The Architecture of a Secure Group Communication System Based on Intrusion Tolerance*

Miguel Correia Paulo Veríssimo Nuno Ferreira Neves
Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande, 1749-016 Lisboa - Portugal
{*mpc,pjv,nuno*}*@di.fc.ul.pt*

Abstract

This paper presents the architecture of a secure group communication system with the fortress model of trust, where the participants of the group equally trust one another. We consider that only a small part of the system, a component called the Trusted Timely Computing Base, has to be entirely trusted. All other components can be corrupted. The overall system will tolerate a certain number of faults of its components and remain behaving correctly.

1. Introduction

Group communication is a well known communication paradigm. The first group communication systems developed had a benign failure model in mind. More recently, a few systems were designed considering malicious faults, such as attacks performed by a hacker [13, 14, 3]. However these systems still have some important drawbacks. For instance, they all assume that the operating system can be considered to be a Trusted Computing Base, i.e., that it does not fail and does not perform attacks on user processes. With current operating systems that is a very optimistic assumption.

This paper presents work in progress on the specification of a secure group communication system that tries to address some of the problems of current systems. The paper describes the architecture and design principles of the system, but leaves the specification of protocols and algorithms for another opportunity. The system considers a *fortress model of trust*, i.e., that processes are equally trusted by one another and that the system guarantees the security of their communication. Our system is being developed within the MAFTIA project middleware architecture [4, 18].

*This work was partially supported by the EC, through project IST-1999-11583 (MAFTIA), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LASIGE) and the project POSI/1999/CHS/33996 (DEFEATS).

The system considers a hybrid failure model, where different components of the system have different failure models. For example, the network can fail arbitrarily (e.g., can perform arbitrary attacks) while software components of the system are fail-controlled (bounded type and number of component failures). We consider that only a small part of the system, a distributed component called the *Trusted Timely Computing Base* (TTCB), has to be entirely trusted. Since it is small, its correctness can be verified.

2. Middleware Architecture of MAFTIA

In this section we briefly describe the features of the middleware architecture of MAFTIA that are most relevant to the description of our system.

2.1. Failure Model

A crucial aspect of any system architecture is the failure model upon which it is conceived and component interactions are defined. There are essentially two different kinds of failure model: controlled failure assumptions (component failures are qualitatively and quantitatively bounded) and arbitrary failure assumptions (there are no bounds on component failures). Hybrid assumptions combining both kinds of failure assumptions would be desirable. Generally, they consist of allocating different assumptions to different subsets or components of the system, and have been used in a number of systems and protocols. Hybrid models allow stronger assumptions to be made about parts of the system that can justifiably be assumed to exhibit fail-controlled behavior, whilst other parts of the system are still allowed an arbitrary behavior. However, this is only feasible when the behavior of every single subset of the system can be modeled and/or enforced with high coverage.

A first step in this direction is the definition of a composite failure model specifically aimed at representing the failures that may result from several classes of malicious faults. A second step is the definition of a set of techniques that act

at different points within this composite failure model and which, combined in several ways, yield dependability vis-à-vis particular classes of faults. MAFTIA identified two guiding principles:

- the sequence: attack + vulnerability → intrusion → failure
- the recursive use of fault tolerance and fault prevention

Vulnerabilities are the primordial faults existing inside the components, essentially design or configuration faults. Attacks are malicious interaction faults that attempt to activate one or more of those vulnerabilities. An attack that successfully activates a vulnerability causes an intrusion, an erroneous state in the system. If nothing is done to process the intrusion, failure of one or more security properties will occur.

The composite failure model is a basis for achieving the objective of a well-founded hybrid failure model. Given a component for which a controlled failure assumption was made, coverage of such an assumption can be achieved with a recursive use of fault tolerance and fault prevention. The component can be constructed through the combined use of removal of internal vulnerabilities, prevention of some attacks, and implementation of intrusion tolerance mechanisms internal to the component, in order to prevent the component from exhibiting failures. Looked upon from the outside, at the next higher level of abstraction, the level of the outer system, the would-be component failures we prevented restrict the system faults the component can produce. In fact we have performed fault prevention, that is, we have a component with a controlled behavior vis-à-vis malicious faults. This principle establishes a divide-and-conquer strategy and can subsequently be used in the construction of fault-tolerant systems.

2.2. Synchrony Model and the TTCB

The Timely Computing Base is a distributed component that is both a timely execution assistant and a timing failure detection oracle, that assists the execution of applications with time requirements in asynchronous environments [17]. The Timely Computing Base is a "small" synchronous component embedded in an asynchronous payload system. It is composed by local parts in hosts (e.g., hardware coprocessors or applications running in a real-time kernel [5]) interconnected by a real-time channel or network. The difficulty of a synchronous component providing useful services for an asynchronous one was approached with the definition of three basic services: duration measurement, timely execution, and timing failure detection.

The Timely Computing Base was defined with a benign failure model in mind. In MAFTIA we extend the Timely Computing Base model to environments with malicious faults [18]. We call the new component the Trusted

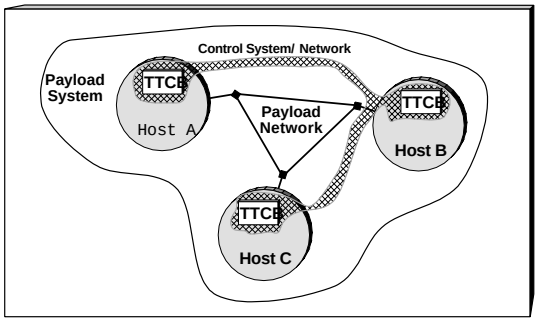

Figure 1. TTCB architecture

Timely Computing Base (TTCB). Like the Timely Computing Base, the TTCB is timely, but also secure (tamperproof) by construction. It is the only component in our system with that characteristic. The architecture of the TTCB is similar to that of the Timely Computing Base (Figure 1). The Timely Computing Base timeliness relied on that component being simple enough to be readily validated. The same applies to the TTCB: it is simple enough for its timeliness and trustworthiness to be validated.

The TTCB has trusted versions of the Timely Computing Base services: trusted duration measurement, trusted timely execution, and trusted timing failure detection. The output of those services can be signed by the TTCB since every *local TTCB* has a private key for which the corresponding public key is distributed to processes that may need it (e.g., using a Public Key Infrastructure). This key pair can be used for strong authentication of a local TTCB or the host where it resides. The TTCB is the correct place to save a long-term key since it is the only component completely trusted.

The TTCB implements some additional services in relation to the Timely Computing Base. The objective is to have a minimal set of secure functions that assist in a useful manner the implementation of building blocks for trusted distributed systems. In this paper we describe how these services can be useful to implement our system modules. The new services are:

- Trusted random number generation. This service generates trustworthy random numbers. These numbers are basic for building cryptographic primitives and it is essential that they are really random.
- Trusted absolute timestamping. This service provides a timestamp, either authenticated (signed with a TTCB private key) or not. The timestamp is meaningful to any local TTCB since clocks are synchronized. In the original Timely Computing Base they were not synchronized but this is an important feature for some security services, e.g., Kerberos authentication protocol.
- Trusted block consensus. This service achieves consensus between local TTCBs on a fixed size block of data.

It may be used to perform simple but crucial decision steps in more complex payload protocols.

- Trusted block equality test. This service is used to test if data that should be equal in a set of sites is corrupted. Every site proposes a value (block) and receives the list of sites that have a value different from the majority.
- Local authentication (and key establishment). This service is used for a participant or process to authenticate the TTCB and obtain a shared key with it. This key can be subsequently used to guarantee the integrity and confidentiality of their communication, therefore establishing a secure channel between them.
- Distributed authentication (and key establishment). This service can be used for two or more distributed participants or processes to authenticate themselves mutually through the TTCB. A shared key is established.

2.3. Architecture of a Node

The architecture of MAFTIA middleware in a node is divided in two levels: participant and site. It is represented in Figure 2. The runtime environment —composed by the operating system, the protocol kernel[1] and the TTCB— is not represented for simplicity. The division in participant and site levels is the materialization of a form of clustering used in MAFTIA: a site is a cluster of participants. This clustering is used in our system as a way to handle scale. A *participant-group* is mapped into a *site-group*, composed by all sites where there are participants[2]. Site level protocols handle inter-site communication. This level has access to, and depends on, a physical networking infrastructure. The participant level offers support to local participants engaging in distributed computations. It multiplexes and demultiplexes participant communication into the site group.

The lowest layer of the architecture is the Multipoint Network module, MN, created over the physical infrastructure. Its main properties are the provision of multipoint addressing and a moderate best-effort error recovery ability, both depending on topology and site liveness information.

In the site level, the Site Failure Detector module, SF, is in charge of assessing the connectivity and correctness of sites, and the MN module depends on this information. The SF module depends on the TTCB to perform its job, and therefore detects the failure of other sites reliably. The Site Membership module, SM, depends on information given by the SF module. It creates and modifies the membership and the view of site-groups. The Communication Support Services module, CS, implements basic cryptographic primitives, group communication with several reliability and ordering guarantees, and other core services. The CS module depends on information given by the SM module about the composition of the groups, and on the MN module to access the network.

[1] By *protocol kernel* we mean a runtime environment for communication protocols such as Ensemble [16] or Appia [1].

[2] More precisely, a participant is a group *member* if it receives messages from the group. If it only sends messages then it is a *sender* and its site does not have to belong to the group [4].

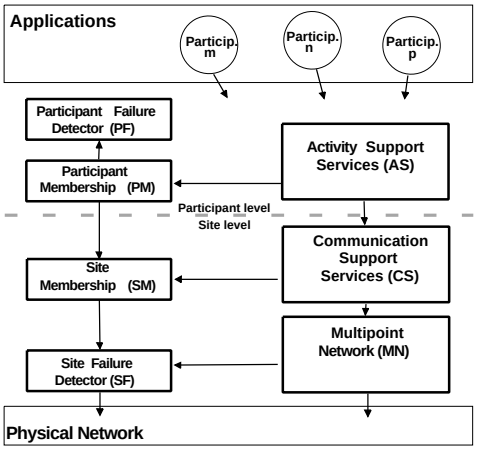

Figure 2. Architecture of a MAFTIA node

In the participant level, the Participant Failure Detector module, PF, assesses the liveness and (optionally) the correctness of local participants. The Participant Membership module, PM, performs similar operations as the SM, on the membership and view of participant-groups. The PM module monitors all groups with local members, depending on information propagated by the SM and by the PF modules, and operating cooperatively with the corresponding modules in the concerned remote sites. The Activity Support Services module, AS, implements building blocks that assist participant activity, such as replication management and transactional management.

3. The Group Communication System Architecture and Security

The group communication system implements the *fortress* model of trust, i.e., it allows a set of participants that equally trust each other to communicate in a trusted way. The architecture of the system is a composition of nodes with the structure described in the previous section. The current section describes how the system is made trustworthy. The components of the system, with the exception of the TTCB, are made fail-controlled with the use of fault tolerance and prevention techniques.

Failure assumptions always have a degree of coverage, i.e., a probability of being held per unit of time. We want to build a system with a high coverage of the failure assumptions —low probability of system corruption— based in components with different coverages. The single component that we consider to have a 100% coverage is the TTCB.

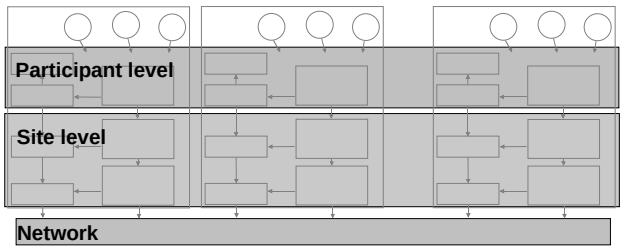

Figure 3. Main system components

The group communication system security architecture has several main components (Figure 3):

Network The physical payload network is fail arbitrary, i.e., it can engage in any attack against hosts. The level of threat posed by the network can be reduced using, e.g., a firewall, but in general its failure model will remain arbitrary since other hosts on the inner side of the firewall may still engage in the same kinds of attacks.

Runtime Environment There is a runtime environment (operating system + protocol kernel + local TTCB) in every host, so it is not a single component but a class of components. The figure does not represent it for simplicity.

Site Level This component is the set of all site levels of all nodes of a site-group (see Figure 3)

Participant Level This component is the set of all local participant levels of all nodes of a site-group (see Figure 3).

The runtime environment has to be made trustworthy because it has the ability to arbitrarily change the code and data of any process and the communication between protocol modules[3]. In the next section we will discuss how this can be achieved. We assume that the assumption of the runtime environment being trusted has a high coverage but, nevertheless, the runtime can still be corrupted. So, both the site level and participant level components have to be made secure against attacks coming from the runtime and from the network (sections 3.2 and 3.3). We assume that the system can be corrupted in a host but that the overall site level and participant level components are trusted. This last assumption has a coverage that is higher that the coverage of the assumption that the runtime environment is not corrupted.

In a host where the system (either the site or participant levels) is corrupted, it may not be possible to prevent the participants from seeing incorrect behavior. To handle these failures the participants on other sites can, e.g.: envelope their communication in long-running atomic transactions; simply to crash the group; or require intervention.

The system API has to be designed carefully in order to avoid attacks from the participants but it cannot do anything if a participant is corrupted and starts to behave differently than specified. We will not discuss how participants can be made trustworthy since it is not a problem of the group communication system.

[3] In fact, only the OS kernel and the protocol kernel have to be made trustworthy since the TTCB is already so by construction.

3.1. Runtime Environment

The *runtime environment* is a "component" that exists in every site. This component includes:

- Operating system kernel (OS kernel). This subcomponent controls every resource of the host: memory, disks, I/O, CPU time (scheduling), etc. Therefore, in principle, if it is corrupted it can arbitrarily modify the code, data and communication of a program.
- Protocol kernel. Our protocols are being programmed using a protocol kernel [16, 1]. This kernel manages issues common to all communication protocols, such as buffers, timeouts and layers. It has full control of the communication between layers of the protocols and between hosts so, if it is corrupted, it can arbitrarily modify the communication between hosts and inside a host.
- TTCB. This component cannot be corrupted so it will never engage in attacks either inside or outside the host.

This discussion leads us to conclude that we cannot let the OS kernel be corrupted. From a theoretical point of view, the trivial solution to protect the kernel would be to consider it a Trusted Computing Base [2]. However this is impractical for common operating systems and so we are not interested in that solution.

We propose to use a combination of different mechanisms to make the OS kernel fail-controlled:

- Select an OS that is as trustworthy as possible.
- Remove known vulnerabilities ("patching").
- Use intrusion detection and countermeasures; although generic Intrusion Detection Systems have a high rate of false detections and non-detections, privilege escalation attacks (like attacks made on the kernel from lower privileged processes) are already detected with a high precision and in run-time [9, 11]. Countermeasures can range from killing a corrupted process to crashing the host.
- Protect the host itself, since most attacks against the OS kernel come from the rest of the host (hackers usually attack first a user account and later the kernel). Some common measures are: close unused user accounts; force the use of strong passwords; deactivate unused network services (e.g., FTP); intrusion detection and countermeasures; remove vulnerabilities; etc.

The protocol kernel also has to be protected. If it runs with kernel privileges it can gain from the protection measures taken for the OS kernel. However, even if it runs with

those privileges, measures have to be taken to protect it from buffer overflow and input validation attacks, since the protocol kernel processes messages coming from the network or user processes. These two categories of attacks have to be handled with techniques such as correct program coding and disabling the execution of code in the stack. The protocol kernel can also be protected using code and data protection, as discussed in the next section.

3.2. Site Level

The composition of all local (node) site levels is a component that we call simply the site level (Figure 3). This component handles the site-group abstraction (see section 2.3). Three issues have to be considered: (1) the code and data of the site level have to be protected from attacks coming from the OS kernels and the network; (2) joins and leaves of sites to and from the group have to be secured; and (3) communication over the network has also to be secured.

Protection of the site level from attacks coming from the OS kernel (first issue) is a complex problem. The similar problems of protecting mobile code from malicious sites and protecting commercial software from reverse engineering are currently being researched. The ideal solution for all three problems would be to execute an encrypted program, that would not be modifiable without detection. Some solutions already exist to do it but the program has to be a polynomial or rational function which is not our case [15]. A partial solution is to *obfuscate* the code and data in order to avoid static analysis [10, 7]. The code and data obfuscation can be parameterized giving several different instances of the same program. An interesting solution in our system would be to instantiate a different version of the site level code in every site, so that reverse engineering one instance would not lead to the immediate compromise of the others. Obfuscation has several problems though: it is prone to dynamic analysis; understanding one instance can help understand the others; and it is possible that a program can be corrupted without a complete understanding of what it does (e.g., by identifying where system calls are made).

Attacks directly from the network to the site level are similar to those against the OS kernel, for instance, buffer overflow and input validation. The solutions are the same as those described in the previous section.

The integrity of important data in local site level subcomponents, such as site-group membership (SM module), can be tested using the TTCB *trusted block equality test* service. For example, every second, local site levels of a site-group can call that service giving a checksum of the membership data. If a host has data different from the others, all are informed, the host is assumed to be corrupt and it is excluded from the site-group.

Secure join (second issue) requires authentication —the site that wants to join has to be who it says it is— and authorization —the site that wants to join has to be allowed to do it (or not). Authentication can be handled using the TTCB to mutually authenticate the site that wants to join and the group. The TTCB's *distributed authentication* service provides strong authentication of the site. Authorization has to be agreed by all the sites of the group therefore solving the problem of one being corrupt.

Secure communication (third issue) is handled using signed and possibly encrypted messages. The sites have to share keys to handle message integrity and (optionally) confidentiality. Keys must be changed (rekeyed) whenever a site joins or leaves a group to avoid that it is able to interfere respectively with past and future communication. Rekeys must also happen with some frequency. The delay between rekeys should be decided with several factors in mind: average time between joins/leaves (that cause a rekey anyway), weight of the rekey protocol vs. encryption, power of the attacker, and kind of attack the attacker can perform to break the key (cipher-text only, known-plaintext, chosen-plain-text).

An additional factor to be considered is the possibility of the keys being compromised. Key disclosure can be detected, in case the attacker is using them to change messages, using the TTCB to send checksums and therefore confirming that messages are not being corrupted. These checksums can be sent for every message or for a number of messages so a tradeoff between additional load and delay of detection has to be considered. Key compromise may have to be handled using higher level mechanisms such as human intervention or coordinated atomic transactions.

3.3. Participant Level

Group participants communicate directly with the participant level, i.e., it is this component that has the system programming interface (see Figure 3). Therefore, this component has to: (1) be built to be trustworthy; and (2) handle the join and leave of participants to and from a group.

The first problem is similar to making the site level trustworthy (previous section). The second is similar to secure site join, but for participants instead of sites. The decision has to be voted by all group participants in order to prevent a corrupted host from introducing a malicious participant in the group. The participant has to be authenticated to the group with information that reliably identifies it: a pair ID/password or a secret key.

4. Related work

Some secure group communication systems exist already: Horus [13], Ensemble [14], and Secure Spread [3]. In these systems, and also in ours, communication and membership are secured using authentication, encryption and signatures with long-term and shared keys. However these other systems assume that their software is not corrupted inside hosts.

This is a strong assumption since trust relies on the security of a very large and unstructured component, such as a Linux or W2K machine. Trust in our system relies to a large extent on the security of a small, therefore provable correct, component, the TTCB.

Our approach is based on the combination of prevention and fault tolerance. We use both approaches to make our model as secure as possible but we accept that assumptions can fail (e.g., keys disclosed, some modules corrupted) and those failures have to be tolerated.

On the other hand, these systems classically consider an asynchronous time model. This can lead to vulnerabilities if protocols, despite looking time free, use hidden timing assumptions, such as timeouts. Our system uses a partially synchronous model whose resilience is based on the TTCB. Finally, a well-founded fault model is sometimes missing in previous works. The MAFTIA composite failure model clearly identifies types of faults and can help asserting the assumption coverage of the system.

Some related work exists in securing a service composed by a set of distributed processes [8, 6, 12].

5. Conclusion

We described the architecture of a secure group communication system. The system is secured using a combination of fault prevention and tolerance techniques. Hybrid component failure assumptions were considered and complete trust is put only on a small component, the TTCB.

Future work will follow several directions. The definition of the services and API of the TTCB is in progress. A classification of the sequences of faults that can lead to system failure (modes of failure) will be done in order to assess the assumption coverage. This classification will lead to a definition of the faults handled (prevented or tolerated) by every component of the system. A class of attacks not discussed here but that is being studied is denial-of-service.

A more formal definition of our system protocols and APIs and an implementation using the Appia protocol kernel are in progress.

References

[1] Appia protocol kernel homepage. http://appia.di.fc.ul.pt/.

[2] M. Abrams, S. Jajodia, and H. Podell, editors. *Information Security*. IEEE Computer Society Press, 1995.

[3] Y. Amir et al. Secure group communication in asynchronous networks with failures: Integration and experiments. In *Proc. The 20th IEEE International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 330–343, Taipei, Taiwan, Apr. 2000.

[4] C. Cachin et al. MAFTIA: Reference model and use cases. DI/FCUL TR 00–5, Department of Computer Science, University of Lisbon, Aug. 2000. Project MAFTIA IST-1999-11583 deliverable D1.

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

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. Third Symposium in Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, USA, Feb. 1999.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 98)*, San Diego CA, USA, Jan. 1998.

[8] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion tolerance in distributed computing systems. In *Proc. 1991 IEEE Symposium on Research in Security and Privacy*, pages 110–121, Oakland, California, USA, May 1991.

[9] A. Ghosh, C. Michael, and M. Schatz. A real-time intrusion detection system based on learning program behaviour. In H. Debar, L. Mé, and S. F. Wu, editors, *Recent Advances in Intrusion Detection - Third International Workshop (RAID)*, volume 1907 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlang, Toulouse, France, Oct. 2000.

[10] 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-Verlang, 1998.

[11] R. Lippmann, J. Haines, D. Fried, J. Korba, and K. Das. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In H. Debar, L. Mé, and S. F. Wu, editors, *Recent Advances in Intrusion Detection - Third International Workshop (RAID)*, volume 1907 of *Lecture Notes in Computer Science*, pages 162–182. Springer-Verlang, Toulouse, France, Oct. 2000.

[12] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, USA, Oct. 1998.

[13] M. K. Reiter, K. P. Birman, and R. van Rennesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, Nov. 1994.

[14] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble security. Technical Report TR98-1703, Cornell University, Sept. 1998.

[15] 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-Verlang, 1998.

[16] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report TR97-1638, Cornell University, July 1997.

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

[18] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the IEEE Third Information Survivability Workshop (ISW-2000)*, Boston, Massachusetts, USA, Oct. 2000.