The Starfish System: Providing Intrusion Detection and Intrusion Tolerance for Middleware Systems

Kim Potter Kihlstrom
Department of Mathematics and Computer Science
Westmont College
955 La Paz Road, Santa Barbara, CA 93108
kimkihls@westmont.edu

Priya Narasimhan
Electrical and Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3890
priya@cs.cmu.edu

Abstract

We introduce the Starfish[1] *system, a new system that provides intrusion detection and intrusion tolerance for middleware applications operating in an asynchronous distributed system. The Starfish system contains a central, highly secure and tightly coupled core. This core is augmented by "arms" that are less tightly coupled and that have less stringent security guarantees, each of which can be removed from the core if a significant security breach occurs. New arms can be "grown" as needed.*

The Starfish system aims to employ a number of techniques for providing intrusion detection and intrusion tolerance. The specific challenges that we will address in this paper are infrastructural support for voting and end-to-end intrusion detection.

1. Introduction

As middleware technologies such as Jini, EJB, CORBA, and DCOM have become more important for designing large software systems, it is has become increasingly important to design algorithms and develop mechanisms to provide intrusion detection and intrusion tolerance capabilities in middleware systems. Intrusion tolerant systems are systems that continue to operate correctly and reliably despite intrusions that cause some nodes to behave in an arbitrary or malicious manner. While traditional computer security concerns keeping intruders out of a system, our goal is detection of an intrusion and continued provision of useful services even during or after a successful intrusion.

It is of crucial importance to protect critical applications such as electronic banking and commence systems, the national power control grid, air traffic control, and medical monitoring and support systems, from malicious intrusions and corruption. While some work has been done in this area, much remains to be investigated and many questions remain open. We must develop algorithms and mechanisms for providing intrusion tolerance to middleware systems, and provide a framework for understanding intrusions in such systems. It is critical that we describe properties that must be achieved by intrusion tolerant systems, so that middleware systems can be evaluated against these properties.

As shown in Figure 1, the Starfish system consists of a tightly coupled and highly secure core containing critical components, as well as arms providing less stringent security guarantees and containing less critical components. In the event of a security breach, an arm can be removed. Additionally, new arms can be grown.

[1]Starfish are known to have small bodies, out of which spring forth a varying number of arms, which break off when damaged. These arms subsequently heal and re-grow. Detached starfish arms (also called "comets") can also regenerate new bodies.

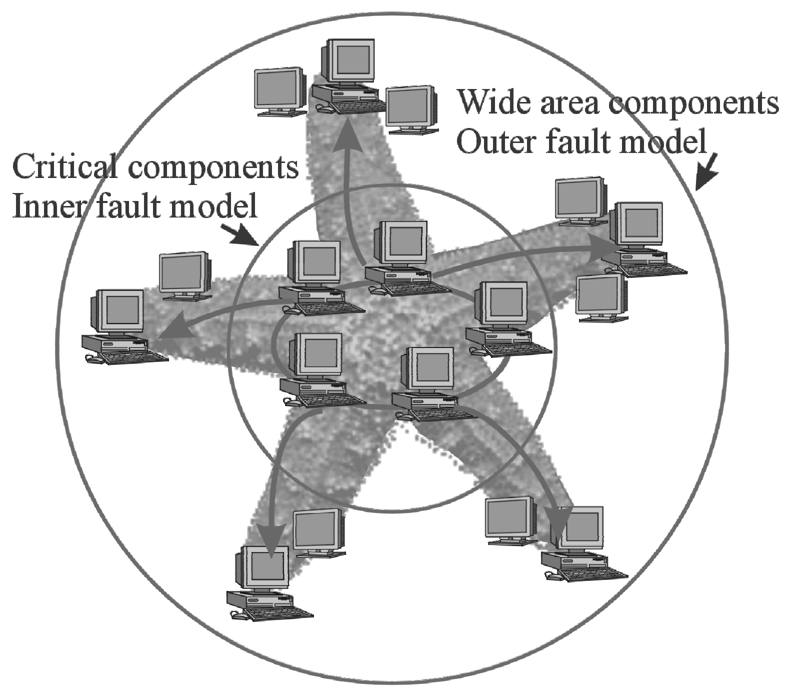


Figure 1. The Starfish system.

Our work explores the challenges underlying providing intrusion tolerance for complex middleware applications in a distributed asynchronous system. The specific challenges that we will address in this paper are infrastructural support for voting and end-to-end intrusion detection. One of the goals of the mechanisms and the algorithms that we have developed, and are continuing to develop, is to be widely applicable across a variety of middleware technologies and, where possible, to transcend differences in diverse middleware implementations and standards. Drawing on our previous work on CORBA, we will investigate, in this paper, generic algorithms, techniques and mechanisms that will be essential foundations for the intrusion tolerant middleware systems of the future.

2. System model

We consider an asynchronous distributed system consisting of processors that communicate via messages over a network. Processors have access to local clocks, but these clocks are not synchronized.

The system is subject to communication, processor, and object faults. Communication between processors is unreliable, and thus messages may need to be retransmitted and may be arbitrarily delayed. Communication channels are not assumed to be FIFO or authenticated.

Processors are either correct or faulty. *Correct processors* always behave according to their specification. *Faulty processors* exhibit arbitrary (*Byzantine*) behavior. Because a Byzantine processor can behave as if it crashed, crash faults are considered as Byzantine faults.

A faulty processor may attempt to disrupt the system by sending different information to different processors, purporting that it is the same information, or by selectively sending messages to some processors and not to others. Further, a malicious processor can attempt to thwart the consistent ordering of messages by sending messages to different processors in a different order. Such a processor can also masquerade as another processor by including the identifier of another processor in the messages it sends. A faulty processor can refuse to send messages, can send messages that are syntactically or semantically incorrect, or can claim that another processor is faulty.

Objects are also subject to crash faults or may behave in an arbitrary or malicious manner. *Correct objects* always behave according to their specification and do not crash. A *faulty object* may fail to send a message as required by the application, which we refer to as a *send omission* fault. A faulty object may also send a message that contains an incorrect (corrupted) invocation or response, which we refer to as a *value* fault.

3. Infrastructural support for voting

In an environment subject to malicious faults due to intrusions, a single object may be corrupted and provide incorrect semantics, including providing an incorrect response to an invocation. Thus, to provide intrusion tolerance, it is crucial to replicate objects. We make use of state machine replication [17] to actively replicate objects. In the case of a deterministic object, if all of the object replicas begin in the same state and process the same updates in the same order, then consistency will be maintained.

However, maintaining replica consistency in an environment subject to malicious faults is a complex problem. Ensuring that all objects process the same updates in the same order is very difficult. We must ensure, for example, that a malicious processor is unable to disrupt the system by sending different information to different processors, purporting that it is the same information. Secure reliable group communication systems can ensure that all of the processors receive the same information, even if some of the processors in the system behave maliciously. Throughout the rest of this paper, we assume the existence of an underlying group communication system such as the SecureRing system [8] that provides secure reliable totally-ordered message delivery and processor group membership services despite malicious processor faults.

We make use of the object group abstraction, in which the replicas of an object form an *object group* and the degree of replication is the size of the object group. An *object group interface* can be employed to hide the details of the underlying group communication protocols. The object group interface allows an object to transparently send an invocation or response to all of the replicas of an object; *i.e.*, to the object group. The object group interface also allows replicas to join and leave (or be removed from) an object group, and for object groups to be created and eliminated.

Fault-tolerant systems typically perform some type of voting. Most work on intrusion tolerance has focused on faulty servers, with clients performing majority voting on responses from replicated server objects. However, in an intrusion tolerant system we must also consider faulty clients, and ensure that such clients are unable to corrupt the state of server objects. We advocate replicating clients as well as servers, and we thus perform majority voting on both invocations and responses, as shown in Figure 2.

There are key questions with regard to voting in a dynamic environment, in which object replicas may be created and removed. In particular, it is crucial to be able to determine how many votes from a particular replicated object constitute a majority, given the current status of the system. To send an invocation or a response to an object group, the originating object does not need to be aware of the number or location of the replicas of the target object. Typically, while the object group layer on a particular processor would process membership changes regarding object groups hosted by the processor, for reasons of scalability it would not receive such membership changes regarding other object groups that it does not host. However, to vote on incoming invocations or responses, the voter at the target replica must know the number of the replicas in the *originating* object group (in order to know how many votes to expect and, thus, how many votes constitutes a majority). To handle this, we propose, in addition to prior work, a hierarchical membership structure that consists of both object groups and *voting groups*, as well as the use of a *voting group interface*, as shown in Figure 3.

The hierarchical group structure allows membership changes to be selectively disseminated only to interested parties, and is thus scalable while providing the information necessary for voting. When a client object group G_{O1} sends an invocation to a server object group G_{O2}, a voting group G_V is formed that contains both G_{O1} and G_{O2}. Membership information for both G_{O1} and G_{O2} is maintained locally and, when there is a change in object group membership of either G_{O1} or G_{O2}, all of the replicas in the voting group G_V are made aware of those changes. The voting group interface allows members of a voting group to transparently vote on the invocations or responses from other members of that voting group. Thus, G_{O2} is able to transparently vote on invocations from G_{O1}, and G_{O1} is able to transparently vote on responses from G_{O2}.

4. End-to-end intrusion detection

Various types of faults can be manifested as the result of an intrusion. Some of these include communication faults, processor faults, and object faults. Once various types of faults have been identified and catego-

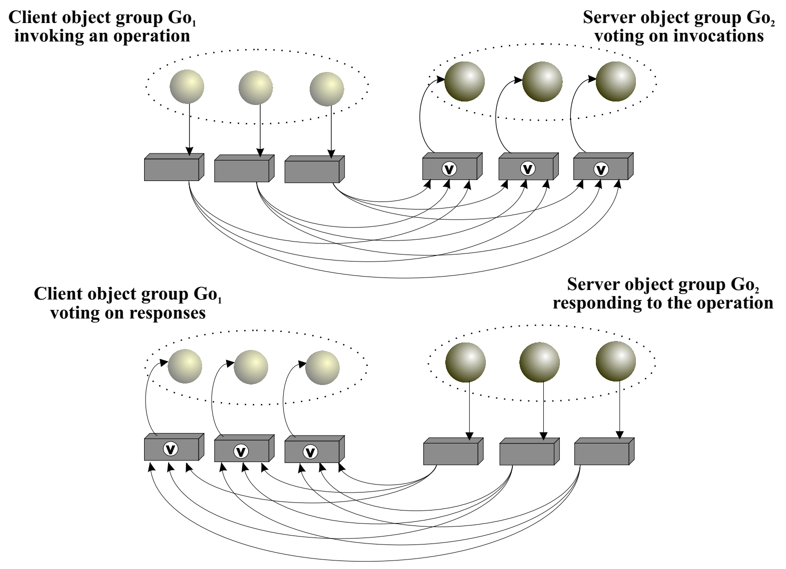

Client object group Go_1 invoking an operation
Server object group Go_2 voting on invocations
V
Client object group Go_1 voting on responses
Server object group Go_2 responding to the operation

Figure 2. Majority voting on invocations and responses.

rized, strategies must be developed to protect systems from these types of faults.

As described above, membership occurs in several levels. At the lowest level, there is the collection of all processors that host objects in the system. We refer to this as *processor* membership. Next, there is voting group membership, which encompasses the client and server object groups that are communicating at any particular point in time. Finally, there is object group membership, in which an object group consists of all of the replicas of an object. All of these memberships are dynamic and change as intrusions are detected, as new objects are created or modified in their degree of replication, and as objects communicate through invocations and responses.

Prior work on Byzantine fault detectors [9] can be applied to intrusion detection in middleware. These techniques allow for the detection and removal of corrupted processors. Each processor has access to a local fault detector module that provides hints about which processors are faulty. The local fault detector monitors messages that are sent and received, and provides an output consisting of a list of processors that it currently suspects.

It is important for intrusion tolerant middleware systems to be adaptive. It is crucial not only to detect Byzantine faults, but also to remove the faulty object or processor. There are many issues to be resolved, such as whether to remove only an object replica that has been found to be faulty, or to remove the processor hosting the replica as well. Another key question is how to ensure that a malicious object replica is not able to cause the removal of a correct object or processor by claiming that it is faulty.

When an object replica exhibits a value fault that is subsequently detected through the voting mechanism, two options are available. The first possibility is simply to remove the faulty replica from the object group of which it is a member, and all voting groups of which its object group is a member. This is the easiest course of action, and may be appropriate when there is reasonable confidence that one faulty object is unable to corrupt another object on the same processor. The process of removing a faulty replica is illustrated in Figure 4. In Figure 4(a), a faulty replica r that is a member of object group G_{O2} sends a vault fault to another object group G_{O3}.

In order to ensure that faulty replica r is removed from its object group G_{O2}, it is necessary that all of the replicas in G_{O2} receive evidence of the value fault. To ensure that this happens, a *Value_Fault_Vote* message is employed. This special message contains embedded votes from each of the object replicas in group G_{O2}, including the faulty vote from replica r. A means of authentication must be employed in order to ensure that a malicious replica is not able to forge an incorrect vote from a correct processor. The *Value_Fault_Vote* message is sent from the target replicas in object group G_{O3} that detected the value fault back to the originating object group G_{O2}, as shown in Figure 4(b). The object group layer makes use of a *value*

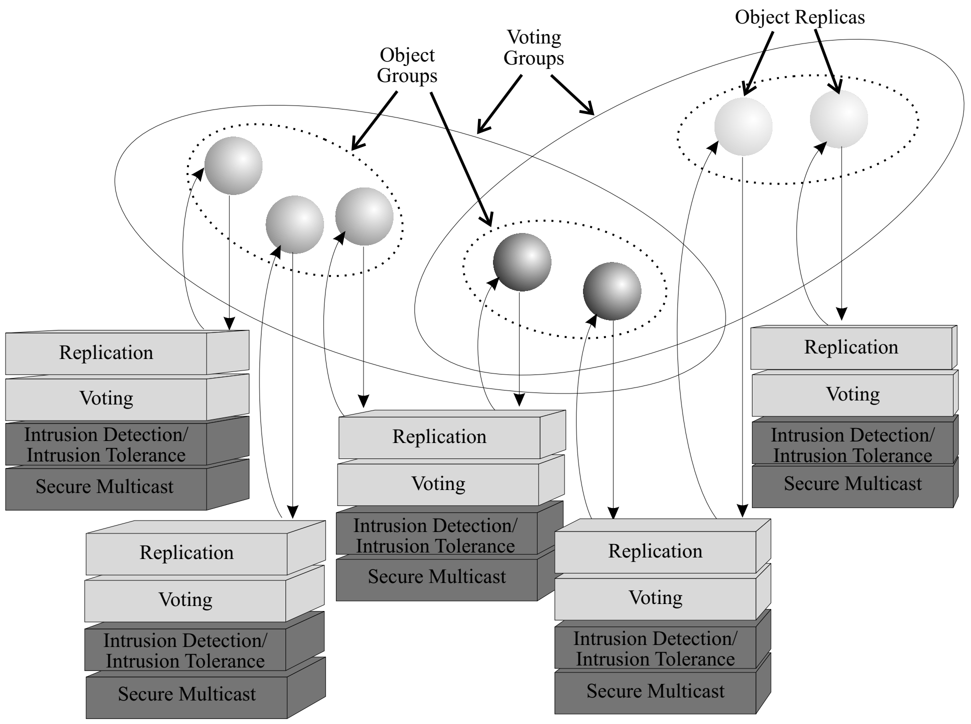


Figure 3. Voting groups and object groups.

fault detector, which votes on the values in the messages of a given set and identifies a corrupt sender replica as well as the object group of which it is a member. The object group interface is then employed to remove the faulty object replica from object group G_{O2}, as shown in Figure 4(c). Due to the voting group mechanism, the change in the object group membership is then propagated to all of the object groups that are members along with group G_{O2} in a voting group (in this example, object groups G_{O1} and G_{O3} in voting groups G_{V1} and G_{V2}). This process is illustrated in Figure 4(d).

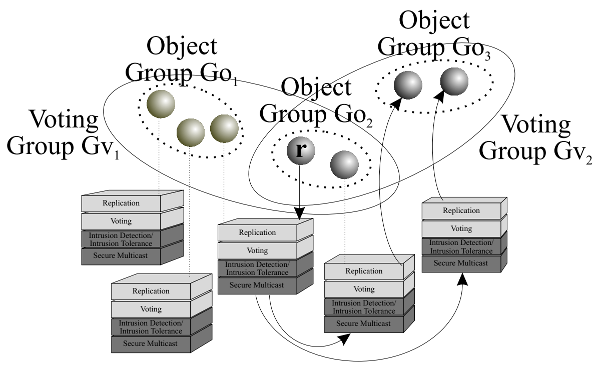
In addition to simply removing a faulty object replica from its object group and voting groups, the second option is to remove the processor hosting the faulty object from the system. This is the safest course of action because the corruption of any object residing on a processor may affect the integrity of other objects in residence. However, this semantic requires additional support mechanisms.

To ensure that all processors handle a value fault due to a corrupt replica as a malicious processor fault, the following completeness property of the Byzantine fault detector module must be satisfied:

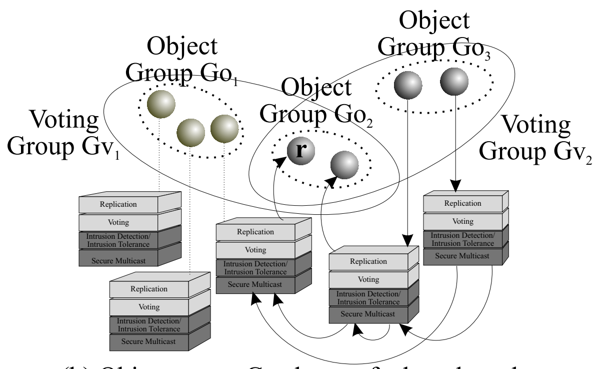
- There is a time after which every processor that has exhibited a fault is suspected by every correct processor

This requires that all of the processors in the system, not merely those that host objects that are members of object or voting groups with the faulty replica, must vote locally on the same set of votes and reach the same decision. To achieve this, a special *base group* containing all of the processors is employed. A value fault detector is used at the processor level and, on detecting an incorrect vote, the value fault detector sends a *Value_Fault_Vote* message to the base group. The value fault detector at each receiving processor compares this set of invocations or responses to determine the corrupt client or server replica and hosting processor.

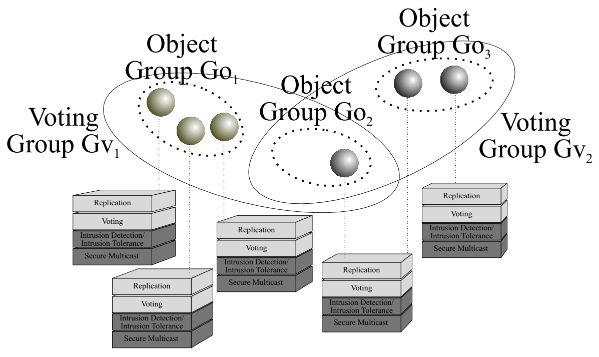
A subtle problem may occur here if care is not taken in the mechanism. If a malicious object is not removed immediately at the application level, a problem may occur with cascading in which the faulty object continues to send incorrect values, each of which causes receiving processors to send a *Value_Fault_Vote* message to the base group. Effectively, this can result in a denial of service attack if not handled properly. It is important to handle



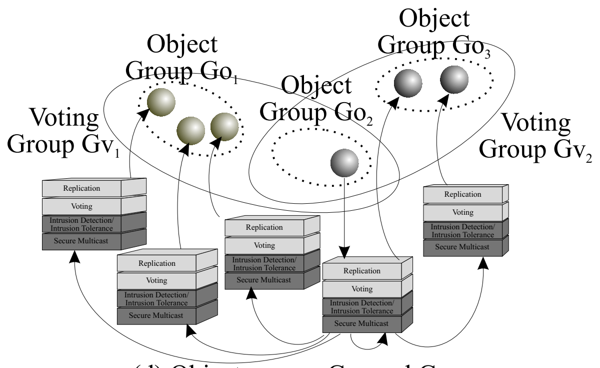
(a) Faulty object replica r (shown shaded) sends value fault to object group Go$_3$



(b) Object group Go$_3$ detects fault and sends *Value_fault_vote* message to object group Go$_2$



(c) Faulty object replica r is removed from object group Go$_2$



(d) Object groups Go$_1$ and Go$_3$ receive membership update on Go$_2$

Figure 4. Steps in removing a faulty replica.

the fault initially at the application level, by removing the faulty replica from its object and voting groups as described above, before handling the fault at the processor level.

5. Related work

This work builds on the authors' experience with the Immune system [15]. The Immune system aims to provide survivability to CORBA applications transparently, enabling them to continue to operate despite intrusions or accidents that damage the underlying distributed system, or faults that occur within the system. The Immune system can protect an existing unmodified CORBA application, running over an unmodified commercial ORB, against arbitrary faults, including those that arise from malicious attacks within the system. Every object within the CORBA application is actively replicated by the Immune system, with majority voting applied on incoming invocations and responses to each replica of the object.

The Immune system grew out of the authors' work on two projects: the Eternal system and the SecureRing system. The SecureRing system [8] provides secure reliable totally-ordered message delivery and group membership services despite the malicious corruption of some of the processors. The protocols multicast messages to groups of processors and deliver messages in a consistent total order to all members of the group. The Eternal system [13] aims to provide support for fault tolerance and evolution to CORBA-based applications. The application objects are replicated and distributed across the system, and the consistency of the states of the replicas is maintained by exploiting the facilities of the underlying group communication protocol for all inter-object communication.

Phalanx [11] is a software system that enables wide-area applications to survive the malicious corruption of clients and servers. Phalanx is a persistent object store that provides support for shared data abstractions such as files and locks. Phalanx is designed to scale to a large number of servers, and employs quorum constructions [10] that enable efficient scaling. Phalanx was developed for use in an electronic election trial during the 1998 Cost Rican presidential elections.

Fleet [12] is a middleware system that implements a distributed object repository. Fleet generalizes Phalanx in several ways, including support for persistent objects of arbitrary types, techniques for fault monitoring, and capabilities for dynamically extending the infrastructure with new object types. Fleet objects are persistent in that the object may outlive the client that created it. Fleet objects are replicated and retain correct semantics despite the malicious corruption of servers. Fleet is designed for Java/Jini applications and makes use of Byzantine quorum systems [10] to achieve scalability.

The Object Group Service [6] is a CORBA-compliant approach that provides fault tolerance for CORBA applications through a set of CORBA services. Mechanisms are provided to allow for the delivery of an invocation (response) to a server (client) object only if a majority of copies of the invocation (response) have been received, though not voted on for verification of their contents or value, from the replicated client (server) object.

The AQuA architecture [4] provides a CORBA-based dependability framework that can tolerate processor crash faults and replica crash faults, as well as send omission faults and value faults due to an corrupted object. The AQuA architecture exploits the facilities of the underlying Ensemble and Maestro toolkits [18], and uses majority voting at the application object level, to detect an incorrect value of an invocation (response) from a replicated client (server). AQuA does not provide for the detection, or tolerance, of malicious processor faults or message corruption faults.

COPE [14] is a set of CORBA security policies and CORBA-based services that is being developed to provide intrusion tolerance. An implementation has been developed using the commercial ORB, OrbixWeb, from Iona Technologies. COPE allows a method of any CORBA object in the application to be invoked only as determined by the application access policy for that object, based on permissions and authentication. Optimistic protocols serve to contain the damage done by an intruder before it is detected.

The BFT library [2] is a state machine replication system that tolerates Byzantine faults. It achieves high performance because it uses digital signatures only for view-change and new-view messages, and authenticates other messages using message authentication codes (MACs). A more recent extension [3] provides for replicas to be recovered proactively, allowing the system to tolerate any number of faults over its lifetime provided that fewer than 1/3 of the replicas are faulty within a small time period. A state transfer mechanism is also provided. BASE [16] is a further extension of BFT that enables replicas to run different or non-deterministic implementations. This is done by defining a common abstract specification for a service and then implementing a wrapper for each distinct implementation to make it behave according to the common specification.

The FRIENDS [5] system aims to provide mechanisms for building fault-tolerant applications in a flexible way through the use of libraries of metaobjects. Separate metaobjects can be provided for fault tolerance, security and group communication. FRIENDS is composed of a number of subsystems, including a fault tolerance subsystem which provides support for replication and detection of faults. The secure communication subsystem includes support for authentication, authorization and audit. While every method invocation on a server can be signed at the client and authenticated at the server, majority voting is not used to determine the delivery of the invocation at the server.

The Cactus project [7] is a framework aimed at providing a flexible way of customizing different levels and strategies for survivability through the combination of components known as micro-protocols, each implementing a specific property. Fault tolerance is implemented as a micro-protocol which supports active and passive replication. Security is provided through micro-protocols that implement various authentication and encryption methods. GroupRPC is a middleware service, implemented using the Cactus approach, which would allow an application to tolerate crash, omission, timing and Byzantine faults.

The WAFT system [1] is a distributed object system that aims to support replication and security strategies suited to wide-area applications. The fault detection mechanism that WAFT employs is able to handle performance faults, such as a failure to comply with a specific quality of service requirement.

6. Conclusions and future directions

The increasingly complex and unpredictable world in which we live requires a greater understand of intrusion detection and the issues involved in the design of intrusion tolerant middleware systems. We must continue to develop mechanisms, tools, and techniques to aid in the development of such systems.

We have presented two general mechanisms for building intrusion tolerant systems. The first is the use of hierarchical groups for scalability. We introduced the idea of a voting group and a voting group interface for selective propagation of membership information and transparent voting. Our second mechanism is that of end-to-end intrusion detection at the object group, voting group, and processor level. These mechanisms, as well as others, are illustrated in the Starfish system currently under development.

Areas for future work include investigation into scalable techniques to achieve replication. Byzantine quorums [10] are a recent advance in this direction, but more work is necessary. Another key question is how to deal with denial of service attacks; very little progress has been made in this area to date. Work is also needed in implementing diversity in servers so that a successful attack on one server cannot be easily extended to other servers.

Another area for future work lies in considerations for recovery of processors and replicas. It is important that repaired objects and processors be allowed to rejoin the system. However, this is a difficult problem in systems subject to malicious faults. Issues of what to recover and what state to transfer when a new or repaired processor comes up, or when a new or repaired replica comes up, must be considered.

References

[1] L. Alvisi and K. Marzullo. WAFT: Support for fault-tolerance in wide-area object oriented systems. In *Proceedings of the 1998 Information Survivability Workshop*, pages 5–10, Orlando, FL, Oct. 1998.

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–86, New Orleans, LA, Feb. 1999.

[3] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 273–87, San Diego, CA, Oct. 2000.

[4] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, Oct. 1998.

[5] J.-C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[6] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[7] M. A. Hiltunen, R. D. Schlichting, and C. A. Ugarte. Survivability issues in Cactus. In *Proceedings of the 1998 Information Survivability Workshop*, pages 85–88, Orlando, FL, Oct. 1998.

[8] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, Nov. 2001.

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[10] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[11] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, West Lafayette, IN, Oct. 1998.

[12] D. Malkhi, M. Reiter, D. Tulone, and E. Ziskind. Persistent object in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition*, pages 126–36, Anaheim, CA, June 2001.

[13] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, Apr. 1998.

[14] C. Namprempre, J. Sussman, and K. Marzullo. Implementing causal logging using OrbixWeb interception. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 257–267, San Diego, CA, May 1999.

[15] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

[16] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *Operating Systems Review (Proceedings of the 18th ACM Symposium on Operating System Principles)*, 35(5):15–28, Oct. 2001.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[18] A. Vaysburd and K. P. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.