Design and Implementation of a Pluggable 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

In this paper we present the design and implementation of a Pluggable Fault Tolerant CORBA Infrastructure that provides fault tolerance for CORBA applications by utilizing the pluggable protocols framework that is available for most CORBA ORBs. Our approach does not require modification to the CORBA ORB, and requires only minimal modifications to the application. Moreover, it avoids the difficulty of retrieving and assigning the ORB state, by incorporating the fault tolerance mechanisms into the ORB. The Pluggable Fault Tolerant CORBA Infrastructure achieves performance that is similar to, or better than, that of other Fault Tolerant CORBA systems, while providing strong replica consistency.

1 Introduction

Several different approaches have been developed to provide fault tolerance for distributed applications based on the Common Object Request Broker Architecture (CORBA) standard [14] promoted by the Object Management Group (OMG). Early research efforts to enhance CORBA with fault tolerance took an integration approach, with the fault tolerance mechanisms implemented inside the ORB [2, 6]. That approach requires extensive modifications to both the ORB core and the applications and, therefore, is regarded as an intrusive approach. Later research efforts adopted a non-intrusive approach that is either service-based [4, 7, 12] with fault tolerance provided through service objects above the ORB, or interception-based [9, 10, 11] with the fault tolerance mechanisms implemented underneath the ORB (in a different address space).

In 1998 the OMG issued a Request For Proposals for Fault Tolerant CORBA (FT CORBA) with the aim of standardizing these efforts. The specification of FT CORBA was finalized by the OMG in April 2000 [13]. FT CORBA defines a set of service interfaces and underlying mechanisms that provide fault tolerance for CORBA applications through object replication, fault detection and notification, and logging and recovery. The FT CORBA standard mandates strong replica consistency for the replicated application objects.

To maintain strong replica consistency, some degree of integration of the fault tolerance mechanisms and the ORB is necessary to retrieve and assign the ORB state properly during the recovery of a replica and the transfer of state from a primary replica to the backup replicas [9, 10, 11]. In this paper, we present a new approach that incorporates the fault tolerance mechanisms into the ORB, while providing maximum transparency to both the ORB and the application. It uses the Pluggable Protocols Framework [5] that comes with many modern CORBA ORBs.

The Pluggable Protocols Framework (PPF) separates the messaging and network protocols from other parts of the ORB core and from the application objects. The PPF allows the network protocol to be replaced, and may also allow the messaging protocol to be replaced. Thus, the PPF makes it possible to develop CORBA applications for, and to deploy them in, environments for which the standard IIOP protocol is not appropriate [5]. The PPF also provides CORBA applications with improved quality of service by enabling the use of customized protocols that are tailored to those applications and their environments.

In this paper we present the design and implementation of a FT CORBA compliant infrastructure, the Pluggable FT CORBA Infrastructure, that is based on the PPF. We replace only the network protocol and not the messaging protocol. The Pluggable FT CORBA Infrastructure achieves performance that is similar to, or better than, that of existing FT CORBA systems [12, 19], while providing strong replica consistency and supporting a wider range of applications.

[*] This paper was published in *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

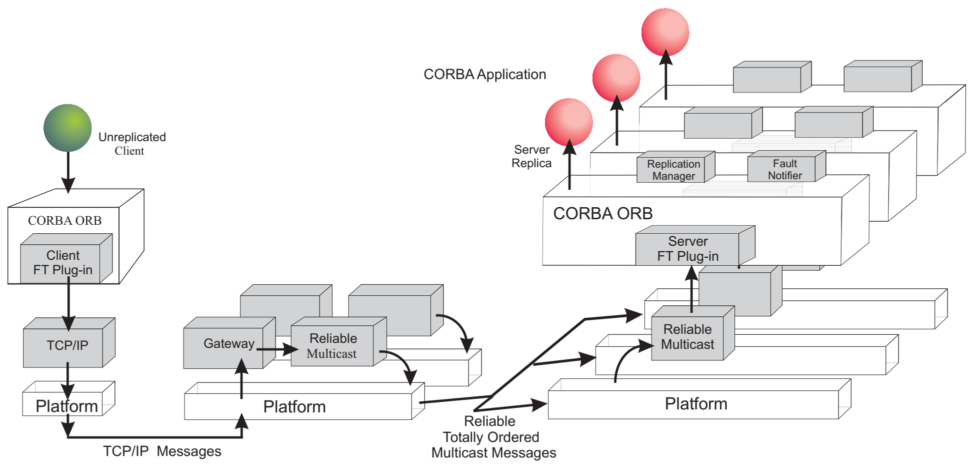

Figure 1. The Pluggable FT CORBA Infrastructure.

2 The Pluggable Protocols Framework

The Pluggable Protocols Framework (PPF) is now provided by many CORBA ORBs, including Orbix 2000 from Iona, ORBacus from Object Oriented Concepts, VisiBroker (real-time version) from Highlander Engineering, ORBexpress from Objective Interface Systems, TAO from Washington University, St. Louis, and e*ORB from Vertel, just to name a few. Although the PPFs differ slightly from one ORB to another, most of them are based on similar reactor-based design patterns [17].

The PPF provides a set of abstract base classes for the ORB-level transport interfaces. For example, the ORB- acus PPF provides `Connector Factory`, `Connector`, `Transport`, `Acceptor Factory` and `Acceptor` interfaces). The protocol plug-in must inherit from these classes in order to provide a concrete implementation for a specific protocol. The PPF provides concrete classes for the `Factory Registry` interfaces. Concrete implementations of the Factory classes register with the corresponding Factory Registries in order to be loaded into the ORB runtime.

3 Fault Tolerant CORBA

The Fault Tolerant (FT) CORBA standard defines interfaces and mechanisms for object replication, fault detection and notification, and logging and recovery.

The FT CORBA standard defines a Replication Manager that handles the creation, deletion and replication of application objects. Although each replica of an object has an individual object reference, the Replication Manager fabricates an Interoperable Object Group Reference (IOGR) for the replicated server object (object group) that clients use to contact that replicated server. The IOGR contains a profile for each of the replicas in the server object group. The Replication Manager inherits several interfaces for the purpose of fault tolerance property management, and replica group management.

The FT CORBA standard also specifies interfaces and mechanisms for hierarchical fault monitoring and notification at the object, type and/or location (e.g., host) level. To enable the health of its replica to be checked, an application object must inherit the `FT::PullMonitorable` interface, which allows the object to be pinged.

To enable the recovery of a replica that has failed, the FT CORBA standard defines interfaces for checkpointing that are inherited by each application object and that are used to retrieve and assign an application object's state. To allow its state to be transferred in this manner, an application object must inherit the `FT::Checkpointable` interface.

4 The Pluggable FT CORBA Infrastructure

The Pluggable Fault Tolerant (FT) CORBA Infrastructure is shown in Figure 1. On the server-side, fault tolerance is provided by the FT protocol plug-in (described in Section 4.1), the Totem group communication system [8] and the Replication Manager, Fault Notifier and Fault Detectors (described in Section 3). The Replication Manager and the Fault Notifier are implemented as CORBA objects running as separate processes, and are replicated using the active replication style. The Fault Detector that detects faults at the object level is implemented as a component of the FT protocol plug-in.

To accommodate the FT protocol plugged into the ORB, we have rewritten the process group layer of Totem [8]. In addition to providing the reliable totally-ordered multicast

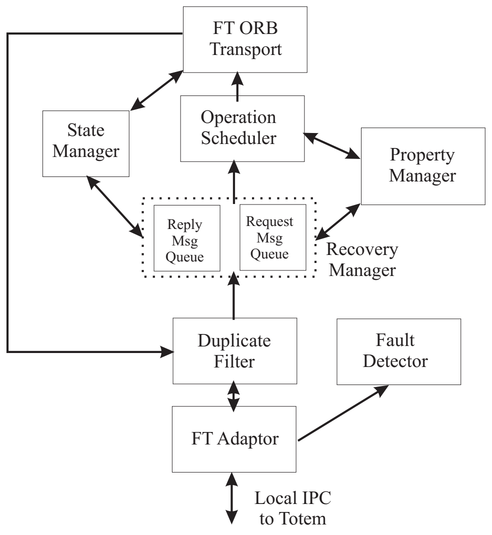

Figure 2. Components of the server-side FT protocol plug-in.

service, Totem also serves as a process-level and host-level fault detector. A fault report is created by Totem once it has detected a fault, and then delivers the report to the Fault Notifier. Totem also conveys the fault reports generated by the object-level fault detector to the Fault Notifier.

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

4.1 Server-Side FT Protocol Plug-in

The server-side Fault Tolerant (FT) protocol is plugged into the ORB using the pluggable protocols framework. As shown in Figure 2, the server-side FT protocol plug-in consists of the following components: FT Adaptor, Duplicate Filter, Recovery Manager, Property Manager, State Manager, Operation Scheduler and FT ORB Transport.

4.1.1 FT Adaptor

The FT Adaptor handles the connection to Totem, passes incoming messages to the Duplicate Filter, and routes outgoing messages to Totem. Currently, a Unix socket is used to connect the FT Adaptor and Totem.

The FT Adaptor attaches a FT protocol header to all messages. Totem uses the FT protocol header information to demultiplex the messages that are received and to deliver them to the correct FT Adaptor. The FT protocol header also contains information that is needed to detect and suppress duplicate invocations and responses, as explained later.

The FT Adaptor sends and receives two different types of messages: (1) the GIOP messages that the CORBA application objects use, and (2) the control messages that the FT Adaptor uses. An example of such a control message is the disconnection indication message that the FT Adaptor sends, when the ORB destroys the Transport instance, to ensure that the peer Transport instance is shutdown and garbage collected properly.

4.1.2 FT ORB Transport

The FT ORB Transport component implements the mandatory interfaces specified in the PPF for each ORB. For ORBacus, it includes the implementation of the `Connector Factory`, `Connector`, `Transport`, `Acceptor Factory` and `Acceptor` interfaces. The Pluggable FT CORBA Infrastructure retains the GIOP messaging protocol and changes the transport protocol. Instead of using TCP/IP, all of the transport instances share the same connection from the FT Adaptor to Totem. Synchronization mechanisms are built into the component based on the pthread library interfaces.

The instances of the concrete classes in the FT ORB Transport component, including the `Connector`, `Transport` and `Acceptor`, are owned by the ORB runtime. The `Transport` instances may be created either directly by the ORB runtime, or by the FT protocol during the setting of the ORB-level state in which case they are subsequently handed over to the ORB runtime. The FT Adaptor registers callbacks with these `Transport` instances to receive notification of the creation and deletion events. For these creation and deletion events, the FT Adaptor sends connection and disconnection indication control messages so that the FT protocol plug-in can take appropriate actions.

Each `Transport` instance has a definite role as a *server* or a *client* during its lifetime. A client `Transport` instance is used by a `Connector` instance to send request messages to a remote server and to receive the corresponding reply messages. A server `Transport` instance is used by an `Acceptor` instance to receive request messages from a remote client and to send the corresponding reply messages.

Each `Transport` instance is assigned a unique *transport id* that consists of a 4-tuple (`source group id`, `destination group id`, `connection sequence number`, `role`) by the FT protocol. The first two identifiers represent the identities of the sending and receiving groups, respectively. The connection sequence number *n*, which is determined by the client process that initiates the connection, indicates that this `Transport` instance is created for the *n*th connection requested by the particular client. The `Transport` instance on the server-side, that is created on a connection request from a client, uses the same connection sequence number as that used by the corresponding `Transport` instance on the client-side.

4.1.3 Duplicate Filter

The Duplicate Filter detects and suppresses duplicate messages, and passes only non-duplicate messages to the Recovery Manager.

For the purpose of duplicate detection, each `Transport` instance contains a `message filter id`, which counts the number of messages sent through a connection. In the FT protocol, duplicate incoming messages can occur only if they arrive for the same Transport instance (identified by the transport id). The Duplicate Filter stores the transport ids that it has seen, and also the last message filter id for each transport id (this suffices because the message filter id increases monotonically). The Duplicate Filter checks the transport id and the message filter id in the message header against its record. A message is deemed to be a duplicate if the Duplicate Filter finds a record with the same transport id but with a higher (or identical) message filter id. The Duplicate Filter drops a duplicate message once it has detected that the message is a duplicate.

For outgoing message duplicate detection, the Pluggable FT Infrastructure exploits the fact that the delivery of invocations on each Transport instance is serialized, regardless of the degree of concurrency that the ORB uses. For an outgoing reply message, if the Duplicate Filter finds that a new invocation has arrived for the `Transport` instance, it drops the outgoing reply message. For an outgoing (nested) request message, if the Duplicate Filter finds that the reply for the request message has already arrived, it drops the request message.

4.1.4 Recovery Manager

The Recovery Manager implements the Logging and Recovery Mechanisms. The log used by the Recovery Manager consists of two message queues, one for request messages and the other for reply messages. Non-duplicate request messages are stored in a total order in the request queue, and non-duplicate reply messages are stored in a total order in the reply queue. The log is used to guarantee the local single thread of control needed to achieve strong replica consistency, and it is also used for recovery. If only one queue were used, the request and reply messages would be interleaved and a linear search would be required to find and deliver a request or reply message to the ORB Transport. By using two separate message queues, one for requests and one for replies, the Operation Scheduler can retrieve a message simply from the top of each queue and deliver it to the ORB Transport.

During recovery, the Recovery Manager retrieves both the application-level state and the ORB-level state from the State Manager, fabricates a single message that contains both kinds of state, and multicasts the message using Totem. The Recovery Manager at a backup replica (for passive replication) or a newly started replica (for passive, active or semi-active replication) forwards the aggregate state transfer message to the State Manager. The State Manager then decomposes the state transfer message, recreates the FT Transport instances, if necessary, sets the message filter id for each Transport instance, and applies the application-level state. Once the application-level state and the ORB-level state are set at the recovering replica, the Recovery Manager replays the queued messages that it received after it had initiated the state transfer.

4.1.5 Property Manager

The Property Manager stores the fault tolerance properties, such as the replication style, the fault monitoring interval, and the checkpoint interval, for each object replica in the process. It also stores the information as to whether or not the local replica is the primary.

4.1.6 State Manager

The State Manager implements an interface with a method that allows the application to register its factory object. This factory object implements the `FT::GenericFactory` interface. The State Manager delegates the creation/deletion request from the Replication Manager to the application factory object. It keeps track of the object ids and the object references for the application objects that have been created and not yet deleted. The State Manager retrieves the application-level state by invoking the `FT::Checkpointable::get_state()` method of the application objects (using the stored object references). For the `FT::Checkpointable::set_state()` method, the State Manager first creates new application objects in the recovering process using the application factory object (the object ids are used for this purpose), if necessary, and then assigns the application-level state to each object by invoking the `FT::Checkpointable::set_state()` method. In addition, the State Manager is responsible for collecting the ORB-level state, most of which is stored in the ORB Transport component that is part of the FT protocol plug-in.

4.1.7 Operation Scheduler

The Operation Scheduler retrieves the request and reply messages from the respective message queues managed by the Recovery Manager. Except in some special cases where optimization is possible, the Operation Scheduler enforces a single-thread of control by delivering a new request message to the Transport component only after the process has no pending request unserviced.

4.1.8 Object-Level Fault Detector

The object-level fault detector is implemented as a C++ object. CORBA application objects that are monitored must inherit the `FT::PullMonitorable` interface. The object-level fault detector uses a pool of threads to pull monitor each registered CORBA object. Based on the user-specified Fault Monitoring Interval property, the `FT::PullMonitorable::is_alive()` method is invoked periodically. If the invocation does not return within a specified timeout period, or if it returns `CORBA::FALSE`, the Fault Detector determines that the monitored object is faulty.

The object-level fault detector is, in turn, monitored by Totem, which schedules special events periodically on each node in a deterministic manner. Whenever such a special event occurs, Totem sends a fault monitoring control message to the FT protocol plug-in. The object-level fault detector responds to this control message immediately, if it is alive. If the Fault Detector determines that a monitored CORBA application object is faulty, it indicates this fact in its response to the control message. If Totem receives a response with a faulty indication, or if it does not receive a response from the object-level fault detector within a specified timeout period, it shuts down the faulty process and fabricates a fault report to the Fault Notifier. Usually, one faulty object will affect a number of objects in the same process. Therefore, it is practical to shutdown the entire process when a faulty object is detected.

The Fault Monitoring Granularity property, defined by the FT CORBA standard, supports hierarchical fault detection by allowing the user to select object, location or location and type as the value of that property. The location value restricts the monitoring to a particular representative object at the given location (process, processor, network, etc), and the location and type value restricts the monitoring further to a particular representative object at the given location of the given type. Thus, regardless of the value of the Fault Monitoring Granularity, the object-level fault detector can be used.

4.2 FT Protocol Plug-in for Unreplicated Clients

The client-side fault tolerance mechanisms required by the FT CORBA standard can be implemented using the PPF for the ORBs that do not yet provide such support. The architecture of the client-side FT protocol plug-in is shown in Figure 3.

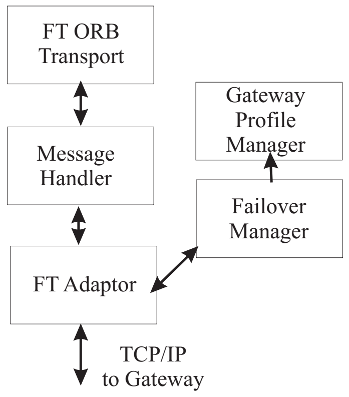

Figure 3. Components of the client-side FT protocol plug-in.

The FT Adaptor makes connections and communicates directly with the primary gateway of the fault tolerance domain using TCP/IP. The FT ORB Transport component uses the interfaces provided by the Message Handler to send and receive GIOP messages. The connections to the servers are separated from the FT ORB Transport component so that it is not aware of any lost connections that occur. The Message Handler inserts a fault tolerance service context into each outgoing message and delivers incoming replies to the FT ORB Transport.

The application must register the server Interoperable Object Group Reference (IOGR) with the Gateway Profile Manager, because the ORB itself is likely to keep only the first IIOP profile contained in the IOGR. The Failover Manager runs on a separate thread in order to heartbeat the primary gateway. If the Failover Manager determines that the primary gateway has crashed, it instructs the FT Adaptor to shutdown the current connection and to reconnect to the new primary gateway. The new primary gateway is found by cycling through the profiles in the IOGR that the Gateway Profile Manager maintains. No exception is thrown back to the application object unless all profiles in the IOGR have been tried and have failed.

4.3 Optimization and Practical Considerations

4.3.1 Semi-Active Replication Style

For active replication, duplicate messages in the network traffic are one of the major sources of overhead.

Duplicate detection at the sender is necessarily best-effort, because of the asynchronous nature of the replicas' concurrent sending of the same messages. Only the Duplicate Filter at the receiver can be guaranteed to capture and suppress duplicates.

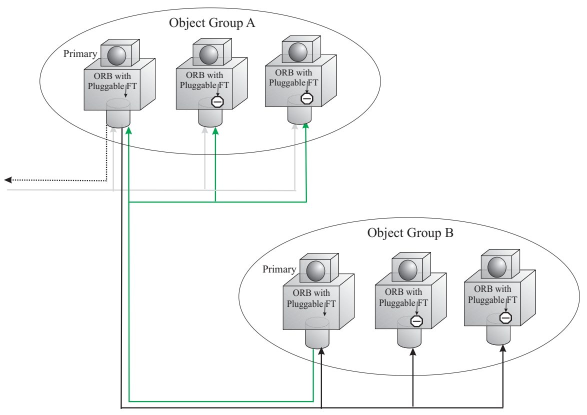

Figure 4. Interactions of object groups using the semi-active replication style.

Duplicate messages have the following negative effects: (1) they compete for network bandwidth, (2) they incur additional processing and transmission for reliable totally-ordered message delivery, and (3) they incur extra processing for duplicate filtering at the receiver. The negative effects of duplicate messages are more pronounced for large messages and for high replication degrees.

To address these negative effects, we advocate the use of *semi-active* replication, to eliminate duplicate messages under fault-free conditions and to provide fast recovery in the event of a fault. As shown in Figure 4 for the semi-active replication style, all replicas in each object group receive requests and replies from the other object groups with which they are interacting, but only one replica in each group (the primary replica) issues outgoing messages (replies or nested invocations). All outgoing messages from the secondary replicas are suppressed *within* the ORB.

In the semi-active replication style,

- All replicas within an object group process incoming messages in the same order; therefore, after each message has been processed, the states of the replicas within the group are consistent. Moreover, under fault-free conditions, all outgoing messages from the secondary replicas are suppressed.

- State transfer from the primary replica to the secondary replicas is not necessary. If the primary fails, one of the secondary replicas is elected as the new primary. The new primary starts by sending the last logged outgoing message. This works, by default, because the FT protocol plug-in enforces a single logical thread of control.

- In general, semi-active replication achieves better performance in terms of round-trip latency than
  - Active replication, because it eliminates duplicate messages
  - Passive replication, because it eliminates periodic state transfer messages from the primary replica to the backup replicas.

- With semi-active replication, the failure of the primary replica results in a brief hiatus in service. The factors that contribute to this delay include the fault detection time and the time to multicast a short control message. The control message informs the surviving replicas of the failure of the primary and the decision as to which replica is to serve as the new primary. In general, for semi-active replication, the delay is shorter than that for passive replication because, in passive replication, the new primary must process the logged invocations after the last checkpoint.

4.3.2 Flow Control for Active Replication

With active replication, flow control problems can arise, even if there are only slight variations in the processing speeds of the processors hosting the replicas, if those processors are loaded heavily and running close to their maximum processing capacities. In such a case, the progress of the clients is determined by the fastest server replica; as a consequence, the arrival rate of the incoming messages will exceed the processing capacity of the slow replicas. The message queues (*i.e.*, log) on the slow replicas will fill up due to either a pre-set limit on the number of messages allowed or a limit imposed by the finite physical memory of the processor. Eventually, the slow replicas will stop functioning, which defeats the purpose of active replication.

To avoid this backlog buildup problem at the slow replicas, the Pluggable FT CORBA Infrastructure employs several flow control mechanisms. Each FT protocol plug-in maintains two thresholds. The Recovery Manager checks the number of messages in the log against these two thresholds. If the Recovery Manager finds that the number of messages in the log exceeds the upper threshold, it asks the FT Adaptor to multicast a control message to the other replicas in the group. Upon receiving this control message, the FT protocol plug-in checks if the number of messages in the log is below the lower threshold. If so, the FT Adaptor slows down the rate of handling new messages until the number of messages in the log rises above the lower threshold.

Note that these flow control mechanisms are best-effort, in that the mechanisms cannot eliminate the overflow of the log at the slow replicas in all circumstances. The reason is that the FT protocol plug-in supporting the fastest replica is not aware of the situation and cannot take any action until it receives the control message. In the meantime, the incoming messages might have already caused the log overflow at the slow replicas.

5 Performance

We have implemented a prototype of the Pluggable FT CORBA Infrastructure using ORBacus 4.0.4 [15] and e*ORB 2.1 [18] and the C++ programming language. In this paper we provide experimental results for the pluggable protocol infrastructure based on ORBacus.

The experiments were performed on six Pentium III PCs over a 100Mbps local-area network. Each PC is equipped with a 1GHz CPU, 256MB of RAM, and runs the Mandrake Linux 7.2 operating system.

A simple client/server application was developed for the experiments. The server contains a method that takes a sequence of octet and returns an identical sequence of octet to the client, so that the request message is similar in length to the reply message, thus reflecting the round-trip nature of the messages.

For the active and semi-active replication styles, we focused on two performance measures: (1) runtime overhead, and (2) fault detection and recovery time. We investigated the runtime overhead as a function of payload length and replication degree. We measured the fault detection time for each process crash fault, the recovery time of loading a new replica as a function of state size, and the recovery time of the primary replica fault for the semi-active replication style.

5.1 Round-Trip Latency

For the overhead measurements, we ran one or more unreplicated clients on one PC and replicated servers on the other PCs, each on a distinct PC. The average latency for two-way invocations was read at the client-side.

5.1.1 Message Length Dependency

The FT protocol plug-in is layered underneath the GIOP messaging protocol and, therefore, does not parse or interpret the GIOP message body. Consequently, the overhead of the FT protocol depends on the lengths of the messages, rather than on the complexity of the data types or the signatures of the methods that are invoked across the network.

Figure 5 shows the average latency for synchronous remote invocations with various payloads for a three-way replicated server using both the active and semi-active replication styles, and also for the unreplicated client/server case.

The overhead of the Pluggable FT CORBA Infrastructure is due primarily to the following aspects of active replication:

- The Totem reliable totally-ordered multicast protocol involves significant CPU usage (10%+) and takes additional network bandwidth. In our experimental configuration, the network bandwidth taken by the rotating token of Totem is about 4Mbps.

- With active replication on identical processors, the duplicate filtering mechanisms cannot effectively suppress duplicate outgoing messages. Therefore, most of the duplicate messages show up in the Totem network traffic, wasting the processing power and the network bandwidth.

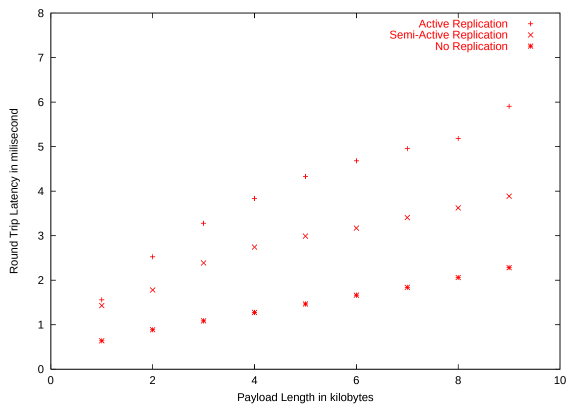

Figure 5. Round-trip latency as a function of payload length.

As Figure 5 shows, semi-active replication exhibits a significant performance gain over active replication, particularly for large messages.

5.1.2 Replication Degree Dependency

Figure 6 shows the round-trip latencies for two different payload sizes as a function of replication degree for the active and semi-active replication styles.

The Totem single-ring protocol uses a token rotating around a logical ring to order messages [8]. The ring size (*i.e.*, the number of processors running Totem) increases with the degree of replication, given that the replicas run on different processors. Therefore, as the degree of replication increases, the latency overhead increases because (1) the token rotation time increases, (2) the collective processing time at a node increases, and (3) for active replication, the number of duplicate messages in the network that must be delivered reliably and in order increases.

As Figure 6 shows, by eliminating the duplicate messages in the network for semi-active replication, the latency is reduced significantly over active replication, especially for invocations with large payload and for a high degree of replication.

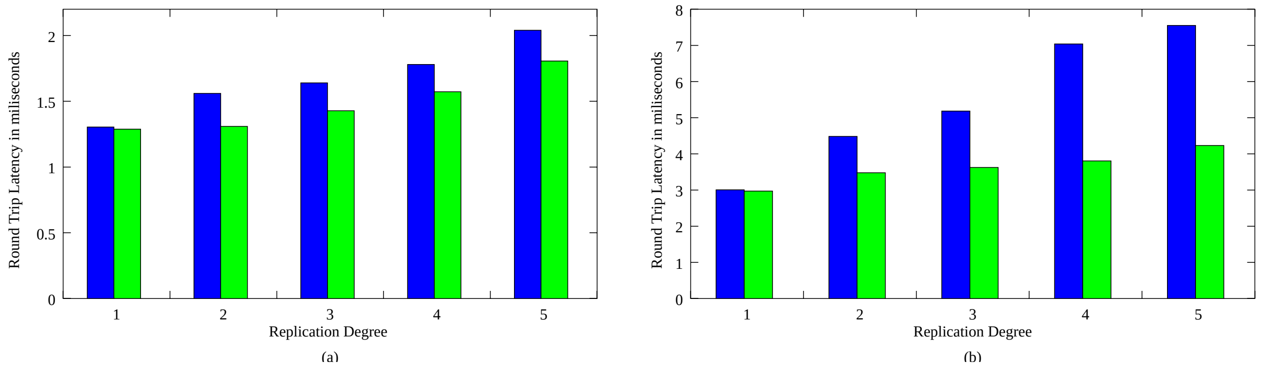

Figure 6. Round-trip latency as a function of replication degree for a payload size of (a) 1KByte, and (b) 8KByte. The dark shaded bars represent active replication, and the light shaded bars represent semi-active replication.

5.2 Fault Detection and Recovery Time

We also measured the time for Totem to detect a process crash fault. In the experiment, we select one of the processes on the same node as the victim. We record a start time before we send a kill signal to the process, and an end time when the notification regarding the lost socket event is received from its reactor. The process crash fault detection time is the difference between the end time and the start time. On average, the process crash fault detection time is about 3 ms.

Next, we measured the recovery time for adding a new replica into an existing group. To obtain a clear picture of the cost of recovery at various stages, we measured the recovery time using a number of metrics:

Application. The time interval between the start of the application replica as indicated by the start of the `main()` function and the completion of the `set_state()` method invocation. This time is closest to the time perceived by the end user (only the time spent in loading the application is not counted). This time includes the cost of initializing the ORB and the application.

Totem. The time interval between the detection of the join of a new replica and the completion of writing the `SET_STATE` protocol message to the replica (the reply of the `set_state()` invocation is not written to Totem). This metric best reflects the actual cost of the Pluggable FT Infrastructure for application recovery. It does not include the cost of initializing the ORB or the application.

`SET_STATE` **Transmission Time.** The time interval between the receiving of the `GET_STATE` protocol message and the completion of the `set_state()` method invocation on the application object, measured at the application process. Assuming that the existing replicas and the newly started replica receives the `GET_STATE` protocol message almost at the same time, this time approximates the cost of retrieving and setting the application-level state, the ORB-level state and the infrastructure-level state, and the transmission and delivery of the `SET_STATE` message. In our experiments, the cost of retrieving and setting the different levels of state is negligible; therefore, the cost of the `SET_STATE` message transmission dominates.

The results of the recovery time measurements are shown in Figure 7. As expected, the recovery time using the first metric (application) is significantly larger than the time measured for the other two metrics (Totem and `SET_STATE` Transmission Time). The recovery time for the second and third metrics show a nice linear dependency on the application-state size. For an application with a state as large as 100KByte (the ORB-level and infrastructure-level state is negligible compared with an application state of this size), the recovery time is about 100ms, 40ms and 10ms, for the three metrics respectively.

The time needed to recover from the failure of the primary replica for the semi-active replication style is also measured. To avoid the need for clock synchronization, both the primary replica and a secondary replica (to be promoted to primary) are run on the same processor. We send a signal to the primary replica. The primary replica then records a time and exits. The loss of the primary replica is detected and the secondary replica is promoted to be the new primary replica. The time at which the secondary replica is promoted is then recorded. The difference between the two recorded times is regarded as the recovery

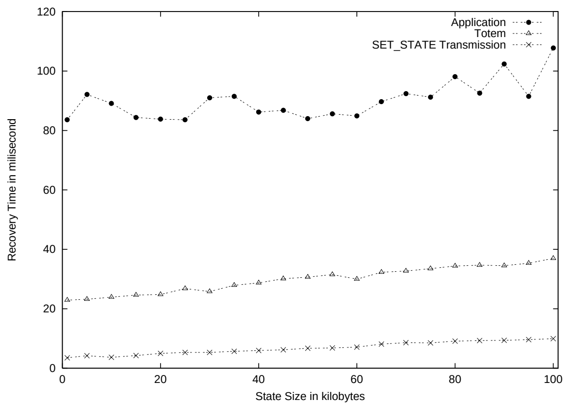

Figure 7. Recovery time as a function of state size using different metrics.

time from the primary replica fault. This recovery time increases with the replication degree and the Totem ring size. In the case of four processors, the recovery time is approximately 10ms.

6 Related Work

Three Fault Tolerant CORBA systems that implement all or part of the FT CORBA standard are DOORS [12], Eternal [9, 10, 11] and IRL [7]. All three systems use a non-intrusive approach. On top of an interception-based fault tolerance infrastructure, Eternal implements the Replication Manager, the Fault Detector and the Fault Notifier as CORBA objects to provide fault tolerance services to application objects. By using CORBA's portable interceptors, IRL embeds a set of CORBA objects that implement mechanisms for transparent reinvocation and redirection, and for duplicate detection and suppression, into the application address space.

Although the Pluggable FT CORBA Infrastructure described in this paper bears some similarities to the interception approach used by Eternal, in that messages are routed through a group communication system, it is more similar to the integration approach because the fault tolerance mechanisms are incorporated into the ORB. Of course, some management components (namely, the Replication Manager and the Fault Notifier) are implemented as CORBA services, which the FT CORBA standard requires.

Embedding the fault tolerance mechanisms into the ORB facilitates the transfer of ORB-level state, as well as application-level state, in an accurate and efficient manner. This approach also provides the opportunity for improved quality of service for the applications by allowing different protocols to be plugged into the ORB. To retrieve and assign the ORB-level state properly, the interception-based and the service-based FT CORBA systems involve proprietary interfaces to be supplied by the ORB vendors.

Nevertheless, the Pluggable FT CORBA Infrastructure differs from the integration-based systems in several respects:

- The Pluggable FT CORBA Infrastructure enters the ORB address space by using the PPF provided by the ORB, with no modification to the ORB core. The integration-based systems require extensive modifications to the ORB to accommodate the fault tolerance mechanisms.

- Because the PPFs offered by different CORBA ORBs follow similar object-oriented design patterns, the Pluggable FT CORBA Infrastructure can be easily ported from one ORB to another, whereas integration-based systems must be tightly integrated into a particular ORB.

- The Pluggable FT CORBA Infrastructure requires only minimal modification to the application program for loading the FT protocol plug-in if the PPF (*e.g.*, ORBacus 4.0 or e*ORB) allows the user to select the particular protocol by invoking an API within `main()`, or no modification at all (other than those required by the FT CORBA specification) if the PPF (*e.g.*, ORBacus 4.1) allows the ORB to load the particular protocol at runtime. Like the interception- and service-based approaches, the Pluggable FT CORBA Infrastructure provides fault tolerance to applications in a transparent manner, whereas for integration-based systems the applications program fault tolerance directly using their own interfaces.

A performance comparison between the Pluggable FT CORBA Infrastructure and other Fault Tolerant CORBA systems is difficult because of platform differences for the measurements and/or the lack of data for those systems. Because a well-engineered group communication system, such as Totem, has better performance than multiple point-to-point TCP/IP communications for the same group size, the Pluggable FT CORBA Infrastructure outperforms existing service-based Fault Tolerant CORBA systems in terms of run-time overhead. Moreover, the Pluggable FT CORBA Infrastructure, based on Totem, achieves performance that is similar to, or better than, that of the interception-based Eternal system [19] for active replication.

Delta-4 [16] introduced semi-active replication, which has since been used in several other systems [1, 3]. These systems differ from the Pluggable FT CORBA Infrastructure in that they use semi-active replication for handling non-determinism, rather than for performance considerations. By using semi-active replication, the Pluggable FT

CORBA Infrastructure and other fault-tolerant systems that are based on group communication can achieve better performance than if they use active replication, because duplicate messages that might otherwise appear on the network are eliminated. Semi-active replication also provides faster recovery from faults than passive replication.

7 Conclusion

We have described the design, implementation and performance of the Pluggable FT CORBA Infrastructure, a fault-tolerant CORBA system based on the pluggable protocols framework. Applications can achieve higher reliability and availability by plugging in the FT protocol with minimal modifications to their existing code. The Pluggable FT CORBA Infrastructure has the advantages of both the intrusive (integration-based) and non-intrusive (serviced-based and interception-based) approaches, while overcoming the drawbacks of each. It achieves performance that is similar to, or better than, that of existing FT CORBA systems, while providing strong replica consistency and supporting a wider range of applications.

References

[1] S. Bestaoui, A. M. Deplanche, and Y. Trinquet. Redundancy management in the SCEPTRE2 real-time executive. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 373–378, Le Touquet, France, October 1993.

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

[3] A. M. Deplanche, P. Y. Theaudiere, and Y. Trinquet. Implementing a semi-active replication strategy in Chorus/ClassiX, a distributed real-time executive. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 90–101, Lausanne, Switzerland, October 1999.

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

[5] 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. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks*, pages 81–98, Salem, MA, August 1999.

[6] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

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

[8] 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*, 39(4):54–63, April 1996.

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

[10] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

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

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

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

[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-Oriented Concepts, Inc. *ORBacus for C++ and Java*, 1998.

[16] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1. Springer-Verlag, 1991.

[17] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture, volume 2*. John Wiley & Sons, Ltd, 2000.

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

[19] W. Zhao, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith. Experimental evaluation of a fault-tolerant CORBA system. In *Proccedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 390–396, Las Vegas, NV, June 2001.