# End-to-End Latency of a Fault-Tolerant CORBA Infrastructure *

W. Zhao, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
wenbing@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

## Abstract

*This paper presents measured probability density functions (pdfs) for the end-to-end latency of two-way remote method invocations from a CORBA client to a replicated CORBA server in a fault-tolerance infrastructure. The infrastructure uses a multicast group-communication protocol based on a logical token-passing ring imposed on a single local-area network.*

*The measurements show that the peaks of the pdfs for the latency are affected by the presence of duplicate messages for active replication, and by the position of the primary server replica on the ring for semi-active and passive replication. Because a node cannot broadcast a user message until it receives the token, up to two complete token rotations can contribute to the end-to-end latency seen by the client for synchronous remote method invocations, depending on the server processing time and the interval between two consecutive client invocations.*

*For semi-active and passive replication, careful placement of the primary server replica is necessary to alleviate this broadcast delay to achieve the best possible end-to-end latency. The client invocation patterns and the server processing time must be considered together to determine the most favorable position for the primary replica. Assuming that an effective sending-side duplicate suppression mechanism is implemented, active replication can be more advantageous than semi-active and passive replication because all replicas compete for sending and, therefore, the replica at the most favorable position will have the opportunity to send first.*

**Keywords:** End-to-End Latency, Probability Density Function, Fault Tolerance, CORBA, Multicast Group Communication Protocol

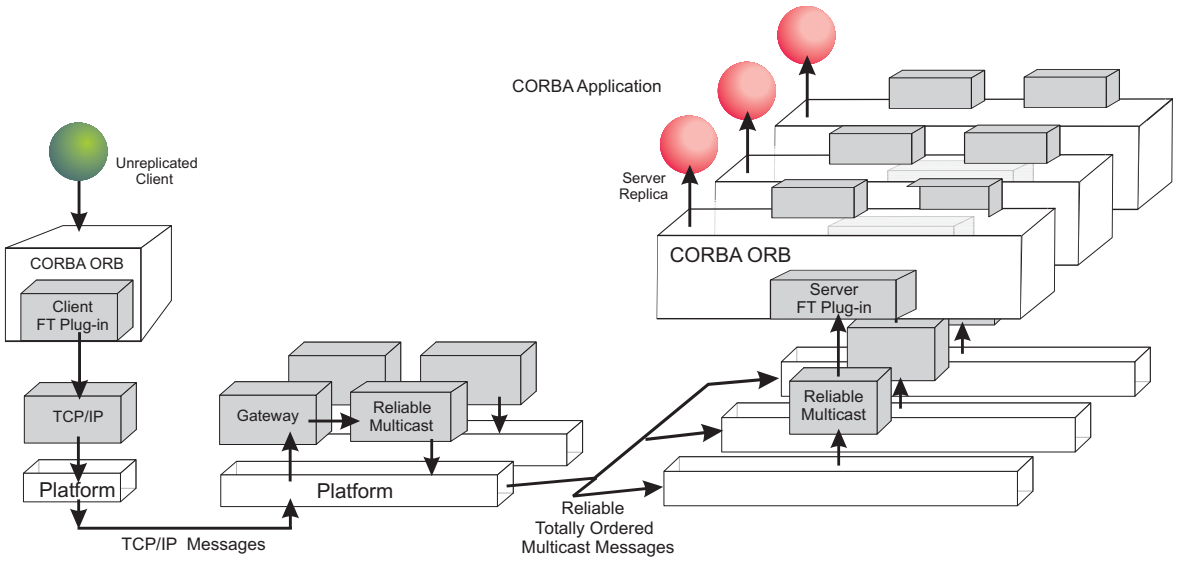
*This research has been supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

## 1 Introduction

The Common Object Request Broker Architecture (CORBA) [14] supports distributed object applications with client objects invoking server objects, which return responses to the client objects after performing the requested operations. CORBA's Object Request Broker (ORB) acts as an intermediary in the communication path between a client object and a server object, shielding them from differences in their programming languages, hardware architectures, operating systems and physical locations.

Several researchers [3, 4, 7, 8, 10, 12, 13] have developed infrastructures that provide CORBA applications with fault tolerance, and several of those systems depend on reliable totally-ordered multicast group-communication protocols to guarantee strong replica consistency. Previous experiments [1, 2] have shown that, over local-area networks, group-communication protocols can achieve throughputs for reliable totally-ordered multicast communication that exceed the throughput that TCP achieves for point-to-point communication. Other researchers [5, 17] have analyzed and measured the latency of group-communication protocols, where a test driver is collocated with each protocol instance. However, to determine whether a fault-tolerant distributed application can meet its real-time requirements, it is more important to characterize the variation of the *end-to-end latency* for remote method invocations in a fault-tolerance infrastructure where a group-communication protocol is used as a component.

In this paper we present measurements of the probability density functions (pdfs) for the end-to-end latency of synchronous remote method invocations from a CORBA client to a replicated CORBA server in a fault-tolerance infrastructure [19]. The infrastructure uses a logical token-ring based protocol [1] to achieve reliable totally-ordered delivery of messages. For a number of common scenarios, we investigate the reasons for the pdfs that we have measured. The measurements show that the peak probability density for the end-to-end latency is affected by the presence of duplicate messages for active replication, and by the position


CORBA Application
Server
Replica
Unreplicated
Client
CORBA ORB
Client
FT Plug-in
CORBA ORB
Server
FT Plug-in
TCP/IP
Gateway
Reliable
Multicast
Reliable
Multicast
Platform
Platform
TCP/IP Messages
Reliable
Totally Ordered
Multicast Messages

detected, Totem creates a fault report and delivers it to the Fault Notifier. Totem also conveys, to the Fault Notifier, the fault reports that the object-level fault detector generates.

Unreplicated clients connect to the replicated servers through passively replicated gateways that provide access to the replicated servers. The client-side failover mechanisms specified by the FT CORBA standard are also implemented as a protocol plug-in.

### 1.3 The Totem Protocols

Totem [9] is a suite of group-communication protocols that provide reliable totally-ordered multicasting of messages to processors operating in a single local-area network (LAN), or in multiple LANs interconnected by gateways. In this study, we employ only the Totem single-ring protocol [1] that works for a single LAN. Totem provides a process group interface that allows applications to be structured into process groups. The Totem process group layer maintains information about the current memberships of the process groups that it supports.

The Totem single-ring protocol provides reliable totally-ordered delivery of messages using a logical token-passing ring superimposed on a local-area network, such as an Ethernet. The token circulates around the ring as a point-to-point message, with a token retransmission mechanism to guard against token loss. Only the processor holding the token can broadcast a message. The token conveys information for total ordering of messages, detection of faults and flow control.

The sequence number field in the token provides a single sequence of message sequence numbers for all messages broadcast on the ring and, thus, a total order on messages. When a processor broadcasts a new message, it increments the sequence number field of the token and gives the message that sequence number. Other processors recognize missing messages by detecting gaps in the sequence of message sequence numbers, and request retransmissions by inserting the sequence numbers of the missing messages into the retransmission request list of the token. If a processor has received a message and all of its predecessors, as indicated by the message sequence numbers, it can deliver the message.

The all-received-up-to field of the token enables a processor to determine, after a complete token rotation, a sequence number such that all processors on the ring have received all messages with lower sequence numbers. A processor can deliver a message as "stable" if the sequence number of the message is less than or equal to this sequence number. When a processor delivers a message as stable, it can reclaim the buffer space used by the message because it will never need to retransmit the message subsequently.

The token also provides information about the aggregate message backlog of the processors on the ring, allowing a fair allocation of bandwidth to the processors. The flow control mechanism provides protection against fluctuations in the processor loads, but is vulnerable to competition for the input buffers from unanticipated traffic.

## 2 Measurements of the Latency

### 2.1 Experimental Setup

We have conducted measurements of the latency for the fault-tolerant infrastructure using a system of four Pentium III PCs, each with 1GHz CPU, 256MBytes of RAM, running the Mandrake Linux 7.2 operating system, over a 100Mbit/sec Ethernet, using Vertel's e*ORB [18]. During the measurements, there was no other traffic on the network.

Four copies of Totem run on the four PCs, one for each PC. We refer to these PCs as $\text{node}_0$, $\text{node}_1$, $\text{node}_2$ and $\text{node}_3$, in the order of the logical token-passing ring imposed by Totem, where $\text{node}_0$ is the ring leader. Because one and only one instance of Totem runs on each node, we use the same node designation to refer to the Totem instance on a particular node.

In the experiments, a CORBA client sends a sequence of 256 longs to a replicated server, and the replicated server echos back the same sequence of longs. In effect, there is about 1KByte payload being transmitted from the client to the server, and back to the client. A message, which comprises the GIOP protocol header, the FT infrastructure protocol header, the Totem protocol header and the message payload is contained in one Totem packet (1.4KByte).

The client runs on the same node as the ring leader, $\text{node}_0$. The server is three-way replicated, where one replica runs on each of the other three nodes, $\text{node}_1$, $\text{node}_2$ and $\text{node}_3$. For each run, the client issues 10,000 remote method invocations on the server.

### 2.2 Measurement Methodology

For all of the measurements, the current time is obtained by calling `gettimeofday()`. At the client, the current time is recorded immediately before issuing a synchronous remote method invocation and after the invocation returns. The difference between the two measured times constitutes the end-to-end latency for each remote method invocation. Unless stated otherwise, the client issues a new remote method invocation immediately after it completes one remote method invocation. To simulate the random "think time" for the client (*i.e.*, the interval between consecutive remote method invocations), the `nanosleep()` function is invoked between successive remote method invocations. On Linux, the actual sleep time is rounded to a multiple

of two clock ticks, with random variations that are significantly larger than the complete token rotation time.

To facilitate understanding of the end-to-end latency profile, and also of the performance of Totem for supporting synchronous remote method invocations, the Totem code is instrumented to collect the following data:

- **Complete token rotation time.** As soon as a Totem instance receives the token, it obtains the current time. The difference between the current time and the time when the token was last seen is the complete token rotation time as seen by this Totem instance.
- **Message-Send delay.** The send delay of a message is measured as the difference between the time the message is written to the Totem send buffer, and the time immediately before the message is sent when the Totem instance receives the token. The send delay increases the end-to-end latency, and might cause unpredictability of the end-to-end latency within the range of one (two) complete token rotation times for one-way (two-way) remote method invocations.
- **Application-processing time.** The application-processing time is measured as the difference between the time a Totem instance delivers a message to the application, and the time it receives a message from the application. The processing time includes the cost of the fault-tolerance mechanisms that reside in the process space of the CORBA application. This measurement provides an estimate of the processing overhead imposed by the fault-tolerance mechanisms.

In all cases, the raw data are stored in a buffer and written to a set of files when Totem and the CORBA client exit. The raw data files are processed off line to produce the pdfs. The resolution for the pdf calculation is set to $1\mu$sec. The cost of each clock-reading is about $0.5\mu$sec on average, which causes insignificant overhead in the latency measurements.

## 2.3 Results

We present here the experimental results of the pdfs for the end-to-end latency, the complete token rotation time, the message-send delay and the application-processing time, for the following five scenarios:

(a) Semi-active replication, with $\text{node}_1$ running the primary server replica

(b) Semi-active replication, with $\text{node}_2$ running the primary server replica

(c) Semi-active replication, with $\text{node}_3$ running the primary server replica

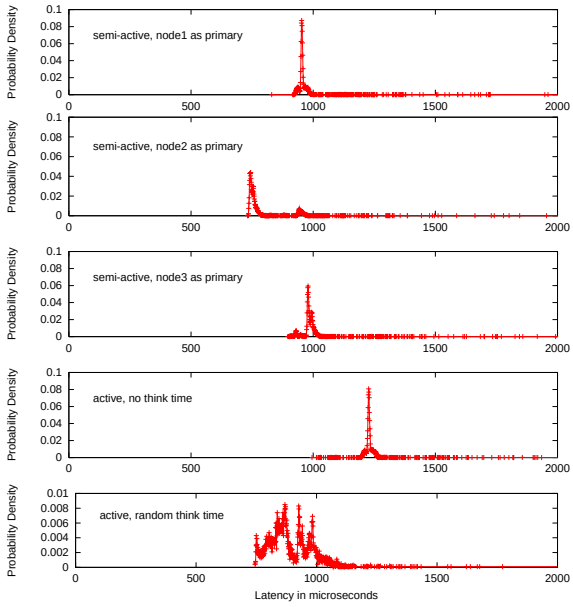



**Figure 2. The measured pdfs for the end-to-end latency as seen by the client corresponding to the five scenarios.**

(d) Active replication, with no "think time" at the client

(e) Active replication, with a random "think time" at the client.

**End-to-End Latency.** The results of the end-to-end latency measurements, for the above scenarios, are shown in Figure 2, as probability density functions (pdfs). For semi-active replication, the highest probability density is reached at a latency of $953\mu$sec, $741\mu$sec and $979\mu$sec, for running the primary on $\text{node}_1$, $\text{node}_2$ and $\text{node}_3$, respectively. Running the primary server replica in the middle of the logical ring ($\text{node}_2$) gives the lowest latency. Note that, in all three scenarios for semi-active replication, the peaks of the pdfs are quite narrow, indicating a highly predictable behavior, which is desirable for real-time applications. For active replication, with no "think time," the peak latency is significantly larger than that for semi-active replication, at $1227\mu$sec. For active replication, with a randomly chosen "think time," the peak latency has a spread in the range of about $250\mu$sec, starting from about $740\mu$sec.

**Complete-Token-Rotation Time.** The measured pdfs for the complete-token-rotation time, as seen by each Totem instance for the different scenarios, are shown in Figures 3 and 4. The peak near $205\mu$sec corresponds to the complete token rotation time when there is no interference between

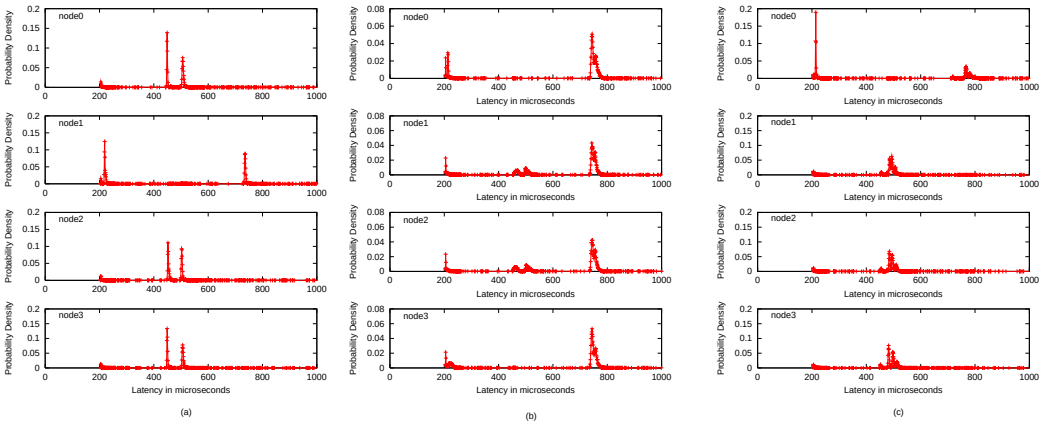
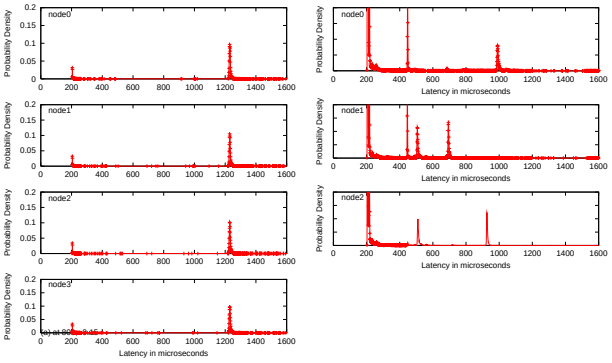



**Figure 3. The measured pdfs for the complete token rotation time as seen by $node_0$ to $node_3$ (from top to bottom) for semi-active replication, with the primary running at (a) $node_1$, (b) $node_2$, and (c) $node_3$.**

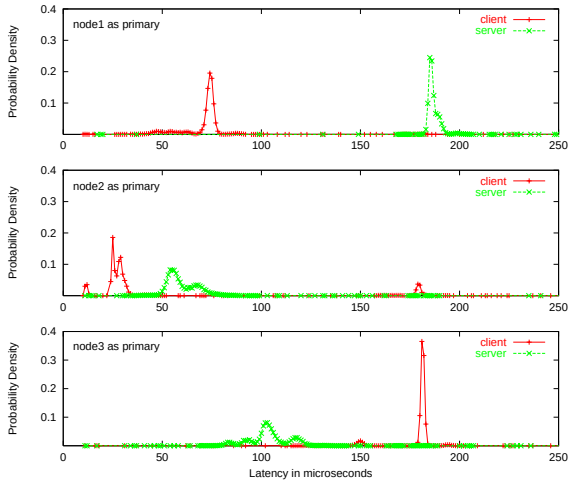
the processing of the token and the application messages sent over Totem (which has been confirmed by running four Totem instances on the network without running any application over them). This scenario corresponds to a token-passing time of about 51$\mu$sec. Other peaks with larger latencies are the results of sending reliable totally-ordered application messages. For a given run, not all Totem instances have the same view of the complete token rotation time.

Each run contains 10,000 synchronous remote method invocations and, thus, there are 20,000 non-duplicate messages (10,000 requests and 10,000 replies) that are broadcast over the Totem logical ring. For semi-active replication, where $node_1$ and $node_3$ run the primary server replica, the observed total number of token rotations during each run is about 21,000. For semi-active replication, where $node_2$ (which is two hops away from the client) runs the primary server replica, the total number of token rotations during each run is about 12,500. For active replication with no "think time," the total number of token rotations during each run is about 11,000.




**Figure**

**Message-Send Delay.** A Totem instance can broadcast a user message only when it receives the token. Thus, when Totem is used in the fault-tolerance infrastructure, with the fault-tolerance mechanisms in the FT-plugin, there can be a significant delay after a message is written to the Totem send buffer and before Totem broadcasts the message. Figure 5 shows the measured delays at the Totem instances that support the client, or the primary server replica, for semi-active replication. To avoid repetition, we do not show the plots for the send delay for active replication. In general, the send delay for active replication is longer because it takes the token longer to circulate around the logical ring, due to the presence of duplicate messages.



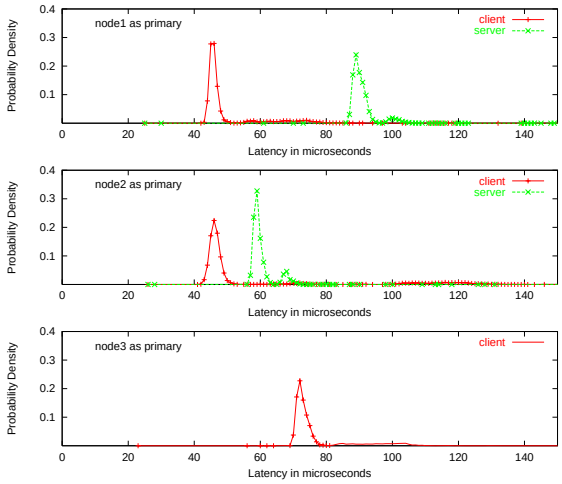




**Figure 5. The measured pdfs for the send delay for the client and the primary server replica in semi-active replication, with the primary running at (a) $node_1$, (b) $node_2$, and (c) $node_3$.**

The processing can be affected by Totem's processing of an incoming token immediately after it delivered a message to the application. The application processing can be delayed by 25$\mu$sec to 30$\mu$sec, as a result of the competition for the single CPU, with the server process being affected slightly more than the client process.

## 2.4 Discussion

The results of the pdfs for the end-to-end latency for the complete token rotation time, the message-send delay and the application-processing time are consistent with each other, and reveal a clear picture of message transmission and delivery for remote method invocations over a fault-tolerance infrastructure using a logical token-passing ring to provide reliable totally-ordered message delivery.

Figure 7 show the steps of a remote method invocation over the fault-tolerance infrastructure when the server is semi-actively replicated with $node_1$ running the primary server replica. In the diagram, starting at the top, the four nodes ($node_0$, $node_1$, $node_2$, $node_3$) are distributed over the logical ring in counter-clockwise order. $Node_0$ is the ring leader and runs the client; the other three nodes run the server replicas. The steps of operation are as follows:

(a) The client on $node_0$ issues a request to the server, but the Totem instance on $node_0$ does not hold the token. Consequently, the request must remain queued in the Totem send buffer at $node_0$. Meanwhile, the token is being passed from $node_2$ to $node_3$.

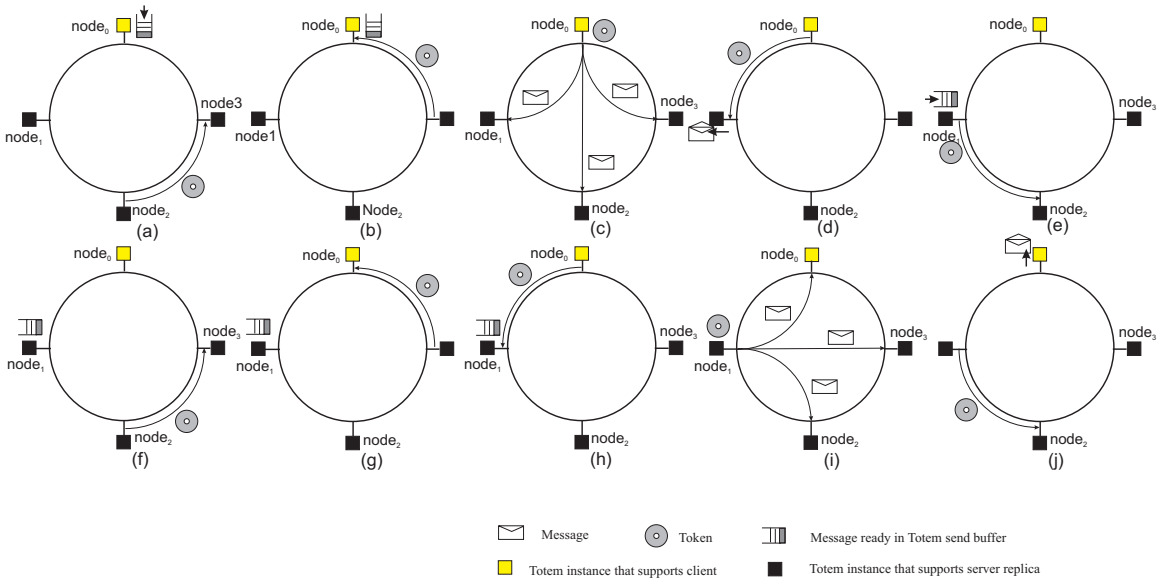



**Figure 7. Steps of a remote method invocation over the fault-tolerance infrastructure when the server is semi-actively replicated, with node$_1$ running the primary server replica.**

(i) The Totem instance on node$_1$ has received the token and broadcasts the reply message. The reply message has been delayed at the server for almost a complete token rotation time (180$\mu$s), as shown in Figure 5(a).

(j) The client at node$_0$ processes the reply and starts to generate the next request. Meanwhile, the Totem instance on node$_1$ passes the token to node$_2$.

These explanations are consistent with the pdfs for the complete token rotation time, as given in Figure 3(a). As seen by node$_0$, node$_2$ and node$_3$, most (20,000) of the token rotations involve the sending and receiving of request and reply messages. Thus, there are two peaks, with latencies of approximately 450$\mu$sec and 500$\mu$sec, which are significantly longer than the peak latency of approximately 205$\mu$s when no message is transmitted.

However, node$_1$, which runs the primary server replica, sees a different view. For node$_1$, 10,000 token rotations carry both the request message and the reply message (Steps (i), (j), (a), (b), (c) and (d) above). For each of those 10,000 token rotations, node$_1$ starts by broadcasting a reply message (Step (i)). Then, the client running on node$_0$ has the opportunity to broadcast a request message, because it has enough time to produce one (Step (c)). Given that there are two user messages sent in each such token rotation, we predict that the latency for the complete token rotation is the sum of four token-passing times (205$\mu$sec) plus the sum of the latencies for the two user messages (245$\mu$sec and 295$\mu$sec), which equals 745$\mu$sec. The actual peak position (740$\mu$sec), shown in Figure 3(a), is in line with the predicted value.

For the other 10,000 token rotations measured at node$_1$ (Steps (e), (f), (g) and (h) above), there is no user message sent because, in Step (e), node$_1$ has no user message to send before it passes the token to node$_2$. Node$_1$ has the opportunity to send the reply message only when the token returns to it. During this token circulation, node$_0$ cannot send because the client running on it is still waiting for a reply. Thus, there are 10,000 token rotations that carry no user message (as seen by node$_1$). These token rotations require about 205$\mu$s, corresponding to the measured peak, shown in Figure 3(a).

The pdf profiles for the other primary replica positions for semi-active replication, and for active replication with no "think time," can be explained in a manner similar to that for node$_1$ as the primary server replica, with some differences. In particular, when node$_2$ runs the primary server replica, node$_2$ is able to generate and queue the reply message immediately before the token arrives from node$_1$, thus allowing the reply to be transmitted without a lengthy delay. Similarly, node$_0$ is able to generate and queue the next request message immediately before the token arrives from node$_3$. Consequently, we see the best end-to-end latency when node$_2$ is the primary (Scenario (b)), as shown in Figure 2(b).

The time required to process a message depends on the type of message (request or reply), on whether it is a duplicate of a message already sent by another node, and on the

type of application (client or server) that runs on the node. The cost for the send operation (sending a user message and sending the token) is similar in the different scenarios. The latency differences for the different scenarios are due primarily to the operation at the node immediately after the sending node on the ring. The latency costs for the different cases are summarized below:

**Case 1.** Request message at a client, reply message at a server. Low cost, because the message is not delivered to the application.

**Case 2.** Duplicate request message at a server, duplicate reply message at a client. Intermediate cost, because the message is discarded immediately as soon as it is determined to be a duplicate.

**Case 3.** Request message at a server, reply message at a client. High cost, because the message is delivered to the application process, and the reception of the token that follows the message must wait until the application message has been delivered. The application processing of the request message further inhibits the processing of the token.

In Scenario (a), where $\text{node}_1$ runs the primary replica, Cases 1 and 3 both apply at $\text{node}_2$ and $\text{node}_3$. Because Case 1 costs less time than Case 3, $\text{node}_0$ sees a different token rotation time for rotations that contain a request message (Case 3) from rotations that contain a reply message (Case 1). Cases 1 and 3 also apply to Scenario (b), where $\text{node}_2$ runs the primary replica. Because $\text{node}_0$ ($\text{node}_2$) can send a request (reply) message the first time the token visits it after it has received the reply (request) message, the end-to-end latency for Scenario (b) is approximately one token rotation time (205$\mu$sec) less than that for Scenario (a). Cases 1 and 3 apply to Scenario (c), where $\text{node}_3$ runs the primary server replica. Because the processing of a request message in Case 3 is slightly more expensive than the processing of a reply message, the token is delayed slightly more by such processing in Scenario (c) and, thus, the end-to-end latency is slightly more than that for Scenario (a).

Cases 1, 2 and 3 apply to Scenario (d) with high probability in each token circulation, where the server is actively replicated. In accordance with our previous explanation for semi-active replication, we conclude that $\text{node}_2$ always broadcasts the reply message to the client ahead of the other two replicas with high probability. When the reply message broadcast by $\text{node}_3$ arrives at $\text{node}_0$ (the client), it is most likely to be a duplicate. Because Cases 1, 2 and 3 all show up in a single token circulation, the observed peak of the pdf for a complete token rotation time occurs at about 1,230$\mu$sec.

For Scenario (e), there are several possibilities for each token circulation, as seen by the ring leader, $\text{node}_0$. For some token circulations, only Case 1 applies, *i.e.*, $\text{node}_1$ broadcasts the reply message the second time it receives the token after it receives the request, because the client is "thinking" (again the "think time" is about two clock ticks, which is much longer than a complete token rotation time). For some other token circulations, Case 1, 2 and 3 all apply, *i.e.*, $\text{node}_0$ broadcasts a request message (Case 3), then $\text{node}_2$ broadcasts the reply for the request message (Case 1) and, finally, $\text{node}_3$ broadcasts the (duplicate) reply for the same request message (Case 2). The pdfs viewed by the other nodes can be explained similarly.

### 2.5 Summary

There are a number of factors that affect the end-to-end latency of synchronous remote method invocations. In addition to the well-known factors such as the quality of the networks, the network and CPU load, the message length, the application (including the ORB) processing time, etc., we have discovered that the position of the primary server replica and the client-invocation patterns also play important roles in a fault-tolerance infrastructure that uses a token-based, group-communication system. In such an infrastructure, assuming that other factors are constant, pre-knowledge of the application processing time and the client invocation pattern help in choosing the most favorable node to run the primary replica.

In our study, we have used a high-quality embedded CORBA ORB that incurs very low overhead for remote method invocations, and a test application that involves essentially no processing at either the server or the client. Consequently, it is possible for a replica to generate a reply after receiving a request, or to generate the next request after receiving a reply, within 100$\mu$sec, which is slightly shorter than the time for two token-passings. Our measurements show that when the client adopts a zero-think-time invocation pattern, $\text{node}_2$ is the most favorable position to run the primary because $\text{node}_2$ has the same distance of two token steps between the client and the primary for the request and reply messages. Thus, for a client that follows the zero-think-time invocation pattern, the end-to-end latency is a step function of the primary position and the server-processing time. The size of the step is determined by the complete token rotation time. This means that increasing the server-processing time might not lead to an increase in the end-to-end latency. For example, in the case that the most favorable primary position is $\text{node}_1$, when the server-processing time is increased by less than one token-passing time, we can move the primary to $\text{node}_2$ to keep the end-to-end latency the same as before. Increasing the server-processing time further will lead to a jump of one full token rotation time.

If the client follows an invocation pattern with the "think time" significantly larger than a complete token-rotation time, the task of choosing the most favorable position for the primary actually becomes simpler. In this case, the primary should run on a node that is closest to the client (measured from the primary to the client in the direction that the token circulates) such that the primary can generate and queue the reply before the token arrives.

In active replication, the server replicas on the logical ring compete for the sending of replies to the client. The first non-duplicate reply is sent by the replica that is located at the most favorable position on the ring. Consequently, active replication is a better solution than semi-active or passive replication if duplicate replies can be suppressed effectively at the less favorable server replicas.

Note that duplicate replies are most likely to be broadcast by Totem after the replicas have received the first reply. This manifests a negative effect for the otherwise good and common design of our fault-tolerance infrastructure. In our current implementation, we follow a layered approach. Totem is used as a group-communication service by the upper layers of the fault-tolerance infrastructure. When the replication mechanisms have a message to send, the message is handed to the Totem layer, Totem then packetizes the message and stores the message in its send buffer in packetized form. Totem does not interpret the message semantics and does not have enough knowledge to determine whether or not a packet is a duplicate. For highly deterministic server processing, as is the case of e*ORB, the sending-side duplicate detection mechanism can hardly capture any duplicates because all of the replicas receive the request simultaneously and send the corresponding reply simultaneously (to Totem's send buffer). At the time a replica is ready to send a reply, the replication mechanisms at the replica are unlikely to have heard from any of the other replicas. This is particularly true if the primary at the most favorable position has to wait to send a message. Consequently, by the time the replication mechanisms at the replicas at the less favorable positions have received the message sent by the replica located at the most favorable position, most likely they have handed the duplicate message to Totem. Thus, virtually all of the duplicate messages are sent by Totem eventually.

Indeed, a more tightly coupled design of the replication mechanisms with Totem can provide more effective sending-side duplicate suppression. User messages to be sent by Totem are logged at the replication mechanisms, rather than directly written to Totem's send buffer, until Totem notifies the replication mechanisms that it has received the token. This scheme gives the replication mechanisms a chance to capture and eliminate the sending of duplicate messages to the Totem layer.

## 3 Related Work

Several researchers have developed object replication and fault tolerance infrastructures for CORBA. Some of these infrastructures [3, 4, 7, 10] were developed before the adoption of the Fault Tolerant CORBA standard [15]. Others [8, 12, 13, 19] were developed after that standard was adopted and implement that standard partially or completely. None of those papers present analysis or measurements of the pdfs for the end-to-end latency of a Fault Tolerant CORBA system, as we have presented here.

Substantial work, both analytic and experimental, has been undertaken on the performance of the Totem protocols. Significant differences between these results can be attributed to the experimental setup and methodology. Budhia [2] measured performance with a test message driver located in the Totem protocol stack, so that the test messages never waited for the arrival of the token, while Thomopoulos [17] assumed that, at the moment that a message is generated, the token is randomly located on the ring. Neither of these assumptions is substantiated by our work. Karl *et. al.* [5] investigated the effects of faults and of scheduling delays on the latency of the Totem protocol, and showed that both can induce considerable variation in the latency. We have not, in this work, investigated the effects of either. Narasimhan [10, 11, 12] has measured the performance of the Totem protocol and of the fault-tolerance mechanisms mounted on the Totem protocol. Her measurements focused primarily on throughput rather than on latency.

## 4 Conclusion

We have measured the probability density functions for the end-to-end latency for synchronous remote method invocations from a CORBA client to a replicated CORBA server in a fault-tolerance infrastructure. The infrastructure is based on the pluggable protocols framework and a group-communication protocol that uses a logical token-passing ring imposed on an Ethernet. As the measurements show, the peaks of the pdfs for the latency are affected by the presence of duplicate messages for active replication, and by the position of the primary server replica on the ring for semi-active and passive replication.

The token-based multicast protocol has the characteristic that a message might have to wait up to a complete token rotation time after it is written into the send buffer and before it is actually sent when the token arrives. This characteristic implies, and our measurements show, that as much as two complete token rotation times can contribute to the end-to-end latency as seen by the client for synchronous remote method invocations. For semi-active and passive replication, careful placement of the primary server replica is necessary to alleviate this broadcast delay to achieve the

best possible end-to-end latency. The client-invocation patterns and the server-processing time should be considered together to determine the most favorable position for the primary. Assuming an effective sending-side duplicate suppression mechanism is implemented, active replication can be more advantageous than semi-active and passive replication in that all replicas compete for sending, and therefore, the replica at the most favorable position always has the opportunity to send first.

## Acknowledgment

The authors wish to thank Dr. Marion Ceruti for her valuable comments on the measurement results presented in this paper.

## References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems* 13, 4 (November 1995), pp. 311-342.

[2] R. K. Budhia, L. E. Moser and P. M. Melliar-Smith, "Performance engineering of the Totem group communication system," *Distributed Systems Engineering*, vol. 5, no. 2 (June 1998), pp. 78-87.

[3] 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," *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, West Lafayette, IN (October 1998), pp. 245-253.

[4] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA object group service," *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 93-105.

[5] H. Karl, M. Werner and L. Kuttner, "Experimental investigation of message latencies in the Totem protocol in the presence of faults," *IEE Proceedings–Software*, vol. 145, no. 6 (December 1998), pp. 219-227.

[6] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman and J. Parsons, "The design and performance of a Pluggable Protocols Framework for Object Request Broker middleware," *Proceedings of the IFIP Sixth International Workshop on Protocols for High-Speed Networks*, Salem, MA (August 1999), pp. 81-98.

[7] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1 (1997), pp. 31-43.

[8] C. Marchetti, M. Mecella, A. Virgillito and R. Baldoni, "An interoperable replication logic for CORBA systems," *Proceedings of the International Symposium on Distributed Objects and Applications*, Antwerp, Belgium (September 2000), pp. 7-16.

[9] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.

[10] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 81-92.

[11] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Message packing as a performance enhancement strategy with application to the Totem protocols," *Proceedings of IEEE Global Telecommunications Conference GLOBECOM'96*, vol. 1, London, UK (November 1996), pp. 649-653.

[12] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Strong replica consistency for fault-tolerant CORBA applications," *Proceedings of the IEEE 6th Workshop on Object-Oriented Real-Time Dependable Systems*, Rome, Italy (January 2001), pp. 10-17.

[13] B. Natarajan, A. Gokhale, S. Yajnik and D. C. Schmidt, "DOORS: Towards high-performance fault-tolerant CORBA," *Proceedings of the International Symposium on Distributed Objects and Applications*, Antwerp, Belgium (September 2000), pp. 39-48.

[14] Object Management Group, The Common Object Request Broker: Architecture and Specification (2.4 edition). OMG Technical Committee Document (formal/2001-02-33) (February 2001).

[15] Object Management Group. Fault Tolerant CORBA (final adopted specification). OMG Technical Committee Document (ptc/2000-04-04) (April 2000).

[16] D. Powell, "Delta-4: A Generic Architecture for Dependable Distributed Computing," Springer-Verlag (1991).

[17] E. Thomopoulos, L. E. Moser and P. M. Melliar-Smith, "Analyzing and measuring the latency of the Totem multicast protocols," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 31, no. 1-2 (1999), pp. 59-78.

[18] Vertel Corporation, e*ORB C++ User Guide (December 2000).

[19] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Design and implementation of a pluggable Fault Tolerant CORBA infrastructure," To appear in *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL (April 2002).