End-To-End Latency of a Fault-Tolerant CORBA Infrastructure

Wenbing Zhao
Cleveland State University, w.zhao1@csuohio.edu

Louise E. Moser
University of California-Santa Barbara, moser@ece.ucsb.edu

P. Michale Melliar-Smith
University of California - Santa Barbara, pmms@ece.ucsb.edu

Follow this and additional works at: http://engagedscholarship.csuohio.edu/enece_facpub

Part of the Digital Communications and Networking Commons, and the Electrical and Computer Engineering Commons

How does access to this work benefit you? Let us know!

Publisher's Statement

NOTICE: this is the author's version of a work that was accepted for publication in Performance Evaluation. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Performance Evaluation, 63, 4-5, (05-01-2006); 10.1016/j.peva.2005.03.002

Original Citation


Repository Citation

Zhao, Wenbing; Moser, Louise E.; and Melliar-Smith, P. Michale, "End-To-End Latency of a Fault-Tolerant CORBA Infrastructure" (2006). Electrical Engineering & Computer Science Faculty Publications. 65.
http://engagedscholarship.csuohio.edu/enece_facpub/65

This Article is brought to you for free and open access by the Electrical Engineering & Computer Science Department at EngagedScholarship@CSU. It has been accepted for inclusion in Electrical Engineering & Computer Science Faculty Publications by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.
End-to-end latency of a fault-tolerant CORBA infrastructure

W. Zhao*, L.E. Moser, P.M. Melliar-Smith

Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA
1. Introduction

The demand for dependable distributed computing has led to the design and implementation of many fault-tolerant infrastructures [5,8,9,11–14,18,22] that work with distributed object standards such as CORBA, DCOM, and EJB/J2EE. A fault-tolerant infrastructure often uses a group communication protocol [1–3,20] for reliable totally ordered message delivery. Such a protocol significantly simplifies the replication, logging, and recovery mechanisms that are needed to achieve strong replica consistency. While there has been extensive research on the performance of group communication protocols, there has been little investigation of the performance of fault-tolerant infrastructures that use group communication protocols.

There is little doubt that understanding how a fault-tolerant infrastructure works under fault scenarios is important, but fault scenarios are unpredictable and unrepeatable and, thus, it is difficult to characterize the performance of the infrastructure under such scenarios. The study of the fault-free performance of a fault-tolerant infrastructure is important for several reasons. First, it provides insight into the practicality of the design. Secondly, it helps optimize the infrastructure by identifying bottlenecks in the design and implementation. Thirdly, it provides guidance on how to deploy the infrastructure to achieve the best performance.

In this paper we present an investigation, by analyses and by measurements, of a fault-tolerant CORBA infrastructure [25] that uses the Totem group communication system [1], which is based on a logical token-passing ring. In particular, we investigate the fault-free end-to-end latency, as seen by a client that invokes remote methods of a replicated server synchronously, i.e., we investigate the time it takes the client to issue an invocation on the replicated server and to receive a reply. To obtain an accurate assessment of how the infrastructure operates with respect to different asynchronous factors (e.g., the time it takes to transmit a message from one node to other nodes, the time it takes for the operating system to schedule the execution of an instruction, etc.), we measure the probability density functions (pdfs) of the end-to-end latency of remote method invocations, the pdfs of the token rotation time, the pdfs of the message-send delay, and the pdfs of the application-processing time. The mean latency is a common metric, but it abstracts away the true behavior of the system and obscures the principal factors that determine the end-to-end latency.

Our analyses and measurements show that, because of the potential message-send delay imposed by the Totem group communication system, care must be taken in selecting the node to run the primary server replica for passive and semi-active replication. The position of the primary server replica on the ring, and the server processing time, must be considered together when making this choice. If an effective sending-side duplicate detection mechanism is implemented in the fault-tolerant infrastructure, active replication is more advantageous than both passive and semi-active replication, because it benefits from the automatic selection of the most favorable server replica for sending replies and, thus, it minimizes the total message-send delay. Our experimental investigations show that, in addition to communication-related run-time overhead, the fault-tolerant infrastructure can incur significant computation-related run-time overhead, if the replicated server engages in intensive computation.

This paper is a comprehensive report of our study. In addition to earlier results presented in conference papers [23,24], we include a more general analysis model of the system, more experimental results, and extensive discussion of many of the fine points relating to the end-to-end latency of the fault-tolerant infrastructure.
2. Background

First we present background material regarding the replication styles, the Totem protocol, and the fault-tolerant infrastructure that is used in this study.

2.1. Replication styles

In a fault-tolerant distributed application, critical components are replicated to mask faults. To maintain strong replica consistency, and to provide a single-copy image to the rest of the system, such components are run on top of a fault-tolerant infrastructure. Group communication systems [1–3,20] are often used in fault-tolerant infrastructures to provide reliable totally ordered multicast message delivery and group membership services.

Active replication and passive replication are the two most common replication styles. In active replication, all replicas perform exactly the same operations in the same order, and the reply or request message that arrives at the destination first is delivered to the application process. In passive replication, only one replica (the primary replica) executes in response to the client’s requests.

Semi-active replication is a hybrid replication style introduced in the Delta-4 architecture [17] that exploits the benefits of both active and passive replication. In semi-active replication, all replicas execute the same operations in the same order, but only one replica (the primary replica) issues responses and nested invocations. Unlike active replication duplicate messages are not sent under fault-free conditions, and recovery is generally faster than for passive replication.

2.2. The Totem system

Totem [10] is a group communication system that provides reliable totally ordered delivery of multicast messages to processors operating in a single local-area network (LAN), or in multiple LANs interconnected by gateways. In this investigation, we employ the Totem single-ring protocol [1] that operates over a single LAN. Totem provides a process group interface that allows applications to be structured into process groups and that maintains information about the current memberships of the groups. There are many group communication systems besides Totem, such as [2,3,20].

The Totem single-ring protocol uses 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 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 delivers 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 the sequence number in the all-received-up-to field of the token. 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 processor loads, but is vulnerable to competition for the input buffers from unanticipated traffic.

2.3. Pluggable FT CORBA infrastructure

The Pluggable Fault-Tolerant (FT) CORBA infrastructure [25] used in this study employs a novel non-intrusive approach to render CORBA applications fault-tolerant. The fault-tolerant mechanisms are plugged into the CORBA ORB by exploiting the pluggable protocols framework that most modern ORBs provide [7]. On the server-side, fault tolerance is provided by an FT protocol plug-in, the Totem protocol, the Replication Manager, the Fault Notifier and the Fault Detectors. The Replication Manager and the Fault Notifier are implemented as CORBA objects running as separate processes, and are themselves replicated. The Fault Detector that detects faults at the object level is implemented as a component of the FT protocol plug-in.

In addition to providing the reliable totally ordered message delivery service, Totem also serves as a process-level and host-level fault detector. When it detects a fault, Totem creates a fault report and delivers it to the Fault Notifier. Totem also conveys, to the Fault Notifier, fault reports generated by object-level fault detectors.

3. Related work

Several researchers have developed object replication and fault-tolerant infrastructures for CORBA [16]. Some of those infrastructures [5,8,11,18] were developed before the adoption of the Fault-Tolerant CORBA standard [15]. Others [9,13,14,25] were developed after the standard was adopted, and implement the standard partially or completely. None of the aforementioned papers presents analyses or measurements of the pdfs for the end-to-end latency of a fault-tolerant CORBA infrastructure.

Substantial work, both analytical and experimental, has been undertaken on the performance of the Totem protocols. Budhia et al. [4] measured the performance of the Totem protocol using a test message driver in the Totem protocol stack, so that the test messages never wait for the arrival of the token. Thomopoulos [19] analyzed and measured the probability density functions of the latency for sending one-way messages in Totem. In his analyses, he assumed that, at the moment that a message is generated, the token is randomly located on the ring. In addition, he assumed that the send events are independent of each other. In essence, he compared the tails of the probability density functions.

Karl et al. [6] 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 et al. [11,13] measured the performance of a fault-tolerant CORBA infrastructure that is different from the fault-tolerant CORBA infrastructure that we have implemented and used for the
performance measurements in this study. Her measurements focused primarily on throughput rather than end-to-end latency.

In our analyses and measurements, we consider both constant and random distributions of the client “think” times. In particular, we recognize that, for synchronous remote method invocations, the send events are not independent. We study the consequence of the correlation of the send events for requests and replies, and present guidelines for achieving the best end-to-end latency by minimizing the message-send delays at the server.

4. Latency analyses

We investigate the end-to-end latency of a single round of remote method invocations issued by the client when there are no faults. The analyses correspond to the determination of the end-to-end latency at the peak probability densities of our measurements. We ignore factors that affect the latency with low probabilities, such as message (or token) loss and retransmission, replica processing uncertainties, operating system scheduling uncertainties, etc. The purpose of the modeling and analyses is to provide an understanding of the run-time behavior of the infrastructure and the sources of delay in the end-to-end latency.

4.1. The model

We assume the use of a general token-based group communication system. The nodes in the system are organized into a logical ring topology. A single client–server application runs on the nodes. We assume that the client is unreplicated and runs as the ring leader and that the server replicas are distributed across two or more other nodes on the ring.

We let \( n \) denote the number of nodes on the ring and label the nodes node\(_0\), node\(_1\), \ldots, node\(_{n-1}\), starting with the ring leader which runs the client. A server replica running on node\(_k\) is \( k \) steps away from the client, and the client is \( n - k \) steps away from the server, in the direction in which the token circulates.

The token circulates around the ring sequentially from node\(_0\) to node\(_1\) and so on to node\(_{n-1}\) and then back to node\(_0\). We let \( T_{\text{step}} \) denote the time for the token to pass from node\(_k\) to node\(_{k+1}\) on the ring, where \( 0 < k < n - 2 \), and \( T_{\text{step}}^{n-1} \) denote the time for the token to pass from node\(_{n-1}\) to node\(_0\) on the ring.

We let \( T_r \) denote a full token rotation time around the ring. In general, \( T_r \) varies for different token rotations. Sometimes, we use the notation \( T_{j}^{r} \), when it is necessary to refer to a particular token rotation \( j \). When there is no user message sent during a round of token rotation, \( T_r \) is simply the sum of the token-passing times for all of the steps. We denote this special case by \( T_{\text{idle}}^r = \sum_{i=0}^{n-1} T_{\text{step}}^r \).

When the client issues a synchronous remote method invocation, the CORBA ORB prepares a request message and sends it to the fault-tolerant infrastructure. The fault-tolerant infrastructure prefixes the message with a protocol header and then sends the message to the Totem layer. The Totem layer multicasts the message to the server replicas when it receives the token. When the request message from the client arrives at the node on which the server replica runs, the Totem layer picks up the message and delivers it to the fault-tolerant infrastructure layer. The fault-tolerant infrastructure checks if the message is a duplicate and if the server object is the correct target for the request message. If so, the infrastructure delivers the message to the CORBA ORB and the ORB dispatches the request to the server application.
Fig. 1. The client and server applications with the supporting software layers and the round-trip path of a synchronous remote invocation. The various terms that contribute to the end-to-end latency are also illustrated.

The client and server applications with the supporting software layers, and the round-trip path of a synchronous remote method invocation, are illustrated in Fig. 1. The server application object then processes the request and sends a reply back to the client. The reply message goes through the different layers in the reverse direction and eventually arrives at the client.

As can be seen, the end-to-end latency of the remote method invocation, denoted $T_{e2e}$, is the difference between the time at which the client receives a reply to a request and the time at which it issues that request (i.e., $T_{e2e} = t_{15} - t_{0}$).

The client processing time, denoted $T_{cproc}$, consists of two parts. The first part is the time it takes for the request to pass from the client application down to the Totem layer. This includes the time it takes the CORBA ORB to marshal a remote method invocation into a request message and the handling time of the fault-tolerant infrastructure (i.e., $t_2 - t_0$). The second part is the time it takes for the reply message to pass from the Totem layer up to the client application. Again, this includes the message handling time and the time it takes the CORBA ORB to unmarshal the reply message and dispatch it to the object (i.e., $t_{15} - t_{12}$).

The server processing time for the server running on node $k$ is denoted $T_{sproc}$. In addition to the time it takes for the request and reply messages to traverse the software layers (i.e., $t_7 - t_4$ and $t_{10} - t_8$), $T_{sproc}$ includes the time for the server replica to process the request (at the application layer) and prepare the reply (i.e., $t_8 - t_7$). Because the reply of only one of the server replicas is delivered to the client and only that server replica’s processing time contributes to the end-to-end latency, we let $T_{sproc}$ represent the server processing time when it is clear from context which node is running the server replica or when such information is not important. The client and server processing time is referred to collectively as the application-processing time.

The interval between two consecutive remote method invocations issued by the client (the “think” time) is $T_{think}$. $T_{delay}$ is the delay after a client issues a request and before the client’s node receives the token and multicasts the message (i.e., $t_1 - t_2$). Likewise, $T_{delay}$ is the delay for the reply message to be
sent at the server running on node $k$ (i.e., $t_{11} - t_{10}$). We drop the superscript denoting the node number when it is clear from context.

The latency for Totem to send and handle a user message $m$ is $T_m$, which depends on the length of $m$. In Fig. 1, for the request this latency is $T_{req} = t_4 - t_3$, and for the reply it is $T_{req} = t_{12} - t_{11}$. There are several other factors that affect the latency $T_m$, including the type of message and the relationships between the replicas at the sending node and at the neighboring node in the next token step. In the analyses, we ignore these factors. Again, sometimes we drop the superscript for convenience.

### 4.2. Determination of the end-to-end latencies

Assuming that a reply message sent by the server replica running on node $k$ reaches the client first, the end-to-end latency $T_{e2e}$ consists of the following terms: the processing time $T_{cproc}$ at the client, the message-send delay $T_{cdelay}$ at the client, the transmission time $T_{req}$ for the request, the processing time $T_{k}$ at the server, the message-send delay $T_{k}$ for the request, and the transmission time $T_{rep}$ for the reply message. Thus, $T_{e2e}$ is given by:

$$T_{e2e} = T_{cproc} + T_{cdelay} + T_{req} + T_{k} + T_{k} + T_{rep}.$$  \tag{1}$$

To determine the send delay for the reply message from a server replica, we need to establish a common reference point in time for the server processing and for the token rotation. We choose to use the point in time when the request message is (fully) received at the server replica, i.e., $t_4$ in Fig. 1. This leads to $T_{k} = T_{11} - T_{10} = (T_{11} - t_4) - (T_{10} - t_4)$. The first part, i.e., $(t_4 - t_3)$, represents the duration between the first token visit after the reply message is produced by the server replica and the receipt of the request message, and the second part, i.e., $(t_4 - t_4)$, is $T_{step}$. Because the message transit time over the network is negligible, $t_4$ is very close to the time at which node 0 (i.e., the node on which the client runs) is ready to pass the token to the next node (or to multicast other user messages if there are other clients on the same node). Assuming that the token has circulated $p$ times after the first visit to node $k$ after the multicast of the request message and that there are $u$ user messages multicast in the meantime (where both $p$ and $u$ can be zero), the sending delay $T_{k}^{delay}$ for the reply message is given by:

$$T_{k}^{delay} = pT_{step} + \sum_{i=0}^{u-1} T_{msg} + \sum_{j=0}^{k-1} T_{msg} - T_{proc}.$$  \tag{2}$$

Similarly, $T_{delay}$ we choose to use the point in time when the reply message is (fully) received at the client, i.e., $t_{12}$ in Fig. 1. Assuming that the token has circulated $q$ times after the first visit to node $k$ after the multicast of the request message and that there are $v$ user messages multicast during the token rotation (where both $q$ and $v$ can be zero), the send delay $T_{delay}$ for the request message at the client is given by:

$$T_{delay} = qT_{step} + \sum_{i=0}^{v-1} T_{msg} + \sum_{j=0}^{v-1} T_{msg} - (T_{msg} + T_{thick}).$$  \tag{3}$$

Note that we have used simplifying notation for the user message passing time $T_{msg}$ in Eqs. (2) and (3) but that it is not exact because different terms refer to different messages.
The token rotation time $T_i$ is given by

$$T_i = T_i^{idle} + \sum_{i=0}^{s-1} T_i^{msg},$$

(4)

where $s$ is the number of user messages sent in one token rotation ($s$ may be zero). In general, $T_i$ varies for different rotations because of the second term of the right side of Eq. (4).

After expanding the terms in Eq. (1), and recalling that $T_i^{idle} = \sum_{i=0}^{s-1} T_i^{idle}$, we have

$$T_{2i} = (p + q + 1)T_i^{idle} + \sum_{i=0}^{s-1} T_i^{msg} + T_i^{req} + T_i^{rep} - T_{think}.$$  

(5)

It might appear to be counterintuitive that the end-to-end latency does not depend on the server-side and client-side processing times. Actually, the processing time is included in the first term of the right side of Eq. (5), i.e., $(p + q + 1)T_i^{idle}$.

Eq. (5) predicts that the client will not see a continuous increase in the end-to-end latency if the server processing time varies continuously, assuming that $T_{think}$ is constant. Note that the presence of duplicate messages, checkpoint messages, and messages sent by other applications can result in greater end-to-end latency, as indicated by the second term of the right side of Eq. (5).

Another implication of Eq. (5) is that the client invocation pattern, reflected by the client “think” time $T_{think}$, also affects the end-to-end latency. In particular, $T_{think}$ affects the end-to-end latency when duplicate replies are present for active replication. If $T_{think}$ is greater than the time that it takes the client to receive duplicate replies and filter them, the duplicate messages do not have a negative effect on the end-to-end latency as seen by the client. However, if $T_{think}$ is less than that time, the send delay for the next request might increase, due to Totem’s sending duplicate replies, because the client must wait to receive the token.

If there are no additional messages beyond the non-duplicate request and reply for each remote method invocation, Eq. (5) can be simplified to:

$$T_{2i} = (p + q + 1)T_i^{idle} + T_i^{req} + T_i^{rep} - T_{think}.$$  

(6)

Assuming that the “think” time at the client is zero, and that no duplicate message appears on the ring, Eq. (5) can be further simplified to:

$$T_{2i} = (p + q + 1)T_i^{idle} + T_i^{req} + T_i^{rep}.$$  

(7)

If the request and reply messages have the same length (the message passing time for each is $T_i^{msg}$), and the token-passing time for each step is the same (denoted $T_0$), the end-to-end latency can be reduced to an even simpler form:

$$T_{2i} = (p + q + 1)nT_0 + 2T_i^{msg}.$$  

(8)

Because the position of the primary server replica on the ring can affect the values of $p$ and $q$, running the primary server replica on different nodes can lead to end-to-end latencies that differ by a complete token rotation time. Thus, care must be taken to determine the best position on the ring to run the primary server replica to achieve the best end-to-end latency. However, for active replication, the server replicas actually enter a competitive mode for sending replies. The server replica at the most favorable position on the ring successfully sends the reply first. The replies sent by the server replicas at the less favorable positions are deemed to be duplicates.
The analyses above are intended to capture the latency, at the peak probability density, for the fault-tolerant infrastructure under investigation. A more extensive model is required to represent the fault-tolerant infrastructure more completely. For example, the CPU contention between the token handling and the application process can affect the end-to-end latency, as shown by the measurement results presented below.

5. Measurements

In this section we present the experimental setup, the measurement methodology, and the measurement results for the probability density functions of the end-to-end latency in several different configurations, and compare the results with our analyses given in the previous section. We investigate the end-to-end latency dependency on the server sending node position, the client invocation pattern, and the server computation load. We summarize the overhead of the fault-tolerant infrastructure at the end.

5.1. Experimental setup

The testbed for our experiments consisted of four Pentium III PCs, each with 1GHz CPU, 256 MB of RAM, running the Mandrake Linux 7.2 operating system, over a 100 Mbit/s Ethernet local-area network, using e*ORB [21]. During the measurements, there was no other traffic on the network.

The testbed used a high-quality Ethernet. We determined that the loss rate of packets is less than $10^{-5}$ for a test client–server application, using UDP to send and receive messages with 1 kB payload synchronously. Retransmissions have an insignificant effect on the pdfs that we measured. Note that ruling out message loss in the experiment does not automatically render the end-to-end latency deterministic or predictable. There are various asynchronous factors, such as message transmission time and operating system scheduling, that contribute to the uncertainty of the run-time behavior.

Four copies of Totem run on the four PCs, one for each PC. We refer to these PCs as node0, node1, node2 and node3, in the order of the logical token-passing ring imposed by Totem, where node0 is the ring leader. Because one instance of Totem runs on each node, we use the same node designation to refer to the Totem instance on a node.

In the experiments, a CORBA client sends 1 kB payload (in the form of a sequence of 256 longs) to a replicated server, and the replicated server echoes back the same payload. The 1 kB payload length is chosen so that the complete user message can be put into a single Totem packet (1.4 kB). The Totem flow-control mechanism is not exercised in our experiments.

The client runs on the ring leader, node0. The server is three-way replicated, with one replica running on each of the other three nodes, node1, node2 and node3. For each run, the client issues 10,000 remote method invocations of the server.

5.2. Measurement methodology

The end-to-end latency is measured at the client as the interval between the time at which the client issues a request and the time at which the reply has returned. The rest of the measurements are carried out at the Totem layer, unless indicated otherwise.

The token rotation time (complete rotation of the token) is measured at each node in the Totem layer as the time interval between two consecutive token visits. The token-passing time is obtained by
dividing the idle token rotation time (when there is no user message) by the number of nodes on the ring.

The client (server) processing time is measured at the Totem layer as the interval between the time at which a reply (request) is delivered to the client (server) and the time at which the next request (corresponding reply) is passed to the infrastructure. The client processing time measured in this way inevitably includes the client “think” time, if any. The message-send delay is also measured at the Totem layer.

The message-send delay and the application-processing time are measured at each relevant node, namely, the client node, the primary server node (if semi-active replication is used), and each server replica (if active replication is used).

Because the operating system rounds the time of the sleep calls (e.g., `nanosleep()`, `select()`, etc.) to clock ticks, we use an alternative way to control the intervals between two subsequent remote method invocations (“think” times) at the client. We insert an empty iteration loop (i.e., a `for` loop) between two consecutive remote method invocations, and control the client “think” time by setting different upper bounds for the iteration loop. The same approach is used to represent the server processing time. A drawback of this approach is that the intended distribution of the “think” time at the client, or the computation time at the server, might be distorted by the interference of token processing, as shown in the next section. To control the server processing time for each remote method invocation, we generate the upper bound for the iteration at the client and piggyback it in the payload of the request message, so that the server replicas have consistent information about the computational load. We record the actual processing time at the server replica and piggyback it in the payload of the reply message.

For active replication, each server replica encodes its node position in its reply message. The client extracts the server’s node position from the reply message delivered to it and records that position. This helps correlate the server sender node position with the end-to-end latency.

The experimental data are stored in a buffer and are written to a set of files when Totem and the applications exit. The experimental data files are processed offline to produce the pdfs. The resolution for the pdf calculation is $1\mu s$. The cost of each clock reading, i.e., `gettimeofday()` is about $0.5\mu s$, which contributes insignificant overhead to the latency.

5.3. Latency dependency on the sending node position

In this experiment a client issues synchronous remote method invocations with zero “think” time for a three-way semi-actively replicated server with minimum processing. To gain a thorough understanding of the system in this basic configuration, we observe the system from several points of view (other than end-to-end latency) and show that these different views are consistent and complementary.

The client has zero “think” time, and the server echoes the client’s request without executing the additional iteration loop that we inserted. The server is semi-actively replicated. Each of the three scenarios, where node$_1$, node$_2$ or node$_3$ hosts the primary server replica, is investigated. In addition to the end-to-end latency, we instrumented Totem and the replication infrastructure code to collect the token rotation time, the message-send delay, and the application-processing time. We show that all of these data are indeed consistent and complementary.

The pdfs for each of the three scenarios, where node$_1$, node$_2$, or node$_3$, is running the primary server replica are shown in Fig. 2. The principal peak when node$_2$ hosts the primary server replica is located at $745\mu s$. It is approximately one idle token rotation time ($205\mu s$) to the left of the principal peak positions
Fig. 2. The measured pdfs for the end-to-end latency as seen by the client. From top to bottom, the plots are pdfs for running the primary server replica running on (a) node1, (b) node2, and (c) node3.

for the other two configurations. The reason for this displacement is that, with high probability, both the client and the server processing times are less than the time for two token-passing steps, as Fig. 3 shows. Therefore, one remote method invocation can be performed in a single token rotation. Consider Eq. (8) for the case in which $p=q=0$ and $n=4$. The cost $T_{msg}$ for one user message can be inferred from Eq. (8) to be $(745 - 205)/2 = 270 \mu s$. A secondary peak exists when node2 hosts the primary server replica. This secondary peak is also located about one idle token rotation time to the right of the principal peak position, suggesting that occasionally the primary server replica or the client cannot get the next message ready for the first token visit.

When node1 hosts the primary server replica, two complete token rotations are required for each remote method invocation, because of the non-favorable position of the primary server replica. As shown in Fig. 4, when node1 hosts the primary server replica, it does not have enough time to issue the reply message during the first token visit after it receives the request and, consequently, the reply can be sent only when the token visits node1 again. This causes an idle token rotation starting at node1. Likewise, when node3 hosts the primary server replica, the client is not ready for the next request at the first token visit after it receives the previous reply and, therefore, an idle token rotation occurs starting at node3.

That the principal peak latency when node3 hosts the primary server replica is slightly greater than the peak latency when node1 hosts the primary server replica reveals that $T_{msg}$ is not exactly the same for every message even if the messages have the same payload length, or the payloads are identical. This is a consequence of using single CPU computers in our testbed, and the sequential nature of the Totem single-ring protocol. When a node sends a message, the node at the next token step also handles the message. Depending on the relationships between the application processes, and the type of message, the message might be delivered to the application process running on the neighboring node, or it might be dropped.
Fig. 3. The measured pdfs for the application-processing latency for both the client and the primary server replica. The plots from top to bottom are for semi-active replication with (a) node 1 running the primary server replica, (b) node 2 running the primary server replica, and (c) node 3 running the primary server replica.

Fig. 4. The measured pdfs for the send delay at the client and the primary server replica. The plots from top to bottom are for semi-active replication with (a) node 1 running the primary server replica, (b) node 2 running the primary server replica, and (c) node 3 running the primary server replica.
by the fault-tolerant infrastructure. In summary, the processing cost of request and reply messages at a client and a server can be categorized as follows:

**Case 1.** *Incoming request message at a client, incoming reply message at a server.* The processing cost is low, because the message is dropped immediately after the replication mechanisms receive the message from Totem and determine that it is not intended for an application process running on this node. Therefore, the message is not delivered to the application.

**Case 2.** *Duplicate incoming request message at a server, duplicate incoming reply message at a client.* The processing cost is slightly more than Case 1 because the message must pass through the duplicate detector, which involves extra processing, before it is dropped.

**Case 3.** *Incoming request message at a server, incoming reply message at a client.* The processing cost is significantly more than the previous two cases because the message is delivered all the way up the protocol stack to the application process.

We let the message passing time corresponding to the above three cases be $T_{\text{msg}}^1$, $T_{\text{msg}}^2$, and $T_{\text{msg}}^3$, respectively. Obviously, they have the following relationship: $T_{\text{msg}}^1 < T_{\text{msg}}^2 < T_{\text{msg}}^3$. Note that Case 2 occurs only for active replication and during the recovery stage for semi-active replication.

Any processing of the message by the replication infrastructure or the application process on a node would delay the handling of the token arriving at that node immediately after the reception of the message because of CPU contention. Moreover, Totem competes for the CPU to handle the incoming token, while the application process is processing the message that it just received, which increases the application-processing time of a message, as shown in Fig. 3.

For example, when node$_3$ hosts the primary server replica, the neighboring node (node$_0$) hosts the client, and the reply message sent by node$_3$ is delivered to the client (Case 3). Because of the single CPU, the client (for processing the reply and preparing the next request) and Totem (for token processing) contend for the CPU. Hence, an extra delay is seen by both the client (due to message handling and processing) and the server (due to token handling). On the other hand, when node$_3$ hosts the primary server replica and node$_2$ hosts a server replica, the message is dropped early without reaching node$_2$ (Case 1) and the CPU contention is kept to a minimum. Thus, $T_{\text{msg}} = T_{\text{rep}}^3$ for the reply message when node$_3$ hosts the primary server replica and the same message when node$_2$ hosts the primary server replica. Because all of the replicas perform the same processing, the request message sent by the client is the same for both scenarios and does not contribute to the latency difference (i.e., for both scenarios the latency for passing the message is $T_{\text{msg}}^3$). Therefore, the end-to-end latency in Fig. 2(c) is greater than that in Fig. 2(a). In fact, the message passing time of 270 µs from the end-to-end latency measurement is the average of $T_{\text{ring}}^3$ and $T_{\text{msg}}^3$.

The above observation is further reinforced by looking at the measured pdfs for the complete token rotation time as seen by each Totem instance for the different scenarios, as shown in Fig. 5. The peak near 205 µs corresponds to the complete token rotation time when there is no interference between the processing of the token and the application messages broadcast by Totem. Other peaks with larger latencies are the results of multicasting 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 from the ring leader, which hosts the client) hosts the primary server replica, the total number of token rotations during each run is about 12,500.

From our analyses, the “loaded” token rotation time is expected to be the sum of the idle token rotation time and the total message passing time, i.e., \(4 \times T_0 + kT_{msg}\), where \(k\) is the number of messages broadcast by Totem during this token rotation. Using the measured average \(T_{msg} = 270\) µs, we expect to see a peak at \(205 + 270 = 475\) µs for one-broadcast-per-token-rotation and at \(205 + 2 \times 270 = 745\) µs for two-broadcasts-per-token-rotation.

What we observed is close to the results of our analyses, with several deviations regarding the position and the number of expected peaks, which should be no surprise after our previous discussion of the effect of CPU contention. As shown in Fig. 5(a) and (b), there are two peaks present around 475 µs. One peak is located around 450 µs, which corresponds to \(4 \times T_0 + T_1\), and the other is located around 500 µs, which corresponds to \(4 \times T_0 + T_3\). There is a single peak located around the expected value for the two-broadcasts-per-token-rotation case. The values of \(T_{msg}\) and \(T_3\) can be estimated to be 245 µs and 295 µs, respectively.

Fig. 5. 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 (a) node 1 running the primary server replica, (b) node 2 running the primary server replica, and (c) node 3 running the primary server replica.
5.4. Latency dependency on the client invocation pattern

We also studied the latency dependency on the client invocation pattern, by varying the client “think” times. For semi-active replication, and active replication with a smart sending-side duplicate filter [25], which involves no extra messages being sent other than the request and (non-duplicate) reply for each remote method invocation, the end-to-end latency is simply offset by the “think” time of the client, compared with the zero “think” time discussed above.

For active replication without effective sending-side duplicate filtering, however, the end-to-end latency seen by the client depends on the interval between two consecutive invocations (i.e., the client invocation pattern), because duplicate messages can delay the token rotation and enlarge the end-to-end latency, as predicted by Eqs. (4) and (5).

The pdfs of the end-to-end latency for active replication without and with smart sending-side duplicate filtering, with zero client “think” time, are given in Fig. 6. The effect of duplicate messages is obvious—the end-to-end latency is significantly greater without sending-side duplicate suppression. From Eq. (5) and the analyses of the effect of CPU contention on the message passing time, it follows that the end-to-end latency with the presence of duplicate messages is obtained by adding the following:

- The idle token rotation time $4 \times T_0$;
- The message passing time $T_{msg}$ for the request message sent by node0 (i.e., the client), which is an incoming request at a server process;
- The message passing time $T_{msg}$ for the reply message sent by node1, which is an incoming reply at the neighboring node that hosts a server replica;
- The message passing time $T_{msg}$ for the reply message sent by node2, which is an incoming reply at the neighboring node that hosts a server replica;
- The message passing time $T_{msg}$ for the reply message sent by node3. Because the token visits node2 before node3, and because it is highly probable that node2 has the reply message ready before the first token visit, the reply message sent by node3 is most likely to be a duplicate.

---

Fig. 6. Probability density functions of the end-to-end latency for active replication, (a) without and (b) with effective sending-side duplicate suppression. The client has zero “think” time in these two measurements.
Thus, the end-to-end latency $T_{e2e} = 4 \times T_0 + 2 \times T_{msg}^{1} + T_{msg}^{2} + T_{msg}^{3}$. By substituting the values obtained from our measurements, and by assuming $T_{msg}^{1} \approx T_{msg}^{2}$, we expect to see a peak located at $205 + 2 \times 245 + 245 + 295 = 1235 \mu s$. The observed peak at $1232 \mu s$ is close to the expected location.

It is interesting to note that the pdf with sending-side duplicate suppression is almost identical to that of semi-active replication with node2 running the primary server replica.

For active replication, the client “think” time is modeled by inserting an iteration loop and controlling the upper bound of the iteration. The upper bound for each run follows a Poisson distribution. The actual client “think” time distributions for four different runs, together with the corresponding pdfs for the end-to-end latency, are shown in Fig. 7. To display the effects of the client invocation pattern, we turned off the sending-side duplicate suppression mechanism. Even though the computation load at the client follows a Poisson distribution, the contention for the single CPU between the client process and Totem caused a longer than expected busy waiting time and, hence, distorted the distribution, as Fig. 7(a) shows. (Note that the Totem process and the application processes all run as normal priority user-level processes. However, the Linux scheduler dynamically assigns and adjusts the priorities of different processes.) The corresponding pdfs for the end-to-end latency for each distribution are shown in Fig. 7(b). The effects of the client’s busy waiting and the token handling contention are also obvious in the pdfs for the end-to-end latency. Recall that the latency for sending each message is about $270 \mu s$. When the mean “think” time is greater than twice that amount, as is the case shown at the bottom of Fig. 7, the effect of duplicates is

![Fig. 7](image-url)
eliminated. We see a broad plateau that starts at approximately 730 µs (which matches the measurement for semi-active replication when node₂ hosts the primary) and extends to approximately a complete token rotation time, with a dip at the right end that is caused by CPU contention between the client process and Totem.

5.5. Latency dependency on the server computation load

For this experiment we use a zero client “think” time for all of the runs. The server processing time is varied by setting a different computation load (the upper bound for an empty for loop) for each invocation. The upper bound is increased from 0 to $7 \times 10^5$ with a step size of $10^4$ for each run. Measurements are performed for both active and semi-active replication.

Fig. 8(a) shows only the pdfs for the first 28 runs for active replication, because the pdfs for the end-to-end latency for the two replication styles are similar. The corresponding server computation load is labeled on the right-hand vertical axis. As can be seen from the figure, except for the first 10 runs, the peak pdf positions remain the same for eight runs (the server processing time varies by one complete idle token rotation time $4 \times T_0$) and, then, the peak pdf positions jump by another idle token rotation time. They then remain the same for the next cycle of eight runs. Note that the peak latencies for the first two runs in each cycle are slightly greater than those for the other runs. This effect is due to the selection of node₃, rather than node₂, as the sending node.

Fig. 8(b) shows the actual server processing time at the peak probability densities for each computation load. There are noticeable periodic discontinuities in server processing time as the computation load increases. Again, this is a result of Totem’s interference in the user process computations. The measured processing time for the same set of computation loads on an unloaded node (i.e., there is no other CPU-intensive process such as Totem running on the same node), which is also shown in the figure, does not exhibit the latency jumps.

Except for small $T_{sproc}$, the end-to-end latencies at the peak pdf positions (i.e., $T_{e2e}$) are a step function of $T_{sproc}$, which Eqs. (5)–(8) have predicted. This relationship is plotted in Fig. 8(c) for active replication and in Fig. 8(d) for semi-active replication.

For small server processing time (i.e., the first 10 runs), the peak positions of the pdfs deviate from our previous analyses. Again, this effect is due to the CPU contention between the server process and Totem. For active replication, a request message from the client is first delivered to the server replica that runs on node₁, and the token then arrives immediately afterwards. Because the server replica just obtained the CPU for processing the request message, the Totem process is not scheduled until some amount of time $T_{sched}$ has passed. Only then does Totem handle the token. Consequently, if the actual server processing time is smaller than $T_{sched}$, the token will have to wait until the server replica finishes the processing before Totem handles the token. (This does not mean that the replica running on node₁ has the opportunity to send the reply at this token visit because, when the replica sends the reply through the Unix socket to Totem, Totem gains the CPU and starts processing the token. By the time Totem receives the reply from the client, it has already passed the token to the next node.) As long as the condition $T_{sched} < T_{sproc}$ is satisfied, node₂ is the most favorable position for sending the reply. Consequently, we see a linear increase in the end-to-end latency with the increase of $T_{sproc}$ for $T_{sched} < T_{sproc}$.
5.6. Overhead

To evaluate the overhead of the fault-tolerant infrastructure, we measured and compared the pdfs of the end-to-end latency of the test application without and with replication. For completeness, we also measured the end-to-end latency of the application on top of our fault-tolerant infrastructure unreplicated on two nodes, i.e., the server runs on one node and the client runs on the other node of a two-node Totem ring with zero “think” time.

The end-to-end latency measurements for the three scenarios mentioned above (with no additional server computation load) are summarized in Fig. 9. Three-way active replication is used for the replicated scenario. It might appear to be surprising that the overhead is greater when the application runs on top
Fig. 9. A comparison of the end-to-end latency under the following three scenarios (from left to the right): (i) replicated server using TCP, (ii) unreplicated server running with the fault-tolerant infrastructure on a two-node Totem ring, and (iii) three-way actively replicated server running with the fault-tolerant infrastructure on a four-node Totem ring.

of our fault-tolerant infrastructure unreplicated compared with the three-way actively replicated case. However, it can be easily understood by considering the extra send delay of reply messages due to the two-node Totem ring setup.

We further characterized the overhead when the server computation load varies. The overhead for active replication (i.e., the additional delay for each remote method invocation) is shown in Fig. 10(a). Because of the CPU contention between Totem and a server replica, the time to perform the computation at the server replica is increased. This reveals that, besides the communication-related overhead incurred by the fault-tolerant infrastructure, which is typically proportional to the message size, there exists additional computation-related overhead for single CPU computers. (Bear in mind that, with a single CPU, if the CPU is engaged in computation for message ordering and delivery, it cannot be used for application computation. Therefore, the extra computation for the replication-related tasks is overhead incurred by the fault-tolerant infrastructure.) The computation-related overhead can constitute a large portion of the total run-time overhead when the server computation load is high, as shown in Fig. 10(a). The results for semi-active replication are similar, but with an additional complete token rotation time for some server computation loads, depending on the position of the primary server replica. For active replication and semi-active replication, where node2 hosts the primary server replica, the end-to-end run-time overhead starts at 70% and decreases to 50% as the server computation load increases.

The oscillation of the total run-time overhead is due to the variation of message-send delays. As can be seen from Fig. 10(a), the message-send delay can be a significant source of the end-to-end run-time overhead. The oscillation amplitude for the run-time overhead, and also the oscillation period, are about one complete token rotation time. Because active replication automatically selects the server replica at the most favorable position to send reply messages, it minimizes the server-side send delay. Thus, with active replication, the client-side send delay dominates the total message-send delay. This is true regardless of the distribution of the “think” time at the client. The client-side send delay, in turn, depends on the selection of the sending node for the different server processing times, as shown in Fig. 10.
Fig. 10. For active replication, (a) the run-time overhead of the end-to-end latency and computation overhead at the server, for different server computation loads, and (b) the effective sending node position for the corresponding measurements given in (a) at different server computation loads.

Because for different runs, reply messages from different server replicas might arrive first at the client, we calculate an overall effective sending node position for each server computation load. The effective sending node position is determined by taking the sum of each sending node position weighted by the probability that the node transmits the reply message. For example, if the observed probabilities for node1, node2 and node3 as the sending node are 0.05, 0.8 and 0.15, respectively, the effective sending node position is $1 \times 0.05 + 2 \times 0.8 + 3 \times 0.15 = 2.1$, i.e., very close to node2.

The effective sending node position as a function of the server computation load is shown in Fig. 10(b). As can be seen, the oscillation period is indeed about one complete token rotation time (i.e., 205 μs).

6. Conclusion

We have performed in-depth analyses and measurements of the end-to-end latency of a fault-tolerant CORBA infrastructure under fault-free conditions, with focus on the pdfs for the latency at the peak probability densities for different replication styles, positions of the primary server replica on the ring, client invocation patterns, and server processing times.

For semi-active replication, and for the same server processing load, our experiments have shown that different choices of the position of the primary server replica on the ring can affect the latency by a complete token rotation time, which can become significant as the ring size increases. For active
replication, the presence of duplicate messages can adversely affect the end-to-end latency. By effectively suppressing duplicate messages at the sender, active replication exhibits advantages over passive and semi-active replication; in particular, the message-send delay is minimized at the server replica as a result of the automatic selection of the sending node in the most favorable position.

Although we considered a single-client scenario, our results can be extended to multiple client scenarios. When multiple clients are present on the logical ring, the request and reply messages for one client appear to another client as extra messages and, thus, the end-to-end latency for each client is increased. The message-send delay (caused by waiting for the token) and CPU contention (caused by the token handling) are still the two primary factors that contribute to the overhead of the fault-tolerant infrastructure for multiple client scenarios. Moreover, for an individual remote method invocation, selecting which server replica to send the reply still has a significant effect on the message-send delay and, hence, on the end-to-end latency. However, because the token rotation is affected by the presence of multiple clients, a good primary server position for one invocation might not be the same as that for another invocation. Consequently, using a fixed primary server might not work as well as for the single-client scenario.

Given these results and observations, we recommend that, when a token-based group communication system such as Totem is used in practical applications, the following guidelines be followed to reduce the end-to-end latency:

- If a good sending-side duplicate suppression mechanism is available, active replication should be used and the replicas should be distributed around the logical token ring so that one of them can send the reply message with minimum delay. This is important, particularly if there are multiple clients because then it takes longer for the token to circulate and also because an optimum sending node can be automatically selected for different invocations.
- If a good sending-side duplicate suppression mechanism is not available, one should estimate the cost of sending duplicate messages using active replication with respect to the potentially long message-send delay due to the fixed primary server position in semi-active or passive replication if there are multiple clients. If the number of clients is large and the replication degree is small, active replication should still be used. Otherwise, if semi-active or passive replication is used, the position of the primary server replica must be determined carefully to avoid excessive message-send delays.
- If dual-processor computers are available, they should be used so that one processor can be used for token processing, reliable message delivery and message ordering and the other processor can be dedicated to the application processes.

Future work includes the characterization of the end-to-end latency under a variety of fault conditions (e.g., different message loss rates and different kinds of faults) and in different operating environments (e.g., in larger networks and on multi-processor computers).

References


