

Providing Support for Survivable CORBA Applications with the Immune System*

P. Narasimhan, K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
{priya, kimk, moser, pmms}@alpha.ece.ucsb.edu

Abstract

The Immune system aims to provide survivability to CORBA applications, enabling them to continue to operate despite malicious attacks, accidents, or faults. 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 Secure Multicast Protocols are employed to enable the majority voting to be effective, even when processors within the network and objects within the application become corrupted.

1 Introduction

Distributed computing and object-oriented programming are paradigms that are increasingly favored for today's applications. The Common Object Request Broker Architecture (CORBA) [12] is a commercial standard for building distributed object applications that frees application developers from dealing with the issues of distribution and networking. This is accomplished through the Object Request Broker (ORB), which serves to communicate operations between the CORBA objects that comprise the distributed application. The location transparency, interoperability and portability that CORBA provides make it an attractive framework for building distributed applications.

The distribution of computing and resources that characterizes a distributed application can enhance its performance. However, it also raises concerns about the reliability and security of the application. Reliability encompasses the system's ability to tolerate accidents or faults. Security involves protecting the resources of the system against malicious attacks. *Survivability* [2] goes beyond security and reliability in that survivable systems must continue to pro-

vide critical services even after the occurrence of attacks, accidents, or faults.

Unfortunately, while CORBA provides many desirable features, it lacks intrinsic support to meet the survivability requirements of critical applications. The goal of the Immune system is to protect an existing CORBA application against intrusions or accidents that damage some portion of the underlying distributed system, or faults that occur within the system.

2 The Immune System

The Immune system, illustrated in Figure 1, transparently enhances any commercial implementation of CORBA with support for survivability. Unmodified CORBA client and server objects that comprise the application can benefit from the services that the Immune system provides, while operating over an unmodified commercial ORB.

The interfaces for the server objects within the application are defined in the Interface Definition Language (IDL), and are used to generate a stub and a skeleton for each server object. At a CORBA client, the stub serves to marshal, and dispatch to the ORB, the client's invocation of the server object. At the CORBA server, the skeleton is used to unmarshal the invocation for delivery to the server object. The client's invocations and the server's responses are conveyed by the ORB over CORBA's Internet Inter-ORB Protocol (IIOP).

The Immune system transparently intercepts [10] the IIOP messages, which are intended for TCP/IP, and passes them to the Replication Manager. The Immune system exploits the facilities of the underlying Secure Multicast Protocols to provide secure reliable totally ordered message delivery. By mapping the intercepted IIOP messages onto the Secure Multicast Protocols, the Replication Manager ensures that the client-server interactions are communicated in multicast messages, without modification of either the application objects or the ORB.

Because the Immune system is aimed at supporting critical applications that must tolerate arbitrary faults,

*Effort sponsored by DARPA and Air Force Research Laboratory, Rome, under contracts F30602-95-1-0048 and F30602-97-1-0284, and by DARPA and the Office of Naval Research, under contract N00174-95-K-0083.

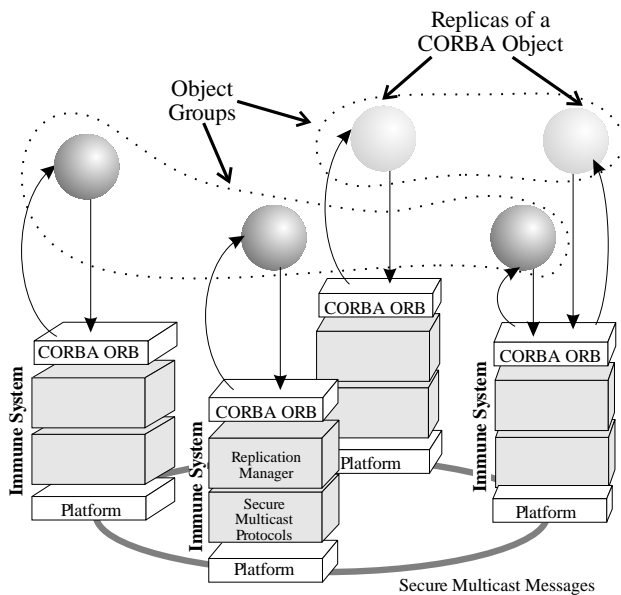


Figure 1: The structure of the Immune system.

including those that arise from the incorrect behavior of the application itself, *active replication with majority voting* must be employed for every client and server object of the application. The Replication Manager exploits the Secure Multicast Protocols to communicate the operations to the replicas and to maintain the consistency of the states of the replicas of each application object.

3 System Model

The underlying model of the Immune system is an asynchronous distributed system, in which processors communicate via messages over a local-area network that is completely connected. The system is asynchronous in that no bound can be placed on the time required for a computation or for the communication of a message.

In the Immune system, the *object group* abstraction is used to model a replicated object, where a group member is one of the replicas of the object. The size of the object group corresponds to the degree of replication of the object. An object group typically spans several processors. A processor may host multiple object replicas and multiple object groups.

The system is subject to communication, processor, and object replica faults, as shown in Table 1. Communication between processors is unreliable and, thus, messages may need to be retransmitted and may be corrupted or arbitrarily delayed. Processors and replicas may crash, and may be corrupted and behave in an arbitrary or malicious manner. Occurrences of such behavior can be handled as *Byzantine*

faults. *Correct* processors and replicas always behave according to their specification and do not crash.

Communication channels are not assumed to be FIFO or authenticated; however, the network is assumed not to partition. Every replicated object is assumed to be deterministic, *i.e.*, if two active replicas of the object start from the same initial state, and apply the same sequence of messages in the same order, they will reach the same final state.

3.1 Considerations for Resilience

The Secure Multicast Protocols detect crashed and Byzantine processors, and ensure the removal of replicas hosted by such processors. To avoid the unnecessary loss of replicas due to the removal of a malicious processor, at most one replica of an object is allocated to a processor. However, replicas of different objects may coexist on the same processor.

If a malicious processor fault is detected, all objects that are hosted by that processor are subsequently excluded from the memberships of all object groups. This is the safest course of action because the corruption of any portion of a processor is likely to affect the integrity of any of the objects that it hosts. The replicas that are lost due to a Byzantine processor must be reallocated to correct processors.

Survivability requirements, such as the degree of replication and the number of processors, are greater for systems that must tolerate malicious faults, compared to systems that tolerate benign faults. We require that at least $\lceil (2n + 1)/3 \rceil$ processors must be correct in a system of n processors. We also require that at least $\lceil (r + 1)/2 \rceil$ replicas must be correct for an object that has r replicas.

4 Requirements for Survivability

With active replication, each replica of a server (client) object must receive and process each invocation (response) addressed to the server (client) object. For replica consistency, the following requirements must be met.

- **Reliable delivery.** Each distinct invocation or response must be received exactly once by every replica of the target object.
- **Total ordering.** All of the server (client) replicas must receive and process the same sequence of invocations (responses) in the same order.

In addition, for the proper operation of the majority voting algorithm, the Immune system requires that the following properties hold.

- **Authentication.** A faulty replica must not send messages that appear to be sent by a correct replica, by masquerading as that replica. If this were allowed, a

Classification	Fault	Description	Mechanisms (Section)
Communication	Message loss	Message not received by a processor or processors	Reliable delivery mechanisms (7.1) Message retransmission (7.1)
	Message corruption	Message corrupted in transit from source to destination	Message digest in token (7.1) Message retransmission (7.1)
Processor	Processor crash	Processor ceases to send messages	Processor membership (7.2) Object group membership (5) Use of replicas on other processors (5)
	Failure to receive message	Processor repeatedly fails to acknowledge a message	Processor membership (7.2) Object group membership (5) Use of replicas on other processors (5)
	Malicious (Byzantine) processor	Processor masquerades as another processor, or sends mutant or improperly formed messages	Message digests in token (7.1) Signature in token (7.1) Checking mechanisms (7.3)
Object Replica	Replica crash	Replica crashes or cannot be contacted	Object group membership (5) Use of replicas on other processors (5)
	Send omission	Failure of a particular client (server) replica to send an invocation (response)	Majority voting on all invocations and responses (6.1)
	Value fault	Incorrect value for invocation (response) received from a particular client (server) replica	Majority voting on all invocations and responses (6.1) Value fault detection (6.2)

Table 1: Types of faults handled by the Immune system.

malicious replica could send an incorrect invocation (response) multiple times, each time masquerading as a correct replica and, thus, could ensure that its incorrect invocation (response) would be selected by the voting algorithm.

- **Eventual exclusion of malicious replicas.** A malicious replica, once detected, must be excluded from the object group membership, and must not be allowed to corrupt the other application objects.
- **Suppression of mutant messages.** Different, or *mutant*, versions of the same invocation (response) message, sent by a faulty client (server) replica to different replicas in a server (client) group, must not be delivered. This prevents such mutant messages from corrupting the states of other application objects.

The underlying Secure Multicast Protocols provide these guarantees on every message that they deliver to the Replication Manager. Thus, all of the messages received by the Replication Manager, and consequently its voters, have these properties. This, in turn, implies the ability of the Replication Manager to provide guarantees (for replica consistency and majority voting) on the invocations and responses that it delivers to the application objects.

Furthermore, at the Replication Manager level, the effect of incorrect behavior of faulty replicas is restricted. For example, while an incorrect response from a faulty server replica will be delivered at the client replicas, different responses to different client replicas will not be delivered.

By requiring the underlying Secure Multicast Protocols to handle all of the processor and communication faults shown in Table 1, the Immune system enables the object replica faults to be handled by the Replication Manager. Thus, the ability of the Secure Multicast Protocols to handle processor corruption enables the Replication Manager to handle replica corruption.

5 Support for Active Replication

With object replication, it is crucial to maintain the consistency of the states of the replicas of an object. To provide continuous service despite the occurrence of faults, it is necessary to be able to detect, and recover from, the failure of either an object replica or a processor.

The object group interface of the Immune system enables an object to invoke the services of another object group in a transparent manner so that the invoker of the operation need not be aware of the exact nature, location, membership, degree of replication, or type of replication of the objects.

Critical applications that must tolerate value faults, in addition to crash faults, require majority voting and, thus, the use of active replication for every object of the application. Thus, in the Immune system, an object group is synonymous with an actively replicated object.

To achieve active replication, the Replication Manager employs the object group interface, as shown in Figure 2, to ensure that each active replica of an object receives all of the messages destined for the group corresponding to the replicated object. To enable the delivery of messages to

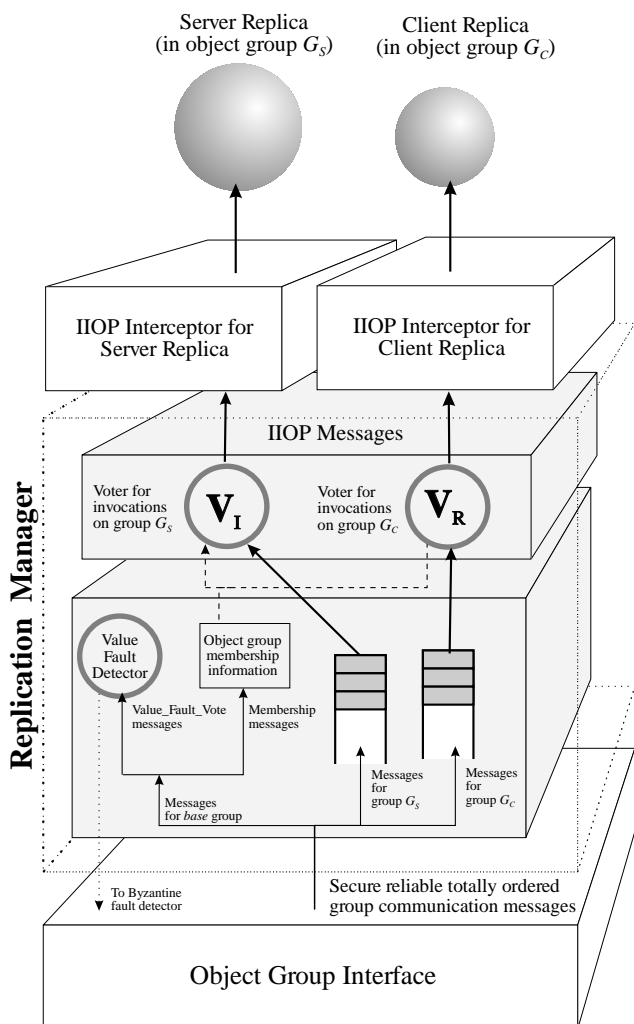


Figure 2: The Replication Manager of the Immune system.

the application through the Secure Multicast Protocols, the Replication Manager encapsulates the server (client) object group identifier into the secure reliable multicast messages that contain the IIOP invocations (responses) from (to) a client object group.

The Secure Multicast Protocols, which communicate operations between object groups, facilitate replica consistency by ensuring the secure reliable total ordering of operations within and across object groups. The object group interface benefits from the secure reliable total ordering guarantees of the underlying Secure Multicast Protocols. The services of the underlying Secure Multicast Protocols are made available to the Replication Manager through the object group interface, which hides the complexity of the protocols from the Replication Manager.

Active replication incurs the computational cost of requiring each active server replica to perform the same task, and the communication cost of multicasting the same invocation (response) from each client (server) replica.

5.1 Duplicate Detection

When a client (server) object is actively replicated, each of the client (server) replicas issues the same invocation (response) to the target server (client). It is essential that these multiple copies of the same invocation (response) are never delivered to the server (client), whose state might then be corrupted by processing the operation multiple times.

The Replication Manager enables the detection, at the target object, of the copies of operations sent by the replicas of an object through the use of unique *operation identifiers*. These operation identifiers give rise to invocation and response identifiers, as shown in Figure 3.

The invocation identifier uniquely identifies an invocation by the client object, and is identical in the first two fields for each of the client replicas. The Replication Manager at each of the server replicas uses a response identifier when returning the results of the invocation. The response identifier is identical to the invocation identifier in the first two fields, and enables the Replication Manager at each of the client replicas to associate the copies of the returned response with the invocation.

6 Support for Majority Voting

In an application subject to arbitrary faults, it is possible for any one of multiple client (server) replicas to send a corrupted invocation (response). Thus, while the copies of the invocation (response) received at the target object represent the same operation, they may differ in value.

Because the Immune system aims to support survivable applications that must tolerate arbitrary faults, including those that arise from incorrect behavior of the application, the Replication Manager employs majority voting on the copies of the invocations and responses delivered to each application object.

6.1 Mechanisms for Majority Voting

To send messages to a target object, the Replication Manager does not need to know the number, or the location, of the target replicas. However, majority voting on the values of the copies of messages representing the same invocation (response) requires the voter at a target server (client) replica to know the number of replicas in the sender client (server) object group.

One way that the voters could obtain this information, is by having every Replication Manager join a special *base group* that serves to disseminate membership changes and membership information. In the Immune system, as

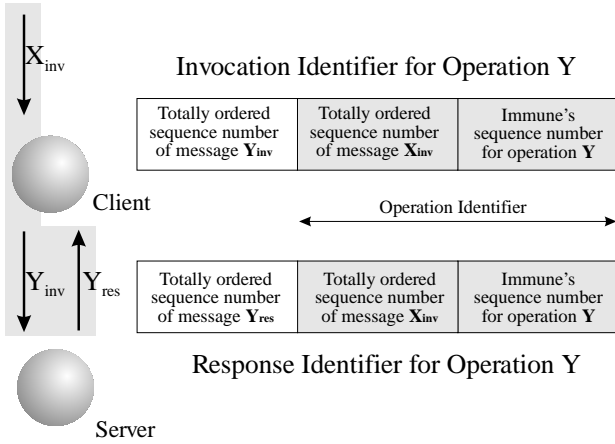


Figure 3: Invocation and response identifiers corresponding to an operation.

shown in Figure 2, object group membership messages (generated by the object group interface due to the addition or removal of a replica) are received, but not forwarded to the application, by every member of the base group. On receipt of such messages, the Replication Manager updates the membership information that it must maintain to perform majority voting.

Because the Immune system is designed to support critical applications that must be survivable, both client and server objects are replicated. Thus, Immune provides support for, and uses, both *input majority voting* (voting on client invocations at the server end) and *output majority voting* (voting on server responses at the client end).

In the case of input (output) majority voting, the Replication Manager at each of the server (client) replicas detects the copies of each invocation (response) using the invocation (response) identifiers in the messages. By virtue of its membership in the base group, the Replication Manager at each of the server (client) replicas knows the current degree r_c (r_s) of replication of the client (server) object or, alternatively, the number of copies of each invocation (response) to expect. From its knowledge of r_c (r_s), the Replication Manager can determine the majority of client (server) replicas. The Replication Manager at each of the server (client) replicas receives the copies of an invocation (response) and dispatches them to a voter for invocation (response) of (to) the server (client) replica.

The voting algorithm is deterministic and produces the same result for each invocation (response) at every server (client) replica. For each invocation (response) from a client (server) object, when the voter receives a majority of copies that it verifies as being identical in value, the voting algorithm produces a single result, which the Replication Manager then delivers as an invocation (response) of (to)

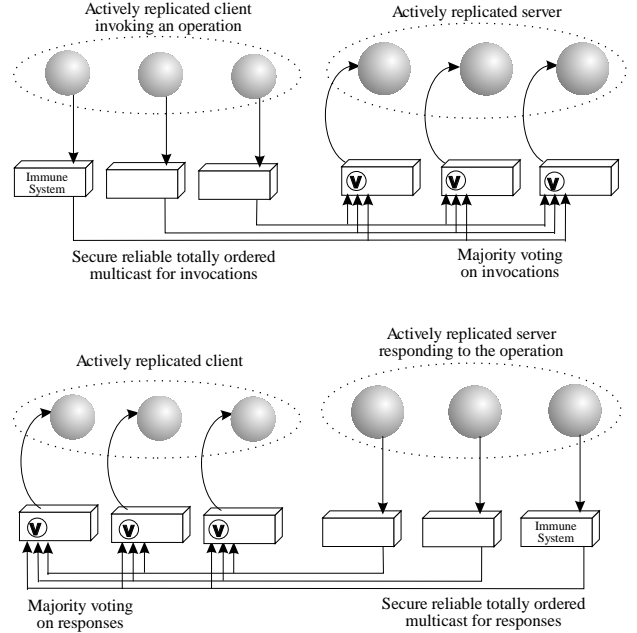


Figure 4: Active replication with input (output) majority voting on invocations (responses).

the target server (client) replica. For every output result that the voter produces, the Replication Manager discards all subsequently received messages that are copies of the delivered invocation (response). Figure 4 shows both input and output majority voting at the target replicas.

Figure 2 shows the interaction between the object group interface and the voting mechanisms. The Replication Manager receives all of the secure reliable totally ordered multicast messages destined for the groups that it hosts, and filters the messages based on their destination groups, passing on to its voters only those messages that are destined for the target replicas with which the voter is associated. The voters then decide on the delivery, to their associated replicas, of the messages that they receive.

6.2 Value Fault Detection

As shown in Figure 2, the voter V_I (V_R) within the Replication Manager can detect an incorrect value of an invocation (response) sent by a corrupted client (server) replica. To handle a value fault due to a corrupt replica as a malicious processor fault, the *eventual strong Byzantine completeness* property (defined in Table 5) of the Byzantine fault detector module within the Secure Multicast Protocols must be satisfied. This requires that all of the Replication Managers within the system (and not merely those that first detected the value fault through their V_I or V_R) must vote locally on the same set of invocations and reach the same decision.

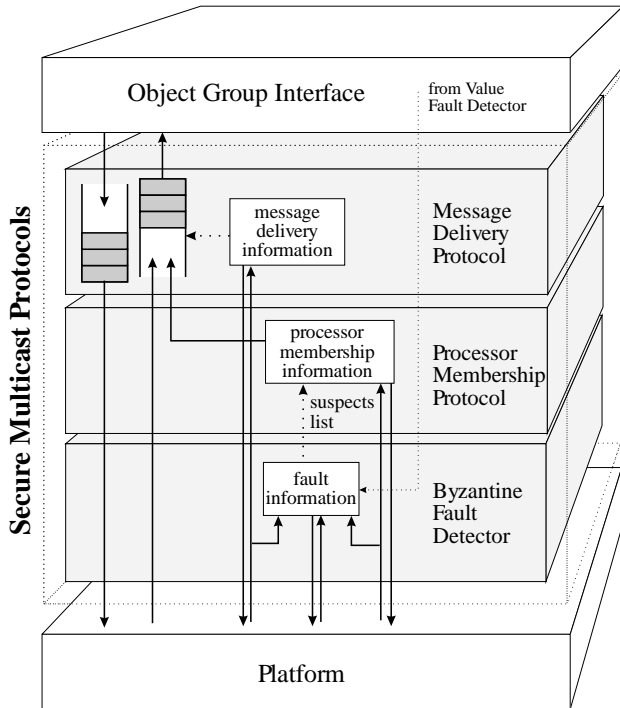


Figure 5: The Secure Multicast Protocols.

To achieve this, each Replication Manager uses a *value fault detector*, as shown in Figure 2. A voter V_I (V_R), on detecting an incorrect value of an invocation (response), sends to the base group, a `Value.Fault.Vote` message encapsulating the set of invocations (responses) on which it voted. This special message is delivered to the value fault detector (within every Replication Manager), which compares this set of invocations (responses) to determine the corrupt client (server) replica and processor.

The value fault detector then communicates the identity of the corrupt processor to the local Byzantine fault detector (within the Secure Multicast Protocols on the same processor) through a `Value.Fault.Suspect` message. This special message is not intended to be transmitted over the network, and is used solely for notifications by the value fault detector to the local Byzantine fault detector. The mechanisms of the Secure Multicast Protocols employ these notifications to decide on, and effect, the removal of the corrupt processor.

7 Secure Multicast Protocols

The Secure Multicast Protocols provide the properties necessary for majority voting, and allow for detection and removal of faulty processors. The protocols consist of a message delivery protocol, a processor membership protocol, and a Byzantine fault detector as shown in Figure 5.

The protocols employ a public key cryptosystem such as RSA [16] in which each processor possesses a private key known only to itself with which it can digitally sign messages. Each processor is able to obtain the public keys of other processors to verify signed messages. The protocols also employ a message digest function such as MD4 [15] in which an arbitrary length input message m is mapped to a fixed length output $d(m)$.

7.1 Message Delivery Protocol

The message delivery protocol delivers two types of messages to the object group interface: regular data messages and `Processor_Membership_Change` messages. A `Processor_Membership_Change` message informs the object group interface of a change in the processor membership, and is delivered in the message sequence along with the regular messages. The processor membership consists of the set of identifiers of all of the processors in the system that are currently operational. For each processor membership of size n , at least $\lceil (2n + 1)/3 \rceil$ processors must be correct. Thus, the number k of faulty processors must satisfy $k \leq \lfloor (n - 1)/3 \rfloor$.

The message delivery protocol provides secure reliable totally ordered delivery of messages multicast by processors in the system. This ensures that all correct processors within a membership deliver the same messages in the same total order and that, if any correct processor delivers a message, then no correct processor delivers a *mutant* message having the same identifier but different contents. The properties of the message delivery protocol are summarized in Table 2.

Imposed on the communication medium is a logical ring with a token that controls the multicasting of messages. To originate a regular message on the ring, a processor must hold the token. The token contains a number of fields that are used for message ordering, requesting retransmission of messages, and fault detection. The fields required to handle various types of faults are summarized in Table 3. Related previous work [6] contains more details on these fields and related mechanisms.

To tolerate faults involving message corruption during transit, the *message_digest_list* field contains digests of the regular messages originated by the token holder. A processor does not deliver any message that does not correspond to a digest in the corresponding token.

To tolerate malicious faults where processors may masquerade as other processors or send mutant messages, additional fields and mechanisms are necessary. To ensure that a malicious processor cannot masquerade as another processor, the token holder digitally signs each token before multicasting it. Additionally, the digest of the token received by the token holder, *i.e.*, the previous token, is contained in the *previous_token_digest* field. This field, along with the *signature* field, allows for the detection of mutant tokens.

Name	Description
Integrity	For any message m , every correct processor p delivers m at most once.
Authentication	For any message m that contains the identifier of a correct processor p , a correct processor q delivers m only if m was originated by p .
Uniqueness	If a correct processor p delivers a message m , then no correct processor q delivers a mutant message m' having the same identifier as m but different contents.
Reliable Delivery	If correct processors p and q are both members of membership M and no membership change occurs, and p originates a message m , then q delivers m . If correct processors p and q both install consecutive memberships M_1 and M_2 , and p originates a message m in membership M_1 , then q delivers m in M_1 .
Total Order	If correct processors p and q both deliver messages m_1 and m_2 , then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2 .

Table 2: Properties of the message delivery protocol.

Fault Type	Token Fields
message loss, processor receive omission, processor crash	$sender_id, ring_id, seq, aru, rtr_list$
message corruption	above fields plus $message_digest_list$
malicious processor	above fields plus $signature, previous_token_digest, rrg_list$

Table 3: Token fields required to cope with various faults.

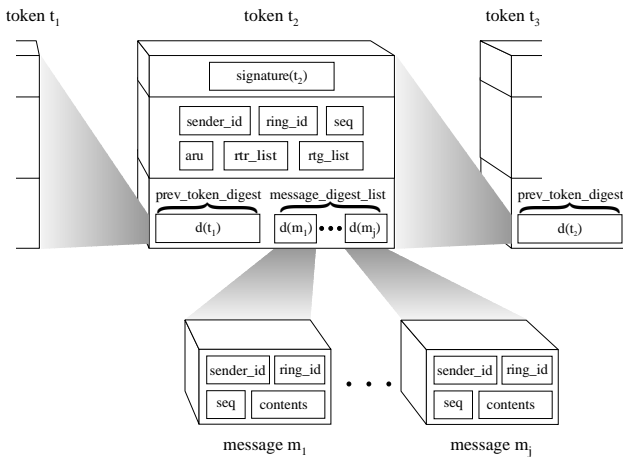


Figure 6: Token and messages sent during one token visit.

Tolerating malicious faults comes with a cost, primarily due to the computation time associated with generating and verifying digital signatures. However, because regular messages are not signed, and because multiple messages are multicast by each token holder, one signature on the token covers multiple messages. The token and messages sent by a processor during one token visit are represented in Figure 6.

7.2 Processor Membership Protocol

The Secure Multicast Protocols include a processor membership protocol that reconfigures the system when one or more processors exhibit faulty behavior. The membership protocol exchanges information via special Membership

messages, and reaches agreement on and installs a new membership consisting of apparently correct processors that are able to communicate with each other. The membership protocol satisfies the properties in Table 4. Termination of the processor membership protocol is ensured by the properties of the Byzantine fault detector. When the fault detector properties hold, all of the correct processors reach agreement and install the new membership.

7.3 Byzantine Fault Detector

The membership protocol depends on a Byzantine fault detector [5] with the properties given in Table 5. The output of the Byzantine fault detector consists of a list of processors that are suspected of being faulty. The particular instances of these faults are:

- Processor fails to send a message that is required by the message delivery or membership protocol
- Processor fails to acknowledge the receipt of a regular message or a token
- Processor sends mutant tokens
- Processor sends a token that is not properly formed.
- Object replica hosted by the processor sends an incorrect value for an invocation or response

The Byzantine fault detector monitors the messages sent by the message delivery and processor membership protocols, and provides its output to the membership protocol in the form of a list that contains the processors currently suspected by the local fault detector module. The fault detector makes use of timeouts for the detection of faults, and also performs the checking of tokens to determine if

Name	Description
Uniqueness	If a correct processor p installs a membership M , then no correct processor q installs a membership M' having the same identifier as M but a different membership.
Self-Inclusion	If a correct processor p installs a membership M , then M includes p .
Total Order	If correct processors p and q both install memberships M_1 and M_2 , then p installs M_2 after M_1 if and only if q installs M_2 after M_1 .
Eventual Exclusion	If p is a correct processor and q is a processor that has exhibited a fault, then there is a time after which p installs a membership that excludes q , and p never subsequently installs a membership that includes q .
Eventual Inclusion	If p and q are correct processors, then there is a time after which p installs a membership that includes q .

Table 4: Properties of the processor membership protocol.

Name	Description
Eventual Strong Byzantine Completeness	There is a time after which every processor that has exhibited a fault is permanently suspected by every correct processor.
Eventual Strong Accuracy	There is a time after which every correct processor is never suspected by any correct processor.

Table 5: Properties of the Byzantine fault detector.

they are of the proper form, as shown in Figure 6. The fault detector also uses information in the token to determine if a malicious processor has sent mutant tokens. In addition, the fault detector makes use of the Value_Fault_Suspect messages from the Replication Manager to identify processors hosting object replicas that have sent an incorrect value of an invocation or response.

8 Implementation and Performance

A prototype version of the Immune system has been implemented. The performance of the prototype is shown in the graph in Figure 7, for a test application developed with the VisiBroker 3.2 ORB from Inprise Corporation. The measurements were taken over a network of six dual-processor 167 MHz UltraSPARC workstations, running the Solaris 2.5.1 operating system and connected by a 100 Mbps Ethernet.

The client object acts as a packet driver, sending a constant stream of one-way invocations at a specified rate to the server object. Each invocation is contained in a fixed-length (64 bytes) IIOP message. The client object's invocation rate is varied to obtain the throughput measurements at the server object. The graph shows the throughputs obtained with this test application for the following cases, corresponding to varying levels of survivability:

- **Case 1:** Unreplicated client and server objects without the Immune system. The throughput is determined by the ORB mechanisms alone.
- **Case 2:** Three-way active replication of both client and server objects without majority voting. Reliable

totally ordered multicasts without either the message digests or the signatures are used. The throughput is dictated by the cost of interception, active replication and multicasting, in addition to the costs of case 1.

- **Case 3:** Three-way active replication of both client and server objects with majority voting. Secure reliable totally ordered multicasts with message digests are used. The throughput is dictated by the cost of message digests, in addition to the costs of case 2.
- **Case 4:** Three-way active replication of both client and server objects with majority voting. Secure reliable totally ordered multicasts with message digests and digitally signed tokens are used. The throughput is dictated by the cost of signatures, in addition to the costs of case 3.

Systems that are designed for a high level of survivability incur a high associated overhead. The greatest cost is that due to signature generation and verification, which are computationally expensive operations that depend on modular exponentiation. The Immune System utilizes CryptoLib [7], a library of routines for public and private key systems. Signatures are computed by RSA decrypting a message digest using the private key, while verification is performed by RSA encrypting the signature using the public key. Because the message digest is a fixed size (16 bytes), the time required for signing is independent of the size of the original message. However, signature generation time is highly related to key modulus size; thus, a tradeoff exists between performance and the level of security attained. The results in Figure 7 were obtained using a key size with a modulus of 300 bits.

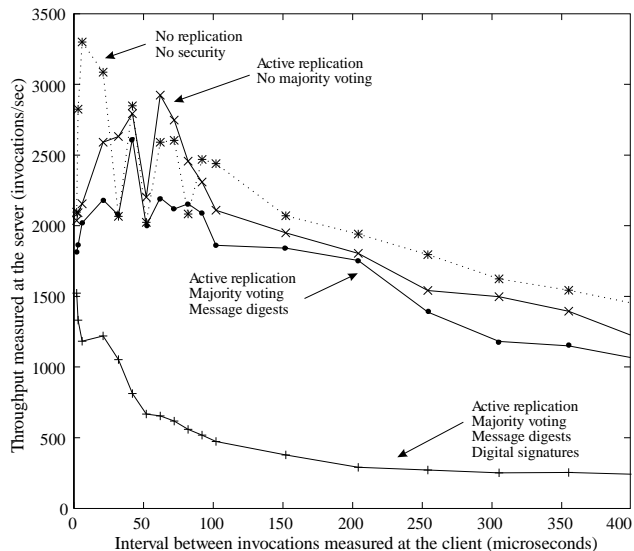


Figure 7: Performance of the Immune system.

The Secure Multicast Protocols have been designed to amortize the cost of computing a signature over the number j of messages m_1, \dots, m_j sent per token visit, as shown in Figure 6. This parameter j can be tuned to achieve optimal performance for different applications. In the performance measurements for cases 2, 3 and 4, up to six multicast messages are sent with each token visit, where each multicast message encapsulates possibly multiple IIOP messages. While the cost of computing a single signature is spread over six messages, the use of signatures is nevertheless computationally expensive, as can be seen from the overheads of the Immune system in case 4. However, the results indicate that the overheads of the Immune system without signatures (cases 2 and 3) are low. In all of the cases, the overheads are quite reasonable given the level of survivability that the Immune system provides.

The graph also indicates some transient behavior, attributable to the ORB, for cases 1, 2 and 3 when the time between consecutive invocations at the client is less than $100 \mu\text{s}$. For such high message generation rates at the client, the ORB batches multiple one-way invocations before transmission. While some performance benefit is gained from this activity of the ORB, the unpredictability of the ORB's batching, evident from the transient behavior in the graph, can lead to undesirable fluctuations in the throughput of the application. This behavior of the ORB is not as significant in case 4, where the computation of the signatures dominates the CPU usage on each processor, effectively reducing the fraction of CPU time allocated to other processing, such as the ORB's batching of IIOP messages.

9 Related Work

The Immune system grew out of the authors' work on two other systems: the Eternal system [9, 11] for providing reliability to CORBA applications and the SecureRing system [6] for ensuring secure reliable totally ordered communication in an environment subject to arbitrary faults. The Immune system goes beyond these systems by aiming to enhance any commercial implementation of CORBA with support for survivability.

Reiter and Birman [13] have presented a method for constructing replicated services that remain correct and available despite the malicious corruption of clients and servers. In their method, a threshold cryptosystem is used such that a client encrypts a request using the public key of the server, and a threshold number of servers can then decrypt the request. Servers then atomically broadcast the request to the other servers. A threshold of the servers is required to sign the response. The disadvantage of this approach is the high overhead associated with the cryptographic operations required for each request and response.

The Rampart system [14] is a toolkit of protocols that facilitate the development of distributed services that remain correct and available despite process corruption. The toolkit includes a group membership protocol, reliable and atomic group multicast protocols, and output voting protocols. In Rampart, a client sends a request to a single server that forwards the request by atomically multicasting it to a group of servers. If the server to which the client sends the request is corrupted, it could refuse to forward the request or could alter the client's request.

The AQuA architecture [1] 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 a corrupted object. The AQuA architecture exploits the facilities of the underlying Ensemble and Maestro toolkits [17], and uses majority voting at the application object level, to detect an incorrect value of an invocation (response) from a replicated client (server). However, in order for majority voting to be effective for survivable applications that must tolerate arbitrary faults, more stringent guarantees, such as those provided by the Secure Multicast Protocols in the Immune system, are required of the underlying multicast protocols.

The FRIENDS [3] 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 that provides support for object 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.

Cactus [4] aims to provide a flexible toolkit for customizing different levels of strategies for survivability through the combination of microprotocols, each implementing a specific property. Fault tolerance is implemented as a microprotocol that supports active and passive replication. Security is provided through microprotocols that implement various authentication and encryption methods. GroupRPC is a middleware service, implemented using Cactus, that allows an application to tolerate crash, omission, timing and Byzantine faults.

Phalanx [8] is a software system that enables wide-area applications to survive the malicious corruption of clients and servers. Phalanx 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 that enable efficient scaling.

10 Conclusion

The Immune system aims to provide survivability for CORBA applications that must continue to operate despite the occurrence of faults and intrusions or accidents that damage the underlying distributed system. Reliability is provided through the replication of CORBA objects, and through mechanisms that ensure replica consistency. The use of secure reliable totally ordered messages to communicate operations between replicated objects facilitates replica consistency. Because the Immune system is designed to support critical applications, both client and server objects are actively replicated, with majority voting applied on all invocations and responses. The measured performance of the Immune system is quite good, given the high level of survivability that it provides.

References

- [1] 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.
- [2] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, and N. R. Mead. Survivable network systems: An emerging discipline. Technical report, CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, Nov. 1997.
- [3] 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.
- [4] 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.
- [5] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 61-75, Chantilly, France, Dec. 1997.
- [6] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Annual Hawaii International Conference on System Sciences*, volume 3, pages 317-326, Kona, Hawaii, Jan. 1998.
- [7] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*, pages 1-17, Santa Clara, CA, Oct. 1993.
- [8] 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.
- [9] 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.
- [10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 81-90, Portland, OR, June 1997.
- [11] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139-150, Sept. 1997.
- [12] Object Management Group. The Common Object Request Broker: Architecture and specification. Revision 2.0, 1995.
- [13] M. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986-1009, May 1994.
- [14] M. K. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems. Lecture Notes in Computer Science 938*, Springer-Verlag, pages 99-110, 1995.
- [15] R. L. Rivest. The MD4 message digest algorithm. In *Advances in Cryptology - Proceedings of CRYPTO '90*, pages 303-11, Santa Barbara, CA, Aug. 1990.
- [16] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, Feb. 1978.
- [17] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software - Practice and Experience*, 28(9):963-79, July 1998.