Project IST-1999-11583

**Malicious- and Accidental-Fault Tolerance
for Internet Applications**

Complete Specification of APIs and Protocols for the MAFTIA Middleware

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

MAFTIA deliverable D9

Public document

July 31, 2002

Research Report RZ 3439, IBM Research, Zurich Research Laboratory
Technical Report DI/FCUL TR-02-11, 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)*
Lau Cheuk Lung, *University of Lisboa (P)*
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

List of Figures

Abstract

This document describes the complete 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 presents the complete specification of the APIs and protocols for the MAFTIA middleware. The architecture of the middleware subsystem has been described in a previous deliverable (D23), where the various system models, modules and services were introduced. A first specification of the APIs and protocols was given one year ago, in another deliverable (D24). This document intends to explain all the protocols that have been produced until now, and to present the external interfaces to the various components of the architecture. These interfaces can be used by the other partners of the project to explore the functionality provided by the middleware in the implementation of the other subsystems of the MAFTIA architecture. The internal interfaces, i.e., the APIs used for the exchange of data among the protocols of the middleware, is only briefly described since this interface is being defined as the middleware implementation progresses. In the next deliverable we will provide a complete prototype of the middleware (D11: Running prototype of MAFTIA middleware, due to 6 month 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. Each part is 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.

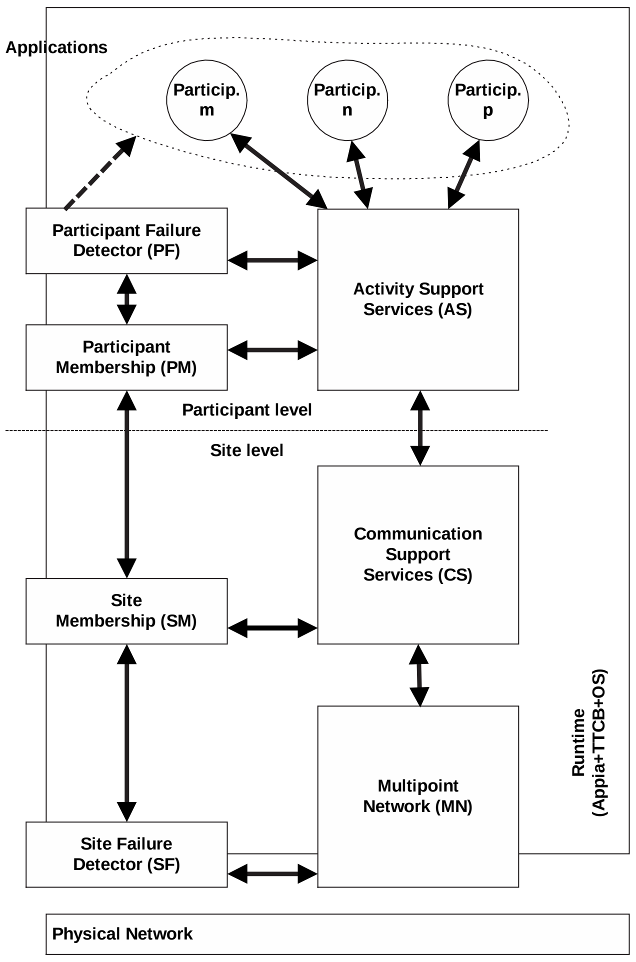

Figure 1.1: Architecture of a MAFTIA Host.

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 secure group communication primitives, such as Byzantine agreement or message multicast, and other core services. The group communication primitives are provided with several reliability and ordering guarantees, such as causal or atomic, and can be defined in terms of several programming models: asynchronous, timed with support from the TTCB, or asynchronous with assistance from the TTCB. At the current time we provide two CS protocol stacks, one that assumes a fully asynchronous network and static, open groups; and another that considers an asynchronous payload network and support from the TTCB, with dynamic open groups.

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 the services, APIs and protocols of a transactional support service (TSS). This service provides multiparty, single level transaction support. The TSS is intended to be used as a building block for intrusion tolerant applications and other AS modules. The architecture of the TSS is derived by applying the general MAFTIA architectural principle of distributing trust to a standard transaction processing architecture. Effectively, the servers implementing the transaction service and optionally the resource managers and resources are replicated.

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.

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 [47, 46]. More information about the framework and the latest updates can be retrieved at the Appia website [1].

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 [55, 56, 53] and the OSI model [75]. 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 [47].

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.

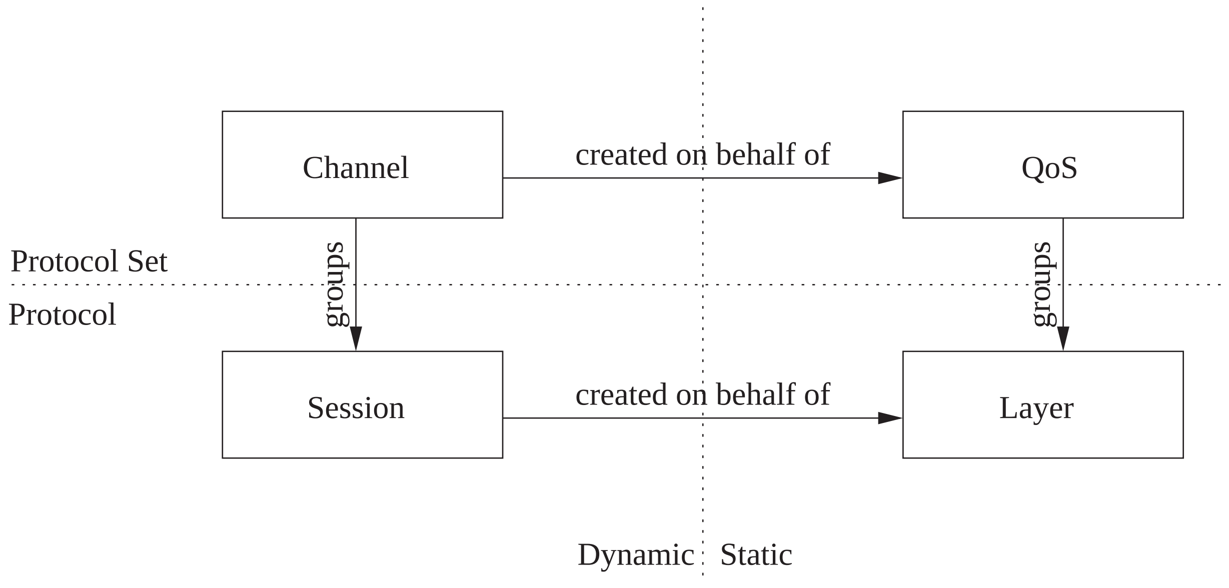

Figure 2.1: Relation between sessions, layers, channels and QoS's.

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.

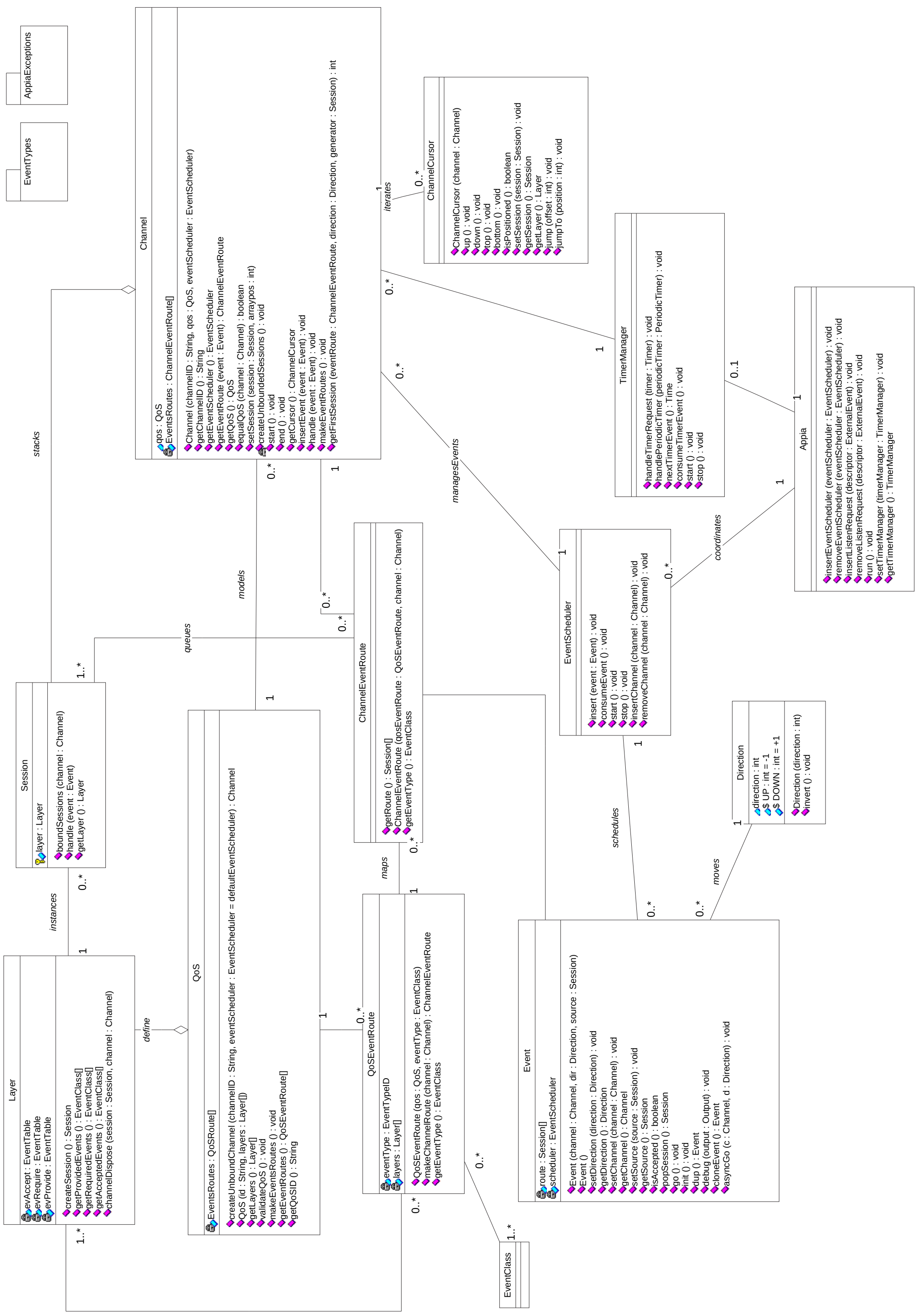
AppiaExceptions
EventTypes
Channel
qos : QoS
EventsRoutes : ChannelEventRoute[]
Channel (channelID : String, qos : QoS, eventScheduler : EventScheduler)
getChannelID () : String
getEventScheduler () : EventScheduler
getEventRoute (event : Event) : ChannelEventRoute
getQoS () : QoS
equalQoS (channel : 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
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
iterates
TimerManager
handleTimerRequest (timer : Timer) : void
handlePeriodicTimer (periodicTimer : PeriodicTimer) : void
nextTimerEvent () : Time
consumeTimerEvent () : void
start () : void
stop () : void
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
stacks
managesEvents
coordinates
models
queues
EventScheduler
insert (event : Event) : void
consumeEvent () : void
start () : void
stop () : void
insertChannel (channel : Channel) : void
removeChannel (channel : Channel) : void
ChannelEventRoute
getRoute () : Session[]
ChannelEventRoute (qosEventRoute : QoSEventRoute, channel : Channel)
getEventType () : EventClass
Session
layer : Layer
boundSessions (channel : Channel)
handle (event : Event)
getLayer () : Layer
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
Direction
direction : int
$ UP : int = -1
$ DOWN : int = +1
Direction (direction : int)
invert () : void
schedules
moves
maps
instances
define
Layer
evAccept : EventTable
evRequire : EventTable
evProvide : EventTable
createSession () : Session
getProvidedEvents () : EventClass[]
getRequiredEvents () : EventClass[]
getAcceptedEvents () : EventClass[]
channelDispose (session : Session, channel : Channel)
QoSEventRoute
eventType : EventTypeID
layers : Layer[]
QoSEventRoute (qos : QoS, eventType : EventClass)
makeChannelRoute (channel : Channel) : ChannelEventRoute
getEventType () : EventClass
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
EventClass
1
0..*
1..*
0..1

Figure 2.2: Appia main UML.

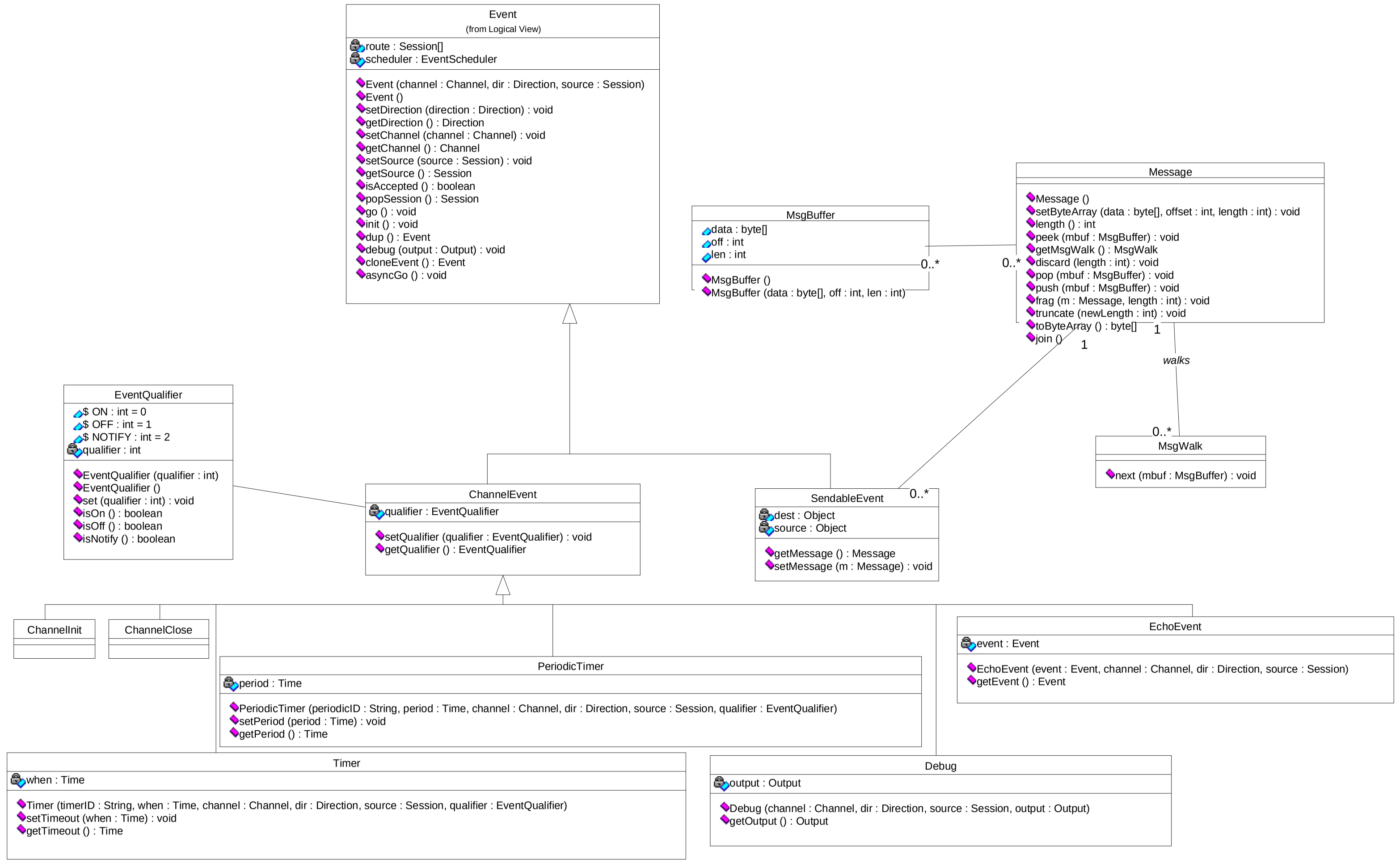
Event
(from Logical View)
route : Session[]
scheduler : EventScheduler
Event (channel : Channel, dir : Direction, source : Session)
Event ()
setDirection (direction : Direction) : void
getDirection () : Direction
setChannel (channel : Channel) : void
getChannel () : Channel
setSource (source : Session) : void
getSource () : Session
isAccepted () : boolean
popSession () : Session
go () : void
init () : void
dup () : Event
debug (output : Output) : void
cloneEvent () : Event
asyncGo () : void
MsgBuffer
data : byte[]
off : int
len : int
MsgBuffer ()
MsgBuffer (data : byte[], off : int, len : int)
0..*
0..*
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 ()
1
1
walks
0..*
MsgWalk
next (mbuf : MsgBuffer) : void
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
ChannelEvent
qualifier : EventQualifier
setQualifier (qualifier : EventQualifier) : void
getQualifier () : EventQualifier
SendableEvent
0..*
dest : Object
source : Object
getMessage () : Message
setMessage (m : Message) : void
ChannelInit
ChannelClose
PeriodicTimer
period : Time
PeriodicTimer (periodicID : String, period : Time, channel : Channel, dir : Direction, source : Session, qualifier : EventQualifier)
setPeriod (period : Time) : void
getPeriod () : Time
EchoEvent
event : Event
EchoEvent (event : Event, channel : Channel, dir : Direction, source : Session)
getEvent () : Event
Timer
when : Time
Timer (timerID : String, when : Time, channel : Channel, dir : Direction, source : Session, qualifier : EventQualifier)
setTimeout (when : Time) : void
getTimeout () : Time
Debug
output : Output
Debug (channel : Channel, dir : Direction, source : Session, output : Output)
getOutput () : Output

Figure 2.3: Appia predefined events UML.

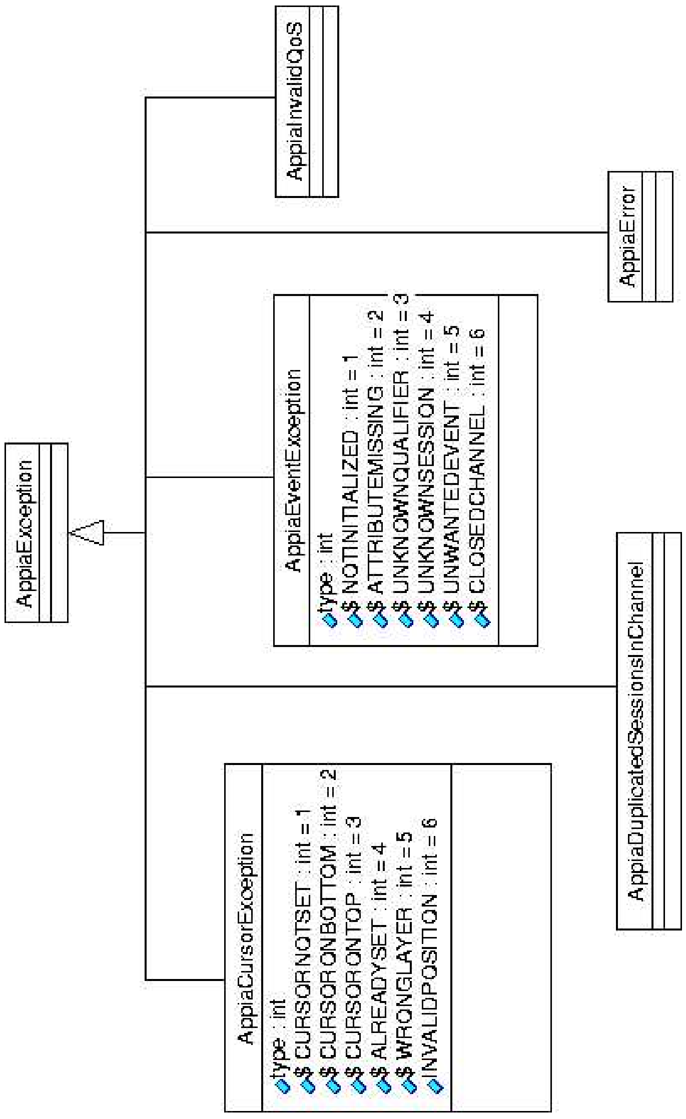
AppiaException
AppiaInvalidQoS
AppiaError
AppiaEventException
type : int
$ NOTINITIALIZED : int = 1
$ ATTRIBUTEMISSING : int = 2
$ UNKNOWNQUALIFIER : int = 3
$ UNKNOWNSESSION : int = 4
$ UNWANTEDEVENT : int = 5
$ CLOSEDCHANNEL : int = 6
AppiaDuplicatedSessionsInChannel
AppiaCursorException
type : int
$ 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 [45].

```
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 [49]. 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 [69, 71]. The implementation of a COTS-based prototype of the TTCB was described in [27, 26]. 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 and local TTCB failure detection.

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

In relation to the TTCB interface, two assumptions are made:

- 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.

On approach (1), security of entity-TTCB communication 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.* This channel is obtained calling the *local authentication service* (Section 2.2.3.1) 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 [26].

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.* On the other hand, a multiparty service like the *agreement service* provides consistent results to all entities involved, and the results are obtained deterministically from the inputs. However, from the point of view of an entity, the correctness of a result may depend on assumptions about majorities of correct entities.

Relatively to timeliness, an example can illustrate approach (2). 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 [70].

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 [69] 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.

Security related services	
Local authentication	For an entity to authenticate the TTCB and establish a secure channel with it.
Trusted block agreement	Achieves agreement on a small, fixed size, data block.
Trusted random numbers	Generates trustworthy random numbers.
Time services	
Trusted timely execution	Executes operations securely and within a certain interval of time.
Trusted duration measurement	Measures the duration of an operation execution.
Trusted timing failure detection	Checks if an operation is executed in a time interval.
Trusted absolute timestamping	Provides globally meaningful timestamps.

Figure 2.5: TTCB Services

The TTCB services are summarized in Figure 2.5. The implementation of the security-related services is presented in [27, 26]. The design and implementation of the time-related services can be found in [70, 19, 20]. The following subsections describe the TTCB services and their APIs. Figure 2.6 shows the meaning of some common API parameters.

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

Figure 2.6: Common TTCB API parameters

2.2.3 Security Related Services

2.2.3.1 Local Authentication Service

The purpose of this service is to allow the entity to authenticate and establish a *secure channel* with a local TTCB. The need for this service derives from the fact that, in general, the communication path between the entity and the local TTCB is not trustworthy. For instance, that communication is probably made through the operating system that may be corrupted and behave maliciously [6]. We assume that the entity–local TTCB communication can be subject to passive and active attacks [44]. A call to the TTCB is composed of two messages, a request and a reply, that can be read, modified, reordered, deleted, and replayed.

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 [44]:

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 keys does *not* allow either (1) a passive adversary to compromise future keys, or (2) impersonation by an active adversary [7].

[6]If the entity is a process or thread, a malicious OS is able to attack not only the entity-TTCB communication but also the entity itself. In these situations, protecting the communication does not add to the application security although, in practice, it prevents some attacks. However, it makes sense to protect the communication if the entities are protected from the OS, e.g., if they are inside a SmartCard, or if they use code protection mechanisms [61, 37].

[7]A passive adversary "attempts to defeat a cryptographic technique by simply recording data and

Every local TTCB has an asymmetric key pair. We assume that correct entities can obtain a correct copy of the local TTCB public key K_u.

A simple protocol with properties SK1 through SK4 can be implemented with two messages, i.e., a single function call. Figure 2.7 shows the protocol. A proof sketch that the protocol verifies SK1 through SK4 can be found in [26].

		Action	**Description**
1	P $\rightarrow$ T	$\langle E_u(K_{et}, X_e)\rangle$	The entity sends the TTCB the new key K_{et} and a challenge X_e, both encrypted with the local TTCB public key K_u
2	T $\rightarrow$ P	$\langle S_r(X_e)\rangle$	TTCB sends the entity the signature of the challenge obtained with its private key K_r

Figure 2.7: Local Authentication Service protocol

The shared key K_{et} has to be generated by the entity, not by the TTCB. We would desire it to be the other way around, but the only key they share initially is the local TTCB public key, that can be used by the entity to protect information that can be read only by the local TTCB (that has the corresponding private key) but not the contrary. K_{et} has to be generated by the entity in such a way that a malicious OS cannot guess or disclose it. The generation of a random key requires sources of randomness (timing between key hits and interrupts, mouse position, etc.), sources that in mainstream computers are controlled by the OS. This means that when an entity gets allegedly random data from those sources, it may get either data given or known by a potentially malicious OS. Therefore, there is the possibility of a malicious OS being able to guess the random data that will be used by the entity to generate the key, and consequently, the key itself. This problem is hard to solve, however, a set of practical criteria can help to mitigate it:

- the entity should use as much as possible sources of random data not controlled by the OS.
- The entity should use as many different sources of random data as possible. Even if an intruder manages to corrupt the OS, it will probably not be able to corrupt its code in many different places and in such a synchronized way, so that it may guess the random number.
- The entity must use a *strong mixing function*, i.e., a function that produces an output whose bits are uncorrelated to the input bits [33]. An example is a hash function such as MD4 or MD5.

thereafter analyzing it (e.g., in key establishment, to determine the key). An active attack involves an adversary who modifies or injects messages." [44]

For similar reasons, the protocol challenge, X_e, has to be generated by the entity using the same approach.

The protocol is implemented in the TTCB API as a single call with the following syntax:

```
(eid, chlg_sign) TTCB_localAuthentication(key, protection, challenge);
```

The input parameters are the key, the communication protection to be used, and the challenge. The output parameters are the entity identification –`eid`– used to identify the entity in the subsequent calls, and the signature of the challenge.

2.2.3.2 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 TTCB_getRandom();
```

2.2.3.3 Trusted Block Agreement Service

The *trusted block agreement service* (or agreement service for short) achieves agreement between a set of distributed entities on a "small" fixed size block of data. 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 [34]); it can be lightweight since it achieves agreement on a small amount of data.

The Agreement Service is formally defined in terms of the functions `TTCB_propose`, `TTCB_decide` and `decision`. An entity *proposes a value* when it calls `TTCB_propose`.

An entity *decides a result* when it calls `TTCB_decide` and receives back a result. The function `decision` calculates the result in terms of the inputs of the service. Formally, the Agreement Service is defined by the following properties:

AS1 Termination. Every correct entity eventually decides a result.

AS2 Integrity. Every correct entity decides at most one result.

AS3 Agreement. If a correct entity decides `result`, then all correct entities eventually decide `result`.

AS4 Validity. If a correct entity decides `result` then `result` is obtained applying the function `decision` to the values proposed.

AS5 Timeliness. Given an instant `tstart` and a known constant $T_{agreement}$, a process can decide by `tstart+T`$_{agreement}$.

The TTCB is a timely component in a payload system with uncertain timeliness. Therefore, the Timeliness property is valid only at the TTCB interface. An entity can only decide with the timeliness the payload system permits.

The interface of the service has two functions: an entity calls `TTCB_propose` to propose its value and `TTCB_decide` to try to decide a result (`TTCB_decide` is non-blocking and returns an error if the agreement did not terminate).

```
out TTCB_propose(eid, elist, tstart, decision, value);
result TTCB_decide(eid, tag);
```

An agreement is uniquely identified by three parameters: `elist` (the list of entities involved in the agreement), `tstart` (a timestamp), and `decision` (a constant identifying the decision function). The service terminates at most $T_{agreement}$ after it "starts", i.e., after either: (1) the last entity in `elist` proposed or (2) after `tstart`, which of the two happens first. That shows the meaning of `tstart`: it is the instant at which an agreement "starts" despite the number of entities in `elist` that proposed. If the TTCB receives a proposal after `tstart` it returns an error.

The other parameters of `TTCB_propose` are: `eid` is the unique identification of an entity before the TTCB, obtained using the Local Authentication Service; `value` is the block the entity proposes; `out` is a structure with two fields, `error`, an error code and `tag`, an unique identifier of the agreement before a local TTCB. An entity calls `TTCB_decide` with the `tag` that identifies the agreement that it wants to decide. `result` is a record with four fields: (1) `error`, an error code; (2) `value`, the value decided; (3) `proposed-ok`, a mask with one bit per entity in `elist`, where each bit indicates if the corresponding entity proposed the value that was decided; (4) `proposed-any`, a similar mask that indicates which entities proposed any value. Some `decision` functions currently available are:

- `TTCB_TBA_RMULTICAST`. Returns the value proposed by the first entity in `elist` (therefore the service works as a reliable multicast) and the two masks.
- `TTCB_TBA_MAJORITY`. This function returns the most proposed value and the same two masks.
- `TTCB_TBA_TEQUALITY`. Compares the values proposed. The entities that proposed the value most proposed decide the result with the masks. The others receive an error.
- `TTCB_TBA_KEY`. Used to establish shared symmetric keys between sets of entities. Returns a symmetric key to the entities involved.

The motivation for the agreement 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. Practical examples of the utility of this service are the BRM and Membership protocols given later.

2.2.4 Time Related Services

2.2.4.1 Trusted Absolute Timestamping Service

Every local TTCB has an internal clock which 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 TTCB_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:

TDM 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.

The service is used calling the functions:

```
tag TTCB_startMeasurement(start_ev);
duration TTCB_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):

TTE 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 TTCB_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 latter 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:

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 TTCB_startLocal(start_ev, spec, handler);
faulty TTCB_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, TTCB_startDistributed), sends the message, the receiver receives the message, executes the remote operation, and tells the TTCB that it is delivered TTCB_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 TTCB_startDistributed(start_ev, spec, mid, elist, handler);
deliv_ev TTCB_delivDistributed(mid, tag);
list_info TTCB_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 TTCB_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 TTCB_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 APIs

This section gives some TTCB APIs that do not correspond to a service.

Each local TTCB has an asynchronous key pair used for authentication and key establishment. A conventional way to distributed public 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 TTCB_getLocalPublicKey that simply returns the local TTCB public key (see below).

```
key TTCB_getLocalPublicKey();
```

A local TTCB identification is returned by the following function:

```
id TTCB_getLocalTTCBid();
```

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 TTCB_crashedLocalTTCBid(id);
out TTCB_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) [55] 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 [53] 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 [56], 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 for details, consult for instance the book by Stevens [67].

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 [65], 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.

Currently most of the IPSec implementations do 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.

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 [54]. 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 [14, 15, 16]). 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 [18]. 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 [5].

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 [30]. 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 [66].

3.1.6 Appia API to Multipoint Network

As an example, we provide the internal interface of the MN, which is based on Appia classes, methods and events. The current API hides the socket interface to the protocols UDP and IP Multicast. Figures 3.1 and 3.2 display these APIs. UdpSimpleSession and UdpCompleteSession classes are the Session subclasses for UdpSimple and UdpComplete protocols, respectively. Concurrently to these session, a reader class, running on another thread listens the sockets for datagrams. Synchronization with Appia is made using async events: SendableEvents are encapsulated in a UdpAsyncEvent and asynchronously inserted in the channel. When the event is delivered, the session extracts the carried SendableEvent and puts it in the channel.

For UDP, the InetWithPort class enriches Java InetAddress class with an integer

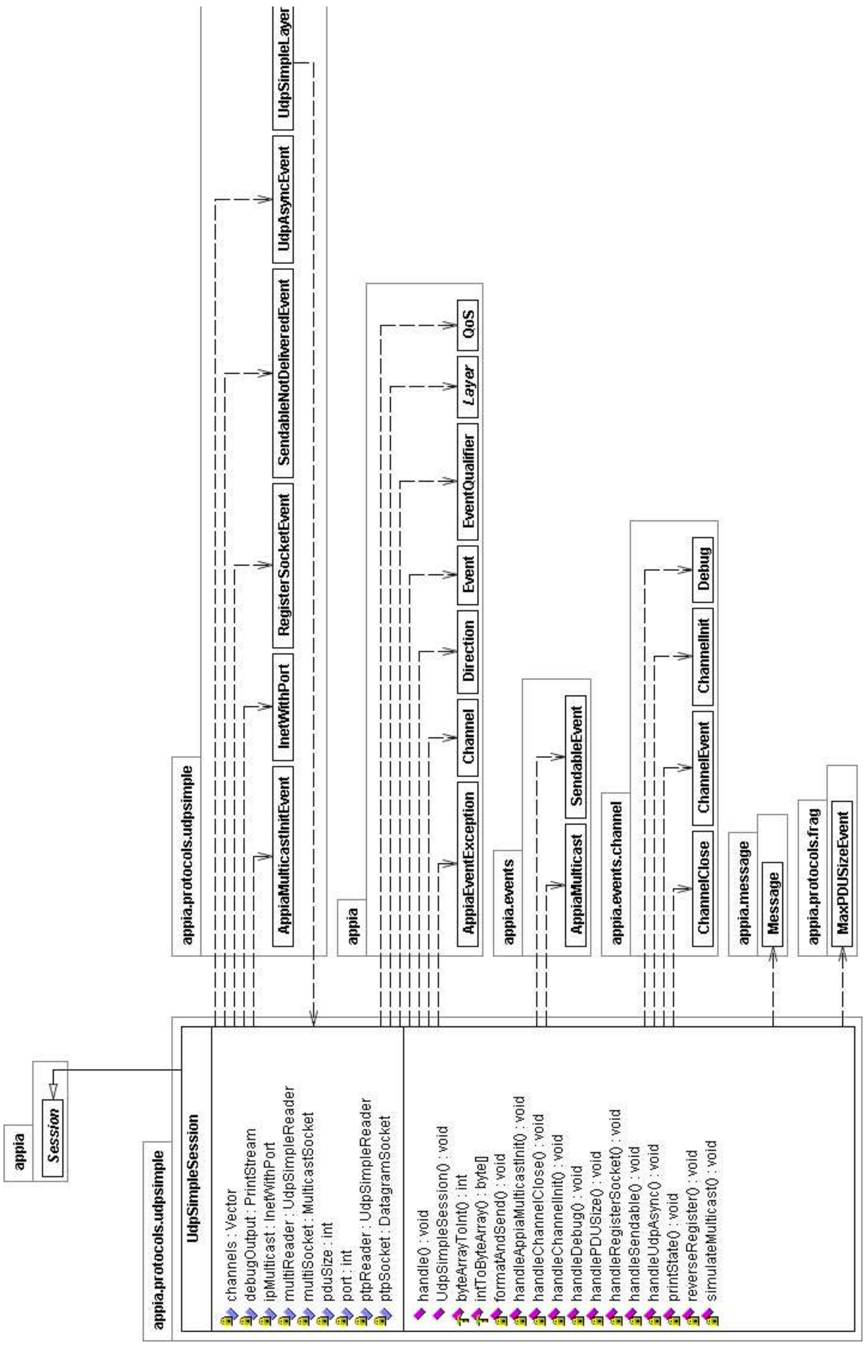

Figure 3.1: udpsimple API.

port number. It intends to be a possible instance for the source and dest attributes of SendableEvents of Appia. The AppiaMulticastInitEvent class is the event that must be received by UdpCompleteSession in order to initialize multicast communication. It carries the multicast address until the udpcomplete layer, notifies the UdpCompleteSession of the

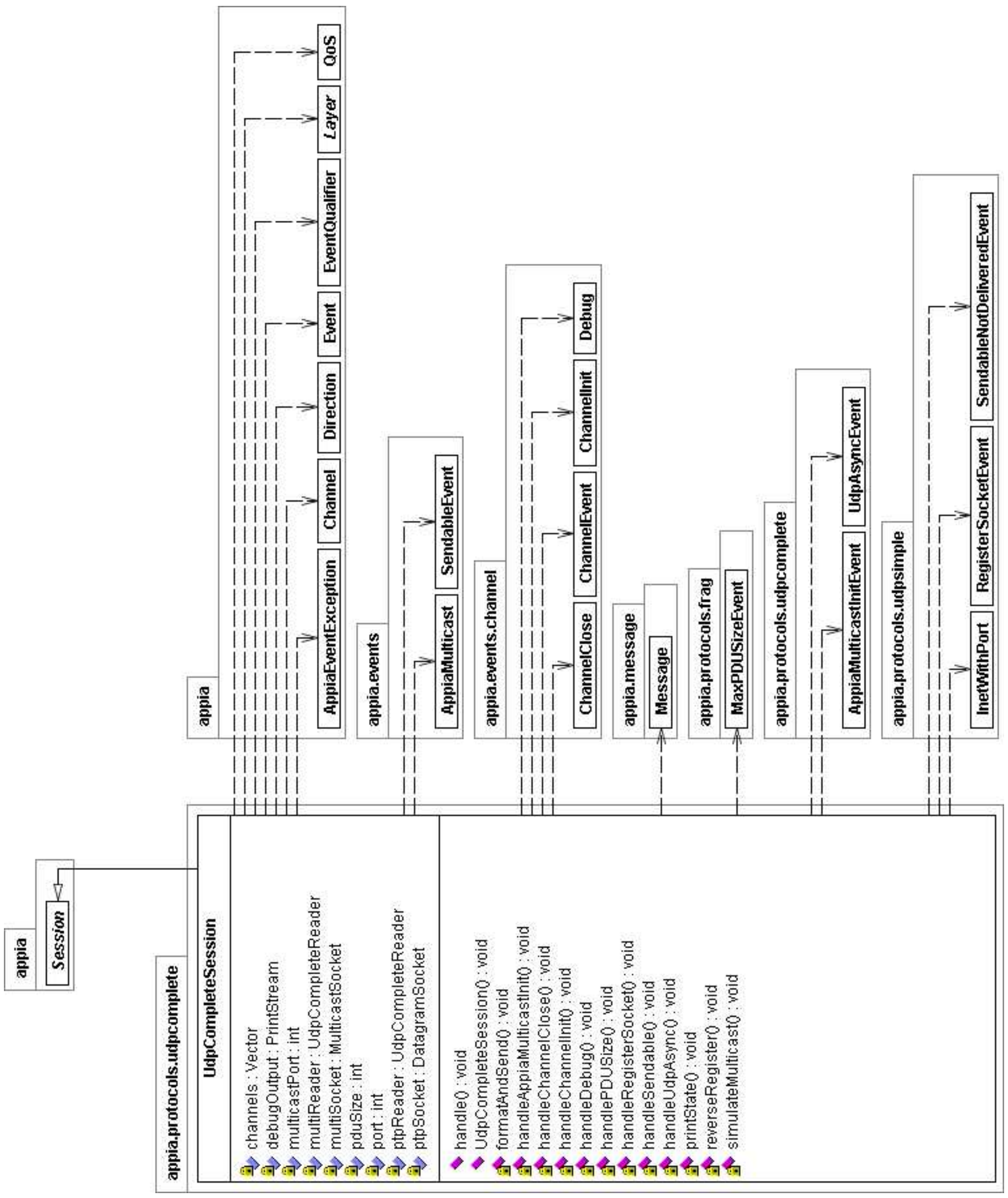

Figure 3.2: udpcomplete API.

multicast address (IP Multicast) to be used, and also specifies if local multicast messages are supposed to be forwarded. The events handled by Appia are SendableEvent for UDP communication, and AppiaMulticast for IP Multicast. When a Appia event is created, the programmer defines which event he needs, defining either InetWithPort for the UDP protocol or a IP Multicast address (in AppiaMulticastEvent) according to the IP Multicast protocol.

3.2 Communication Services

In this section we concentrate on *group communication services* in 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 describe here 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* [71]. 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, where dynamic groups present a host of new complications. Certainly provision must be made for the whole issue of hosts joining, leaving and being excluded from a group. In addition, protocols must be able to cope with the group changing *during* protocol execution. There are also often 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.

Finally, there is the question of within which system model we are working: purely asynchronous or asynchronous with assistance from a TTCB. Using the services provided by the TTCB allows much more efficient communications protocols and also permits fairly direct protocols for changing the group itself (at the cost of the additional infrastructure supplying the TTCB). Purely asychronous protocols usually use more sophisticated cryptographic techniques and sometimes more communication rounds, but work in a more standard network configuration.

In this section we skirt this whole issue of group and system model by working within the context of a fixed group and providing APIs which work in general asynchronous networks and, in certain cases, in the TTCB-enhanced situation as well. (Protocols corresponding to these APIs in the pure asynchronous case are shown below in Section 5.2.1, while those which use a TTCB are in Section 5.2.2.) The unchanging group will be specified by a fixed group identifier String groupID and will imply a fixed (known) list of participants.

Regardless of which 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 that all honest parties will deliver their messages in the same order (*atomic broadcast*) and perhaps, additionally, that the message will be kept in encrypted form until it is delivered (*secure causal 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 do not treat *service replication* explicitly in this section, as it is basically an application-layer task. It can usually be implemented quite simply, however, out of the broadcast and agreement techniques described here.

MAFTIA middleware provides three types of communications primitives in this context: agreement, stream, and sequenced broadcast. The latter two are both examples of what we just described as *broadcast*, divided into the cases of long-lived protocols which act like communications channels for the group, and one-shot protocols which exist merely to deliver a single message from a single sender. We proceed with detailed descriptions of the APIs and semantics of these primitives.

Agreement. We begin with *agreement 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 agreement 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.

MAFTIA middleware provides implementations for the following agreement protocols in the purely asynchronous system model:

binary Byzantine agreement – BinaryAgreement
validated binary Byzantine agreement – ValidatedAgreement
validated multi-valued Byzantine agreement – ArrayAgreement

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

```
Agreement agreement = 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 = agreement.negotiate(value);
```

Here both the answer and the value are objects of a class which extends the Negotiable marker interface in a way useful for the particular protocol, such as containing a boolean value for BinaryAgreement, a boolean value, byte[] proof and a BinaryValidator for ValidatedAgreement, *etc.*; see below for details.

The protocol can also be called without blocking, as for example in the code fragment

```
agreement.propose(value); // returns immediately
while(true) {
   if (agreement.canDecide()) {
      answer = agreement.decisionNegotiable();
      break;
   }
   else
      do something else interesting
}
```

The methods negotiate, propose, canDecide and decisionNegotiable, with the above semantics, arguments and return types, automatically exist for all agreement protocols by virtue of the fact that they all extend the abstract Agreement class. Summarizing, then, this invocation style for agreement protocols is carried by the following publicly-accessible methods of the class Agreement:

```
public abstract class Agreement {
   public Agreement(String protocolID, String groupID);
   public Negotiable negotiate(Negotiable value);
   public void propose(Negotiable value);
   public boolean canDecide();
   public Negotiable decisionNegotiable();
   public void abort();
}
```

Here we have included one more method abort(), which can be used to terminate a running protocol instance, but which leaves that instance and the corresponding instances run by the other protocol participants in an indeterminate state.

We now specify the the specific content of the Negotiable objects needed for each particular agreement protocol, as well as other, more direct method calls for using these particular protocols.

1. The Negotiable for a BinaryAgreement must be a BinaryNegotiable, which is of the form

```
public class BinaryNegotiable implements Negotiable {
   public BinaryNegotiable(boolean value);
   boolean value;
   public boolean getValue();
}
```

In addition, BinaryAgreement overloads the methods propose and negotiate with versions taking a simple boolean value and also provides a method decision() which returns the boolean contents of the BinaryNegotiable decisionNegotiable(); the semantics of these methods are obvious. Hence the full class description of BinaryAgreement includes the additional methods

```
public class BinaryAgreement extends Agreement {
   public boolean negotiate(boolean value);
   public void propose(boolean value);
   public boolean decision();
}
```

2. ValidatedAgreement requires a ValidatedNegotiable, of the form

```
public class ValidatedNegotiable extends BinaryNegotiable {
   public ValidatedNegotiable(boolean value, byte[] proof,
      BinaryValidator validator);
   byte[] proof;
   public byte[] getProof();
   BinaryValidator validator;
}
public abstract class BinaryValidator {
   public abstract boolean isValid(boolean value, byte[] proof, int offset,
      int length);
   public boolean isValid(boolean value, byte[] proof);
}
```

The application programmer must create a class which extends BinaryValidator, providing a body for the isValid(boolean value, byte[] proof, int offset, int length) method that can determine the validity of a ValidatedNegotiable's value given data lying in a byte[] proof starting at location int offset and running for int length bytes, perhaps using other identifying data which would therefore also be in the new class definition. (The other signature of isValid uses the entire argument proof as its validating data.) The field validator will then be a handle to an instance of this new class, and well-formed instances vn of the ValidatedNegotiable class (such as valid propose values in a ValidatedAgreement instance or outputs of ValidatedAgreement.decisionNegotiable()) will satisfy the consistency condition that

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

Note that the ValidatedNegotiable returned by decisionNegotiable() and negotiate(Negotiable value) contains a proof which is not necessarily that which was originally proposed;

it is, however, some data which proves the validity of decided value (perhaps coming from another party participating in the protocol).

In addition, we again use method-overloading and provide additional methods in order to allow an interface which avoids the usage of Negotiable objects, as follows:

```
public class ValidatedAgreement extends Agreement {
   public ValidatedAgreement(String protocolID);
   public ValidatedAgreement(String protocolID, boolean bias);
   public boolean negotiate(boolean value, byte[] proof, BinaryValidator
      validator);
   public void propose(boolean value, byte[] proof, BinaryValidator
      validator);
   public boolean decision();
   public byte[] getProof();
}
```

Here we show the two constructors for this class, one of which functions as usual, the other of which instead implements *biased* validated binary Byzantine agreement, wherein the protocol will always decide for the given bias value if any honest participant proposes that value. The semantics of the next two methods are exactly analogous to their Negotiable versions, while getProof() gives a way to access directly the validating data for a final decision of a completed ValidatedAgreement instance.

3. ArrayAgreement uses a ArrayNegotiable, of the form

```
public class ArrayNegotiable implements Negotiable {
   byte[] value;
   public byte[] getValue();
   ArrayValidator validator;
}
public abstract class ArrayValidator {
   public abstract boolean isValid(byte[] value, int offset, int length);
   public boolean isValid(byte[] value);
}
```

Note that in ArrayNegotiable, there is no need for a separate proof field, as any data required for validation can simply be part of the value array. Of course, 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 ArrayNegotiable.

The Negotiable-less interface of ArrayNegotiable is via the additional methods

```
public class ArrayAgreement extends Agreement {
   public byte[] negotiate(byte[] value, ArrayValidator validator);
   public void propose(byte[] value, ArrayValidator validator);
   public byte[] decision();
}
```

Broadcast. We move on to *sequenced broadcast protocols*, implemented by extensions of the class Broadcast and hence referred to sometimes merely as *broadcast protocols* (but see the discussion of *streams*, below, for some very similar names). These are short-lived, one-time protocols (just as the *agreement* protocols were), 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 protocolID. MAFTIA middleware provides two variants of this type of broadcast: *reliable broadcast*, 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*, which guarantees the uniqueness of delivered messages but not that all honest parties actually deliver all messages. Consistent broadcast is extended also to *verifiable consistent broadcast*, in a manner described below.

The sequenced broadcast protocols presented here all valid in the purely asynchronous system model. In addition, *reliable broadcast* is also available in the TTCB-enabled situation; the corresponding protocol is called (Byzantine) reliable multicast in Section 5.2.2, below.

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

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

where XYZ is either ReliableBroadcast, ConsistentBroadcast or VerifiableConsistentBroadcast, protocolID and groupID are Strings and sender is an int identifying the intended sender within the context of the specified group.

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. The byte[] payload would then be sent by the call

```
broadcast.send(payload);
```

All parties can receive a message with the call

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

which will block until the expected message is delivered. In case the application should not remain blocked while waiting for delivery, the method canReceive() can be called, as follows:

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

For completeness, we also give the public class skeleton of the abstract parent class for sequenced broadcast protocols.

```
public abstract class Broadcast {
   public Broadcast(String protocolID, String groupID, int sender);
   public int getSender();
   public void send(byte[] m);
   public byte[] receive();
   public boolean canReceive();
   public void abort();
}
```

Here, the method abort() attempts to terminate a running protocol instance, leaving it and the corresponding instances run by other parties in an indeterminate state. The method getSender(), as one might expect, returns the identity of the declared sender for that instance.

There is no protocol-specific change of, or additions to, the semantics for ReliableBroadcast or ConsistentBroadcast: both use exactly the same API as given by the parent class Broadcast – they differ only in the guarantees of service they provide. However, VerifiableConsistentBroadcast takes advantage of the fact that the underlying consistent broadcast protocol is in fact an example of a *verifiable* protocol. What this means is that a party who has delivered the payload message can produce a single protocol message that allows any other party to deliver the payload and terminate the broadcast as well. Such a message is called the *closing message* of a broadcast. MAFTIA middleware provides verifiable consistent broadcast in the class VerifiableConsistentBroadcast that extends ConsistentBroadcast. This is a virtual protocol on top of consistent broadcast that requires no additional communication with other parties and uses the following interface:

```
public class VerifiableConsistentBroadcast extends ConsistentBroadcast {
   public byte[] getClosing();
   public void deliverClosing(MessageDigest hash, byte[] v, int offset, int
      length);
   public static byte[] getPayloadFromClosing(byte[] v);
   public static boolean isValidClosing(ThresholdSignature thS,
      MessageDigest hash, String pid, byte[] v, int offset, int length);
}
```

The method getClosing() will return the closing message of a protocol instance which has delivered its payload, while deliverClosingMessage(hash, v, offset, length) will pass to an instance which has not yet delivered its payload a valid closing message sitting in the byte[] v starting at location offset and running for length bytes. The first static method above of VerifiableConsistentBroadcast can be used by applications to extract a the actual payload part of a closing message in a byte[] which was perhaps acquired in some unusual way, i.e., not directly by calling the getClosing() of a finished VerifiableConsistentBroadcast. Similarly, the second static method can determine the validity of a candidate closing message sitting at a certain offset in a byte[] v and running for a certain length. In these extra methods, the MessageDigests and ThresholdSignatures must new instances created with the same parameters as those used by the underlying ConsistentBroadcast.

Stream. We move on to *stream protocols*, a class of broadcast protocols which provide long-lived communications channels 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.) MAFTIA middleware provides four such protocols:

atomic broadcast – AtomicStream
secure causal atomic broadcast – SecureAtomicStream
reliable stream – ReliableStream
consistent stream – ConsistentStream

Atomic broadcast instances simply implement a communication stream upon which group members can transmit messages with the guarantee that all honest parties will receive the messages out 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 [60], 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 remaining two streams, reliable and consistent, are of a somewhat different nature: they are made up of many repeated instances of the corresponding broadcast protocol, aggregated into streams. Thus the overhead of keeping track of which sender is associated with which instance of the broadcast protocol, and of starting another such broadcast instance when one terminates because a message is received, is handled automatically, at the cost of some loss of detailed control.

All of the streams presented here work in the simple asynchronous model. When a TTCB is available, a different protocol can be used to achieve atomic broadcast streams; this is the (Byzantine) atomic multicast protocol of Section 5.2.2.

Note that the names "atomic broadcast" and "secure atomic broadcast" are standard in the literature and thus we use them, despite the confusion to which this could possibly lead with the sequenced broadcast protocols described above; the class names, at least, will make it clear that we are speaking of *streams* and not *broadcast* protocols. Since the idea of aggregated broadcast streams is not well-known, we are instead free to use the new names "reliable stream" and "consistent stream", more consistently with naming conventions of this document.

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);
```

As usual, protocolID and groupID are Strings with application-specific meaning. A byte[] payload may be broadcast on a given stream by the call

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

Here payload will be the message itself for all protocols other than secure atomic broadcast, in which special case the payload will merely be the ciphertext corresponding to the desired message. This ciphertext can be prepared by calling a special static method in class

SecureAtomicStream as follows

```
payload = SecureAtomicStream.encrypt(protocolID, groupID, publicKey,
    cleartext);
```

Here publicKey is a BigInteger public key for the threshold cryptosystem used in secure causal atomic broadcast and cleartext is a byte[].

Note that sending can block, if insufficient progress is being made on the stream protocol and buffers are full. In order to determine if an attempt to send will block, the method canSend() can be called, as for example:

```
while(true) {
  if (stream.canSend()) {
    stream.send(message);
    break;
  }
  else
    do something else interesting
}
```

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 canReceive() may be called, as in the following code fragment:

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

Every party participating in the protocol must be prepared to call receive() for an arbitrary number of payloads, until the stream closes. If the outputs are not so removed, then the stream will stall and eventually block the parties who are trying to send.

When the application has determined that it is ready to close the stream it may call close(), after which is may no longer send any more messages on that stream. It must however continue to receive() messages until isClosed() is true, or else call waitDone(), which will block until the stream has terminated (implicitly receiving and discarding all necessary messages). Instead of calling close() and waitDone() separately, the application may call also closeWait(), which signals the termination and returns only after the stream has been closed.

Note that protocol instances handle these termination requests as normal stream messages, simply counting the number of such which are received from distinct partners and ending their execution when that number exceeds the possible number of corrupted parties. Thus the stream is guaranteed to close when all honest parties together close() it, and it will *not* close if only the adversary so desires.

Finally, calling abort() provides a way to terminate an agreement instance immediately. The local instance of the protocol is cleaned up, but the state of other parties engaged in the protocol is unspecified.

Summarizing, the public interface of the class Stream is:

```
public abstract class Stream {
   public Stream(String protocolID, String groupID);
   public void send(byte[] m);
   public boolean canSend();
   public byte[] receive();
   public boolean canReceive();
   public void close();
   public boolean isClosed();
   public void waitDone();
   public void closeWait();
   public void abort();
}
```

3.3 Activity Services

This section describes the MAFTIA transactional support activity service. The transactional support activity 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 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 3.3.2 we introduce a high-level view of the MAFTIA transactional support activity service architecture, in Section 3.3.3 we introduce the protocols that are used to implement the MAFTIA transactional support activity service, and in Section 3.3.4 we introduce the components that implement the protocols.

3.3.1 Overview of the Transactional Support Activity Service

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 has 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).

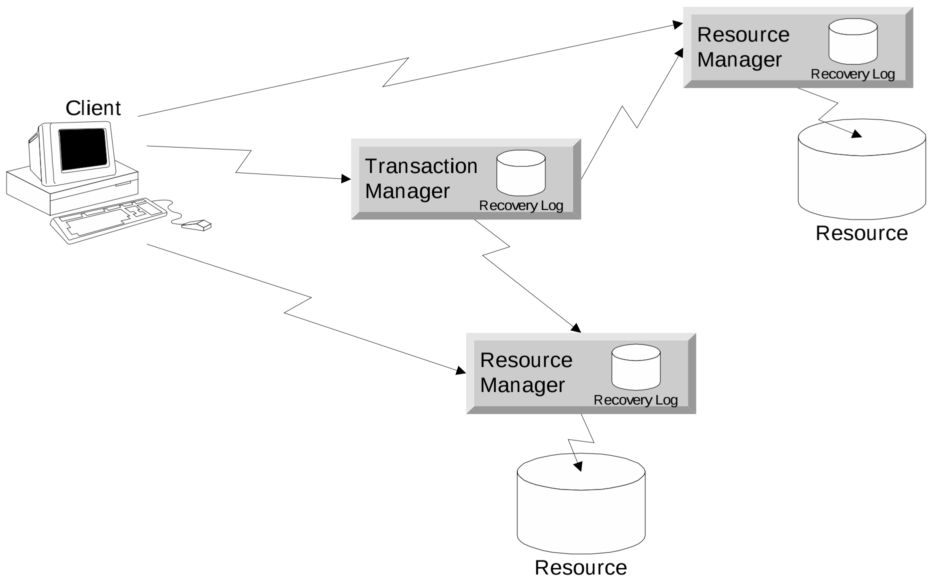

Figure 3.3: High-level View of Transaction Service Architecture

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 the resource managers.

3.3.2 Transaction Support Architecture

A high-level view of the transaction support architecture is shown in figure 3.3. The transaction service architecture is made up of clients, resource managers and transaction managers. In order to provide intrusion-tolerance, the resource managers/resources and transaction managers are 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. There may be multiple resources managed by a each resource manager. 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 can recover after a failure by communicating with their replicas.

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 can recover after a failure by communicating with their replicas.

3.3.3 Transaction Support Protocols

In this section we provide an overview of the protocols used to implement MAFTIA transaction support.

3.3.3.1 Atomic Commitment and Abort Protocols

We bae our atomic commitment and abort protocols on the basic two phase commit (2PC) protocol as described in [48]. We illustrate a successful commit of a transaction in Figure 3.4 which shows the main interactions between a client, transaction manager, resource manager and a resource.

In the figure, a client establishes a transaction by invoking the begin operation on the transaction manager. The transaction manager allocates a unique transaction identifier (tid) and returns it to the client. The client then invokes a method on a resource by sending an invocation request to the resource's resource manager, note that the transaction

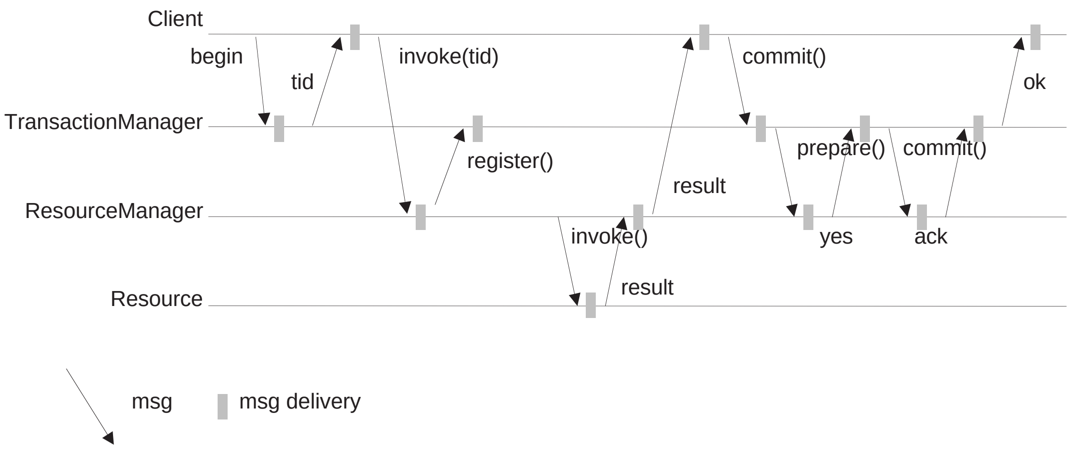

Figure 3.4: Overview of Use of Transaction Service

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 register operation on the transaction manager to register itself for the transaction. After waiting for a reply indicating a successful registration, the resource manager then delegates the invocation to the resource it manages. The resource returns the result to the resource manager, which returns it to the client. Further method invocations proceed in the same manner. Finally, the client asks the transaction manager to attempt a commit by invoking the commit operation on the transaction manager. The transaction manager then invokes the prepare operation on all resource managers participating in the transaction. If all resource managers reply that they are prepared to commit, then the transaction manager tells all participating resource managers to commit their results by invoking the commit operation on the resource manager. After a commit, the resource manager acknowledges the commit to the transaction manager. When all resource managers involved in the transaction have acknowledged a successful commit, then the transaction manager informs the client that the commit was successful.

We do not show in the figure what happens if, in response to a prepare message, a resource manager replies that it cannot commit. In this case, the transaction manager decides upon abort and sends abort messages to all resource managers. The resource managers perform an abort of any changes made within the context of the transaction and acknowledge the abort by sending acknowledgements to the transaction manager. When the transaction manager receives acknowledgements from all resource managers, then the transaction manager informs the client that the commit was unsuccessful.

We extend the basic 2PC by allowing multiple clients to take part in the same transaction. We provide a join operation that allows clients to join an existing transaction, and modify the commit logic so that the wishes of all the participating clients must be taken into accound 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 [74, 58] 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.

3.3.3.2 Distributed Locking Protocols

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. For simplicity we have adopted a pessimistic scheme. A pessimistic 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 unsuccessful and the request must be retried, essentially the client must block until the request is 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.

If locks are acquired and released before the end of a transaction then there is a possiblility that a transaction T might acquire a write lock on a resource, make change and then release the lock before deciding on abort or commit. After the release of the lock, another transaction U might read the value of the resource and take some action. However, transaction T could then abort and require that the resource restore its previous state, prior to the write that took place within the context of the transaction. Now transaction U will be acting upon an incorrect value. To prevent this situation, strict two phase locking scheme (strict 2PL), see [2] for more detail, is used. In strict 2PL, transactions obtain and release locks in two phases. In the growth phase, transactions obtain locks on the resources that are involved in the transaction. In the release phase, locks are only released when the transaction commits or aborts.

3.3.3.3 Recovery Protocols

In a standard distributed transaction system, recovery is required to cope with the failure of transaction managers and resource managers. As will be explained in the protocols section, the MAFTIA distributed transaction system can tolerate a proportion of failures of both transaction managers and resource managers, yet still provide correct service. However, it is desirable that we provide some support for recovery to allow failed managers to be restarted under the control of an outside (trusted) agency in order to prevent creeping intrusion. Since we are dealing with intrusion tolerance we assume that failure is detected through the use of the MAFTIA intrusion detection system and recovery involves the possible rebuild of an entire host where the manager is located rather than a simple restart as in a conventional system.

Since we assume that there are correct replicas always available in the system then we propose a recovery scheme that does not rely upon local durable logs. Instead, we assume that when a manager restarts it simply queries its peers in order to find out its current status. Essentially, we implement a state transfer protocol.

3.3.4 Transaction Support Components

In this section we provide an overview of the components implementing transactional support activity service.

3.3.4.1 Resource Managers

Each resource manager implements the semantics required for participation in the transactional protocols, and recovery semantics. It is implemented using the Appia framework, and is implemented using the following protocols: ResourceManager, LockManager, and RecoveryManager. The intrusion-tolerance properties of these protocls rely upon the VotingMulticast, Invocation and OpenGroup protocols that provide transparent service replication. As we are not addressing service replication in this part of the deliverable we leave the discussion of these protocol to Section 5.3.3. All communication is via events that are sent via Appia channels. Typically, there is one Appia channel for communicating with the transaction manager group, one Appia channel for communicating with other resources in the same resource manager group, and one Appia channel for communicating with clients.

Each resource manager may manage multiple resources that can be created and destroyed dynamically. We assume that each resource has a resource identifier (ObjectID) that is unique to each resource manager (and when resource managers are replicated, unique to each resource manager group).

The resource manager does not share the same application interface as the resources it manages, primarily because it may manage resources of many different types. Instead we provide a generic invocation method (invoke) that can be used to invoke a particular method of a managed resource.

When a resource manager is invoked in the context of a transaction, the resource manager registers itself with the transaction manager by invoking the transaction manager's registerResource method. The transaction manager keeps a list of resources participating within each transaction.

When a resource is invoked for the first time, the resource manager invokes the save method of the resource in order to record its state prior to any changes within the transactional context. Should the transaction be commited then the saved state is thrown away when commit is invoked on the resource, but if abort is invoked then the current state is returned to the save state so that the effect of any changes is undone.

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 is not 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 passing a transaction identifier argument whenever a resource operation is invoked via its resource manager. For example, if a resource has a deposit operation with the signature:

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

Then when the operation is invoked by a client via the resource manager, the transaction identifier passed as a parameter to the resource manager's invoke method in addition to the resource identifier, method name and array of method arguments:

```
invoke(transactionId, objectId,"deposit", new Object[] {amt})
```

During the execution of 2PC, the transaction manager invokes the resource manager prepare method to determine if a resource can be successfully committed. On receiving the event, the resource manager invokes the operation prepare on the encapsulated resource to determine if it is capable of committing. 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 asking all resources registered as taking part in the transaction to prepare, it either asks each resource manager to commit or abort on each resource manager. A resource manager receiving a CommitEvent invokes the commit operation on the resource, this makes permanent any changes made within the transactional context. A resource manager receiving an AbortEvent invokes the abort operation on a resource, forcing 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. For simplicity we have adopted a pessimistic scheme. A pessimistic 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.

We implement a "one writer, multiple reader" concurrency control protocol for each resource managed by a resource manager.

High level API. A resource manager implements the ResourceManager interface while a resource implements the Resource and application interface. For example,

```
class ApplicationResourceMgr implements ResourceManager {


}
```

The full ResourceManager interface is shown below:

```
interface ResourceManager {

    //ask the object if it can commit
    public boolean prepare(Transaction tid, ObjectID oid);
    // force a commit - save changes
    public void commit(Transaction tid, ObjectID oid);
    // abort changes
    public void abort(Transaction tid, ObjectID oid);
    // request a lock on a resource
    public void setLock(Transaction tid, ObjectID oid, int lockType);
    // unlock a resource (not done by client but by transaction
    // manager)
    public boolean unLock(Transaction tid, ObjectID oid);
    // invoke an object method
    public Object invoke(Transaction tid, ObjectID oid, String method,
          Object[] args);
    // initiate recovery
    public void initRecovery();
    // provide resource manager status
    public log getStatus();
}
```

Every resource managed by the resource manager must implement the Resource interface shown below:

```
interface Resource {

    //checkpoint object state
    public void save();
    //ask the object if it can commit
    public boolean prepare(Transaction tid);
    // force a commit - save changes
    public void commit(Transaction tid);
    // abort changes
    public void abort(Transaction tid);
}
```

High level Appia API. We describe the resource manager API in terms of the events that it accepts and provides in Table 3.1. We describe the protocol, using a tabular format that briefly describes each event, and which channel it is sent via. For brevity, we do not show all low level events such as ChannelInit. Nor do we show the subprotocols that are explained in the protocols section of this deliverable.

Event	**Purpose**	**A/P**	**Channel**
InvokeEvent	Invoke an operation on a resource	A	client
PrepareEvent	Can a resource be committed?	A	TM group
AbortEvent	Abort changes made to a resource	A	TM group
CommitEvent	Force a resource to commit changes	A	TM group
RegisterResourceReplyEvent	Confirms registration of resource	A	TM group
ResourceAckReplyEvent	Acknowledges TM requests	P	TM group
RegisterResourceEvent	Requests registration of resource	P	TM group
InvokeReplyEvent	Encapsulates the result of the invocation	P	client
SetLockEvent	Request a lock	A	client
UnlockEvent	Request a lock to be released	A	TM group
LockReplyEvent	Result of a lock or unlock request	P	client
RecoveryEvent	Initiates recovery process	A	trusted source
GetStatusEvent	Request current status	A	RM group
GetStatusReplyEvent	Current resource manager status	P	RM group

RM Resource manager, **TM** Transaction manager, **A/P** Accept/Provide.

Table 3.1: Appia API for ResourceManager protocol

3.3.4.2 Transaction Manager

Each transaction manager implements the semantics required for driving the atomic commitment and abort protocols, as well as recovery semantics. It is implemented using the Appia framework, and is implemented using the following protocols: TransactionManager, and RecoveryManager. As for the Resource Manager, the intrusion-tolerant implementation also relies upon the VotingMulticast, Invocation, and OpenGroup protocols that are described in Section 5.3.3. All communication is via events that are sent via Appia channels. There are typically the following channels: one channel for communicating with the clients, one channel for communicating with other transaction managers, and one channel per resource manager group.

Each transaction manager can support multiple transactions, each transaction has a unique transaction identifier (TID). A client starts a transaction by invoking the transaction manager method begin. The transaction manager generates a unique transaction identifier for the transaction. The TID is returned to the client. This TID is used as a capability and it allows any client possessing it to participate in the transaction. A client that wishes to take part in an active transaction sends the TID as an argument of an invocation of the transaction manager method join. 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 as an argument of an invocation of the transaction manager method commit or abort.

An example of the use of the transaction manager to demarcate a transaction is shown below:

```
// begin the transaction
Transaction myTid = TransactionManager.begin();
// transactional operations
// end the transaction
TransactionManager.commit(myTid);
```

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.

High level API. The full TransactionManager interface is shown below:

```
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, ObjectID oid,
        ResourceManager r);
    // initiate recovery
    public void initRecovery();
    // provide transaction manager
    public log getStatus();
}
```

Transaction Manager Appia API. We describe the Appia API in terms of the events that the Transaction Manager protocol accepts and provides in Table 3.2.

Event	Purpose	A/P	Channel
BeginEvent	Begin a transaction	A	client
JoinEvent	Join a transaction	A	client
CommitEvent	Commit a transaction	A	client
AbortEvent	Abort a transaction	A	client
RegisterResourceEvent	Register a resource	A	RM group
RegisterResourceReplyEvent	Acknowledge registration	P	RM group
PrepareEvent	Can commit?	P	RM group
ResourcePrepareReplyEvent	Acknowledges prepare request	A	RM group
CommitEvent	Force commit	P	RM group
AbortEvent	Force abort	P	RM group
ResourceAckEvent	Acknowledges commit or abort request	A	RM group
RecoveryEvent	Initiates recovery process	A	trusted source
GetStatusEvent	Request current status	A	TM group
GetStatusReplyEvent	Returns the current status	P	TM group
UnlockEvent	Request release of a lock	P	TM group

RM Resource manager, **TM** Transaction manager, **A/P** Accept/Provide.

Table 3.2: Appia API for TransactionManager protocol

3.4 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.

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 *view synchrony* group semantics [4, 3].

Informally, view synchrony provides membership information to participants (or sites) in the form of views and guarantees that all participants (or 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).

MAFTIA middleware implements the group membership API below using a TTCB. There is also an approach envisioned which uses secret resharing to work in the simple asynchronous model. This latter method would again implement the same API below but require some additional functionality and add one more API method, which we now explain.

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, the methods joinGroup, leaveGroup and removeGroup described below change these parameters to $n' = n + 1$ (in the first case) or $n' = n - 1$ (in the latter two cases) but all have $t' = t$.

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 [13]. 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.

3.4.1 Site membership module.

This module offers the following calls:

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

joinSiteGroup makes the site try to join the group with site group id `siteGroupID`. `credential` is data used for joining authorization. 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. 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`.

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.

3.4.2 Participant membership module.

This module offers the following calls:

```
joinGroup(id, participantGroupID, credential);
leaveGroup(id, participantGroupID, credential);
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`. Join authorization is granted or denied based on the `credential` given by the participant. If the group does not exist a new one is created with a single participant (and a corresponding 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.

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`.

4 Runtime Environment Protocols

4.1 Trusted Timely Computing Base

In this section, we describe the protocols that implement the only security related distributed service that is provided by the TTCB, the Agreement Service.

4.1.1 Agreement Service Protocol

The TTCB Agreement Service is implemented using a time-triggered protocol: `TTCB_propose` is called asynchronously, and gives the TTCB data that is stored in tables; then, periodically that data is broadcast to all local TTCBs and, also periodically, data is read from the network and processed.

AN1 Broadcast – The AN has an unreliable packet broadcast primitive

AN2 Integrity – Nodes can detect if packets were corrupted in the network. Corruptions are converted to omission failures

AN3 Omission degree – No more than Od omissions may occur in a given interval of time

AN4 Bounded delay – Any correct packet is received within a maximum delay T_{send} from the send request

AN5 Partition free – The network does not get partitioned

AN6 Broadcast Degree – If a broadcast is received by any local TTCB other than the sender, then it is received by at least Bd local TTCBs

AN7 Confidentiality – The content of network traffic cannot be read by unauthorized users

AN8 Authenticity – Nodes can detect if a packet was broadcast by a correct node

Table 4.1: Abstract Network (AN) properties.

Protocol 1 shows the protocol. It is based on TTCB assumptions that we summarize here for clearness:

- the local TTCBs have clocks synchronized to π;
- the protocol code is executed in real-time (therefore there is a worst case execution time for every section of code);
- every local TTCB communicates with the others exclusively by broadcasting a message with a constant period;
- the network is described by the Abstract Network (AN) model (see Table 4.1).

The protocol uses two tables. The `dataTable` stores all agreements data. Each record has the state of one agreement with the format: `(tag, elist, tstart, decision, vtable)`. All fields have the usual meaning (Section 2.2) except `vtable`, which is a table with the values proposed (one per entity in `elist`). `sendTable` stores data to be broadcast to all local TTCBs. Every record is a proposal with the format: `(elist, tstart, decision, eid, value)`. The agreement is identified by `(elist, tstart, decision)`. `eid` identifies the entity that proposed and `value` is the value proposed.

The protocol has four routines. The *propose routine* is executed when an entity calls the TTCB function `TTCB_propose` (Lines 1-8). The routine begins doing some tests: if the entity already proposed a value for this agreement; if the entity that calls the service is in `elist`; if `tstart` already expired (Line 2). Other tests, are also made but are not represented since they are not related to the algorithm functionality. If the propose is accepted, its data is inserted in sendTable and dataTable, and the `tag` is returned (Lines 4-8). The *broadcast routine* broadcasts data to all local TTCBs every T_s (the period) either if there is data in sendTable or not (Lines 9-13). Every message is broadcasted $Od + 1$ times in order to tolerate omissions in the network (Od is the omission degree). After the broadcast, sendTable is cleaned. The *receive routine* reads and processes messages every T_r (Lines 14-21). Since each message is broadcasted $Od+1$ times, copies of the same message have to be discarded by the function `read` (Line 15). For each message received, the data in each record of sendTable is inserted in dataTable (Lines 16-20). The *decide routine* is executed when an entity calls the function `TTCB_decide`. The routine searches dataTable for the agreement identified by the tag and returns an error if it does not exists. If the instant $tstart + T_{agreement}$ passed or the local TTCB has the values proposed by all entities in `elist`, the result is obtained and returned.

We proved that the protocol implements the Agreement Service and that $T_{agreement}$ can be given by [26]:

$$T_{agreement} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi \qquad (4.1)$$

Protocol 1 Agreement Service internal protocol (at a local TTCB).

```
WHEN ENTITY CALLS TTCB_propose(eid, elist, tstart, decision, value) {propose routine}
if (entity already proposed) or (eid $\notin$ elist) or (clock() $>$ tstart) then
   return error;
insert (elist, tstart, decision, eid, value) in sendTable;
get R $\in$ dataTable : R.elist = elist $\wedge$ R.tstart = tstart $\wedge$ R.decision = decision;
if (R = $\perp$) then
   R := (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
return R.tag;

WHEN CLOCK() = $round_s \times T_s$ {broadcast routine}
repeat
   broadcast(sendTable);
until $Od + 1$ times
sendTable := $\perp$; $round_s := round_s + 1$;

WHEN CLOCK() = $round_r \times T_r$ {receive routine}
while read(M) $\neq$ error do
   for all (elist, tstart, decision, eid, value) $\in$ M.sendTable do
      get R$\in$dataTable : R.elist = elist $\wedge$ R.tstart = tstart $\wedge$ R.decision = decision;
      if (R = $\perp$) then
         R := (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
      insert value in R.vtable;
$round_r := round_r + 1$;

WHEN ENTITY CALLS TTCB_decide(eid, tag) {decide routine}
get $R \in$ dataTable : R.tag = tag;
if (R$\neq\perp$) and [(clock()>R.tstart+$T_{agreement}$) or (all entities proposed a value)] then
   return (calculate result using function R.decision and values in R.vtable);
else
   return error;
```

The constants in the formula have the following meaning: T_s and T_r are respectively the send and receive periods; $WCET_{send}$ and $WCET_{receive}$ are respectively the send and receive routines worst execution times; T_{send} is the maximum communication delay; π is the precision of the clock synchronization algorithm.

4.1.2 Reliable Broadcast Protocol of the Agreement Service

In the Agreement Service protocol in Protocol 1, if a local TTCB crashes during the broadcast, some local TTCBs may receive the message while others may not. Such an inconsistency can lead to different local TTCBs giving different results to one or more agreements. Therefore, informally, when the sender crashes, the broadcast must either deliver the message to all recipients or to none. Such a broadcast is usually called a Reliable Broadcast and this section describes such a protocol. If we replace Lines 10-12 and 15 in Protocol 1 by this protocol, the Agreement Service protocol becomes tolerant to local TTCB crashes. The second condition in Line 24 has also to be substituted by: "all entities *in non-crashed local TTCBs* proposed a value". The complete protocol is shown in [26].

This section presents a time-triggered Timely Reliable Broadcast that tolerates crashes, assumes channel omissions (Abstract Network property AN3), and is lightweight, in the sense that it does not retransmit messages. A timely reliable broadcast is formally defined in terms of two primitives `R-broadcast(M)` and `R-deliver(M)`, where M is the message, that verify the following properties (based on [36]):

Validity. If a correct local TTCB R-broadcasts M then it eventually R-delivers M.

Agreement. If a correct local TTCB R-delivers message M then all correct local TTCBs eventually R-deliver M.

Integrity. For any message M, a correct local TTCB R-delivers M at most once and only if M was R-broadcast by *sender(M)*.

Timeliness. There is a known constant $T_{broadcast}$ such that, if a message is R-broadcast at instant t, then no correct local TTCB R-delivers M after $t + T_{broadcast}$.

The protocol is shown in Protocol 2. The *broadcast routine* is similar to the Agreement Service protocol. Every T_s a message M is broadcasted $Od + 1$ times to tolerate omissions in the channel. The message is broadcasted even if there is no data to be sent. This is important for the protocol to work properly and for the detection of local TTCB crashes (a local TTCB is known to be crashed if a message is not received by its dead-

Protocol 2 Timely reliable broadcast protocol.

```
 1 WHEN CLOCK() = $round_s \times T_s$ {broadcast routine}
 2 sender := my_id(); seq := $round_s$;
 3 M := (sender, seq, higherseqVector, data);
 4 **repeat**
 5   broadcast(M);
 6 **until** $Od + 1$ times
 7 $round_s := round_s + 1$;

 8 WHEN CLOCK() = $round_r \times T_r$ {receive routine}
 9 **while** read(M) $\neq$ error **do**
10   **for all** M-ndlv in notDelivered **do**
11     **if** [(M-ndlv.sender = M.sender) and (M-ndlv.number < M.number)] or (M-ndlv.number <
       M.higherseqVector[M-ndlv.sender]) **then**
12       R-deliver(M-ndlv.data); remove M-ndlv from notDelivered;
13   **if** (higherseqVector[M.sender] > M.number) **then**
14     R-deliver(M.data);
15   **else**
16     put M in notDelivered; higherseqVector[M.sender] := M.number;
17 $round_r := round_r + 1$;
```

line [20]). The message has an header with the sender identifier, a sequence number and the table `higherseqVector`. This table has an entry for every local TTCB that contains, for every other local TTCB, the highest sequence number of a message received from that local TTCB. The *receive routine* starts by reading a message M (Lines 8-17). Copies of messages already received are discarded by the function `read`. For every message received the routine does two things: (1) tests if previously received but not R-delivered messages (stored in `notDelivered`) can be R-delivered (Lines 10-12); (2) tests if M can be R-delivered (Lines 13-16).

Considering AN6, the protocol tolerates Bd local TTCB crashes in a reference interval of time. A message can be R-delivered by a local TTCB when it knows that all other non-crashed local TTCBs will also R-deliver it (Agreement property). A local TTCB can R-deliver a message $M(s,n)$ when it receives (a) $M(s,n+1)$ or (b) $M(s',n')$ with `higherseqVector[s]=n+1` (s is the sender and n the message number). The intuition behind this is: if s crashes during the broadcast of $M(s,n+1)$ but at least one local TTCB receives the message, then at least Bd local TTCBs receive it (AN6) and at most other $Bd-1$ can crash (the protocol tolerates Bd crashes). Therefore, at least one correct local TTCB receives $M(s,n+1)$ and broadcasts $M(s',n')$ with `higherseqVector[s]=n+1` to the other non-crashed local TTCBs. Since messages are broadcast $Od+1$ times, all non-crashed local TTCBs either receive $M(s,n+1)$ or $M(s',n')$ and R-deliver $M(s,n)$. In the protocol, Line 11 tests this condition. However, it considers that any $M(s,n_+)$ with $n_+ > n$ causes $M(s,n)$ to be R-delivered, since the Abstract Network does not guarantee the order of the reception of messages. The same is true for $M(s',n')$ with `higherseqVector[s]` $> n$. Line 13 checks if the message received, $M(s'',n'')$, can be R-delivered immediately.

This is the case if a message from the same sender but with a higher number was received previously, i.e., if `higherseqVector[`s''`]` $> n''$.

We proved these results and also that the protocol R-delivers a message M within $T_{broadcast}$ of `R-broadcast(M)` (the meaning of the constants is the same as before) [26]:

$$T_{broadcast} = 2 \times (WCET_{send} + T_{send} + T_r + WCET_{receive} + T_s) + \pi \tag{4.2}$$

The Agreement Service termination instant is related to $T_{broadcast}$ [26]:

$$T_{agreement} = T_s + T_{broadcast} \tag{4.3}$$

5 Middleware Protocols

5.1 Multipoint Network

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

5.1.1 Internet Protocol

For IP, the two most used protocols are the UDP [53] and TCP [56], which are standard and widely used protocols. Therefore, we will not include a definition of these protocols here.

5.1.2 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.

5.1.3 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 [39] and [40], 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) [42] with Message-Digest Algorithm (MD5) or with Secure Hash Algorithm (SHA-1) [50]. In ESP, IPSec may use also the two above algorithms for authentication or the Data Encryption Standard (DES) [51] in Cipher Block Chaining Mode (CBC), which is applicable to several encryption algorithms and for which a general description is given in [64], 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".

5.1.4 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 [54]:

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 5.1: Summary of ICMP messages.

5.1.5 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 [29].

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 [29] . 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.

5.1.5.1 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 [17], 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.

5.2 Communications Support

In this section we explain how the communications services described in Section 3.2 are realized. We provide two implementations for these services, one that assumes a completely asynchronous model and another that assumes an asynchronous model for the payload system and takes advantage of the support provided by the TTCB.

5.2.1 Asynchronous Communication Protocols

In this section we shall give technical descriptions of how the communications services described in Section 3.2 are realized in the simple asynchronous system model. 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.

As described above in §3.2, the basic group communications primitives provided by MAFTIA middleware are realized by the three classes Agreement, Broadcast and Stream. These classes are extended to form the particular implementations of all individual protocols, while in turn they all extend the class Protocol which carries basic information pertinent to all MAFTIA middleware group communications protocol types.

An overview of these classes and their relationships is shown in Figure 5.1.

This figure shows the **Java** class hierarchy; another important aspect of the overall architecture is the logical dependency of protocols upon one another (partially reflected in the Appia protocol stack as well), which is depicted in Figure 5.2.

We should mention that an important additional piece of infrastructure used in many of these class is provided by the classes ThresholdCoin and ThresholdSignature, which implement threshold cryptographic services of the same names.

Moving now to the algorithms themselves, we first note certain conventions that we will always follow:

1. n and $t < n/3$ are the number of members of the group and the number of permissible corrupted servers, respectively. The groups members will be named $P_1, \ldots, P_n$.
2. All information about the group context is fixed. In particular, we will not explicitly name the group with a groupID.
3. Messages from P_i to P_j are always presumed to be of the form $(\mathsf{protocolID}, i, j, \mathit{payload})$, so that when specifying the message, we will often give only the *payload*; the values of protocolID, i and j are implied by the context.
4. Network messages are also presumed to be MACed, and thus the implicit i and j in such a message cannot be spoofed when both i and j are honest.
5. When a protocol invokes a sub-protocol, it will be often be described as "tagged" with some additional data: this means that the protocolID of the sub-protocol is formed by that of the the ambient protocol with the additional data appended in some unambiguous fashion. Of course, the groupID of the sub-protocol will be identical to that of its parent.
6. Similarly, a threshold coin or other named cryptographic operation will be described as "tagged" with certain data when we wish to indicate that the input of that cryptographic function is to be protocolID, groupID and extra data all concatenated unambiguously.
7. In protocol descriptions, messages are sent between participants, or between the application and the protocol itself, often with some functional label in `this font`, such

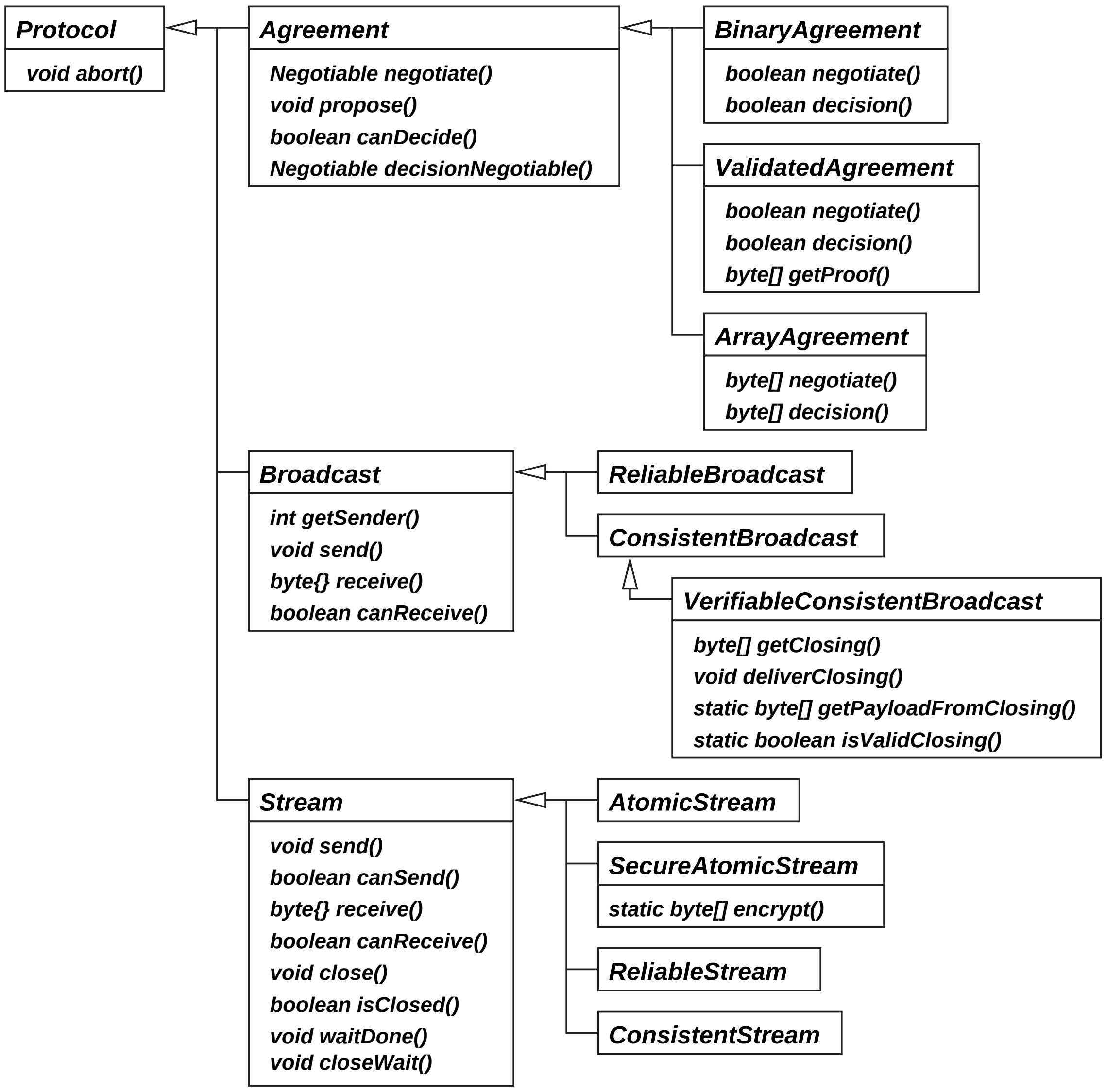

Figure 5.1: Communications services for static groups, class overview

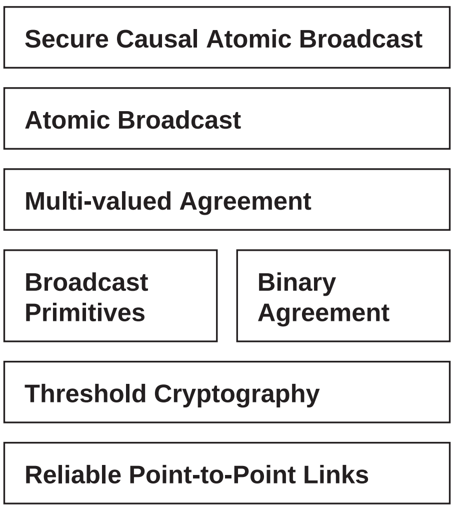

Figure 5.2: Communications services for static groups, logical hierarchy

as “a `pre-process` message” in binary Byzantine agreement. These labels are to be translated into disambiguating numerical values which are unique also among different protocols: for example, there should be no confusion between `deliver` messages in the several protocols which have such messages.

5.2.1.1 Binary Byzantine Agreement

This is the *ABBA* protocol from [10]. It uses an $(n, n-t, t)$ threshold signature scheme $\mathcal{S}$ and an $(n, t+1, t)$ threshold signature scheme $\mathcal{S}_0$, as well as an $(n, n-t, t)$ threshold coin-tossing scheme (see [10] for explanations of these cryptographic parameters). We will let $F(C)$ denote the value of coin with tag C.

Since the protocol is somewhat involved, we first give an overview. For a given instance of the protocol, each party P_i has an initial value $V_i \in \{0,1\}$, and the protocol proceeds in rounds $r = 1, 2, \ldots$ The first round starts with a special pre-processing step:

0. Each party sends its *initial value* to all other parties signed with an $\mathcal{S}_0$-signature share. On receiving $2t+1$ such votes, each party combines the signature shares of the value with the simple majority (i.e., at least $t+1$ votes) to a threshold signature of $\mathcal{S}_0$. This value will be the value used in the first pre-vote.

After that, each round contains four basic steps:

1. Each party casts a *pre-vote* for a value $b \in \{0,1\}$. These pre-votes must be *justified*

by an appropriate $\mathcal{S}$-threshold signature, and must be accompanied by a valid $\mathcal{S}$-signature share on an appropriate message.

2. After collecting $n-t$ valid pre-votes, each party casts a *main-vote* $v \in \{0, 1, \texttt{abstain}\}$. As with pre-votes, these main-votes must be justified by an appropriate $\mathcal{S}$-threshold signature, and must be accompanied by a valid $\mathcal{S}$-signature share on an appropriate message.
3. After collecting $n - t$ valid main-votes, each party examines these votes. If all votes are for a value $b \in \{0, 1\}$, then the party *decides* b, but continues to participate in the protocol for one more round. Otherwise, the party proceeds.
4. The value of coin tagged with r is revealed, which may be used in the next round.

We now proceed with the details of the protocol, shown as Protocol 3.

A crucial part is played here by cryptographic *justifications* of many of the messages, which prove that the sender had in her possession sufficiently many appropriate prior messages from other protocol participants. These justifications are formed as follows:

Pre-vote justification: In round $r = 1$, party P_i's pre-vote is the majority of the pre-processing votes from the pre-processing step. There must be at least $t + 1$ votes for the same value $b \in \{0, 1\}$. For the justification, a party selects $t + 1$ such votes, and combines the accompanying $\mathcal{S}_0$-signature shares to obtain an $\mathcal{S}_0$-threshold signature on the message $(\texttt{pre-process}, b)$. In rounds $r > 1$, a pre-vote for b may be justified in two ways:

- either with an $\mathcal{S}$-threshold signature on the message $(\texttt{pre-vote}, r-1, b)$; we call this a *hard* pre-vote for b;
- or with an $\mathcal{S}$-threshold signature on the message $(\texttt{main-vote}, r-1, \texttt{abstain})$ for the pre-vote $b = F(r - 1)$; we call this a *soft* pre-vote for b.

Intuitively, a hard pre-vote expresses P_i's preference for b based on evidence for preference b in round $r - 1$, whereas a soft pre-vote is just a vote for the value of the coin, based evidence of conflicting votes in round $r-1$. The threshold signatures are obtained from the computations in previous rounds (see below). We assume that the justification indicates whether the pre-vote is hard or soft.

Main-vote justification: A main-vote v in round r is one of the values $\{0, 1, \texttt{abstain}\}$ and, like pre-votes, accompanied by a *justification* as follows:

Protocol 3 Binary Byzantine agreement with *ABBA*

```
1  PRE-PROCESSING:
2  generate an $\mathcal{S}_0$-signature share on the message $(\texttt{pre-process}, V_i)$ and send the message
   $(\texttt{pre-process}, V_i, \textit{signature share})$ to all parties.
3  collect $2t+1$ proper pre-processing messages
4  for round $r = 1, 2, \ldots$ do
5    PRE-VOTE:
6    if $r = 1$ then
7      let $b$ be the simple majority of the received pre-processing votes
8    else
9      select $n - t$ properly justified main votes from round $r - 1$ and let
       $$b = \begin{cases} 0 & \text{if there is a main-vote for 0,} \\ 1 & \text{if there is a main-vote for 1,} \\ F(r-1) & \text{if all main-votes are } \texttt{abstain}. \end{cases}$$
10     produce an $\mathcal{S}$-signature share on the message $(\texttt{pre-vote}, r, b)$
11     produce the corresponding *justification* (see text)
12     send the message $(\texttt{pre-vote}, r, b, \textit{justification}, \textit{signature share})$ to all parties.

13   MAIN-VOTE:
14   collect $n - t$ properly justified round-$r$ pre-vote messages and let
       $$v = \begin{cases} 0 & \text{if there are } n-t \text{ pre-votes for 0,} \\ 1 & \text{if there are } n-t \text{ pre-votes for 1,} \\ \texttt{abstain} & \text{if there are pre-votes for 0 and 1.} \end{cases}$$
15   produce an $\mathcal{S}$-signature share on the message $(\texttt{main-vote}, r, v)$
16   produce the corresponding *justification* (see text)
17   send the message $(\texttt{main-vote}, r, v, \textit{justification}, \textit{signature share})$ to all parties

18   CHECK FOR DECISION:
19   collect $n - t$ properly justified main-votes of round $r$
20   if all main votes are for $b \in \{0, 1\}$ then
21     *decide* for value $b$ but continue one more round up to line 13

22   COMMON COIN:
23   generate a share of the coin with tag $r$
24   send the message $(\texttt{coin}, r, \textit{coin share})$ to all parties
25   collect and combine $n - t$ shares of the coin tagged with $r$
```

- If among the $n - t$ justified round-r pre-votes collected by P_i there is a pre-vote for 0 and a pre-vote for 1, then P_i's main-vote v for round r is `abstain`. The justification for this main-vote consists of the justifications for the two conflicting pre-votes.
- Otherwise, P_i has collected $n - t$ justified pre-votes for some $b \in \{0, 1\}$ in round r, and since each of these comes with a valid $\mathcal{S}$-signature share on the message $(\texttt{pre-vote}, r, b)$, P_i can combine these shares to obtain a valid $\mathcal{S}$-threshold signature on this message. Party P_i's main-vote v in this case is b, and its justification is this threshold signature.

5.2.1.2 Validated Binary Byzantine Agreement

It is only necessary to make two small modifications to the above-described protocol for binary Byzantine agreement in order to extend the functionality with external validations and bias. First, the justifications used in the pre-votes of round 1 are replaced by the proofs submitted with protocol invocation, and their correctness is tested by with the external predicate implemented as the submitted ValidatedNegotiable's BinargyValidator field validator. The logic of the protocol guarantees that either a decision is reached immediately or the validations for 0 and for 1 are seen by all parties in the first two rounds; in either case, the protocol instance can return proof data which validates the value for which it decides.

Second, the protocol can be biased towards a value $b \in \{0, 1\}$ by modifying the use of the threshold coin so that in the first round it always appears to have value b.

5.2.1.3 Validated Multi-valued Byzantine Agreement

We describe the protocol *VBA* from [9] that realizes validated multi-valued Byzantine agreement.

The basic idea of the validated agreement protocol is that every party proposes its value as a candidate value for the final result. One party whose proposal satisfies the validation predicate is then selected in a sequence of binary Byzantine agreement protocols and this value becomes the final decision value. More precisely, the protocol consists of the following steps:

Echoing the proposal (lines 1–4): Each party P_i broadcasts the value that it proposes to all other parties using verifiable consistent broadcast. This ensures that all honest parties obtain the same proposal value for any particular party, even if the sender is corrupted. Then P_i waits until it has received $n-t$ proposals satisfying the external predicate Q before entering the agreement loop.

Agreement loop (lines 5–20): One party is chosen after another, according to a fixed permutation Π of $\{1, \ldots, n\}$. Let a denote the index of the party selected in the current round (P_a is called the "candidate"). Each party P_i carries out the following steps for P_a:

1. Send a `vote` message to all parties containing 1 if P_i has received P_a's proposal (including the proposal in the vote) and 0 otherwise (lines 6–11).
2. Wait for $n - t$ `vote` messages, but do not count votes indicating 1 unless a valid proposal from P_a has been received—either directly or included in the `vote` message (lines 12–13).
3. Run a validated binary Byzantine agreement biased towards 1 to determine whether P_a has properly broadcast a valid proposal. Vote 1 if P_i has received a valid proposal from P_a and validate this by the protocol message that completes the verifiable broadcast of P_a's proposal. Otherwise, if P_i has received $n - t$ `vote` messages containing 0, vote 0; no validation data is needed here. If the agreement decides 1, exit from the loop (lines 14–20).

Delivering the chosen proposal (lines 21–24): If P_i has not yet delivered the broadcast by the selected candidate, obtain the proposal from the validation returned by the Byzantine agreement.

The full validated multi-valued Byzantine agreement protocol is shown as Protocol 4.

5.2.1.4 Reliable Broadcast

MAFTIA middleware uses the protocol *RBC* from [9] (which is a simple adaptation of work of Bracha [6] to reduce message complexity) for reliable broadcast. This protocol uses the hash of a payload message as a short, but unique representation for the potentially much longer message. The idea is that the payload is sent only once by the sender to all parties. When a party is ready to deliver a payload message but does not yet know it, it asks an arbitrary subset of $2t + 1$ parties for its contents and at least one of them will answer with the correct value.

Protocol 4 Validated multi-valued Byzantine agreement with *VBA*

LET $V_a(v, \rho)$ BE THE FOLLOWING PREDICATE:

$$V_a(v,\rho) \equiv (v = 0) \textbf{ or } \big(v = 1 \textbf{ and } \rho \text{ completes the verifiable consistent broadcast of a message } (\texttt{echo}, w_a, \pi_a) \text{ with tag } a \text{ such that } Q(w_a, \pi_a) \text{ holds}\big)$$

UPON RECEIVING MESSAGE $(\texttt{propose}, w, \pi)$:

```
 1 *verifiably consistently broadcast* message $(\texttt{echo}, w, \pi)$ tagged with $\texttt{vcbc}|i$
 2 $w_j \leftarrow \perp; \pi_j \leftarrow \perp \quad (1 \le j \le n)$
 3 **wait for** $n - t$ messages $(\texttt{echo}, w_j, \pi_j)$ to be consistently delivered with tag $\texttt{vcbc}|j$
     from distinct $P_j$ such that $Q(w_j, \pi_j)$ holds
 4 $l \leftarrow 0$
 5 **repeat**
 6   $l \leftarrow l + 1; a \leftarrow \Pi(l)$
 7   **if** $w_a = \perp$ **then**
 8     send the message $(\texttt{vote}, a, 0, \perp)$ to all parties
 9   **else**
10     let $\rho$ be the message that completes the consisten broadcast with tag $\texttt{vcbc}|a$
11     send the message $(\texttt{vote}, a, 1, \rho)$ to all parties
12   $u_j \leftarrow \perp; r_j \leftarrow \perp \quad (1 \le j \le n)$
13   **wait for** $n - t$ messages $(\texttt{vote}, a, u_j, \rho_j)$ from distinct $P_j$ such
       that $V_a(u_j, \rho_j)$ holds
14   **if** there is some $u_j = 1$ **then**
15     $v \leftarrow 1$; $\rho \leftarrow \rho_j$
16   **else**
17     $v \leftarrow 0$; $\rho \leftarrow \perp$
18   propose $v$ validated by $\rho$ in validated binary Byzantine agreement with tag $a$
       biased towards 1, with predicate $V_a$
19   **wait for** the agreement protocol to decide some $b$ validated by $\sigma$ tagged by $a$
20 **until** $b = 1$
21 **if** $w_a = \perp$ **then**
22   use $\sigma$ to complete the verifiable consistent broadcast with tag $\texttt{vcbc}|a$
       and consistenly deliver $(\texttt{echo}, w_a, \pi_a)$
23 output $(w_a, \pi_a)$
24 **halt**
```

A detailed algorithm is presented as Protocol 5, describing P_i's actions when the sender is declared to be j (possibly equalling i). The function H used here is a collision-free hash function.

Protocol 5 Protocol *RBC* for reliable broadcast

```
Initialization:
$\bar{m} \leftarrow \bot; \bar{d} \leftarrow \bot$
$e_d \leftarrow 0; r_d \leftarrow 0$      $(d \in \{0,1\}^{k'})$

Upon receiving message $(\texttt{broadcast}, m)$:
send $(\texttt{send}, m)$ to all parties

Upon receiving message $(\texttt{send}, m)$ from $P_l$:
if $j = l$ and $\bar{m} = \bot$ then
  $\bar{m} \leftarrow m$
  send $(\texttt{echo}, H(m))$ to all parties

Upon receiving message $(\texttt{echo}, d)$ from $P_l$ for the first time:
$e_d \leftarrow e_d + 1$
if $e_d = n - t$ and $r_d \leq t$ then
  send $(\texttt{ready}, d)$ to all parties

Upon receiving message $(\texttt{ready}, d)$ from $P_l$ for the first time:
$r_d \leftarrow r_d + 1$
if $r_d = t + 1$ and $e_d < n - t$ then
  send $(\texttt{ready}, d)$ to all parties
else if $r_d = 2t + 1$ then
  $\bar{d} \leftarrow d$
  if $H(\bar{m}) \neq d$ then
    send $(\texttt{request})$ to $P_1, \ldots, P_{2t+1}$
    wait for a message $(\texttt{answer}, m)$ such that $H(m) = \bar{d}$
    $\bar{m} \leftarrow m$
  output $(\texttt{deliver}, \bar{m})$

Upon receiving message $(\texttt{request})$ from $P_l$ for the first time:
if $\bar{m} \neq \bot$ then
  send $(\texttt{answer}, \bar{m})$ to $P_l$
```

5.2.1.5 *Consistent and Verifiable Consistent Broadcast*

These two protocols are in fact identical, the plain consistent broadcast simply does not provide access to the closing message of a finished instance, nor does it allow the application layer to deliver such a message "from above". We therefore show this protocol, the *VCBC* protocol from [9], only one time, as Protocol 6. Here again we give P_i's actions when the declared sender is j (possibly equalling i) and again H is a collision-free hash function; the protocol also employs an $(n, \lceil \frac{n+t+1}{2} \rceil, t)$ threshold signature scheme $\mathcal{S}$.

In the protocol description, the messages `request` and `answer` encode the verifiability extension mechanisms of asking for the closing message from a VerifiableConsistentBroadcast instance or of giving such a message to such an instance; as such, they are unsupported for bare ConsistentBroadcasts.

Protocol 6 Protocol *VCBC* for [verifiable] consistent broadcast

```
INITIALIZATION:
$\bar{m} \leftarrow \bot; \bar{\mu} \leftarrow \bot$
$W_d \leftarrow \emptyset; r_d \leftarrow 0$      $(d \in \{0,1\}^{k'})$

UPON RECEIVING MESSAGE $(\texttt{broadcast}, m)$:
send $(\texttt{send}, m)$ to all parties

UPON RECEIVING MESSAGE $(\texttt{send}, m)$ FROM $P_l$:
if $j = l$ and $\bar{m} = \bot$ then
  $\bar{m} \leftarrow m$
  compute an $\mathcal{S}$-signature share $\nu$ with tag $(j, \texttt{ready}, H(m))$
  send $(\texttt{ready}, H(m), \nu)$ to $P_j$

UPON RECEIVING MESSAGE $(\texttt{ready}, d, \nu_l)$ FROM $P_l$ FOR THE FIRST TIME:
if $i = j$ and $\nu_l$ is a valid $\mathcal{S}$-signature share then
  $W_d \leftarrow W_d \cup \{\nu_l\}$
  $r_d \leftarrow r_d + 1$
  if $r_d = \lceil \frac{n+t+1}{2} \rceil$ then
    combine the shares in $W_d$ to an $\mathcal{S}$-threshold signature $\mu$
    send $(\texttt{final}, d, \mu)$ to all parties

UPON RECEIVING MESSAGE $(\texttt{final}, d, \mu)$:
if $H(\bar{m}) = d$ and $\bar{\mu} = \bot$ and $\mu$ is a valid $\mathcal{S}$-signature then
  $\bar{\mu} \leftarrow \mu$
  output $(\texttt{deliver}, \bar{m})$

UPON RECEIVING MESSAGE $(\texttt{request})$ FROM $P_l$:
if $\bar{\mu} \neq \bot$ then
  send $(\texttt{answer}, \bar{m}, \bar{\mu})$ to $P_l$

UPON RECEIVING MESSAGE $(\texttt{answer}, m, \mu)$ FROM $P_l$:
if $\bar{\mu} = \bot$ and $\mu$ is a valid $\mathcal{S}$-signature tagged with $(j, \texttt{ready}, H(m))$ then
  $\bar{\mu} \leftarrow \mu$
  $\bar{m} \leftarrow m$
  output $(\texttt{deliver}, \bar{m})$
```

5.2.1.6 Atomic Broadcast Stream

This is the somewhat involved protocol *ABC* from [9]; we outline it first for clarity. In this protocol, each party maintains a FIFO queue of not yet delivered payload messages.

Messages received to be broadcast are appended to this queue whenever they are received. The protocol proceeds in asynchronous global rounds, where each round r consists of the following steps:

1. Send the first payload message w in the current queue to all parties, accompanied by a digital signature σ in a `queue` message.
2. Collect the messages of $n - t$ distinct parties and store them in a vector W, store the corresponding signatures in a vector S, and propose W for Byzantine agreement validated by S.
3. Perform multi-valued Byzantine agreement with validation of a vector $W = [w_1, \ldots, w_n]$ and proof $S = [\sigma_1, \ldots, \sigma_n]$ through the predicate $\Theta_r(W, S)$ which is true if and only if for at least $n - t$ distinct indices j, the vector element σ_j is a valid signature on the tag $(\texttt{queue}, r, j, w_j)$ by P_j.
4. After deciding on a vector V of messages, deliver the union of all payload messages in V according to a deterministic order; proceed to the next round.

The digital signature referred to here is any standard (i.e., non-threshold) secure signature scheme; below it is sometimes denoted $\mathcal{S}$.

In order to ensure liveness of the protocol, there are actually two ways in which the parties move forward to the next round: when a party receives a *broadcast* input message and when a party receives a `queue` message of another party pertaining to the current round. If either of these two messages arrive and contain a yet undelivered payload message, and if the party has not yet sent its own `queue` message for the current round, then it enters the round by appending the payload to its queue and sending a `queue` message to all parties.

A detailed description is found in Protocol 7.

The FIFO queue q is an ordered list of values (initially empty). It is accessed using the operations *append*, *remove*, and *first*, where *append*(q, m) inserts m into q at the end, *remove*(q, m) removes m from q (if present), and *first*(q) returns the first element in q. The operation $m \in q$ tests if an element m is contained in q.

A party waiting at the beginning of a round simultaneously **waits for** `broadcast` and `queue` messages containing some $w \notin d$ in line 2. If it receives a *broadcast* request, the payload m is appended to q. If only a suitable `queue` protocol message is received, the party makes w its own message for the round, but does not append it to q.

The protocol in Protocol 7 is formulated using a single loop that runs forever after initialization; this is merely for syntactic convenience and can be implemented by decom-

Protocol 7 Protocol *ABC* for atomic broadcast

LET $\Theta_r(v, \rho)$ BE THE FOLLOWING PREDICATE:

$$\Theta_r([w_1, \ldots, w_n], [\sigma_1, \ldots, \sigma_n]) \equiv \big(\text{for at least } n - t \text{ distinct } j,\ \sigma_j \text{ is a valid } \mathcal{S}\text{-signature by } P_j \text{ on tag } (\texttt{queue}, r, j, w_j).\big)$$

INITIALIZATION:

```
$q \leftarrow []$          {FIFO queue of messages to *broadcast*}
$d \leftarrow \emptyset$          {set of *delivered* messages}
$r \leftarrow 0$          {current round}
```

UPON RECEIVING MESSAGE $(\texttt{broadcast}, m)$:

```
if $m \notin d$ and $m \notin q$ then
  $append(q, m)$
```

FOREVER:

```
1  $w_j \leftarrow \perp; \sigma_j \leftarrow \perp$     $(1 \le j \le n)$
2  wait for $q \neq []$ or a message $(\texttt{queue}, r, l, w_l, \sigma_l)$ received from $P_l$
     such that $w_l \notin d$ and $\sigma_l$ is a valid signature from $P_l$
3  if $q \neq []$ then
4    $w \leftarrow first(q)$
5  else
6    $w \leftarrow w_l$
7  compute a digital signature $\sigma$ on $(\texttt{queue}, r, i, w)$
8  send the message $(\texttt{queue}, r, i, w, \sigma)$ to all parties
9  wait for $n - t$ messages $(\texttt{queue}, r, j, w_j, \sigma_j)$ such that $\sigma_j$ is a valid
     signature from $P_j$ (including the message from $P_l$ above)
10 $W \leftarrow [w_1, \ldots, w_n]; S \leftarrow [\sigma_1, \ldots, \sigma_n]$
11 propose $W$ validated by $S$ for multi-valued validated Byzantine agreement
     with tag $r$ and predicate $\Theta_r$
12 wait for the validated Byzantine agreement protocol with tag $r$ to decide
     upon some $V = [v_1, \ldots, v_n]$
13 $b \leftarrow \bigcup_{j=1}^{n} v_j$
14 for $m \in (b \setminus d)$, in some deterministic order do
15   output $(\texttt{deliver}, m)$
16   wait for an acknowledgment
17   $d \leftarrow d \cup \{m\}$
18   $remove(q, m)$
19 $r \leftarrow r + 1$
```

posing the loop into the respective message handlers.

5.2.1.7 Secure Causal Atomic Broadcast Stream

We explain the protocol *SC-ABC* from [9] for secure causal atomic broadcast. To *broadcast* a ciphertext c, we simply *broadcast* c on a fixed atomic broadcast stream. Upon *delivery* of a ciphertext c via this sub-stream, a party *schedules* c. Then it computes a decryption share δ and sends this to all other parties in an `decrypt` message containing c. It waits for $t+1$ `decrypt` messages pertaining to c. Once they arrive, it recovers the associated cleartext and *delivers* c to the application. After receiving the acknowledgment, the party continues processing the next atomic broadcast *delivery* by generating the corresponding acknowledgment. Encryption and decryption here refer to a fixed $(n, t + 1)$-threshold cryptosystem (denoted $\mathcal{E}$ below). The details are in Protocol 8. For ease of notation, the

Protocol 8 Protocol *SC-ABC* for secure causal atomic broadcast

```
INITIALIZATION:
        open an atomic broadcast stream $\mathcal{A}$ tagged with scabc

UPON RECEIVING (broadcast, $c$):
        broadcast $c$ on the stream $\mathcal{A}$

FOREVER:
    1  wait for the next message $c$ that is delivered on the stream $\mathcal{A}$.
    2  compute a tagged $\mathcal{E}$-decryption share $\delta$ for $c$
    3  output (schedule, $c$)
    4  send the message (decrypt, $c, \delta$) to all parties
    5  $\delta_j \leftarrow \perp \qquad (1 \leq j \leq n)$
    6  wait for $t + 1$ messages (decrypt, $c, \delta_j$) from distinct parties that contain valid
         tagged decryption shares for $c$
    7  combine the decryption shares $\delta_1, \ldots, \delta_n$ to obtain a cleartext $m$
    8  output (reveal, $m$)
    9  wait for an acknowledgment
   10  acknowledge the last message delivered on $\mathcal{A}$
```

protocol in Protocol 8 is formulated using a FOREVER loop; it can be decomposed into the respective message handlers in straightforward way.

5.2.1.8 Aggregated Broadcast Streams (Reliable and Consistent)

As described above in §3.2, aggregated broadcast streams are built up in a simple way from the corresponding broadcasts; there is no protocol *per se* for these streams (and hence we shall provide no formal exposition of one). Instead, a new instance of such a stream starts n simultaneous broadcasts, one for each possible sending party, and waits for them to deliver their payloads. As soon as one does so, that payload is passed up to the application layer as the next message on the stream, and a brand new instance of the lower-level broadcast is immediately created to wait for the next possible message from that sender. It follows clearly that the order of delivery can vary wildly from participant to participant, depending upon thread scheduling, but the guarantees of reliability and consistency extend in the obvious way to the resulting streams.

5.2.2 Asynchronous Communication Protocols with Support from TTCB

This section describes two protocols, Byzantine Reliable Multicast (BRM) and Byzantine Atomic Multicast (BAM), that were developed using a novel way of designing secure protocols, which is based on a well-founded hybrid failure model. Although these protocols tolerate arbitrary faults, they do not need to necessarily incur the cost of "Byzantine agreement", in number of participants and round/message complexity. They rely on the existence of the TTCB, where the participants only execute crucial parts of the protocol operation, under the protection of a crash failure model. Otherwise, participants follow an arbitrary failure model.

5.2.2.1 Processes and Failures

A process is *correct* if it always follows the protocol until the protocol completion. There are several circumstances, however, that might lead to a process failure. For instance, a process can crash (e.g, due to a power outage) or start to behave maliciously (e.g., produce wrong results). In an arbitrary failure model, which is the model being considered, no restrictions are imposed on process failures, i.e., they can fail arbitrarily. A process can simply stop working, or it can send messages without regard of the protocol, delay or send contradictory messages, or even collude with other malicious processes with the objective of breaking the protocol.

From the various reasons that can cause a process to produce incorrect results, traditionally the most difficult to tolerate is related to attacks made by humans. Once an attacker takes control of a process, it can make that process behave in any way, and if one wants to be conservative, one has to assume that it can cause that process behave in the worse possible manner to the protocol execution. In the rest of this section we will look into a few examples of attacks that are specific to our architecture, and that might lead to the failure of the corresponding process.

A personification attack can be made by a local adversary if it is able to get the pair $(eid, secret)$, which lets a process communicate securely with the local TTCB. Before a process starts to use the TTCB, it needs to call the Local Authentication Service to establish a secure channel with the local TTCB. The outcome of the execution of this procedure is a pair $(eid, secret)$, where eid is the identifier of the process and $secret$ is a symmetric key shared with the local TTCB. If an attacker penetrates a host and obtains this pair, it can impersonate the process before the TTCB and the TTCB before the process. If this pair is kept secret, the attacker can only try to disrupt or delay the communication between the process and the local TCCB – personification attacks are prevented.

Another personification attack is possible if the attacker obtains the symmetric keys that a process shares with other processes. In this case, the attacker can forge some of the messages sent between processes. Most of the messages transmitted by the protocol being proposed do not need to be authenticated and integrity protected because corruptions and forgeries can be detected with the help of the TTCB. The only exception happens with the acknowledgments sent by the protocol, where it is necessary to add a vector of message authentication codes. A successful attack to a host and subsequent disclosure of the shared keys of a process, allows an attacker to falsify some acknowledgements. If the keys can be kept secret, then the attacker can only disrupt or delay the communication, in the host or the network.

A denial of service attack happens if an attacker prevents a process from exchanging data with other processes by systematically disrupting or delaying the communication. In asynchronous protocols typically it is assumed that messages are eventually received (reliable channels), and when this happens the protocol is able to make progress. To implement this behavior processes are required to maintain a copy of each message and to keep re-transmitting until an acknowledgement arrives (which might take a long time, depending on the failure). Here we decided to take a different approach: if an attacker can systematically disrupt the communication of a process, then the process is considered failed as soon as possible, otherwise the attacker will probably disturb the communication long enough for the protocol to become useless. For example, if the payment system of an e-store is attacked and an attempt of paying an item takes 10 hours (or 10 days) to proceed, that is equivalent to a failure of the store.

In channels with only accidental faults it is usually considered that no more than Od messages are corrupted/lost in a reference interval of time. Od is the *omission degree* and tests can be made in concrete networks to determine Od with any desired probability [72]. If a process does not receive a message after $Od + 1$ retransmissions from the sender, with Od computed considering only accidental faults, then it is reasonable to assume that either the process crashed, or an attack is under way. In any case, we will consider the receiver process as failed. The reader, however, should notice that Od is just a parameter that will be used in the protocol. If Od is set to a very high value, then our protocol will start to behave like the protocols that assume reliable channels.

Note that the omission degree technique lies on a synchrony hypothesis: we 'detect' omissions if a message does not arrive after a timeout longer than the 'worst-case delivery delay' (the hypothesis). Furthermore, we 'detect' crash if the omission degree is exceeded. In our environment (since it is asynchronous, bursts of messages may be over-delayed, instead of lost) this artificial hypothesis leads to forcing the crash of live but slow (or slowly connected) processes. There is nothing wrong with this, since it allows progress of the protocol, but this method is subject to inconsistencies if failures are not detected correctly [24].

Another advantage of considering systematically delayed processes as failed is related with the implementation of the TTCB. Since the TTCB is a small component, it can only keep the results of the agreement service for a limited time. If a delayed process asks for a result after it expired the simplest thing to do is to consider the process as failed. Alternatively, the protocol could be made more complex to recover from this situation. However, there is no much justification in doing so for the reason pointed earlier – if a process is too late it is useless.

5.2.2.2 Protocol Definition and Properties

In each execution of a multicast there is one sender process and several recipient processes. A message transmitted to a group should be delivered to all member processes (with the limitations mentioned below), including the sender. In the case of BAM, assurances must be provided about the order of message delivery. Each process, in BAM, must deliver its messages in a same order. In the rest of the document, we will make the classical separation of *receiving* a message from the network and *delivering* a message – the result of the protocol execution.

Informally, a reliable multicast protocol enforces the following [7]: 1) all correct processes deliver the same messages, and 2) if a correct sender transmits a message then all correct processes deliver this message. For atomic multicast, total order must be added in the first assertive. These rules do not imply any guarantees of delivery in case of a malicious sender. However, one of two things will happen, either the correct processes never complete the protocol execution and no message is ever delivered, or if they terminate, then they will all deliver the same message. No assumptions are made about the behavior of the malicious (recipient) processes. They might decide to deliver the correct message, a distinct message or no message.

Formally, a reliable multicast protocol has the following properties [36]:

- *Validity:* If a correct process multicasts a message M, then some correct process in $group(M)$ eventually delivers M.
- *Agreement:* If a correct process delivers a message M, then all correct processes in $group(M)$ eventually deliver M.
- *Integrity:* For any message M, every correct process p delivers M at most once, and only if p is in $group(M)$, and if $sender(M)$ is correct then M was previously multicast by *sender*.

For atomic multicast, the following properties are included:

- *FIFO order:* If a correct process broadcasts a message M before it broadcasts a message M', then no correct process delivers M', unless it has previously delivered M.
- *Total order:* If correct processes p and q both deliver messages M and M', then p delivers M before M' if and only if q delivers M before M'.

These properties place no restriction on the messages delivered by faulty processes. However, as stated in [Hadzilacos94], uniform versions of these properties are only possible for benign failures, thus, in an arbitrary failures model, uniformity is a meaningless concept.

5.2.2.3 The BRM-M Protocol

The Byzantine Reliable Multicast BRM-M protocol is executed in two phases. In the first one, the sender multicasts the message one time to the recipients, and then it securely transmits an hash code through the TTCB agreement service. This hash code is used by the receivers to ensure the integrity and authenticity of the message. If there are no attacks and no congestion in the network, with high probability the message is received by all recipients, and the protocol can terminate immediately. Otherwise, it is necessary to enter the second phase. Here, processes retransmit the message until either a confirmation arrives or the $Od + 1$ limit is reached. Each multicast is performed at most $Od + 1$ times in order to tolerate omissions due to accidental faults (see Section 5.2.2.1).

Figure 9 shows an implementation of the protocol. A message consists of a tuple with the following fields $(type, sender, elist, tstart, data)$. *type* indicates if it is a data message (DAT) or an acknowledgement (ACK). *sender* is the identifier of the sender process, and *data* is either the information given by the application or a vector of MACs (see below). *elist* is a list of *eid* with the format accepted by the TTCB agreement service. The first element of the list is the *eid* of the sender, the others are the *eid* of the receivers. *tstart* is the timestamp that will be given to the agreement service. Each execution of the protocol is identified by $(elist, tstart)$. The protocol uses two low level read primitives, one that only returns when a new message is available, $read\_blocking()$, and another that returns immediately either with a new message or with a non-valid value ($\perp$) to indicate that no message exists, $read\_non\_blocking()$. These two primitives only read messages with the same value of $(elist, tstart)$ which correspond to a given instance of the protocol execution. Other values of the pair are processed by other instances of the protocol. We assume that there is a garbage collector that throws away messages for instances of the protocol that have already finished running (e.g., delayed message retransmissions). This garbage collector can be constructed by keeping in a list the identifiers of the messages already delivered and comparing the identifiers of the arriving messages.

With the exception of the beginning, the code presented in the figure is common both to the sender and the recipients. If the process is a sender, it constructs and multicasts the message to the receivers (Lines 3-4). *tstart* is set to the current time plus a delay T_1. T_1 should be proportional to the average message transmission time, i.e., it should be calculated in such a way that there is a reasonable probability of message arrival before *tstart*. In practice, the value of *tstart* is a tradeoff: if it is too large, the first phase may take longer than what is required; if the value is too small, a correct recipient may not receive the message before *tstart* and the second phase will have to be executed unnecessarily (i.e., the opportunity to terminate the protocol early is lost).

Recipient processes start by blocking, waiting for a message arrival (Line 6). Depending on whether there are or not message losses, the received message might be of type DAT or ACK, or a corrupted message with the fields $(elist, tstart)$ correct. The variable *n-sends* contains the number of messages multicasted and is set initially to 1 for the sender and to 0 to the recipients (Lines 4 and 6). Next, both sender and recipients propose the hash of the message, $H(M)$, to the agreement service (M is the message transmitted by the sender, or the first message received by the recipient), and then they block waiting for the result of the agreement (Line 7-8). The decision function used by the protocol is called $TTCB\_TBA\_RMULTICAST$. This function selects as the result of the agreement the value proposed by the first process in *elist*, which in our case is the sender. An hash function is basically a one-way function that compresses its input and produces a fixed sized digest (e.g., 128 bits for MD5). It is assumed that an attacker is unable to subvert the cryptographic properties of the hash function, such as weak and strong collision resistance [44]. Since the system is asynchronous, there is always the possibility, although highly improbable, that the sender experiences some delay and it tries to propose after *tstart*. In this case, *TTCB_propose* will return the error $TTCB\_TSTART\_EXPIRED$ and the sender process should abort the multicast, and the application can retry the multicast later (for simplicity this condition is omitted from the code). If all processes proposed the same hash of the message, all can deliver and terminate (Line 9). Recall that field *proposed-ok*

Protocol 9 BRM-M Sender and Recipient protocol.

```
{——— Phase 1 ———}
if I am the sender then {SENDER process}
   M := (DAT, my-eid, elist, TTCB_getTimestamp() + $T_1$, data);
   multicast M to elist except sender; n-sends := 1;
else {RECIPIENT processes}
   read_blocking(M); n-sends := 0;
propose := TTCB_propose(M.elist, M.tstart, TTCB_TBA_RMULTICAST, H(M));
repeat
   decide := TTCB_decide(propose.tag);
until (decide.error = TTCB_TBA_ENDED);
if (decide.proposed-ok contains all recipients) then
   deliver M; return;

{——— Phase 2 ———}
M-deliver := $\perp$;
mac-vector := calculate macs of (ACK, my-eid, M.elist, M.tstart, decide.value);
M-ack := (ACK, my-eid, M.elist, M.tstart, mac-vector);
n-acks := 0; ack-set := eids in decide.proposed-ok;
t-resend := TTCB_getTimestamp();
repeat
   if (M.type = DAT) and (H(M) = decide.value) then
      M-deliver := M;
      ack-set := ack-set $\cup$ {my-eid};
      if (my-eid $\notin$ decide.proposed-ok) and (n-acks $<$ Od+1) then
         multicast M-ack to elist except my-eid; n-acks := n-acks + 1;
   else if (M.type = ACK) and (M.mac-vector[my-eid] is ok) then
      ack-set := ack-set $\cup$ {M.sender};
   if (M-deliver $\neq \perp$) and (TTCB_getTimestamp() $\geq$ t- resend) then
      multicast M-deliver to elist except (sender and eids in ack- set);
      t-resend := t-resend + Tresend; n-sends := n-sends + 1;
   read_non_blocking(M); {sets M = $\perp$ if there are no messages to be read}
until (ack-set contains all recipients) or (n-sends $\geq$ Od+1);
deliver(M-deliver);
```

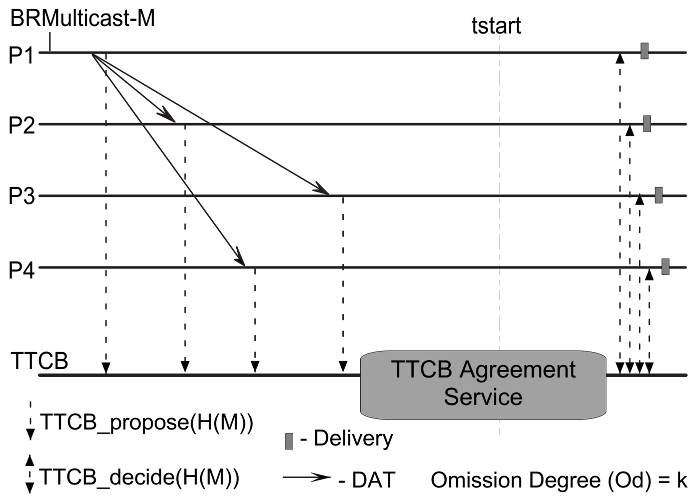

Figure 5.3: Example execution of the BRM-M protocol (best case scenario).

indicates which processes proposed the same value as the one that was decided, i.e., $H(M)$. Figure 5.3 illustrates an execution where processes terminate after this first phase of the protocol. Notice in the figure that the TTCB agreement is initiated immediately after all processes have proposed their value and not by *tstart*.

The second phase is executed if for some reason one or more processes did not propose the hash of the correct message by *tstart*. Variable *M- deliver* is used to store the message that should be delivered, and is initialized to a value outside the range of messages (Line 11). The protocol utilizes message authentication codes (MAC) to protect ACK messages from forgery [44]. This type of signature is based on symmetric cryptography, which requires a different secret key to be shared between every pair of processes. Even though, MACs are not as powerful as signatures based on asymmetric cryptography, they are sufficient for our needs, and more importantly, they are several orders of magnitude faster to calculate. Since ACKs are multicasted to all processes, an ACK does not take a single MAC but a vector of MACs, one per each pair (sender of ACK, other process in *elist*) [21]. A MAC protects the information contained in the tuple *(ACK, my-eid, M.elist, M.tstart, decide.value)*, and is generated using the secret key shared between each pair of processes (Lines 12-13). Next, processes initialize variables *n-ack* and *ack-set* (Line 14). The first one will count the number of ACKs that have been sent. The second one will store the eid of the processes that have already confirmed the reception of the message, either by proposing the correct $H(M)$ to the agreement (Line 14) or with an acknowledge message. *t-resend* indicates the instant when the next retransmission should be done (Line 15). It is initialized to the current time, which means that there will be retransmission as soon as possible.

The loop basically processes the arriving messages (Lines 17-23), does the periodic retransmissions (Lines 24-26), and reads new messages (Line 27). If the message is of type DAT and its hash is the same as the one given by the sender (Line 17) then it is saved for later delivery (Line 18). Next, the eid of the process is added to *ack-set* to indicate that this process has correctly received the message (Line 19). If the process received the message but did not propose the correct hash to the agreement then it needs to confirm the reception by multicasting an ACK (Line 20-21). The ACKs, like the DAT messages, are only transmitted $Od + 1$ times. If the received message is an ACK with a valid MAC, then the eid of the sender is put in *ack-set* (Line 22-23). Next, if it is time, the message is retransmitted to the processes that did not confirm the reception (Lines 24-26). The loop goes on until $Od + 1$ messages are sent or all recipients acknowledged the reception of the message (Line 28). To complete the protocol, the process delivers the message.

As mentioned in Section 5.2.2.2, there are situations where the protocol does not terminate if the sender is malicious or the process is failed. For instance, a malicious sender could propose a false hash of the message, and in this case no correct recipients would be able to deliver the message. To address this problem, a garbage collection mechanism has to prevent correct processes from being clogged with protocol instances that never terminate. This mechanisms should interact with the membership service to determine and remove instances waiting for faulty processes.

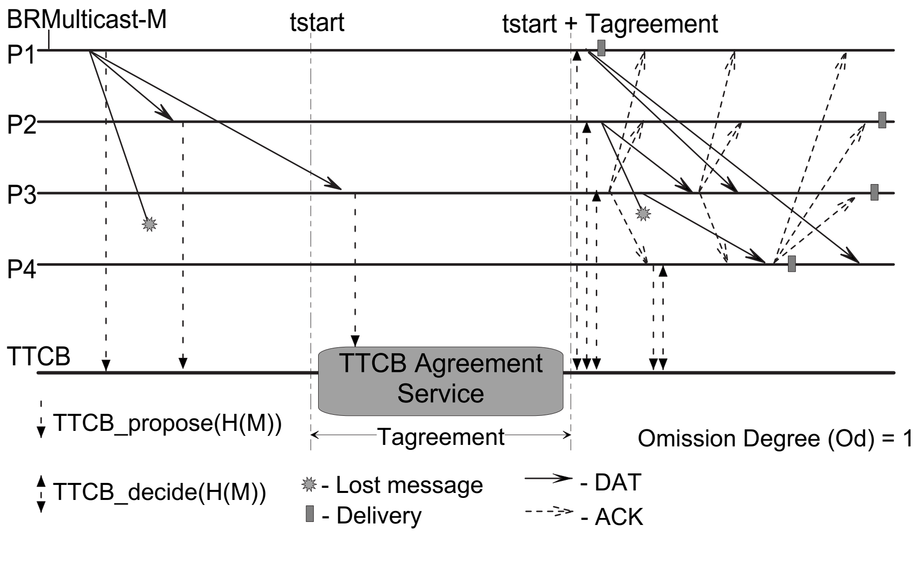

Figure 5.4: Example execution of the BRM-M protocol.

Example of an Execution

Figure 5.4 represents an execution of the protocol. The sender multicasts the message once, P2 receives it in time to propose $H(M)$, P3 receives the message late and P4 does not receive. When the agreement terminates all processes except P4 have the message and get the result from the TTCB (P4 does not even know that the protocol is being executed). At this point, by observing the result of the agreement, all become aware that only P1 and P2 proposed the hash. Therefore, both P1 and P2 multicast the message to P3 and P4. P3 multicasts an ACK to all processes confirming the reception and sends the message to P4. P1 terminates at this moment because it has already sent the message $Od+1$ times. The first message P4 receives is the ACK sent by P3. P4 saves it in *ack-set* and gets the result of the agreement. Then it receives the right message, and multicasts an ACK. At this moment all processes terminate.

5.2.2.4 Overview of the BAM Protocol

The BAM protocol achieves intrusion-tolerance using BRM to multicast messages in a reliable and secure manner. This sub-protocol assures the validity, agreement, and integrity properties, described in the previous section. BAM is an atomic multicast protocol for asynchronous systems with a hybrid failure model. The BAM protocol does not need public key cryptography, since it uses the TTCB to securely exchange a digest of the message.

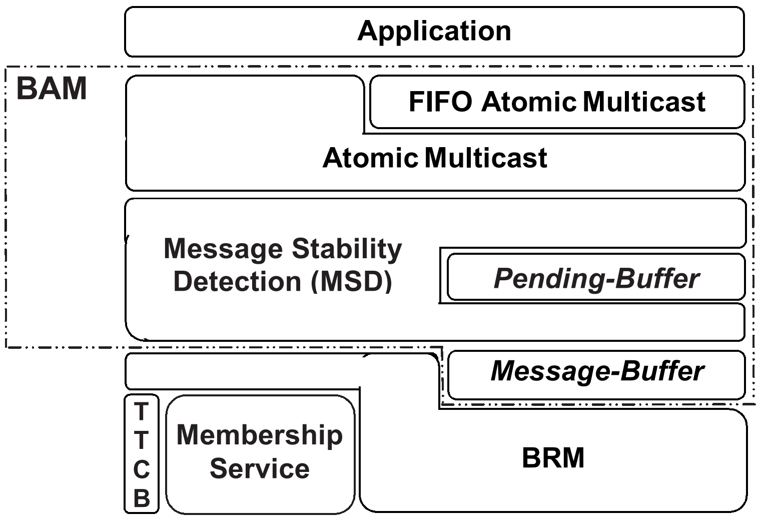

Figure 5.5: Communication services with support from TTCB, logical hierarchy.

Figure 5.5 illustrates the composition of layers required to implement the BAM architecture. Each message delivered by the BRM protocol is inserted in *Message-Buffer*. All messages in the *Message-Buffer* are submitted to the Message Stability Detection protocol in order to determinate which messages are stable. A message is stable if all correct processes already have BR-delivered it, i.e., the BRM protocol has delivered the message to all correct processes . Unstable messages are put in the *Pending-Buffer*, waiting to eventually become stable. In face of malicious processes, the *Pending-Buffer* is useful to detect possible attacks of these processes. For example, to detect delayed messages, when a process p did not receive a message M while all remaining process already received it; or to detect a malicious process p that says it had send/received an inexistent message to/from a process q (*phantom messages*). That information can be useful for the Failure Detection and Membership Service, for instance, to detect intrusions and remove such malicious processes from the current view. After Message Stability Detection, the stable messages can be delivered in some deterministic order (see Figure 5.5). Adopting the modular specification presented in [36], we provide two ways to atomically deliver M:

- *Byzantine Atomic Multicast:* is a Reliable Multicast that satisfies Total Order. The ordering of messages in BAM is based on a timestamp provided by the TTCB.
- *Byzantine FIFO Atomic Multicast:* is a Reliable Multicast that satisfies both FIFO and Total Order. FIFO Atomic Multicast is stronger than both Atomic Broadcast and FIFO Broadcast. In this particular case, the ordering of messages is based on a timestamp provided by the TTCB and a sequence number.

5.2.2.5 The BAM Protocol

As mentioned in the previous section, the BAM protocol depends on BRM to assure the validity, agreement, and integrity properties. Every message M multicasted using the BRM protocol contains the identifier of the sender (*eid*), timestamp (*tstart*), and the last membership view (*elist*) at the multicast event instant. *eid* is obtained when a process starts to use the TTCB, by calling the Local Authentication Service to establish a secure channel with the local TTCB. *tstart* is set to the current time plus a delay *Td*. The current time is obtained from the Trusted Absolute Timestamping Service, which provides globally meaningful timestamps. The *elist* is provided by membership service of the MAFTIA architecture, which sends view updates for each membership changes. BRM is defined in terms of two primitives: *BR-multicast(M)* and *BR-deliver(M)*, where M is the message multicasted by the sender. When a sender process *BR-multicast(M)* that means it used the BRM protocol to multicast M, and when a recipient process *BR-deliver(M)* means that it received a message M delivered by the BRM protocol. In a similar way, BAM is also defined in terms of two primitives: *BA-multicast(M)* and *BA-deliver(M)*, where M is the message atomically multicasted by the sender.

The BAM protocol is described in Protocol 10. When a process p wishes to *BA-multicast* a message M it executes *BR-multicast(M)* (Line 4). When a process *BR-deliver* M, it adds M to the *Message-Buffer*$_p$, which contains all messages locally delivered by process p according to BRM protocol. The variables of the protocol are given below:

- *$elist_{vn}$:* is the current view sent by Byzantine Membership Service and *vn* is the view number;

- *M-header:* is the header of a message *M* in the *Message-Buffer*. It contains the *eid* of the sender, *elist*, *timestamp*, *sn* (sequence number for FIFO ordering) – these parameters are obtained from *BRM-deliver(M))* –, *r* (round number), and *status* (either *unstable*, *stable* or *pending*):

  M-header := (*eid*, *elist*, *TTCB timestamp*, *sn*, *r*, *status*);

- *LVp_r:* is the local vector of the process *p* that contain all *M-header* with either *unstable* or *pending* status in round *r*. The set of message *M* received and not delivered until round *r*:

  $LVp_r := \{\forall M\text{-}header: M \in (Message\text{-}Buffer_p \cup Pending\text{-}Buffer_p)\}$;

- *LMp_r:* is the local matrix of the process *p* containing its own LMp_r and the set of LVq_r's received from each process *q* in the round *r*:

  $LMp_r := \{\forall p,q: p,q \in elist, LVp_r \cup \forall LVq_r\}$;

In order to add a message *M* to *Message-Buffer*$_p$, the header of *M* (*M-header*) is built and added to the body of *M*, and its initial status is set to *unstable*. In order to *BA-deliver(M)*, the status of *M* must be stable. *BA-deliver* happens as follows: the set of messages in *Message-Buffer*$_p$ is submitted to the Message Stability Detection (MSD) layer, the heart of the BAM protocol, which detects the *BA-deliverable* (or *stable*) messages. The *BA-deliverable* is the set of stable messages, stable means that all correct processes have already *BR-deliver* them. Note that for any process *p* and *q*, in a round *r*, *BA-deliverable*$_p$ must be equal to *BA-deliverable*$_q$. Due to the asynchrony assumption of the payload system, not always the set of messages in *Message-Buffer*$_p$ is the same in any correct process q, thus, the remaining messages are put in *Pending-Buffer*$_p$, or more precisely, *Pending-Buffer*$_p$ = *Messages-Buffer*$_p$ - *BA-deliverable*. A message *M'* in *Pending-Buffer*$_p$ (*M'-header.status* = *pending*) could only be delivered when the outcome of one of the following rounds the MSD indicates that *M'* was finally *BR-deliver* by all correct processes.

The MSD of the BAM protocol is based on the virtual synchrony model [3]. The views are ordered in an acyclic sequence. The MSD protocol is executed in each process, in asynchronous rounds, and in a sequential manner – each round is used to detect a set of stable messages (*BA-deliverable* = { ∀M: *M-header.status* = *stable*}). In the Figure 5.6 is presented the communication model of the BAM, the arrows of the upper part show the messages exchanged by the applications, and the lower part represents the messages of

Protocol 10 Byzantine Atomic Multicast (BAM)

```
INITIALIZATION:
$r \leftarrow 0$;

WHEN A PROCESS $p$ EXECUTES $BA\text{-}multicast\ M$
$BR\text{-}multicast(M)$ to $elist_{vn}$;

WHEN A PROCESS $q$ $BR\text{-}deliver\ M$
build $M\text{-}header$; set $M\text{-}header.status$ to $unstable$;
$M \leftarrow M\text{-}header + M$;
$Message\text{-}Buffer_p \leftarrow Message\text{-}Buffer_p \cup M$;

WHEN A PROCESS $p$ EXECUTES MSD IN A ROUND $r$
if $TTCB\_getTimestamp() > to + r \times T$ or $Message\text{-}Buffer_p > Buffer\_limit\_size$ then
    $r \leftarrow r + 1$;
    {Step 1: all process $BR\text{-}multicast$ its $LVp_r$ to participants of $elist_{vn}$}
    $LVp_r \leftarrow \forall M\text{-}header$: $M \in (Message\text{-}Buffer_p \cup Pending\text{-}Buffer_p)$;
    $BR\text{-}multicast(LVp_r)$ to $elist_{vn}$;
    {Step 2: all process gather $\forall LVq_r$: $q \in elist_{vn}$, and $BR\text{-}multicast\ LMp_r$}
    wait until [$\forall q$: delivered $LVq_r$ or $q \notin (elist_{vn} \cup \forall elist_{vn+k}$: k > 0)];
    $LMp_r \leftarrow LVp_r \cup \forall delivered\_LVq_r$;
    $BR\text{-}multicast\ LMp_r$ to $elist_{vn}$;
    {Step 3: all process gather $\forall LMq_r$: $q \in elist_{vn}$}
    wait until [$\forall q$: delivered $LMq_r$ or $q \notin (elist_{vn} \cup \forall elist_{vn+k}$: k > 0)];
    $BA\text{-}deliverable_p \leftarrow \forall M$: $M\text{-}header \in (LMp_r \cap \forall delivered\_LMq_r)$;
    $Pending\text{-}Buffer_p \leftarrow (\forall M$: $M\text{-}header \in LVp_r)$ - $BA\text{-}deliverable_p$;
    $Message\text{-}Buffer_p \leftarrow Message\text{-}Buffer_p$ - $BA\text{-}deliverable_p$ - $Pending\text{-}Buffer_p$;

WHEN A PROCESS $p$ EXECUTES $BA\text{-}deliver(BA\text{-}deliverable_p)$
order the $BA\text{-}deliverable_p$ messages according to timestamp (if FIFO so, uses sequence number too);
$BA\text{-}deliver(BA\text{-}deliverable_p)$;
```

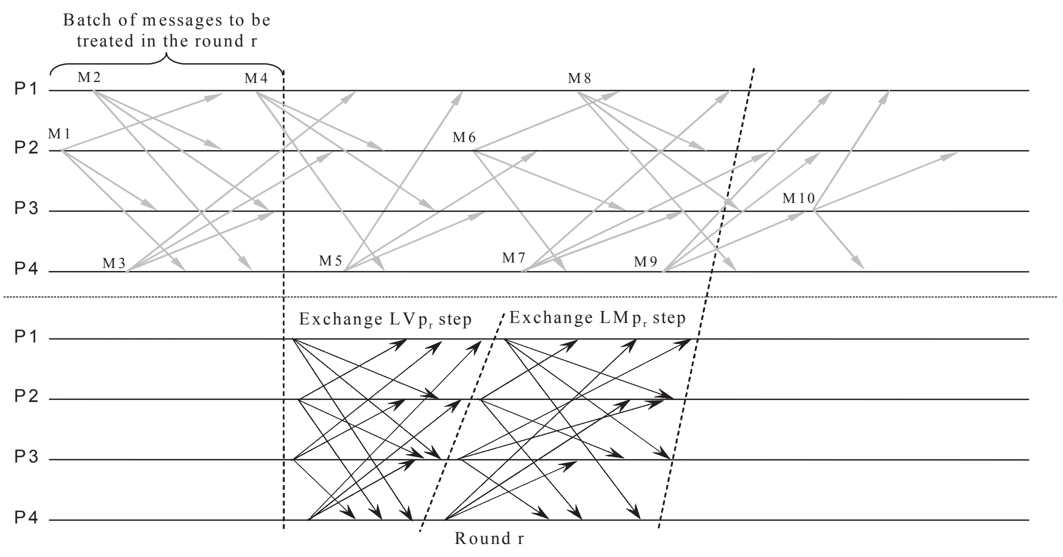

Figure 5.6: Example execution of the BAM protocol

the MSD protocol, which runs in background. The communications in both frames can be executed concurrently.

In a round *r*, the MSD protocol is executed to detect which received messages and not delivered from previous rounds could be delivered in present round. For a better performance, the beginning of a round is defined in either a periodic manner or when the *Message-Buffer* limit is reached (Line 10). By using this optimization we can reduce the frequency of execution of the protocol when there is a small message traffic.

Each round is executed in three steps. On the first step, each process *BR-multicast* its *Local Vector* (*LV*), which contain the set of *M-header* of the batch of messages contained within *Message-Buffer* and *Pending-Buffer* (all *M-header* with status either *unstable* or *pending*). In the following step, each process waits for *Local Vectors* from each correct process, indicated by the Membership Service. Since then, each process builds its own *Local Matrix* (*LM*) and *BR-multicast* it. Within a *Local Matrix* are contained all *Local Vectors* of the group processes (correct or malicious). Finally, on the last step, each process waits for *Local Matrix* from each process in *elist*, and, after that, it selects all of the stable messages according to each *LM* received. The intersection of each *LM* will indicates the stable messages, which already were *BR-delivered* by each process. Thus, that stable messages can be *BA-delivered.* Messages not *BA-delivered* are put within *Pending-Buffer* and their status set to *pending.* If a message *M'* takes too long within *Pending-Buffer* that could be indicative of something wrong with *M'*, and that could be a symptom of a possible intrusion.

5.3 Activity Services

This section describes the protocols that implement the MAFTIA transaction support activity service. Our approach is to implement standard atomic commitment and abort protocols, and distributed locking protocols using service replication protocols that transparently add intrusion-tolerance through replication and voting. This allows us to tolerate the corruption of a certain proportion of replicas executing these protocols. We also take advantage of the replication to implement distributed recovery protocols that depend on the group's continued functioning for recovery. We do not focus on the problems of authorisation, or confidentiality of communications as these can be addressed through the use of the MAFTIA authorisation service and standard IPSec.

In this section we introduce the following protocols:

- Service replication protocols
- Atomic commitment and abort protocols
- Distributed locking protocols
- Distributed recovery protocols

Our protocols depend upon the intrusion-tolerant group communication protocols provided by the lower layers of the MAFTIA middleware.

5.3.1 Fault model

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 [11]. This approach takes into account the diversity of the servers and attempts to find the minimum number with a common failure mode. Our protocols can either be used with the time-free or partially timed communication services. In particular we rely upon atomic broadcast, and the intrusion-tolerant properties of the communication services. Note that, in order to simplify our protocols we assume static groups of replicas.

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. Currently we do not make clients intrusion-tolerant or the transaction service tolerant of misbehaving clients. For example, clients may block the progress of transactions or access to resources managed by resource managers. We have avoided using timeouts to resolve these problems as they introduce a vulnerability that could be exploited by an attacker. Therefore, we propose that future work could look at using the intrusion detection service to drive detection of client misbehaviour.

5.3.2 Protocols

In this section we describe our protocols. Generally, our protocols map to Appia layers. Clients are implemented using the Multicast with Voting protocol layer. Transaction managers are implemented using all the Service Replication protocol layers, TransactionManager protocol layer and the RecoveryManager protocol layer. Resource managers are implemented using the Service Replication protocol layers, ResourceManager protocol layer, LockManager protocol layer and the RecoveryManager protocol layer.

5.3.3 Service Replication

The Service Replication protocols are designed to work together to provide simple service replication. They are based upon the work in [8] which describes the protocols for the implementation of dependable third-parties. The work described here can be seen as another application of these ideas to the servers providing the transactional support service. However, the protocols described in [8] assumed only a time-free model whereas ours are intended to work with the partially-timed model as well. There are three protocols: multicast with voting, opengroup, atomic broadcast and invocation.

Service replication is used to replicate transaction managers and resource managers. The general approach is to replicate the managers using an *active replication* group management policy. In active replication, all the functioning members of the group perform processing [63]. 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.

5.3.3.1 Multicast with Voting

The Multicast with Voting protocol (Protocol 11) imposes a reply-request style of communication between a client and a server group, and implements voting on the replies. When the protocol is initialised a list of group members is passed to it. A client multicasts to the group using the v-multicast-request message. When this is received the protocol blocks, any further requests are queued until the current request has been satisfied. In the protocol the message queue is represented by the set q, and we define the function *append* for adding to the queue and $first$ for removing the first member of the queue.

The multicast is implemented by sending the request to one member of the group who has the responsibility of rebroadcasting it over an atomic broadcast channel to the rest of the group. The advantage of this approach is that this ensures that each member of the group has the message delivered in the same member as each other member. This is required in order to implement state machine replication which requires that each replica of the state machine processes each request in the same order as every other replica. When sufficient $(t+1)$ identical replies for a request have been received then the reply is returned to the client, in the protocol we define a function $count_{unique}$ which counts the number of identical members in a set. We can match replies to requests since a unique sequence number is assigned to each request and is returned with each reply.

As the member of the group who initially receives the request may not be honest, we use a local timeout. Unless the request has been successfully processed within a given time then the request is repeated with the next member of the group. The open group and invocation protocols filter requests, ensuring that identical requests (same message and sequence number) are filtered out. For a complete discussion of this approach see Section 3.4, Deliverable D5 “Full Design of Dependable Third Party Services” [12].

5.3.3.2 Open Group Protocol

The OpenGroup protocol (Protocol 12) implements the rebroadcasting to the other members of the group. It simply waits for a v-multicast message and rebroadcasts the message using atomic broadcast to other members of the group. The message is only rebroadcast if it does not exist in the history of delivered messages (*history*). The history is maintained by the Invocation protocol and contains the sequence number of each message

Protocol 11 Voting Multicast protocol (VMP).

```
INITIALIZATION:
$group \leftarrow$group members
$t \leftarrow$max failures
$T \leftarrow$required timeout
$blocked \leftarrow false$
$q \leftarrow []$ {message queue}
$seq \leftarrow 0$ {unique identifier for messages}

UPON RECEIVING MESSAGE $(ID|j, \texttt{in}, \texttt{v-multicast-request}, m)$:
$append(q, m)$

FOREVER
if $\neg blocked$ and $q \neq \{\}$ then
  $blocked \leftarrow true$
  $seq \leftarrow seq + 1$
  $replies \leftarrow \{\}$
  $curr\_dest \leftarrow 0$
  send $(\texttt{v-multicast}, m|seq)$ to $group[curr\_dest]$
  set timeout to $T$

UPON RECEIVING MESSAGE $(ID|j, \texttt{in}, \texttt{v-multicast-reply}, m|reply\_seq)$:
if $reply\_seq \neq seq$ then
  {ignore, already received sufficient replies}
else
  $replies \leftarrow replies \cup m|reply\_seq)$
  if $count_{unique}(replies) = t + 1$ then
    reset timeout
    output $(ID|j, \texttt{out}, \texttt{v-multicast-deliver}, m)$
    $blocked \leftarrow false$

UPON TIMEOUT
$curr\_dest \leftarrow curr\_dest + 1$
if $curr\_dest > length(q)$ then
  halt and raise exception
else
  $replies \leftarrow \{\}$
  send $(\texttt{v-multicast}, m|seq|j)$ to $group[curr\_dest]$
  set timeout to $T$
```

sent by a particular group. We define a function *group*() which calls the membership service in order to determine which group a given party belongs to. We cannot simply rely upon the sequence number to determine if a message has already been delivered as the sequence number is serializable with respect to a particular group (or in the case of clients who may form a group of one) or party.

Protocol 12 Open Group protocol (OGP).

```
Upon receiving message $(ID|j, \texttt{in}, \texttt{v-multicast}, m|seq)$:
$source \leftarrow j$
$membership \leftarrow group(j)$
if $\neg contains(history, membership|seq$ then
    atomic broadcast $(\texttt{a-broadcast}, m|seq|source)$ to $membership$
```

5.3.3.3 Invocation Protocol

The Invocation protocol (Protocol 13) allows the transactional protocols to be specified independently of the communication protocols. It acts a dispatcher and hides the interface between the service replication protocols and the other functional protocols. The invocation protocol maps multicast requests, multicast replies and atomic broadcast messages onto transactional messages. For example, a begin message received over the atomic broadcast channel by a transaction manager is encapsulated within an atomic broadcast a-deliver message. The transaction manager invocation protocol maps this to a local begin message and appends the source of the message.

The protocol maintains a history of delivered messages in *history*, if a message from a group has already been delivered then it is not subsequently delivered. This removes duplicated messages that occur due to resends due to the multicast protocol.

5.3.4 Atomic Commitment and Abort Protocols

In this section we describe an intrusion tolerant two-phase atomic commitment protocol [48] and an intrusion tolerant abort protocol that gain their intrusion-tolerance from their reliance upon our service replication protocols. The service replication protocols allow these protocols to treat resource manager groups, and the transaction manager group as single entities. The atomic commitment protocol is blocking as a unanimous decision to commit is required from all clients participating in the transaction before commitment can take place. This is a consequence of avoiding the use of timeouts within the resource managers or transaction managers. As a result we propose the use of some form of global

Protocol 13 Invocation protocol (IP).

```
UPON RECEIVING MESSAGE $(ID|j, \texttt{in}, \texttt{a-deliver}, m|seq|source)$:
$membership \leftarrow group(source)$
if $\neg contains(history, membership|seq$ then
    $history \leftarrow history \cup membership|seq)$
    send $m|seq|source$ to self

UPON RECEIVING MESSAGE $(ID|j, \texttt{in}, \texttt{v-multicast-deliver}, m|seq|source)$
send $m|seq|source$ to self

UPON RECEIVING MESSAGE $(\texttt{tm-tid}, m|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{tm-tid}, m|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{tm-join-reply}, result|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{tm-join-reply}, result|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{tm-register-reply}, m|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{tm-register-reply}, result|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{tm-commit-reply}, result|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{tm-commit-reply}, result|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{tm-abort-reply}, result|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{tm-abort-reply}, result|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{rm-prepare-reply}, result|tid\_request|seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{rm-prepare-reply}, result|tid\_request|seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{rm-ack}, seq|source)$:
send $(\texttt{v-multicast-reply}, (\texttt{rm-ack}, seq))$ to $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{rm-prepare}, resource\_id|dest)$:
send $(\texttt{v-multicast-reply}, (\texttt{rm-invoke-result}, result|rtnVal|seq))$ to group containing $P_{source}$
UPON RECEIVING MESSAGE $(\texttt{lock}, resource\_id|tid\_request|source$
send $(\texttt{v-multicast-request}, (\texttt{rm-prepare}, resource\_id))$ to $group(P_{source})$
UPON RECEIVING MESSAGE $(\texttt{rm-abort}, resource\_id|dest)$:
send $(\texttt{v-multicast-request}, (\texttt{rm-abort}, resource\_id|tid\_request))$ to $group(P_{dest})$
UPON RECEIVING MESSAGE $(\texttt{rm-commit}, resource\_id|tid\_request|dest)$:
send $(\texttt{v-multicast-request}, (\texttt{rm-commit}, resource\_id|tid\_request))$ to $group(P_{dest})$
UPON RECEIVING MESSAGE $(\texttt{tm-register}, resource\_id|tid\_request)$:
send $(\texttt{v-multicast-request}, (\texttt{tm-register}, resource\_id|tid\_request))$ to transaction mananger group
UPON RECEIVING MESSAGE $(\texttt{rm-invoke-result}, result|rtnVal|seq|dest))$:
send $(\texttt{v-multicast-request}, (\texttt{lock}, resource\_id|tid\_request))$ to $group(P_{dest)}$
UPON RECEIVING MESSAGE $(\texttt{unlock}, resource\_id|tid\_request|dest)$
send $(\texttt{v-multicast-request}, (\texttt{unlock}, resource\_id|tid\_request))$ to $group(P_{dest})$
```

deadlock detection system that interacts with the MAFTIA intrusion detection system to solve this problem (for similar database approaches see [2].

There are two main protocols, the TransactionManager protocol (TMP) implemented by the transaction manager and the ResourceManager protocol (RMP) implemented by the resource manager. Each protocol is made up of several subprotocols. In this section we provide each subprotocol and interleave them to reflect the progression of the two protocols as the cooperate to realise intrusion-tolerant atomic commitment and abort.

5.3.4.1 Beginning a Transaction

Protocol 14 specifies how transactions are created. When a client requests the creation of a transaction then a new transaction identifier (*tid*) is generated and a transaction record created. The transaction record is stored in a transaction log which is a hashtable (*trans_log*) that uses the *tid* as a key. We define $put(table, key, entry)$ which stores *entry* under the key value *key* in table *table*. Similarly, we define $get(table, key, entry)$ which retrieves the entry *entry* that is stored under the key value *key*. Each transaction record ($clients|resources|state$) maintains the list of clients involved in the transaction, the resources involved in the transaction and the current state of the transaction (for example, `begin` or `end`).

Protocol 14 Transaction Manager protocol (TMP) – creating a transaction.

```
1 UPON RECEIVING MESSAGE ($\texttt{tm-begin}, m|seq|source$): {received from client}
2 $tid \leftarrow tid + 1$ {issue new transaction identifier}
3 $clients \leftarrow \{source\}$ {add client who starts transaction}
4 $resources \leftarrow \{\}$ {initially no resources}
5 $state \leftarrow$ begin
6 $put(trans\_log, tid, clients|resources|state)$ {store transaction record}
7 send ($\texttt{tm-tid}, tid|seq|source$) to self {return transaction id to the client}
```

5.3.4.2 Client Join

Protocol 15 specifies how clients may join a transaction. A client may only join a transaction after it has been created. Any number of clients may be involved in a transaction, they can only join if they know the *tid*. We assume that this is passed between clients via channels external to the transactional support activity service. The protocol receives the join request from the client who wishes to join, the request identifies the transaction that the client wishes to join (*tid_request*). The transaction log is checked to see if the transaction does exist (we define the function $contains(table, key)$ which returns *true* if the hashtable *table* contains an entry for *key*). If it does exist then the state of the transaction is checked to ensure that it is active, in this case its state with be `begin`. Otherwise the transaction is assumed to be either being committed, aborted or has already ended. Finally, the result of the join request is returned to the requesting client.

Protocol 15 Transaction Manager protocol (TMP) – clients joining a transaction.

```
INITIALIZATION:

UPON RECEIVING MESSAGE (tm-join, $tid\_request|seq|source$):
if $\neg contains(trans\_log, tid\_request)$ then
  $result \leftarrow false$ {transaction doesn't exist}
else
  $get(trans\_log, tid, clients|resources|state)$
  if $state \neq begin$ then
    $result \leftarrow false$ {transaction in progress or complete}
  else
    $get(trans\_log, tid\_request, clients|resources|state)$ {get the existing client list}
    $clients \leftarrow clients \cup source)$ {add the requesting client to the client list}
    $put(trans\_log, tid\_request, clients|resources|state)$ {update the transaction log}
    $result \leftarrow true$
send (tm-join-reply, $result|seq|source$) to self {return result to client}
```

5.3.4.3 Resource Registration

Protocol 16 specifies how a resource manager that is asked to invoke a method of a resource requests registration with the transaction (if it is not already a member), protocol 17 specifies how transaction managers register resources.

In order to invoke a method on a resource that is managed by a resource manager, the client sends a `rm-invoke` event to the appropriate resource manager group with the object identifer of the resource that is the target of the invocation, the transactional context for the invocation (*tid_request*), and the details of the method invocation (*method*). For brevity, we assume that *method* includes the name of the method and any arguments supplied to the method.

Assuming that *tid_request* is valid, then we check whether the resource has been registered for the transaction. If it is not registered then we attempt to register it with the transaction managers, assuming that this is successful then the state of the resource is updated. Note that we assume there is a hashtable *state* that stores the transactional state of each resource and uses the *resource_id* as the key. If the resource was already registered or it has now been successfully registered, then the method is invoked on the resource and the result is returned to the client. Should *tid_request* be invalid, or the resource cannot be registered then the method is not invoked and the result `notOk` is returned to the client.

Protocol 16 Resource Manager protocol (RMP) – invoking a resource manager, and requesting the resource manager to commit.

```
UPON RECEIVING MESSAGE (rm-invoke, $resource\_id|tid\_request|method|seq|source$): {received from client}
if $tid\_request > 0$ then
  if $contains(state, resource\_id)$ then
    $get(state, resource\_id, resource\_state)$ {find current state of resource}
  else
    $resource\_state \leftarrow$ non-transactional {no current state for resource}
  if $resource\_state \neq$ transactional and $resource\_state \neq$ prepare then
    {resource not involved in a transaction, so register}
    save state of $resource\_id$
    send (tm-register, $resource\_id|tid\_request$) {send registration request to the transaction manager group}
    wait for tm-register-reply containing result of request
    if $result =$ ok then
      {registration was ok, so update state table}
      $resource\_state \leftarrow$ transactional
      $put(state, resource\_id,$ transactional$)$
    else
      {unable to register resource, cannot make resource transactional}
      $put(state, resource\_id,$ non_transactional$)$
  if $resource\_state =$ transactional then
    {invoke method on resource}
    $rtnVal \leftarrow$ result of invoking $method$ on $resource\_id$
    $result \leftarrow$ ok
  else
    {not transactional, cannot invoke method}
    $rtnVal \leftarrow$ null
    $result \leftarrow$ notOk
else
  {invalid transaction id so cannot invoke method}
  $rtnVal \leftarrow$ null
  $result \leftarrow$ notOk
send (rm-invoke-result, $result|rtnVal|seq|source$ {return result of invocation to client}
```

The transaction manager receives the request from the resource manager group, as long as the transaction exists then the list of resources that are members of the transaction is updated.

Protocol 17 Transaction Manager protocol (TMP) – register a resource.

```
1  UPON RECEIVING MESSAGE (tm-register, $tid\_request|seq|source$): {received from resource asking to
   be registered}
2  if $\neg contains(trans\_log, tid\_request)$ then
3     {the transaction exists, retrieve the list of resources registered for the transaction}
4     $result \leftarrow false$ {transaction doesn't exist}
5     $get(trans\_log, tid\_request, clients|resources|state)$
6     if $state \neq begin$ then
7        $result \leftarrow false$ {transaction is being finalised or has ended}
8     else
9        {add the resource to the list of resources registered for the transaction}
10       $resources \leftarrow resources \cup source|resource\_id$
11       $put(trans\_log, tid, clients|resources|state))$
12       $result \leftarrow true$
13 send (tm-register-reply, $result|seq|source$) to self {return the result to the requesting resource
   group}
```

5.3.4.4 Commit

Protocol 18 shows how transaction managers start the two phase commit process by asking all resource managers if they can commit their resources. All clients must agree on commitment, if any client disagrees then resources are asked to abort.

Protocol 19 shows how the resource manager checks with the resource whether it is prepared to commit and returns the result to the transaction manager.

5.3.4.5 Decide on Commit or Abort

Protocol 20 shows that transaction manager collect replies from resource managers and decide on commit or abort. Once a decision has been made then the resource managers are told to commit or abort, and clients are informed as to whether the transaction has been committed or aborted. Note that when resource managers are asked to commit or abort their resources then they are also asked to release any locks that they hold.

Protocol 21 specifies how a resource manager responds to a decision on commit or abort by a transaction manager. Essentially it applies either commit or abort to the resource being managed and updates its log that records each resource's transaction status.

Protocol 18 Transaction Manager protocol (TMP) – beginning of two-phase commit.

```
UPON RECEIVING MESSAGE (tm-commit, $tid\_request|seq|source$): {received from client}
if $contains(trans\_log, tid\_request)$ then
  {transaction exists}
  $get(trans\_log, tid\_request, clients|resources|state)$
  if $state =$ begin and $contains(clients, source)$ then
    transaction not finalising or ended, and the client requesting the commit is a member of the transaction
    $put(trans\_log, tid\_request,$ voting$)$ {record state as voting, prevents any other clients or resources joining transaction}
    register the client vote
    if all clients agree on commit then
      {ask all registered resources if they can commit}
      $dest = first(resources)$
      for all resources do
        {ask all resource groups to prepare}
        send (rm-prepare, $resource\_id|dest$) to self
        $dest = next(resources)$
    else if all clients voted but disagreement then
      {tell all clients that commit failed}
      $result = notOk$
      $dest = first(clients)$
      for all clients do
        send (tm-commit-reply, $result|seq|dest$)
        $dest = next(clients)$
      {ask all registered resources to abort}
      $dest = first(resources)$
      for all resources do
        send (rm-abort, $resource\_id|tid\_request|dest$) to self
        $dest = next(resources)$
else
  $result = notOk$
  send (tm-commit-reply, $result|seq|source$) {inform client whether request is being acted upon or not}
```

Protocol 19 Resource Manager protocol (RMP) – checking if resource is prepared to commit.

```
UPON RECEIVING MESSAGE ($ID|j$, in, a-deliver, $m|seq|source$) WHERE $m$ IS (rm-prepare, $tid\_request$): {received from transaction manager}
$put(state, resource\_id,$ prepare$)$
$result \leftarrow$ result of invoking prepare on resource $resource\_id$
send (rm-prepare-reply, $result|tid\_request|source|seq$) to self {send vote to transaction manager}
```

Protocol 20 Transaction Manager protocol (TMP) – decision phase of two-phase commit.

```
UPON RECEIVING MESSAGE (rm-prepare-reply, $prepare\_result|tid\_request|seq|source$): {receive vote from resource manager}
if $contains(trans\_log, tid\_request)$ then
  $get(trans\_log, tid\_request, clients|resources|state)$ {get current state}
  if $prepare\_result =$ ok then
    {record vote as yes}
    record resource vote as ok
  else
    {record vote as no}
    record resource vote as notOk
  if all resources have voted or any resource votes no then
    {we can make a decision on transaction commit or abort}
    if all resources vote to commit then
      $status \leftarrow$ commit
      $result = ok$
      $dest = first(resources)$
      for all resources do
        {tell all resources to commit changes and release locks}
        send (rm-commit, $resource\_id|tid_request$) to self
        send unlock, $resource\_id|tid\_request|dest$ to self
        $dest \leftarrow next(resources)$
    else
      $status \leftarrow$ abort
      $result = notOk$
      $dest = first(resources)$
      for all resources do
        {tell all resources to abort changes and release locks}
        send (rm-abort, $resource\_id|tid\_request$) to self
        send unlock, $resource\_id|tid\_request|dest$ to self
        $dest \leftarrow next(resources)$
    {record state of transaction}
    $put(trans\_log, tid, clients|resources|state))$
    $dest = first(clients)$
    for all clients do
      {tell all clients result of attempted commit}
      send (tm-commit-reply, $result|seq|dest$) to self
      $dest = next(clients)$
```

Protocol 21 Resource Manager protocol (RMP) – abort, and commit.

```
1 UPON RECEIVING MESSAGE (rm-abort, $resource\_id|tid\_request|seq|source$): {received from transaction manager}
2 $get(state, resource\_id, current\_state)$ {find out the transactional state of the resource}
3 if $current\_state$ = prepare or $current\_state$ = transactional then
4   {we can only abort if we are in a transaction}
5   invoke abort on resource $resource\_id$
6   $put(state, resource\_id,$ non-transactional$)$
7   send (rm-ack, $seq|source$) to self {send acknowledgement to transaction manager}

8 UPON RECEIVING MESSAGE (rm-commit, $resource\_id|tid\_request|seq|source$): {received from transaction manager}
9 $get(state, resource\_id, current\_state)$ {find out the transactional state of the resource}
10 if $current\_state$ = prepare then
11   {can only commit if already prepared}
12   invoke commit on resource
13   $put(state, resource\_id, strnon - transactional)$
14   send rm-ack, $tid\_request|seq|source$) to self {send acknowledgement to transaction manager}
```

5.3.4.6 Acknowledge Resource Manager Commit or Abort

Protocol 22 specifies how a transaction manager handles acknowledgement of resource manager commit or abort. When all acknowledgements have been received then it marks the transaction as ended.

Protocol 22 Transaction Manager protocol (TMP) – handling acknowledgements of commit or abort.

```
1 UPON RECEIVING MESSAGE (rm-ack-reply, $tid\_request|seq|source$):
2 if $contains(trans\_log, tid\_request)$ then
3   $get(trans\_log, tid\_request, clients|resources|state)$
4   count the number of resources
5   if all received then
6     $state \leftarrow$ end
7     $put(trans\_log, tid\_request, clients|resources|state))$
```

5.3.4.7 Abort

Protocol 23 specifies how a transaction manager implements abort when requested to abort by a client. Essentially it contacts all resources and asks them to abort, it then informs all clients of abort. Any client can force an abort of a transaction.

Protocol 23 Transaction Manager protocol (TMP) – requesting an abort.

```
Upon receiving message (tm-abort, $tid\_request|seq|source$): {received from client}
$result \leftarrow$ notOk
if $contains(trans\_log, tid\_request)$ then
   $get(trans\_log, tid, clients|resources|state)$
   if $state =$ begin and $contains(clients, source)$ then
      {valid abort request, transaction active and the client is a member of the transaction}
      $result \leftarrow$ ok
      $state \leftarrow$ abort
      $put(trans\_log, tid\_request, clients|resources|state))$
      {ask all resources to abort}
      $dest = first(resources)$
      for all $resources$ do
         send (rm-abort, $resource\_id|tid\_request|dest$) to self
         $dest \leftarrow next(resources)$
if $result =$ notOk then
   {unable to abort, inform requesting client}
   send (tm-abort-reply, $request\_tid|result|seq|source$) to self {sent to client}
else
   {inform all clients of abort}
   $dest = first(clients)$
   for all clients do
      {tell all clients result of attempted commit}
      send (tm-commit-reply, $result|seq|dest$) to self
      $dest = next(clients)$
```

5.3.5 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 that clients within the same transactional context take care of concurrency between them using an application specific protocol.

The lock compatibility scheme implemented by the protocol is the commonly used lock compatibility scheme of *one writer and multiple readers*. When an operation requests a lock for a resource via a resource manager within the context of a transaction then one of the following takes place (based on scheme in [28]):

- If the resource is not already locked, then the requested lock is granted.
- If the resource has a conflicting lock set by another transaction then the lock is not granted, for example a write lock is considered to conflict with a read lock.
- If the resource has a non-conflicting lock set by another transaction then the lock is granted.
- If the resource has already been locked in the same transaction then the lock is promoted, for example a read lock can be promoted to a write lock.

The LockingManager protocol (Protocol 24) specifies the distributed locking protocol.

5.3.6 Distributed Recovery Protocol

We propose a simple recovery protocol (the RecoveryManager protocol 25) based upon state transfer between replicas. This approach assumes that eventually a majority of transaction managers will recover and accordingly resource managers will commit or abort. We assume that no more that t replicas may fail and that some trusted process starts the recovery process. This recovery process may require complete reinitialisation of the host where the replica is executed as not only the state, but also the software, may have been corrupted. While the system is recovering we assume that all other replicas queue requests until recovery is over and the recovering replica is reintegrated into the group.

Protocol 24 Locking Manager protocol (LMP).

```
1  UPON RECEIVING MESSAGE $(ID|j, \texttt{in}, \texttt{a-deliver}, m|seq|source)$ WHERE $m$ IS
   $(\texttt{lock}, lock\_request, resource\_id, tid\_request)$: {request from client}
2  $get(lock\_log, resource\_id, lock|tids)$ {retrieve the current lock type and the transactions holding the
   shared lock}
3  if $lock = \emptyset$ then
4    $result \leftarrow ok; lock \leftarrow lock\_request; tid \leftarrow tid\_request$
5    $append(tids, tid\_request)$
6    $put(lock\_log, resource\_id, lock|tids)$
7  else if $lock\_request = read$ then
8    if $lock = read$ then
9      $result \leftarrow ok$
10   else
11     $\{lock = write\}$
12     $result \leftarrow notOk$
13 else
14   $\{lock_request = write\}$
15   if $lock = read$ then
16     if $length(tids) = 1$ then
17       {only one transaction holds read lock, so promote}
18       $result \leftarrow Ok$
19       $lock \leftarrow lock\_request$
20       $put(lock\_log, resource\_id, lock|tids)$
21     else
22       $result \leftarrow notOk$
23 send $(\texttt{lock-result}, result|seq|source$ to self {return result of request to client}

24 UPON RECEIVING MESSAGE $(\texttt{unlock}, resource\_id|tid\_request|seq|source)$: {request from transaction
   manager}
25 $lock \leftarrow get(lock\_log, resource\_id)$
26 if $tid\_request = tid$ then
27   $result \leftarrow ok; lock \leftarrow \texttt{none}; tid \leftarrow 0$
28   $put(lock\_log, resource\_id, lock)$
29 else
30   $result \leftarrow notOk$
31 send $(\texttt{lock-result}, result|seq|source$ to self {return result of request to transaction manager}
```

Protocol 25 Recovery Manager protocol (RMP)

```
1  INITIALIZATION:
2  $log \leftarrow$ initialise pointer to data structure representing the recovery log
3  {in the case of a transaction manager this is $trans\_log$,}
4  {in the case of a resource manager this is $lock\_log$ and $state$}
5  UPON RECEIVING MESSAGE (recovery-getStatus, $seq|source$):
6  send (recovery-status-reply, $trans\_log|seq$) to $P_{source}$ {return log to requesting replia}

7  UPON RECEIVING MESSAGE ($ID|j$, in, restart) FROM A TRUSTED SOURCE
8  send (recovery-getStatus) to own group

9  UPON RECEIVING MESSAGE (recovery-status-reply, $received\_log|source$)
10 $log \leftarrow received\_log$ {replaced current log with received log}
```

5.3.7 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 [73]). 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.

It has been proposed by the group communications community that distributed transactions are unnecessary [35, 62]. Their argument is based upon the argument that the atomicity property is guaranteed when using atomic broadcast and therefore transactions are not needed when dealing with updates to replica groups. However, it is not clear that these approaches address isolation, subtransactions or recovery after failure. Also it is not clear how serializability is maintained if transactions are allowed to interleave operations.

More recently it has been suggested to use group communications to implement transactions. For example, [52] model have implemented 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 [38]. However, it assumes a model structured in terms of replicated databases rather than a CORBA style transaction architecture.

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 for a CORBA style transaction architecture. This differentiates our work from approaches that make use of new or modified group communication primitives (for example, optimistic broadcast) [35, 62, 73]. 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.

Our system could be used to implement error confinement at the application level. An example of this approach in the database world is [43] where transactions are used to allow rollback or compensatory action after an intrusion has been detected.

5.4 Membership

This section describes the dynamic membership service. This service provides basically three operations: the addition of sites to a group, the removal of sites from the group due to a request from a member, and the removal of a site due to its failure.

5.4.1 Membership in the Asynchronous Model

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. Since this work is currently being developed, we only give a brief overview of these here; more details can be found in [68].

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 [68].

5.4.2 Membership with Support from the TTCB

This section is concerned with *site level membership*, i.e., with groups of hosts. The membership service handles basically three operations: the addition of sites to a group, and the removal of sites from the group either due to a failure or to a specific request from that site.

The membership service relies on a safety invariant that at most f out of $n = 3f+1$ sites are failed in each group. Some membership services for fail-silent systems that have been proposed in the literature handle not only these operations but also network partitions. In our case, with an arbitrary failure model, given the invariant above, it is not possible to handle partitions, since no partition would guarantee the invariant and be able to make progress or merge back the main partition [1]. In fact, handling partitions seams to be incompatible with an arbitrary failure model, unless we assume than no more than a fraction out of $f = \frac{n-1}{3}$ members can fail.

The membership service generates *views*, i.e., numbered events containing the group members. A view is generated by the service whenever the membership is changed due to a member join, leave or failure. We consider that a view is *defined* at a site if the site is in that view. A group of sites is created when the first member joins and installs the first view.

We present the membership service as if a site could be in a single group. We do this without loss of generality, to avoid constant references to the group identification.

The view a site S_i is an array V_i^n kept at that site with current members of the group. The index n reflects the n^{th} view of the group. The protocol guarantees that every correct site has the same view at every instant of logical time, i.e., after the delivery of the same (totally ordered) view in every site.

[1]We could assume that the invariant holds for any partition. However, this would imply, for instance, that if a single site became partitioned from the group then it would be necessarily correct!

The membership protocol is defined formally in terms of the following properties (inspired in [59]):

- *Uniqueness.* If views V_i^n and V_j^n are defined, and sites S_i and S_j are correct, then $V_i^n = V_j^n$.
- *Validity.* If site S_i is correct and view V_i^n is defined, then $S_i \in V_i^n$ and, for all correct sites $S_j \in V_i^n$, V_j^n is eventually defined.
- *Integrity.* If site $S_i \in V_i^n$ and V_i^{n+1} is not defined then at least one correct site detected that S_i failed or S_i requested to leave. If site $S_i \in V_i^{n+1}$ and V_i^n was not defined then at least one correct site authorized S_i to join.
- *Liveness.* If $\frac{2}{3}|V^n| + 1$ correct sites detect that S_i failed or receive a request to join, or one correct site requests to leave, then eventually V^{n+1} is installed, or the join is rejected.

Uniqueness guarantees that all correct sites in a group see the same membership. Validity guarantees that if a view is defined at a site then the site is in the view (often called Self-Inclusion property). Validity guarantees also that every correct site in a view will eventually install the view. Integrity prevents isolated malicious sites from removing or adding sites to the group. Liveness guarantees that the view changes when a number of correct sites detectes a failure, or a correct site wants to join or leave (or the join is rejected).

The *Byzantine Group Membership Protocol (BGM)* evolves at each site in three states: *Normal*, *Agreement* and *Stabilize*. BGM works roughly the following way. When a site joins a group it enters the Normal state. When another site wants to join or leave, or when a site is detected to be failed, certain *events* are generated and the protocol changes to the Agreement state. In this state, the sites of the current view try to agree on the next view, running the *View Change Agreement protocol* (VCA or agreement protocol, not to be confused with the TTCB Agreement Service). Each site proposes the view changes it thinks that should be performed. When an agreement between at least $(n - f)$ of those sites is reached, the protocol changes to the Stabilize state. That state *stabilizes* the communication according to the group semantics. When the communication is stabilized, the new view is installed.

BGM verifies the four properties above. An extra *economy principle* for BGM is that no *useless* executions of VCA should be performed. The reasons for this are (1) VCA uses communication and TTCB resources, and (2) during VCA group communication probably has to stop (depends on the group semantics). Therefore, the protocol enforces:

- Correct sites do not try to start an agreement without a *reasonable expectation* that a new view will be installed or a site join rejected.
- Malicious sites are prevented from starting a VCA, even if the maximum number of f failed sites are malicious and collude in trying to do so.

Let us illustrate the first objective looking at failure detection. Imagine that a site detected a failure. Even if all correct sites detect the failure, the detection is not simultaneous. Therefore, if one site detects a failure ahead and tries immediately to change the view proposing the removal of the failed site, probably the other sites will reject the change. This VCA execution would be useless. The second objective is clear.

The following section presents the site failure model. Section 5.4.2.2 gives two *basic* multicast protocols used by BGM. Then, three sections describe what happens when a site is detected to be failed, or a site wants to join or leave (Sections 5.4.2.3 – 5.4.2.5). These sections describe how the events that feed the BGM protocol are generated. BGM is presented in Section 5.4.2.6.

5.4.2.1 Site Failure Model

We denote by *site* the software entity that executes the protocols in a host. Therefore, we are interested here in the failure model of sites. We say generically that a site is *correct* if it follows the protocol. There are several circumstances, however, that may lead to the site failure. For instance, a site can crash (e.g., due to a host crash) or can be corrupted by a hacker. In an arbitrary failure model, which is what is being considered in this document, sites can fail arbitrarily. Therefore, a site can simply stop working, can send messages disregarding the protocol, can delay messages, and even collude with other malicious sites trying to break the protocol.

A site uses a number of secrets for authentication and to protect its communication: the pair $(eid, secret)$ used to communicate with the TTCB and a set of symmetric keys to communicate with other sites. If an attacker manages to discover these secrets, we consider that the site is failed, since it can be impersonated.

An attacker with access to the network (or even to the host) may be able to disrupt the communication of one or more sites. In that case we also consider those sites to be failed. In channels with only accidental faults it usually considered that no more than Od messages are corrupted/lost in a reference interval of time. Od is the *omission degree* and tests can be made in concrete networks to determine Od with the desired probability [72]. Given this parameter, some protocols will retransmit message $Od+1$ times; if the message is not received then the communication is being disrupted by an attacker and the site is considered to be failed. If the communication is very delayed then we also consider it failed.

5.4.2.2 Basic Multicast Protocols

This section presents two multicast protocols that guarantee the following properties:

- *Validity:* If a correct process multicasts a message M, then some correct process in $group(M)$ eventually delivers M.
- *Integrity:* For any message M, every correct process p delivers M at most once and only if p is in $group(M)$, and if $sender(M)$ is correct then M was previously multicast by $sender(M)$.

The predicate $sender(M)$ gives the message field with the sender, and $group(M)$ gives the "group" of processes involved, i.e., the sender and the recipients (note that we consider that the sender also delivers). Here, *process* should be read *site*. The site model can be found in Section 5.4.2.1.

These two properties are common to reliable multicast protocols but the two protocols do not guarantee the Agreement property, i.e., that either all correct processes deliver the message or none [36]. The two protocols are very simplified versions of the BRM protocol.

We provide two similar protocols to be used in different cases. Mcast$_{MAC}$ does not use the TTCB but requires the sites to share symmetric keys (Protocol 26). Since these keys are not always available, Mcast$_{TTCB}$ uses the TTCB to avoid this requirement (Protocol 27).

Mcast$_{MAC}$ uses message authentication codes (MAC) to guarantee the integrity and authenticity of messages [44]. This type of signature is based on symmetric cryptography, that requires a different secret key to be shared between every pair of sites. Since messages are multicast to a set of processes, a message does not take a single MAC but a vector with one MAC per recipient [21]. Mcast$_{MAC}$ multicasts M $(Od+1)$ times to tolerate accidental faults in the network. If an attacker disrupts the communication, malicious faults, one of the sites is considered to be malicious (Section 5.4.2.1).

Mcast$_{TTCB}$ uses the TTCB Agreement service to guarantee the integrity and authenticity of messages. The sender uses the service to send the recipient a reliable hash of the message. Given the properties of hash functions, the recipient can verify if the message

Protocol 26 Mcast$_{MAC}$ protocol.

```
M ←(my-eid, elist, data); {Sender}
mac-vector ←calculate macs of M;
M ←M ∪ mac-vector;
repeat
   multicast M to elist except sender;
until done $Od + 1$ times
deliver(M);

When (message M received) and (M not delivered) {Recipient}
if M.mac-vector[my-eid] is ok then
   deliver(M);
```

Protocol 27 Mcast$_{TTCB}$ protocol.

```
M ←(my-eid, elist, TTCB_getTimestamp() + $T_1$, data); {Sender}
repeat
   multicast M to elist except sender;
until done $Od + 1$ times
propose ←TTCB_propose(M.elist, M.tstart, TTCB_TBA_RMULTICAST, H(M));
deliver(M);

read(M); {Recipient}
propose ←TTCB_propose(M.elist, M.tstart, TTCB_TBA_RMULTICAST, H(M));
if propose.error = OK then
   repeat
      decide ←TTCB_decide(propose.tag);
   until (decide.error = TTCB_TBA_ENDED);
   while H(M) $\neq$ decide.value do
      read(M);
   deliver(M);
```

it received is correct [44]. Mcast$_{TTCB}$ also multicasts the messages $(Od + 1)$ times for the same reason as Mcast$_{MAC}$. These protocols are used to transmit short control messages, so to multicast $(Od + 1)$ copies is not a high overload.

5.4.2.3 Removing Failed Sites

Each site has a Site Failure Detector module (SF, Figure 1.1). SF gives failure detection events about sites (E_{FD}*(site)*) to the Site Membership module (SM). The membership protocol, BGM, does not use the failure detector to guarantee the safety or liveness of the agreement protocol, VCA, but only to remove failed sites from the membership. Chandra and Toueg classified crash failure detectors in terms of two properties: Accuracy (ability to avoid false detections) and Completeness (ability to detect actual failures) [24]. BGM assumes that the failure detector has Strong Accuracy: no site is detected failed before it fails. The reason for this is that the membership protocol uses these detections to remove sites and only failed sites can be really removed. BGM does not require a specific type of Completeness. A site is removed when *enough* sites detect its failure.

BGM is independent of the types of failures detected. It can work with failure detectors that detect from crash to Byzantine failures. We present one possible failure detector that detects Byzantine failures in Section 5.4.2.8. We do not consider site recoveries since we assume a failure detector with Strong Accuracy. However, a removed site can try to rejoin the group executing the join protocol.

Taking in account the economy principle given previously, when a site detects a failure, event E_{FD}*(site)*, it multicasts that event using Mcast$_{MAC}$ (see Protocol 31). The delivery of the multicast at a site is an event E_{dlvFD}*(sender,object)* (*sender* is the site that generated the detection; *object* is the site whose failure was detected). When a site delivers the $(2f + 1)^{th}$ event of the kind, about the same site but incoming from different sites, it starts VCA. Waiting for $(2f + 1)$ events guarantees that the view change will be performed (economy principle).

5.4.2.4 Site Join

In the crash model, when a site wants to join a group it has to find out a non-crashed contact site. In the arbitrary failure model, joining a group is more complex since (1) ex-members may be malicious and provide false information about membership; (2) finding out actual members is not enough since some members may also be malicious. The solution is to contact a number of sites from which a majority is correct. Since we do not make assumptions on the number of failed sites in the universe of possible sites, these sites have to be members of the group. However, when the site wants to join it does not know the membership of the group, so how can it contact the members?

A solution for this problem is to have a trusted third party in charge of providing membership information. This *membership server* (MServer) has to be secure and to have high availability. We assume the existence of MServer.

We assume that sites can get a reliable copy of the MServer's address and public key K_u. This key corresponds to a private key K_r known only by the server and is used for every site to establish a shared symmetric key $K_{\langle MS,S_i \rangle}$. This key can be used to simulate a *secure channel* between the site and MServer, in a similar way to the entity-TTCB secure channel [26].

MServer is a "database" that contains pairs *(siteGroupId, view)*. The database is used in the following way:

1. When a group is created, the initial member, S_1 establishes a secure channel with MServer (in case it does not have one yet) and gives MServer the group identification and the initial view, that contains only S_1 itself.
2. When a new view V^{n+1} is installed, each correct member site $S_i \in V^n$ sends MServer a pair *(siteGroupId,* V^{n+1}*)*. The group view in MServer is changed when it receives $\frac{|V^n|-1}{3} + 1 = (f+1)$ identical pairs. The information in MServer is reliable due to the assumption that no more than f sites in a view fail. Therefore, receiving $(f+1)$ copies guarantees that this information is correct.

When a site S_i wants to join a group the following happens (Protocol 28):

1. S_i contacts MServer and establishes a secure channel with it.
2. S_i makes a request to MServer and gets a view V^n of the group it wants to join.
3. S_i sends a request to join to the members in V^n using Mcast$_{TTCB}$ [2]; the request can contain additional information used for the sites to grant or deny *authorization* to join; several authorization schemes can be used so we do not give details about it.
4. Each site that gets the request (event $E_{dlvReqJoin}$*(sender,credential)* in Protocol 31, where *sender* is the joining site and *credential* is information used for authorization) does Mcast$_{MAC}$ of the request to all group members. The delivery of this message is

[2]It uses Mcast$_{TTCB}$ instead of Mcast$_{MAC}$ because it does not have shared keys with the other sites yet.

the event $E_{dlvMcastJnRq}$*(sender,object)* (*sender* is the site that sent the Mcast$_{TTCB}$; *object* is the site that wants to join).

5. When a site receives $(2f + 1)$ events for the same site, it changes to the Agreement state. Waiting for $(2f + 1)$ events guarantees that the join will be explicitly accepted or rejected by the group (economy principle).

6. When the agreement (VCA) terminates the group sites inform S_i if it can join or not.

Protocol 28 Joining site protocol.

```
WHEN SITE $S_i$ CALLS JOINSITEGROUP(SITEGROUPID, CREDENTIAL)
establishKeyMServer(); {if not established yet}
$V^n$ ←MserverGetView(siteGroupId);
Mcast$_{TTCB}$($V^n$, RequestToJoin);

WHEN $(f+1)$ REPLIES WITH SAME ANSWER DELIVERED
if Accepted then
  establishKeys($V^n$);
  start BGM;

WHEN ESTABLISHKEYS(L) IS CALLED {L is a list of sites}
for all sites $S_i \in L$ that not me do
  if my-eid < eid($S_i$) then
    elist = (my-eid,eid($S_i$));
  else
    elist = (eid($S_i$),my-eid);
  propose ←TTCB_propose(elist, $tstart_{ke}$, TTCB_TBA_KEY, 0);
  repeat
    decide ←TTCB_decide(propose.tag);
  until (decide.error = TTCB_TBA_ENDED);
  $K_{S_i}$ = decide.value;
```

Sites that join a group need to receive the new view information (since they did not participate in the VCA) and to establish shared keys with all the other group members. This section details how this is done.

When the BGM protocol state changes from Agreement to Stabilize and there are new sites, the (correct) sites that transited from the previous view do a Mcast$_{TTCB}$ to the new sites. This multicast contains two pieces of information: the new view V^{n+1} and an instant $tstart_{ke}$. When the joining site gets $(f + 1)$ identical copies of that information it installs the view and goes to the Normal state.

The instant $tstart_{ke}$ has to be defined deterministically so that all correct sites send the same value. It is given by $tstart_{ke} = tstart_f + T_{ke}$, where $tstart_f$ is the $tstart$ value used in the last TTCB Agreement of the VCA protocol that decided the new view and T_{ke} is a constant.

$tstart_{ke}$ is used to establish the shared keys. This operation is based on the TTCB Agreement service and a decision function $decision = TTCB\_TBA\_KEY$ that establishes a shared key. A non-optimized version of the key establishment algorithm is shown in Protocol 28 [3]. Sites that transited from the previous view execute the same protocol but establishing keys only with the sites that joined.

5.4.2.5 Site Leave

When a site decides to leave a group, it calls `leaveSiteGroup(siteGroupId)`. This call generates an internal E_{leave} event. The site does an Mcast$_{MAC}$ with the event and starts a VCA to change the view. VCA and Mcast$_{MAC}$ authenticate the sites, therefore correct sites know if it is the site that is trying to leave or if it is a malicious site trying to remove it. If a site delivered an Mcast$_{MAC}$ with a $E_{dlvLeave}$*(site)* event, when it proposes the view change it requests that leave. Requests to leave are always accepted by all correct sites.

5.4.2.6 BGM Core Protocol

The *Byzantine Group Membership protocol (BGM)* (Protocol 31) has the three states mentioned above: Normal, Agreement, Stabilize. BGM handles a set of events that we resume here:

- E_{FD}*(object)*: local failure detector detected the failure of site *object*
- E_{dlvFD}*(sender,object)*: delivered an Mcast$_{MAC}$ with an indication that site *sender* detected the failure of site *object*
- $E_{dlvReqJoin}$*(sender,credential)*: delivered an Mcast$_{TTCB}$ with a join request from site *sender* (*credential* is information used for authorization)
- $E_{dlvMcastJnRq}$*(sender,object)*: delivered an Mcast$_{MAC}$ from site *sender* with a request to join for site *object*
- E_{leave}: local site wants to leave group
- $E_{dlvLeave}$*(sender)*: delivered an Mcast$_{MAC}$ with a request to leave from site *sender*
- $E_{dlvGMcast}$*(sender, view number, Bvc)*: delivered a GMcast message from site *sender* for *viewnumber*; *Bvc* is a bag with view change proposals

[3]This version is more readable than the optimized version. The latter executes the TTCB agreements in parallel.

The Normal state corresponds to a stable membership and normal communication. When certain combinations of the events above happen in a site, that site engages in the *View Change Agreement protocol (VCA or agreement protocol)* (Protocol 30) and the state changes to Agreement. This engagement consists in proposing a set of view changes using the *Group Membership Multicast protocol (GMcast)* (Protocol 29). There is a set of GMcast tasks per VCA: one used by the process to propose (send) view changes; one per each other site in the current view that proposes a view change (at most f failed sites may not propose).

The combinations of events that cause a site do GMcast are (Protocol 31):

- the delivery of $(2f+1)$ $E_{dlvMcastJnRq}$*(sender,object)* events from different sites, about the site *object*
- the delivery of $(2f+1)$ E_{dlvFD}*(sender,object)* events from different sites, about the site *object*
- the event E_{leave}
- the delivery of a GMcast with a request to leave from the sender of the GMcast
- the delivery of $(f+1)$ GMcasts

The fact that the protocol handles these combinations of events instead of the events themselves, e.g., the delivery of $(f+1)$ GMcasts instead of the delivery of a single one, is a consequence of the economy principle.

The GMcast protocol is used to propose *view changes*. These view changes are:

- VC_{Join}*(site,accept)*: *site* wants to join; *accept* indicates if the site is authorized to join or not
- $VC_{RmvFailed}$*(site)*: proposes the removal of *site* since it is failed
- VC_{Leave}*(site)*: remove *site* from the group

View Change Agreement

VCA reaches agreement in a distributed way, i.e., the decision of a result is taken locally by each site. At each site, VCA selects a set of at least $(n-f) = (2f+1)$ GMcasts, out of $n = |V^n|$ GMcasts that should be sent (see Protocol 30). The function *vote* chooses from these GMcasts which view changes are performed:

- a site S_i with at least $(f + 1)$ $VC_{RmvFailed}(S_i)$ proposals is removed
- a site S_i with at least $(f + 1)$ $VC_{Join}(S_i, accept)$ proposals accepting/rejecting it is allowed/rejected to join
- a site S_i with (1) a $VC_{Leave}(S_i)$ sent by S_i or (2) at least $(f + 1)$ $VC_{Leave}(S_i)$ is removed

GMcast is basically a reliable multicast protocol with some similarities to BRM [25] (Protocol 29). Sites use GMcast to propose view changes. Since GMcast is a reliable multicast: (1) all correct sites deliver the same view change proposals; (2) if a correct sender transmits a view change proposal, then all correct sites deliver that message [25]. Additionally to being a reliable multicast, each GMcast task periodically (every $Tagr$) makes a TTCB_propose for a TTCB agreement with the hash of the message it sent or received. When GMcast gets the result of one of these TTCB agreements (call to TTCB_decide), it blocks calling the function *block*. When all GMcast tasks corresponding to a VCA block, VCA processes the result of these TTCB agreements and unblocks them. These TTCB agreements are used by VCA to choose the above mentioned $(n-f)$ messages which are used to select view changes (see Protocol 30).

The comparison of the TTCB agreements puts some difficulties, addressed in Protocols 29 and 30. The *first* is that a malicious process can try to send a GMcast message with a $tstart$ far in the future. This would make all GMcast tasks block waiting for the result of that TTCB agreement. To avoid this, if $tstart$ is after $t + Tagr$, where t is the present instant, it is normalized (line 8 in Protocol GMcast). To be normalized means to be put in the interval $[t, t+Tagr]$. If $tstart$ is "too much" in the past, it is also normalized (for instance, if it is older than the previous view change). The *second* difficulty is that GMcast message may be received in a site after its TTCB agreement terminates, i.e., after the other GMcast tasks for the same VCA to unblock. When the other tasks block again, they will be blocked in TTCB agreement with different $tstart$'s. The solution for this is to unblock only the late task while keeping those ahead blocked (lines 16–18 in VCA). The late one will block right away and it will be possible to compare the results of corresponding TTCB agreements.

5.4.2.7 Membership Protocol and FLP

Fischer, Lynch and Paterson proved that no deterministic protocol can solve the agreement problem in asynchronous systems if even a single process is allowed to crash [34]. In practical systems this FLP impossibility result has to be circumvented, for instance using randomization [57], partial-synchrony [32] or unreliable failure detectors [22]. The

Protocol 29 Group Membership multicast protocol (GMcast).

```
WHEN GMCAST(V, $n_{view}+1$, BVC) CALLED {task $S_i$ = self}
tstart$_0$ ←TTCB_getTimestamp() + $T_1$;
M ←(DAT, my-eid, elist($S_i$, $V^n$), tstart$_0$, ($n_{view}+1$,Bvc));
multicast M to $V^n \backslash S_i$; n-sends ←1;
goto Common-code;

WHEN M FOR A NEW GMCAST RECEIVED {task $S_i$ = sender(M)}
read_blocking(M); n-sends ←0;
tstart$_0$ ←normalize(M.tstart); {only if out of bounds}
{COMMON-CODE}
propose ←TTCB_propose(M.elist, tstart$_0$, TTCB_TBA_RMULTICAST, H(M));
**repeat**
   decide ←TTCB_decide(propose.tag);
**until** (decide.error = TTCB_TBA_ENDED);
**if** (decide.proposed-ok contains all recipients) **then**
   GMcast-deliver M;
ret ←block(tstart$_0$, decide.proposed-ok);
**if** ret $\neq$ END **then**
   tstart$_0$ ←tstart$_0$ + Tagr;
   propose ←TTCB_propose(M.elist, tstart$_0$, TTCB_TBA_RMULTICAST, H(M));
M-deliver ←$\perp$; hash ←decide.value;
mac-vector ←calculate macs of (ACK, my-eid, M.elist, M.tstart, decide.value);
M-ack ←(ACK, my-eid, M.elist, M.tstart, mac-vector);
n-acks ←0; ack-set ←eids in decide.proposed-ok;
t-resend ←TTCB_getTimestamp();
**repeat**
   **if** (M.type = DAT) and (H(M) = hash) **then**
      **if** M not delivered **then**
         GMcast-deliver(M);
      M-deliver ←M; ack-set ←ack-set $\cup$ {my-eid};
      **if** (my-eid $\notin$ decide.proposed-ok) and (n-acks < Od+1) **then**
         multicast M-ack to $V^n \backslash S^i$; n-acks ←n-acks + 1;
   **else if** (M.type = ACK) and (M.mac-vector[my-eid] is ok) **then**
      ack-set ←ack-set $\cup$ {M.sender};
   **if** (M-deliver $\neq \perp$) and (TTCB_getTimestamp() $\geq$ t-resend) **then**
      multicast M-deliver to $V^n \backslash \{\{S_i\} \cup$ack-set}
      t-resend ←t-resend + Tresend; n-sends ←n-sends + 1;
   **if** ret $\neq$ END **then**
      decide ←TTCB_decide(propose.tag);
      **if** decide.error = TTCB_TBA_ENDED **then**
         ret ←block(tstart$_0$, decide.proposed-ok);
         **if** ret $\neq$ END **then**
            tstart$_0$ ←tstart$_0$ + Tagr;
            propose ←TTCB_propose(M.elist, tstart$_0$, TTCB_TBA_RMULTICAST, H(M));
   read_non_blocking(M); {sets M = $\perp$ if there are no messages to be read}
**until** ((ack-set contains all recipients) or (n-sends $\geq$ Od+1)) and (ret = END);
```

Protocol 30 View Change Agreement protocol (VCA).

```
WHEN ALL GMCAST($S_i$) CORRESPONDING TO A VCA CALLED *block(tstart$_i$, proposed-ok$_i$)*
if $\forall_{S_i,S_j \in V^n}$ : tstart$_i$ = tstart$_j$ = tstart then
  Bts ←Bts ∪ {tstart};
  for all $S_i \in V^n$ do
    if ∃ task GMcast($S_i$) then
      A[i] ←proposed-ok$_i$; {A[]: array to save results of agreements}
    else
      propose ←TTCB_propose(elist($S_i$, $V^n$), tstart, TTCB_TBA_RMULTICAST, 0);
      repeat
        decide ←TTCB_decide(propose.tag);
      until (decide.error = TTCB_TBA_ENDED);
      A[i] ←decide.proposed-ok; {⊥ if error}
else {handle tasks still in earlier TTCB agreements}
  tstart$_{min}$ ←tstart$_i$ : $\forall_{S_i,S_j \in V^n}$ tstart$_i$ ≤ tstart$_j$;
  if tstart$_{min}$ ∈ Bts then {that TTCB agreement was already handled}
    for all $S_i$ : tstart$_i$ = tstart$_{min}$ do
      wake up task GMcast($S_i$) returning CONTINUE;
      return;
  else {that TTCB agreement was not handled yet}
    for all $S_i \in V^n$ do
      if tstart$_i$ = tstart$_{min}$ then
        A[i] ←proposed-ok$_i$;
      else
        propose ←TTCB_propose(elist($S_i$, $V^n$), tstart$_{min}$, TTCB_TBA_RMULTICAST, 0);
        repeat
          decide ←TTCB_decide(propose.tag);
        until (decide.error = TTCB_TBA_ENDED);
        A[i] ←decide.proposed-ok;
Bmsg ←⊥;
for all $S_i \in V^n$ do
  if #{sites that proposed-ok for GMcast($S_i$)} ≥ f+1 then
    Bmsg ←Bmsg ∪ {$S_i$}; {Bmsg saves senders of messages chosen}
if #Bmsg ≥2f+1 then
  wake up tasks returning END;
else
  wake up tasks returning CONTINUE; {wait for next set of TTCB agreements}

WHEN (#BMSG $\geq 2f+1$) AND (GMCAST DELIVERED ALL MESSAGES IN BMSG)
VCA-deliver vote(Bmsg); {VCA protocol ends}
```

Protocol 31 Byzantine Group Membership protocol (BGM).

```
INITIALIZATION:
n$_{view}$ ←1; {view number} $V^1$ ←$S_1$; {view} Bvc ←⊥; Bfd ←⊥; Bjn ←⊥;

WHEN $E_{FD}(object)$
Bvc ←Bvc ∪ {$VC_{RmvFailed}$(object)}; {Bvc: bag with view changes the site wants to do}
Mcast$_{MAC}$($V^n$, FD(object));

WHEN $E_{dlvFD}(sender, object)$ AND ($FD(sender, object) \notin$ BFD)
Bfd ←Bfd ∪ {$VC_{RmvFailed}$(sender,object)}; {Bfd: bag with incoming failure detections}
if (state = NORMAL) and (#{$VC_{RmvFailed}$(s,o) ∈ Bfd : o = object} ≥ 2f+1) then
   GMcast($V^n$, n$_{view}$+1, Bvc); state ←AGREEMENT;

WHEN $E_{dlvReqJoin}(sender, credential)$
Bvc ←Bvc ∪ {$VC_{Join}$(sender,accept(sender))}; {accept(sender) indicates if the site authorizes the join}
Mcast$_{MAC}$($V^n$, Join(sender));

WHEN $E_{dlvMcastJnRq}(sender, object)$
Bjn ←Bjn ∪ {$E_{dlvMcastJnRq}$(sender,object)}; {Bjn: bag with incoming requests to join}
if (state = NORMAL) and (#{$E_{dlvMcastJnRq}$(s,o) ∈ Bfd : o = object} ≥ 2f+1) then
   GMcast($V^n$, n$_{view}$+1, Bvc); state ←AGREEMENT;

WHEN $E_{leave}$
Bvc ←Bvc ∪ {$VC_{Leave}$(self)}; Mcast$_{MAC}$($V^n$, Leave(self));
if state = NORMAL then
   GMcast($V^n$, n$_{view}$+1, Bvc); state ←AGREEMENT;

WHEN $E_{dlvLeave}(sender)$
Bvc ←Bvc ∪ {$VC_{Leave}$(sender)};

WHEN (STATE = NORMAL) AND (($E_{dlvGMcast}(sender, n_{view}+1, Bvc'')$ AND $VC_{Leave}(sender)$) OR
   (($(f+1)^{th}$ $E_{dlvGMcast}(-, n_{view}+1, Bvc'')$))
GMcast($V^n$, n$_{view}$+1, Bvc); state ←AGREEMENT;

WHEN (STATE = AGREEMENT) AND (VCA DELIVERS BVC')
Bvc ←Bvc \ Bvc'; Bfd ←Bfd \ {$VC_{RmvFailed}$(s,o) ∈ Bfd : $\exists_{VC_{RmvFailed}(o')\in Bvc'}$, o = o'};
Bjn ←Bjn \ {$E_{dlvMcastJnRq}$(s,o) ∈ Bjn : $\exists_{VC_{Join}(o',-)\in Bvc'}$, o = o'};
if $\exists_{VC_{Join}(s,a)\in Bvc'}$ : a = Accepted then
   L ←{s: $VC_{Join}(s,a) \in Bvc'$ and a = Accepted}; Mcast$_{TTCB}$(L, view info); establishKeys(L);
if $\exists_{VC_{Join}(s,a)\in Bvc'}$ : a = Rejected then
   L ←{s: $VC_{Join}(s,a) \in Bvc'$ and a = Rejected}; Mcast$_{TTCB}$(L, Rejected); establishKeys(L);
if Bvc' contains only rejected joins then
   state ←NORMAL; {stay in the same view}
else
   state ←STABILIZE; {stabilize to install new view}

WHEN (STATE = STABILIZE) AND (FINISHED STABILIZING)
n$_{view}$ ←n$_{view}$+1;
if ($\forall_{S_i\in V^n}$, #{$VC_{RmvFailed}$(s,$S_i$) ∈ Bfd} < 2f+1) and ($\forall_{S_i}$, #{$E_{dlvMcastJnRq}$(s,$S_i$) ∈ Bjn} < 2f+1)
   and ($E_{leave} \notin$ Bvc) then
   state ←NORMAL;
else
   GMcast($V^n$, n$_{view}$+1, Bvc); state ←AGREEMENT;
```

membership problem was proved to be equivalent to agreement so it is also bound by FLP [23]. Therefore, how does VCA circumvent the FLP impossibility result?

The first thing to be noted is that the system model considered is not purely asynchronous (see MAFTIA deliverables D1 and D23). The payload system has unreliable timeliness, so it is asynchronous, but the TTCB is a synchronous, timely subsystem, that provides synchronized clocks and a set of timely services (timely at the TTCB interface).

We want to show how VCA circumvents FLP. Recall how VCA works (Protocol 30). VCA does not try to get contributions from all group sites but only from $(n - f)$. Each correct site makes a view change proposal using GMcast. Periodically every correct site tries to TTCB_propose a hash of the GMcast it sent and the others it received. Therefore, periodically a set of TTCB agreements give every correct site a consistent *snapshot* of which sites delivered which GMcasts messages.

Since every correct site gets the same snapshot, they can reach agreement applying a deterministic stop (and decision) criterium. This set of TTCB agreements work as a sort of Unreliable Failure Detector, with weak accuracy but that gives precisely the same suspicions at every site. With the passage of time, the probability that all correct sites deliver all GMcasts tends to 1. The stop criterium is: stop when $(f+1)$ sites proposed the hashes of at least the same $(2f+1)$ messages. This condition means two things: (1) at least one correct site delivered those messages (since only f can fail), therefore all correct sites will deliver the messages (GMcast is a reliable multicast); (2) at least half of the messages chosen plus one were sent by correct sites (since at most f messages can be sent by failed sites). Therefore, given this condition all correct sites will agree on a majority of messages sent by correct sites, and all correct sites will eventually deliver the messages chosen.

5.4.2.8 Site Level Failure Detection

GMP uses a site failure detector to remove failed sites from the group. The Site Failure Detector module (SF, see Figure 1.1) is an abstraction that can hide several mechanisms. Here we consider that the module encompasses two classes of mechanisms:

- *Crash FD.* Detects if a site crashed.
- *Byzantine FD.* Byzantine faults depend on the protocols being executed [4]. Therefore, a Byzantine FD must be designed or tailored to monitor a specific protocol and its Byzantine failures [31]. In this context, the protocol to be monitored is BGM.

[4]Byzantine or arbitrary faults encompass crash faults, but these are not considered here.

Additionally to these two kinds of failure detection we could also consider an Intrusion Detection System (IDS). An IDS can not detect if a protocol fails, i.e., if it performs invalid interactions, but can detect if a host is successfully attacked (a fault) or if it is corrupted (an error). These faults and errors may lead to the site failure, therefore they can be detected as a preliminary indication that the site may fail. We do not consider this kind of detector here.

The membership protocol requires a failure detector with Strong Accuracy: no site is detected failed before it fails. The effective removal of failed sites from the membership will depend on the failure detector Completeness.

Crash Failure Detector

In asynchronous system it is not possible to distinguish a crashed host from a slow host. Therefore, crash failure detection in asynchronous systems is intrinsically unreliable. The TTCB provides a function call to detect crashed local TTCBs, and consequently crashed hosts (when a host crashes its local TTCB is forced to crash and vice-versa). However, a site (software) may crash without its host crashing, so that function is not particularly useful here.

The Crash FD we propose is based on the TTCB Agreement service (Protocol 32). The correctness of the algorithm relies on a *host synchrony assumption*: there are maximum bounds on the times for a process to be scheduled by the operating system and for its execution. Usually, crash FDs rely on network synchrony assumptions, i.e., in maximum bounds for message transmissions (timeouts). Host synchrony assumptions have a much higher coverage in current systems.

Protocol 32 Crash failure detection algorithm (at every host).

```
{all sites start this task with the same $t_{speak}$ (next instant to say “I am alive”)}
loop
  t ←TTCB_getTimestamp();
  if $t > t_{speak}$ then {$T_{max}$: maximum time for the task to be scheduled and call TTCB_propose}
    propose ←TTCB_propose(elist, $t_{speak} + T_{max}$, TTCB_TBA_RMULTICAST, 0); {“I am alive”}
    repeat
      decide ←TTCB_decide(propose.tag);
    until (decide.error = TTCB_TBA_ENDED);
    for all eid ∈ elist do
      if bit corresponding to eid in decide.proposed-any is reset then
        event $E_{FD}$(site(eid));
    $t_{speak}$ ←$t_{speak} + T_{detect}$; {$T_{detect}$: detection period; $T_{detect} \geq T_{max} + T_{agreement}$}
```

The idea of the algorithm is the following. Each site waits for t_{speak} and tries to propose a value to the TTCB Agreement service. Given the assumption above, every correct site will propose before $tstart = t_{speak} + T_{max}$, where T_{max} is the maximum take for the site code to be scheduled and run. When the TTCB agreement terminates, every site will get a mask with the sites that proposed (and also a value and another mask, but these are not used here). The sites that did not proposed are crashed or malicious (if they did not propose on purpose) and are detected failed equally by all correct sites.

A hacker with access to a host can try to attack the corresponding site in such a way that it does not propose in time. This is once again the problem of not being possible in an asynchronous system to distinguish a delay from a crash, although in this case the problem are delays in hosts, not in the network. A site that is delayed in such a way is considered faulty (see Section 5.4.2.1), so the algorithm will detect its failure correctly, although the failure is not a crash.

The coverage of this algorithm can be improved in several ways. For instance, T_{max} can be increased, or a failure detection can be delayed until a site is detected failed in N successive rounds. However, these solutions delay the detections.

This failure detector is a *perfect failure detector ($\mathcal{P}$)*, if the assumptions hold:

- *Strong Completeness:* Eventually every site that fails is permanently suspected by every correct site.
- *Strong Accuracy:* No site is suspected before it fails.

Doudou et al. use Muteness Failure Detectors instead of crash [31]. The idea is to detect not only crashes but also malicious muteness. These detectors detect if a process q stops sending messages from protocol $\mathcal{A}$ to p. We do not detect these malicious failures because Strong Accuracy would require synchrony assumptions on the network to define maximum delays [5].

Byzantine Failure Detector

The Byzantine FD is based on the idea of *monitored Byzantine failures*, i.e., the FD does not try to detect all Byzantine failures but only a subset that it monitors. Detecting all Byzantine failures of a component is usually considered to be impossible, since it would require a reliable detector capable of observing all component interactions in a timely manner, and comparing those interactions with the component behavior specification. Monitored Byzantine failures are equivalent to Kihlstrom et al. *detectable Byzantine*

[5]Doudou et al. use them to solve Consensus so they need only Eventual Weak Accuracy.

faults, however, not all "detectable" failures are practical to detect, so we prefer the word "monitored" [41].

The Byzantine FD is a *perfect failure detector ($\mathcal{P}$)*, characterized by the following two properties:

- *Strong Accuracy:* No correct site detects that another site failed before the latter produces a monitored Byzantine failure.
- *Strong Completeness:* Eventually every site that produces a monitored Byzantine failure is permanently suspected by every correct site.

This definition imposes a limit on the failures the failure detector can monitor. Strong Completeness requires that the failure detector monitors only failures that can be detected by all correct sites. This is not a problem since BGM is based on GMcast that does reliable multicast, therefore every correct site delivers the same messages.

The classes of failures detected by the Byzantine FD are:

- *Semantic failures.* A site sends a message whose meaning is incorrect (not as specified).
- *Syntactic failures.* A site sends a message whose format is incorrect.

Examples of semantic failures are: GMcast messages sent to a view with an *elist* that does not match the membership; propose a view change to remove a site that is not in the membership. Examples of syntactic failures are: GMcast messages without *elist*; GMcast messages with *elist* in an invalid format.

The Byzantine FD algorithm is simple, so we skip the code for brevity. When a GMcast message is received, its syntax and semantics are tested and and event is generated if it is incorrect.

5.4.2.9 Participant Level Membership and Failure Detection

Groups of participants are mapped into groups of sites. This means that the participant membership module (PM) uses the services of the site membership module (SM) (Figure 1.1).

The participant membership protocol is shown in Protocol 33. The protocol simply adds or removes participants from a bag with the local participants in a group (of participants). If the participant is the first to join a group in a site, the site joins the corresponding group of sites (line 10). If the group of sites does not exist it is created. If the site is the last to leave a group of participants in a site, the site leaves the corresponding group of sites. If it is the last site to leave, the group is destroyed.

As for site failure detection, we consider that the participant failure detector has Strong Accuracy, i.e., that no participant is detected failed before it really fails. Participant failure detection is a local matter. If a participant is detected failed it is removed from all groups.

Protocol 33 Participant Level Membership (at every host).

```
1  Initialization:
2  LV $\leftarrow \bot$; {contains a bag with the local participants of each group}

3  When participant calls joinGroup(id, participantGroupID, credential)
4  **if** $\exists$ participantGroupID entry in LV **then**
5     **if** credential accepted **then** {otherwise an error is returned}
6        LV[participantGroupID] $\leftarrow$LV[participantGroupID] $\cup$ {id};
7  **else**
8     establishKeyMServer(); {if not established yet}
9     siteGroupId $\leftarrow$MserverGetSiteGroupId(participantGroupID); {$\bot$ if does not exist}
10    joinSiteGroup(siteGroupId, credential); {creates a new one if does not exist}
11    **if** Accepted **then** {otherwise an error is returned}
12       create LV[participantGroupID]; LV[participantGroupID] $\leftarrow${id};

13 When participant calls leaveGroup(id, participantGroupID, credential)
14 **if** credential accepted **then**
15    LV[participantGroupID] $\leftarrow$LV[participantGroupID] $\setminus$ {id};
16    **if** LV[participantGroupID] $= \bot$ **then**
17       delete participantGroupID entry in LV;
18       leaveSiteGroup(siteGroupId); {siteGroupId corresponding to participantGroupID saved}

19 When FDparticipant(id)
20 **for all** participantGroupId : id $\in$ LV[participantGroupID] **do**
21    LV[participantGroupID] $\leftarrow$LV[participantGroupID] $\setminus$ {id};
22    **if** LV[participantGroupID] $= \bot$ **then**
23       delete participantGroupID entry in LV;
24       leaveSiteGroup(siteGroupId); {siteGroupId corresponding to participantGroupID saved}
```

6 Conclusion

This deliverable presents the complete specification of the APIs and protocols for the MAFTIA middleware. The first half of the deliverable starts by describing the interfaces of the runtime environments that will support the middleware architecture and other components in general, namely the Appia protocol kernel and the Trusted Timely Computing Base (TTCB). Next, the interfaces of the following modules were introduced: Multipoint Network; Communication Services; Activity Services; Site Membership; Participant Membership. The APIs that were described can be used not only by end-user level programs, but also recursively by other modules of the architecture. The second half of the deliverable, explains the various protocols that implement the functionality provided by both the runtime environments and the middleware modules. In particular, there were protocols described for the components: TTCB, CS in the asynchronous model, CS with an asynchronous payload system and support from the TTCB, a transactional support AS, and a dynamic membership service.

In a next deliverable of WP2, "D11: Running prototype of MAFTIA middleware, due to 6 month from now, we will provide an implementation of the APIs and protocols described in this document.

Bibliography

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

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[3] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 123–138, November 1987.

[4] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):46–76, 1987.

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

[6] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, 1984.

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

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

[9] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In Joe Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[10] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132, June 2000.

[11] Christian (editor) Cachin. Specification of Dependable Trusted Third Parties. Technical Report RZ 3318, IBM Research, Zurich Reseach Laboratory, 22 January 2001.

[12] Christian (editor) Cachin. *Full Design of Dependable Third Party Services.* Deliverable D5. Project MAFTIA IST-1999-11583, February 2002.

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

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

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

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

[17] 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.

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

[19] 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, September 2000.

[20] A. Casimiro and P. Veríssimo. Timing failure detection with a Timely Computing Base. In *Proceedings of the European Research Seminar on Advances in Distributed Systems*, April 1999.

[21] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 22–25, February 1999.

[22] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[23] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, May 1996.

[24] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[25] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In *Proc. of the 21th IEEE Symposium on Reliable Distributed Systems*, October 2002.

[26] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel (extended version). DI/FCUL TR 01–12, Department of Computer Science, University of Lisbon, December 2001.

[27] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Fourth European Dependable Computing Conference*, October 2002.

[28] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design.* Addison-Wesley Publishing Company, third edition, 2001.

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

[30] M. Daniele, B. Wijnen, Ed. M. Ellison, and D. Francisco. Ed. Agent Extensibility (AgentX) Protocol. RFC 2741, January 2000.

[31] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash-stop to byzantine failures. In *International Conference on Reliable Software Technologies*, May 2002. (invited paper).

[32] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[33] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. RFC 1750, December 1994.

[34] 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.

[35] 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.

[36] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[37] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 92–113. Springer-Verlag, 1998.

[38] 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.

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

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

[41] K. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proc. Int'l Conference on Principles of Distributed Systems*, pages 61–75, December 1997.

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

[43] P. Luenam and P Liu. The design of an adaptive intrusion tolerant database system. In *Proceedings of the Workshop on Intrusion Tolerant Systems*, June 2002.

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

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

[46] 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*, pages 707–710, April 2001.

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

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

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

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

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

[52] 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*, pages 5–15, 2001.

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

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

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

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

[57] M. O. Rabin. Randomized Byzantine Generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[58] 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.

[59] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

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

[61] T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 1998.

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

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

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

[65] A. Shacham, R. Monsour, R. Pereira, and M. Thomas. IP Payload Compression Protocol (IPComp). RFC 2393, December 1998.

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

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

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

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

[70] 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, June 2000.

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

[72] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. In *SIGCOMM*, pages 83–93, 1989.

[73] 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*, pages 264–274, 2000.

[74] 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, 1995.

[75] 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.