates the corresponding request id. This is to allow the client to order results and thereby achieve sequential consistency for the results of requests.

Protocol *data-replication*:

1. the client uses sc-broadcast(resource manager replica group, request) to multicast to the resource managers replica group,
2. each resource manager (RM) waits for sc-delivery(request) and then executes the request,
3. the resource manager uses send(client, result) to send the result back to the client,
4. the client waits for majority(delivery(reply)) to deliver the result.

5 Conclusion

This deliverable presented the first specification of the APIs and protocols for the MAFTIA middleware. It started by describing the interfaces of the runtime environments that will support the middleware architecture and other components in general that might want to use their services, namely the Appia protocol kernel and the Trusted Timely Computing Base (TTCB). Next, the interfaces of the following modules were presented: Multipoint Network; Communication Services; Activity Services; Site Membership; Participant Membership. The APIs described can be used not only by end-user level programs, but also recursively by other modules of the architecture. Since many of the protocols are currently under development, the protocol chapter was mainly devoted to definition of the underlying principles and in some cases a few protocols were briefly described.

In a next deliverable of WP2, "D9 - Complete specification of APIs and protocols for the MAFTIA middleware", this specification will be further refined with the inclusion of updates to the APIs and with a complete specification of the protocols.

Bibliography

[1] N. Abghour, Y. Deswarte, V. Nicomette, and D. Powell. Specification of authorisation services. Technical Report LAAS Report 01.001, LAAS-CNRS, 23 January 2001.

[2] The *Appia* website. http://appia.di.fc.ul.pt.

[3] U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). RFC 2264, January 1998.

[4] C. Cachin, editor. *Specification of Dependable Trusted Third Parties*. Deliverable D26. Project MAFTIA IST-1999-11583, January 2001.

[5] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1), 1997.

[6] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1067, August 1988.

[7] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1098, April 1989.

[8] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[9] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Management information base for version 2 of the simple network management protocol (snmpv2). RFC 1907, January 1996.

[10] J. Case, S. Waldbusser, M. Rose, and K. McCloghrie. Introduction to Community-based SNMPv2. RFC 1901, January 1996.

[11] A. Casimiro, P. Martins, and P. Veríssimo. How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–134, Porto, Portugal, September 2000. IEEE Industrial Electronics Society.

[12] M. Daniele, B. Wijnen, and Ed. D. Francisco. Agent extensibility (agentx) protocol. RFC 2257, January 1998.

[13] M. Daniele, B. Wijnen, Ed. M. Ellison, and D. Francisco. Ed. Agent Extensibility (AgentX) Protocol. Network Working Group, Request for Comments: RFC 2741, January 2000.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[15] R. Guerraoui and A. Schiper. The Transaction Model vs The Virtual Synchrony Model: Bridging the gap. In *Theory and Practice in Distributed Systems*, pages 121–132. Springer-Verlag, 1995.

[16] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press / Addison-Wesley, 1993.

[17] R. Jiménez-Peris, M. Patiño Martínez, S. Arévalo, and F. J. Ballesteros. Translib: An ada 95 object oriented framework for building dependable applications. *International Journal of Computer Systems: Science & Engineering*, 15(1):113–125, 2000.

[18] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, November 1998.

[19] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, November 1998.

[20] J. Kienzle. *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming.* Phd, EPFL, 2001.

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[22] M. C. Little and S. K. Shrivastava. Integrating group communication with transactions for implementing persistent replicated objects. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, pages 238–253. Springer-Verlag, 2000. process groups, object replication and atomic transactions.

[23] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[24] H. Miranda. Plataforma de suporte ao desenvolvimento e composição de malhas de protocolos. Master's thesis, Departamento de Informática - Universidade de Lisboa, May 2001.

[25] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, April16–19 2001. IEEE Computer Society.

[26] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cacún, México, September 1999. IEEE.

[27] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Data Base Management System. 11(4), 1986.

[28] Network Systems Research Group. *x-kernel Programmer's Manual (Version 3.3)*, June 1997.

[29] National Institute of Standards and Technology (NIST). Announcing the secure hash standard. FIPS 180-1, U.S. Department of Commerce, April 1995.

[30] National Institute of Standards and Technology (NIST). Data Encryption Standard. FIPS 46-3, U.S. Department of Commerce, October 1999.

[31] M. Patiño Martínez, R. Jiménez-Peris, and S. Arévalo. *Group Transactions*: An integrated approach to transactions and group communication. In *Workshop on Concurrency in Dependable Computing (at 22nd International Conference on Application and Theory of Petri Nets and 2nd International Conference on Application of Concurrency to System Design*, pages 5–15, Newcastle upon Tyne, UK, 2001.

[32] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[33] J. Postel. Internet Control Message Protocol. RFC 792, September 1981.

[34] J. Postel. Internet Protocol. RFC 791, September 1981.

[35] J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[36] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report TR 595, Department of Computing, University of Newcastle upon Tyne, 1997.

[37] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[38] L. Rodrigues, K. Guo, A. Sargento, R. Van Renesse, B. Gladeand, P. Veríssimo, and K. Birman. A transparent light-weight group service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, October 1996.

[39] A. Schiper and M. Raynal. From group communications to transactions in distributed systems. *Communications of the ACM*, 39(4):84—87, 1996.

[40] F. B. Schneider. Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial. 22(4):299–319, 1990.

[41] B. Schneier. *Applied Cryptography Second Edition.* John Wiley & Sons, New York, NY, 1996.

[42] A. Shacham, R. Monsour, R. Pereira, and M. Thomas. Ip payload compression protocol (ipcomp). RFC 2393, December 1998.

[43] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):627–645, July 1987.

[44] W. Stallings. *SNMP, SNMPv2, SNMPv3, RMON 1 and 2*. Addison-Wesley Longman, Inc., 3rd edition, 1998.

[45] W. Richard Stevens. *Unix Network Programming*. Prentice Hall, New York, NY, 1990.

[46] R. Strobl. Dynamic groups in threshold cryptography. Diploma Thesis, Department of Computer Science, ETH Zürich, Winter 2001.

[47] R. J. Stroud and I. S. Welch, editors. *Reference Model and Use Cases for MAFTIA*. Deliverable D1. Project MAFTIA IST-1999-11583, August 2000.

[48] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.

[49] P. Veríssimo and N. Ferreira Neves, editors. *Service and Protocol Architecture for the MAFTIA Middleware*. Deliverable D23. Project MAFTIA IST-1999-11583, January 2001.

[50] P. Veríssimo and L. Rodrigues. A posteriori Agreement for Fault-tolerant Clock Synchronization on Broadcast Networks. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing*, Boston - USA, July 1992. IEEE. INESC AR/65-92.

[51] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C, 2000. IEEE Computer Society.

[52] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *FTCS-25*, pages 499–509, California, USA, 1995.

[53] H. Zimmermann. OSI Reference model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.