Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

Running Lab Prototype of MAFTIA Middleware

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

MAFTIA deliverable D25

Public document

March 1, 2002

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)*
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)*
Robert Stroud, *University of Newcastle upon Tyne (UK)*
Paulo Veríssimo, *University of Lisboa (P)*
Michael Waidner, *IBM Zurich Research Lab (CH)*
John Warne, *University of Newcastle upon Tyne (UK)*
Ian Welch, *University of Newcastle upon Tyne (UK)*

Contents

List of Figures

Abstract

This document briefly describes three prototypes, each one corresponding to a subset of the MAFTIA middleware architecture. Together, these prototypes represent the most important components of the architecture, and constitute Deliverable D25 - Running Lab Prototype of MAFTIA Middleware. The code distribution of the prototypes is available for review, and it includes a more extensive documentation.

The first prototype is composed of an implementation of the Trusted Timely Computing Base (TTCB) on a real-time Linux kernel, and two secure reliable multicast protocols that explore the services provided by the TTCB. The second prototype implements the Communication Support (CS) layer of the architecture, and supplies a number of protocols including binary and multi-value agreement and atomic broadcast. The third prototype represents an Activity Support (AS) module and offers a transactional support service. This prototype implements a number of protocols and activities that facilitate the creation of resource and transaction managers.

1 Introduction

This document briefly presents three prototypes that together represent the most important components of the MAFTIA middleware architecture. The main ideas and the rationale for this architecture has been presented in a previous deliverable (D23 [9]), where the various system models, modules and services were introduced. The APIs and some of the protocols that are provided by the prototypes have also been previously described in another deliverable (D24 [7]). In the following year, we expect to consolidate and integrate the prototypes to create a complete implementation of the middleware subsystem (D11: Running prototype of MAFTIA middleware – due to T0 + 36 months).

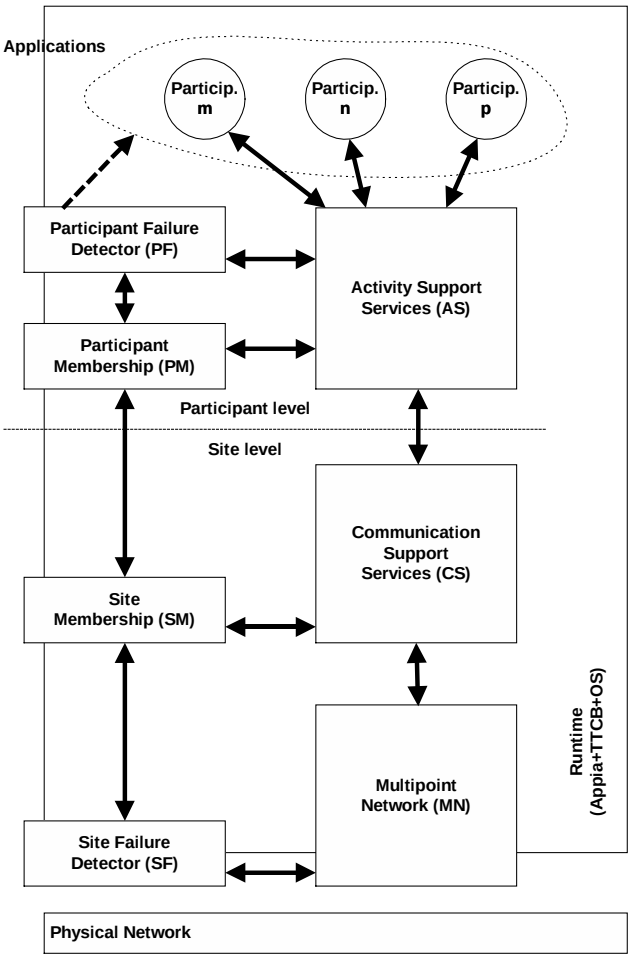

Figure 1.1: Architecture of a MAFTIA Host

Figure 1.1 represents the main components of the architecture of a MAFTIA host, in which the relations between modules are depicted by the 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 Trusted Timely Computing Base (TTCB).

This document is organized in three parts, each one corresponding to one of the prototypes. The first prototype implements the TTCB, which is a distributed security kernel that might have its services called by every other module. 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. Examples of these services are: trusted block agreement, local authentication, and trusted duration measurement. The current implementation is built on a real-time Linux kernel (RT-Linux) that has been enhanced with several modifications to improve its security. The control channel is based on an extra fast Ethernet that can only be accessed by the TTCB. Even though we have only now started an extensive testing phase, we expect that the current prototype already exhibits a good coverage of the TTCBs properties, such as fail-controlled behavior (i.e., it fails only by crashing).

This prototype also includes the implementation of two secure reliable multicast protocols that are capable of tolerating Byzantine failures. These protocols use the services provided by the TTCB to execute a small number of crucial steps in a secure way. Even though the protocols have some similarities, they have been optimized for different working conditions, which can be useful for building systems which can adapt to distinct levels of threat. In the architecture displayed in Figure 1.1, these protocols will be part of the Communication Support (CS) component. Currently, they have been implemented in Java as a set of Appia layers. Appia is a layered communication framework. Each layer represents a micro-protocol that has a uniform interface which allows each layer to be combined with other layers in a stack to provide a service. The development of the of Multipoint Network (MN) did not require too much effort since the Appia distribution already provides a number layers that can be included in this component. Nevertheless, it was necessary to produce an adaptation module from the interface offered by the TTCB at the kernel-level to the interface provided by Appia to the layers.

The second prototype offers an hierarchy of secure protocols that correspond to a CS component implementation. It supplies primitives for group communication such as binary and multi-valued agreement, reliable and consistent broadcast, and an atomic broadcast stream. Atomic broadcast immediately provides secure state-machine replication. MAFTIA architecture documents describe a multi-dimensional parameter space of system models, such as: purely asynchronous, partially synchronous or fully equipped with access to a Trusted Timely Computing Base; static, dynamic, open and closed groups; etc. This prototype resides in only one point of this space, with a fully asynchronous network and static, open groups. This makes it useful on an asynchronous wide-area network such as the Internet, where messages may be delayed indefinitely and protocol participants do not have access to a common clock.

The tools provided by this prototype enable Activity Support (AS) modules to achieve fault tolerance in this system model by distributing their trust to several servers, some of which may be failing in a malicious manner. Care must be taken in implementing such AS modules as the failed servers must not learn any secrets by participating in some protocol, nor should they have the power to act in unauthorized ways on the group's behalf or to prevent the other, honest servers from properly processing a request. Such robustness is achieved by Byzantine Agreement and Atomic Broadcast protocols which make extensive use of threshold cryptography, as well as by the direct application of such threshold cryptographic primitives. For example, the Distributed Certificate Authority (DCA) described in Deliverable D5 [3] uses Atomic Broadcast to circulate client requests among the DCA servers, Binary Byzantine Agreement to decide whether or not to honor certain requests, and Threshold Signatures on the certificates to ensure that they reliably represent the will of the honest DCA servers.

The third prototype offers a transactional support service (TSS), which is an AS module. It 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 [8] to a standard transaction processing architecture. Effectively, the servers implementing the transaction service and optionally the resource managers and resources are replicated.

The current version of the TSS is implemented in Java as a set of Appia layers. The layers can be combined to implement different TSS components, for example Resource Managers or Transaction Managers. The prototype depends upon group communication layers that are provided as part of the standard Appia distribution. In the current implementation we assume only benign failures so the standard group communications layers are sufficient. However, the next version of the TSS will be built on top of the CS intrusion tolerant group communication modules and local platform services. It is assumed that both TTCB-based and time-free communication modules will have similar interfaces which will make it relatively simple to switch from a benign fault model to one incorporating malicious faults. We also assume that under either fault model that simple local platform services such as local cryptographic support, limited secure storage for keys, and durable storage are available.

2 Trusted Timely Computing Base Prototype

This section presents a brief overview of the prototype of the TTCB on a RT-Linux. It also describes two secure reliable multicast protocols that take advantage of the services offered by the TTCB in certain crucial steps of their execution. These protocols will be included in the future in a CS module.

2.1 Architecture

Each host contains the typical software layers, such as the operating system and runtime environments, and an extra component, the TTCB. The TTCB is a distributed entity with local parts in the hosts and a control channel (see Figures 2.1 and 2.2). The local parts, or *local TTCBs*, are computational components with activity, conceptually separate from the hosts' operating system. The *control channel* is a private communication channel or network that interconnects the local TTCBs. It is conceptually separated from the payload network, the network used by the hosts to communicate.

2.2 Implementation

The current TTCB implementation is a preliminary version of the design presented in [5], with the programming interface presented in [7]. The overall TTCB architecture is quite simple: local TTCBs in hosts, interconnected by a (private) control network.

For very high coverage, local TTCBs would normally be built on dedicated tamper-proof hardware modules with a dedicated network attachment, such that the modules can be plugged with the proper physical isolation into the host hardware. However, the current design is based on the same COTS hardware and software which runs the payload, with isolation implemented in software. The local architecture of this software-based solution consists of a small secure real-time kernel running on the bare machine hardware, on top of which the regular operating system runs, as all the rest of the host software. The local TTCB is basically a privileged and highly protected set of real-time tasks of that kernel.

For the kernel, we use a general purpose real-time executive, RT-Linux, based on a modified Linux kernel [2, 10]. This approach has the advantage of running Linux and Linux applications on top of it, but it controls all hardware resources in order to safeguard the timeliness of real-time tasks, which are used to support the TTCB. As a disadvantage, it can be as insecure as Linux. The TTCB is built on the RT-Linux kernel, and in consequence, our design concentrates mainly on securing the design principles that make-up a

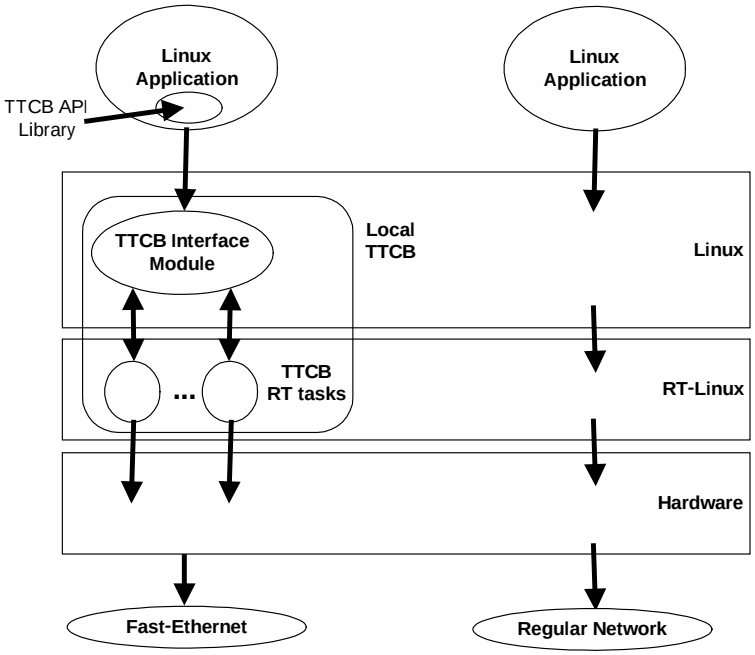

Figure 2.1: Architecture of a host with a Local TTCB.

trustworthy implementation of a security kernel— interposition and shielding [5]. This is achieved removing a set of Linux capabilities during the normal system operation [1].

The control channel can assume several forms. To pursue the COTS strategy, our architecture is based on Fast Ethernet, for campus-wide systems: we provide each host having a TTCB with an extra LAN adapter.

Figure 2.1 presents the architecture of a host with a local TTCB. The local TTCB is composed by a Linux kernel module ("TTCB Interface Module") and a set of real-time tasks that run in the RT-Linux kernel. The former module handles the communication with the applications and executes some non-real-time basic services, such as giving timestamps or random numbers. The real-time tasks include always one for reading packets from the Fast-Ethernet device driver and another to send packets periodically to this same network.

Applications communicate with the TTCB using RT-Linux fifos. However, a library with a set of C functions is provided so that the programmer does not need to go into that level of detail (see figure). Currently, a Java library is also being developed. It was used to implement the reliable multicast protocol described below.

2.3 Example

The BRM is a package that provides secure reliable multicast, as a component of MAFTIA architecture, which tolerates arbitrary faults, including Byzantine faults. It is a building block that can be used for applications that required intrusion tolerance. Furthermore, it is intended to be combined with other protocols to provide warranties more restrictive of agreement and ordering (e.g. atomic multicast [6]). There are two protocols in this package, BRM-T and BRM-M, first one protocol exhibits fast termination in the presence of attacks and/or crash or malicious process failures, but has a higher message complexity. On the other hand, the BRM-M uses fewer messages and terminates faster in fault-free situations, but takes longer to terminate in the presence of attacks/faults. These two similar protocols, optimized for different conditions, can be useful for building systems which adapt to, e.g., different levels of threat.

The BRM prototype is implemented using Appia, a layered communication framework, and the TTCB, a distributed secure kernel. Appia provides the facilities for group communication under the assumption of a benign fault model. The BRM protocols are built using the facilities provided by IP Multicast layer of Appia. The TTCB is used in the BRM to execute crucial parts of the protocols operation, under the protection of a crash failure model. The TTCB provides only a few basic services, using TTCB allows to BRM not to need of a public key infrastructure. Furthermore, our protocols to have efficiency similar to that of accidental fault-tolerant protocols: for f faults, our protocols require $f+1$ processes, instead of the typical $3f+1$ of Byzantine systems. More details about Appia and TTCB can be found in Deliverable D24 [7], and the protocols in [4].

2.3.1 BRM Architecture

The architecture of the prototype of MAFTIA secure reliable multicast is shown in Figure 2.2. Four hosts are shown, one Certified Authority Manager host and three MAFTIA hosts equipped with TTCBs. Actual deployments may have more MAFTIA hosts.

The Certified Authority Manager manages and provides the secret keys (symmetric cryptography) for each pair of participants. These keys are used to create the Message Authentication Codes (MAC) that are included in ACK messages from the BRM-M protocol. These keys are transmitted using the SSL Layer provided by Appia.

The MAFTIA hosts make use of the BRMulticast Layer which relies upon the TTCB, the Appia IP Multicast Layer, the Appia SSL Layer and the SKMClient Layer (for simplicity, the last two layers are not displayed). The BRMulticast Layer provides an

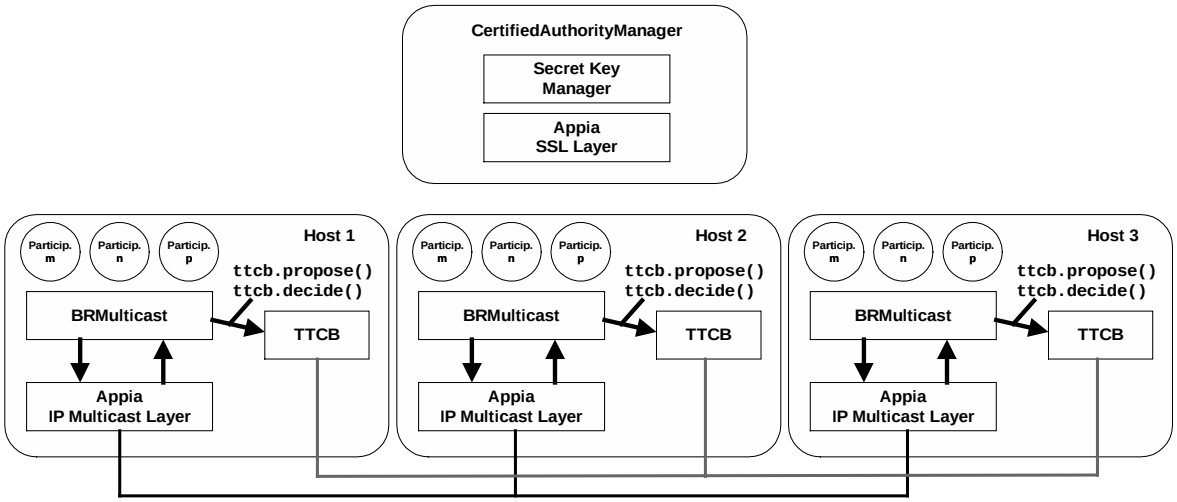

Figure 2.2: Architecture of a distributed TTCB with the BRM protocols.

implementation of a secure reliable multicast protocol. It provides methods for multicasting a message and receiving a callback when a message is received by the protocol layer. The BRM protocols uses the TTCB Trusted Block Agreement Service to multicast the hash of the message to the recipients. The hash is used by the recipients, according the protocol, to validate the message M transmitted by Multicast Layer (payload network). The IP Multicast Layer is an Appia layer that provides access to low-level IP multicast sockets. The Appia SSL Layer is used to connect to the Certified Authority Manager via a secured connection, and the SKMClient Layer manages locally the secret keys provided by Certified Authority Manager.

3 Asynchronous Group Communication Prototype

This section describes an hierarchy of secure group communication protocols that correspond to a CS component implementation. The prototype implements one of the various system models that are being considered in MAFTIA, where the network is asynchronous and the groups are static and open.

3.1 Architecture

The CS prototype is built in a modular way, as shown in Figure 3.1. Modularity greatly simplifies the construction and analysis of the complex protocols needed to tolerate Byzantine (malicious) faults.

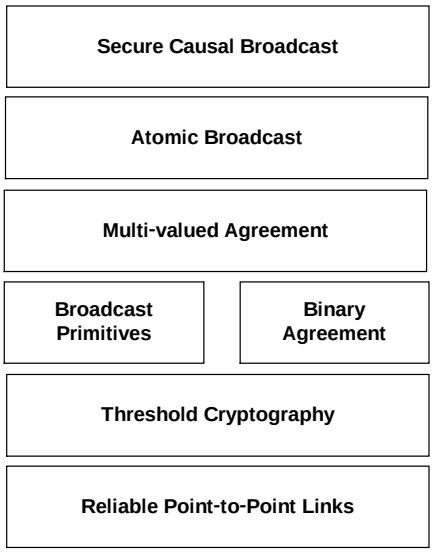

Figure 3.1: Architecture of the CS protocols.

Currently, the CS prototype needs a trusted dealer to generate the secret keys for a particular configuration. The dealer is needed only once, when the system is initialized. The keys must be distributed to all servers in a trusted way. The point-to-point links are authenticated using a message authentication code (MAC) for which one symmetric key for every pair of servers is also generated by the dealer.

The point-to-point links in CS are currently implemented by reliable TCP streams for simplicity and are therefore subject to a denial-of-service attack by sending forged TCP acknowledgements. It is planned to replace TCP by a custom sliding-window management

over UDP which will provide authenticated acknowledgments and more appropriate session management.

Link authentication is provided over TCP using SHA1 (160-bit output). SHA1 is also used as the hash function in the threshold signature and standard signature schemes, and in the threshold coin-tossing protocol.

A single configuration file contains all important parameters, such as the identities of all parties, the over-all system parameters, cryptographic key sizes, *etc.* A party is identified by an Internet address of the form **hostname:port**, which denotes the socket endpoint through which it connects to the others.

3.1.1 Threshold Cryptography

Threshold cryptography is crucial for several of the CS protocols and forms a core component of the architecture. Threshold schemes are used for *digital signatures*, *coin-tossing*, and *public-key encryption*. These are all non-interactive and robust; they are implemented by a collection of algorithms for generating, verifying, and assembling shares of the cryptographic operation. In this way, they do not require any particular communication pattern and can be easily integrated into asynchronous protocols.

In contrast to the other threshold schemes, true threshold signatures are not essential for the operation of the CS protocols. Since the group is static and a standard digital signature scheme is also available, whose public keys are known to all servers, it is sometimes more efficient to use a vector of standard signatures instead of a threshold signature. This is called a *multi-signature* and requires no change to the protocols that use threshold signatures. Multi-signatures are particularly suited for small groups when fast communication is available and computation is expensive.

The keys for all threshold schemes are generated by the dealer and included in the initial state of each server.

3.2 Implementation

The prototype is implemented in Java (version 1.2 or later), with each communications protocol type using a particular extension of an abstract base class **Protocol**. Similar protocols with largely identical APIs are grouped together by being extensions of a common class, as follows:

Broadcast – **ReliableBroadcast** and **ConsistentBroadcast** provide reliable and consistent broadcast, respectively

Agreement – **BinaryAgreement** provides binary Byzantine agreement, **ValidatedAgreement** provides validated binary Byzantine agreement, and **ArrayAgreement** provides multi-valued agreement

Stream – **AtomicStream** provides an atomic broadcast stream, **SecureAtomicStream** provides a secure causal atomic broadcast stream, and **ReliableStream** and **ConsistentStream** provide reliable and consistent broadcast streams, respectively

In addition, the threshold cryptographic algorithms used within the CS layer and also accessible to AS modules are in the following classes:

ThresholdCoin – manipulates a threshold coin

ThresholdSignature – has extensions **RSAThresholdSignature** and **MultiSignature** with the two different approaches to threshold digital signatures described in Section 3.1.1

ThresholdCipher – provides the threshold public-key encryption scheme

The CS module also includes a stand-alone Java program which reads the configuration file and creates keys of the desired strength for all threshold cryptographic tools and point-to-point authentication MACs.

3.3 Example

The various protocols and tools of the CS prototype have been extensively tested across a variety of different operating systems, physical hardware and network configurations, including LANs on a single Ethernet backbone and WANs covering three continents. The most difficult test, using full-strength cryptography (1024-bit RSA and discrete logarithm keys), the global WAN and servers ranging from a 200 MHz Pentium Pro to a 1GHz Pentium III, gave times of a few seconds to agree to the "next" message in atomic broadcast. This indicates that these protocols are practical today in certain critical contexts, and are likely soon to be practical, with an optimized implementation and some protocol improvements, for a much broader spectrum of applications.

4 Transaction Support Prototype

This section provides an overview of the prototype of MAFTIA transactional support [7] which is an Activity Service. In the following sections we provide a brief overview of the architecture, the implementation of the architecture and the demonstration application provided with the prototype.

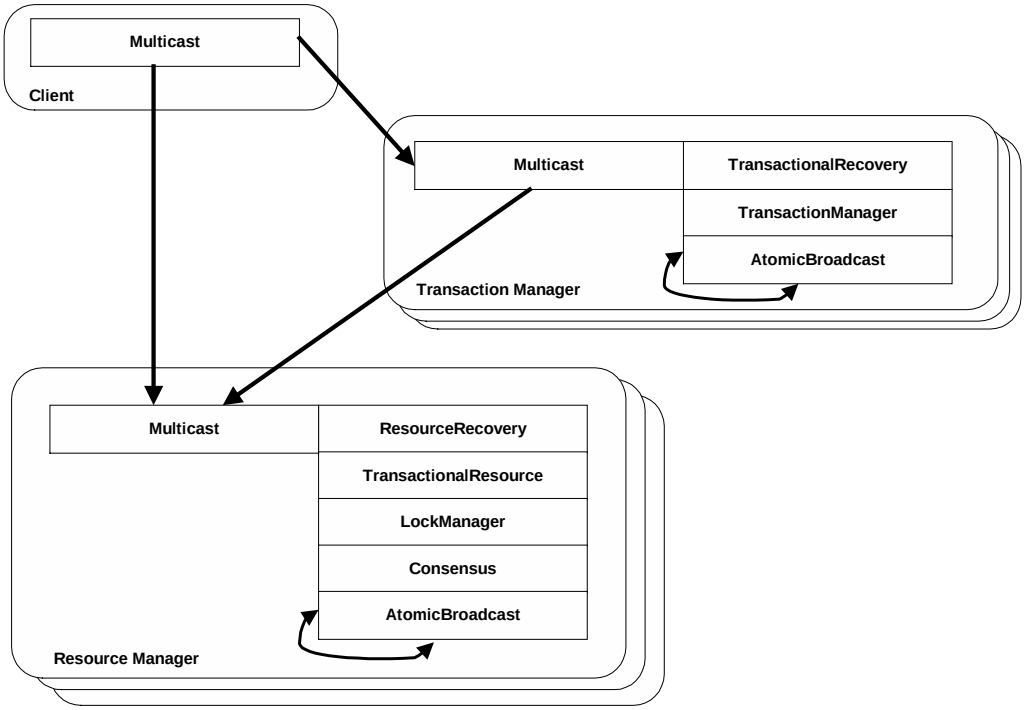

Figure 4.1: Architecture of the Transactional Support Service.

4.1 Architecture

The TSS architecture is composed of clients, resource managers and transaction managers. Only the resource managers and transaction managers are replicated. Each component is implemented in a modular way out of Appia layers, as shown in Figure 4.1. The replicated components use a closed group style of communication which is implemented by the atomic broadcast layer. Other communication, for example between clients and resource managers, is an open group style of communication which is implemented by the multicast layer. The role of each component is introduced below.

Clients. Clients use the transaction manager to begin and end transactions. Within

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

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

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

4.2 Implementation

The prototype is implemented in Java using the Appia framework. In this section we provide a brief description of each Appia layer. We have grouped layers according to their relationship to the middleware modules.

4.2.1 Communication Service Layers

These layers provide abstractions for various modules of the communication service, in particular multicast, atomic broadcast, and consensus. In future versions of the prototype we will use the TTCB-based and time-free communication services instead of the ones described here. However, we expect that the future versions will have similar interfaces thereby making the transition from group communications that assume benign failures to group communications that assume Byzantine failures.

Multicast Layer – Provides an open group style of group communication. A client sends its request to a single member of a closed group, the recipient then uses the Atomic Broadcast layer to send the request to all group members. Ordering of requests

is FIFO but dependent on the order in which that the first member of the group received the requests from the client.

Atomic Broadcast Layer – Provides a closed group style of group communication and supports FIFO-ordered, view-synchronous multicast.

Consensus Layer – As solving consensus is equivalent to solving reliable and totally ordered multicast we build a consensus layer on top of the atomic broadcast layer.

4.2.2 Activity Service Layers

These layers provide the TSS-specific protocols as described in Deliverable D24.

ResourceRecovery Layer – Controls recovery of a resource after a benign failure. Resources make use of local durable logs to support recovery. This layer is required by the ResourceManager layer.

TransactionalResource Layer – Allows a resource to take part in the two phase commit and abort protocols. It also implements the registration of ResourceManagers with TransactionManagers. This layer is required by the ResourceManager layer.

LockManager Layer – Provides a distributed pessimistic locking protocol. The isolation property depends upon this to prevent clients that are not within a transaction observing the effects of the transaction until it is complete. This layer is required by the ResourceManager layer and in turn depends upon the Consensus layer.

TransactionManagerRecovery Layer – Controls recovery of a transaction manager after a benign failure. During recovery, Transaction Managers rely upon other members of the Transaction Manager group to determine the outcome of a transaction. This layer is requires the TransactionManager layer.

TransactionManager Layer – Implements the two phase commit protocol and abort protocol. It 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.

4.3 Example

A text-based demonstration application is provided with the prototype. The application is made up of implementations of clients, transaction managers and resource

managers. When using the demonstration application, the user must first start a group of transaction managers, and two groups of resource managers. The user can then begin a transaction, send requests to the resource managers and either commit or abort a transaction. The transaction managers and resource managers echo their responses to the console window so that the user can verify their correct behaviour. Instructions on the use of demonstrator is included with the TSS distribution.

5 Conclusion

This deliverable contains three prototypes, each one of them represents a subset of the MAFTIA middleware architecture. The first prototype implements the TTCB distributed security kernel on a RT-Linux. To ensure good coverage of the TTCB properties, this version of the RT-Linux suffered several enhancements to improve its overall security. The local TTCB is composed by a Linux kernel module, which is responsible for some non-real-time services and interactions with the applications, and a set of real-time tasks that handle all to communication through the control channel. This prototype also includes the implementation of two secure reliable multicast protocols that are capable of taking advantage of the services provided by the TTCB to execute a small number of crucial steps in a secure way. The second prototype offers an hierarchy of secure protocols that correspond to a CS component implementation. It supplies primitives for group communication such as binary and multi-valued agreement, and an atomic broadcast stream. The system model assumed by this prototype is a fully asynchronous network, and static, open groups, which makes it useful on an asynchronous wide-area network such as the Internet. The protocols being provided enable AS modules to achieve fault tolerance in this system model by distributing their trust to several servers, some of which may be failing in a malicious manner. The third prototype offers a transactional support service, which is an AS module. It provides multiparty, single level transaction support that can 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.

In the next deliverable of WP2, "D11 - Running prototype of MAFTIA middleware", we will expand and integrate the current prototypes to produce a complete implementation of the middleware subsystem.

Bibliography

[1] Linux Capabilities FAQ 0.2. ftp://ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt, 2000.

[2] Michael Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Socorro (New Mexico), USA, June 1997.

[3] C. Cachin, editor. *Full Design of Dependable Third Party Services. Project MAFTIA IST-1999-11583 deliverable D5.* January 2002.

[4] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In *Submitted for publication*, 2002.

[5] M. Correia, P. Veríssimo, and N. Neves. The architecture of a secure group communication system based on intrusion tolerance. In *Proceedings of the International Workshop on Applied Reliable Group Communication*, pages 17–22, April 2001.

[6] 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.

[7] N. F. Neves and P. Veríssimo, editors. *First Specification of APIs and Protocols for MAFTIA Middleware. Project MAFTIA IST-1999-11583 deliverable D24.* August 2001.

[8] D. Powell and R. J. Stroud, editors. *MAFTIA: Conceptual Model and Architecture. Project MAFTIA IST-1999-11583 deliverable D2.* November 2001.

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

[10] Victor Yodaiken and Michael Barabanov. RTLinux version two. http://www.fsmlabs.com/archive/design.pdf, 1999.