Report Number: WUCSE-2005-59

2005-11-01

Design and Performance of a Fault-Tolerant Real-Time CORBA Event Service

Huang-Ming Huang and Christopher Gill

Developing distributed real-time and embedded (DRE)systems in which multiple quality-of-service (QoS) dimen-sions must be managed is an important and challenging R&D problem. This paper makes three contributions to re-search on multi-dimensional QoS for DRE systems. First, itdescribes the design and implementation of a fault-tolerantreal-time CORBA event service for The ACE ORB (TAO).Second, it describes our enhancements and extensions tofeatures in TAO, to integrate real-time and fault toleranceproperties. Third, it presents an empirical evaluation ofour approach. Our results show that with some refinements,real-time and fault-tolerance features can be integrated ef-fectively and efficiently in a CORBA event service.
... Read complete abstract on page 2.

Design and Performance of a Fault-Tolerant Real-Time CORBA Event Service

Huang-Ming Huang and Christopher Gill

Complete Abstract:

Developing distributed real-time and embedded (DRE)systems in which multiple quality-of-service (QoS) dimen-sions must be managed is an important and challenging R&D problem. This paper makes three contributions to re-search on multi-dimensional QoS for DRE systems. First, itdescribes the design and implementation of a fault-tolerantreal-time CORBA event service for The ACE ORB (TAO).Second, it describes our enhancements and extensions tofeatures in TAO, to integrate real-time and fault toleranceproperties. Third, it presents an empirical evaluation ofour approach. Our results show that with some refinements,real-time and fault-tolerance features can be integrated ef-fectively and efficiently in a CORBA event service.

2005-59

Design and Performance of a Fault-Tolerant Real-Time CORBA Event Service

Authors: Huang-Ming Huang, Christopher Gill

Corresponding Author: cdgill@cse.wustl.edu

Abstract: Developing distributed real-time and embedded (DRE) systems in which multiple quality-of-service (QoS) dimensions must be managed is an important and challenging R&D problem. This paper makes three contributions to research on multi-dimensional QoS for DRE systems. First, it describes the design and implementation of a fault-tolerant real-time CORBA event service for The ACE ORB (TAO). Second, it describes our enhancements and extensions to features in TAO, to integrate real-time and fault tolerance properties. Third, it presents an empirical evaluation of our approach. Our results show that with some refinements, real-time and fault-tolerance features can be integrated effectively and efficiently in a CORBA event service.

Notes:
This research was supported in part by the DARPA PCES program.

Type of Report: Other

Design and Performance of a Fault-Tolerant Real-Time CORBA Event Service *

Huang-Ming Huang and Christopher Gill
{hh1,cdgill}@cse.wustl.edu
Department of Computer Science and Engineering
Washington University, St.Louis, MO, USA

Abstract

Developing distributed real-time and embedded (DRE) systems in which multiple quality-of-service (QoS) dimensions must be managed is an important and challenging R&D problem. This paper makes three contributions to research on multi-dimensional QoS for DRE systems. First, it describes the design and implementation of a fault-tolerant real-time CORBA event service for The ACE ORB (TAO). Second, it describes our enhancements and extensions to features in TAO, to integrate real-time and fault tolerance properties. Third, it presents an empirical evaluation of our approach. Our results show that with some refinements, real-time and fault-tolerance features can be integrated effectively and efficiently in a CORBA event service.

1. Introduction

Recent research efforts have extended middleware that implements the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [9] standard, to support distributed real-time and embedded (DRE) system applications such as avionics mission computing [11], distributed interactive simulation [21], and computer-aided stock trading [3]. A common goal of these efforts is to examine how the specific requirements of each DRE system shape the middleware itself. Many DRE systems have the following common requirements.

Distributed processing. DRE system components are deployed across multiple endsystems. It is necessary for a component to be able to invoke operations on other components regardless of their locations.

Timeliness and real-time predictability. Many DRE systems have stringent timing constraints with severe consequences if the specified deadlines cannot be met.

High reliability. Applications like avionics computing systems may require a very high degree of reliability even in the face of faults. Failures in some critical components, though ultimately unavoidable, must not be allowed to compromise the overall reliability of the system.

The CORBA standard addresses the issue of distributed processing by providing a method invocation model, where a client invokes an operation on a target object that may reside locally or on a remote server. This model, however, may be too restrictive because of the tight coupling between client and server lifetimes it assumes.

A CORBA event service provides support for decoupled communication between objects. Instead of using point-to-point communication, interested event consumers subscribe for the types of events they need from the event service. Event suppliers push events to the event service instead of directly to the consumers. The event service is responsible for managing how to dispatch the events. This approach reduces coupling between suppliers and consumers, but it poses the following new challenges. First, the event service becomes a mediator for all events and thus might become a bottleneck for event delivery. Therefore, how to ensure end-to-end timeliness is a concern. Second, the event service itself becomes a potential single point of failure. Therefore, how to provide fault-tolerance for the event path from suppliers to consumers is also a concern.

Hence, how to integrate fault-tolerance and real-time abilities in a CORBA event service is an important research problem. Fault-tolerance can be achieved by providing redundancy. Real-time support requires elimination of delays to meet timing constraints. It is therefore necessary to determine how to trade off fault-tolerance and real-time properties carefully, which is the research problem this paper addresses. We focus on an application domain that has been important to the DRE systems R&D community over the past decade, notably systems that use event services to mediate communication and/or concurrency among local and remote software objects. Accordingly, we focus our efforts on policies and mechanisms to achieve both real-time predictability and fault-tolerance within an open-source event service built on top of an open-source real-time CORBA object request broker, The ACE ORB (TAO) [14].

*This research was supported in part by DARPA contracts F33651-01-C-1847 and F33651-03-C-4111 (PCES).

In this paper we describe the design, implementation and performance of a fault-tolerant real-time event service (FTRTES) and compare its performance to that of TAO's real-time event service (RTES) upon which it is based. In this research we have focused on the robustness of event service subscriptions, so that if an event service crashes the event delivery paths between event suppliers and event consumers are still preserved, and after a crash events can still be delivered. Furthermore, our solution approach offers configuration options for trading off the latency of supplier/consumer subscriptions for the number of channel crashes that are assured to be tolerated.

The rest of this paper is organized as follows. Section 2 gives an overview of TAO's RTES and FT-CORBA features, which we use and extend in our FTRTES implementation. Section 3 describes key challenges and our solution approach for designing the FTRTES. Section 4 describes experiments we conducted to evaluate our FTRTES implementation. Section 5 describes related work and Section 6 offers concluding remarks.

2. Overview of TAO's Real-Time Event Service and Fault-Tolerant CORBA Features

The OMG Event Service is a standard CORBA Service that allows applications to use a decoupled communication model instead of direct client-to-server method invocations [13]. In the OMG Event Service model, *suppliers* produce events and *consumers* receive events. Before sending or receiving events, both suppliers and consumers have to connect to an *event channel* which is responsible for event delivery. In this paper, we refer to the connection establishment operation as an event *subscription*.

The OMG Event Service provides two models for event delivery: *push* and *pull*. In the push model, suppliers send events to the event channel and the event channel sends them to the consumers. In the pull model, the event channel polls the suppliers to obtain events,and the consumers then poll the event channel. The Event Service also supports hybrid push/pull models which allow the suppliers to push events and consumers to pull events or the event channel to pull events from suppliers and push them to consumers. The TAO Real-Time Event Service is an extension to OMG Event Service that provides real-time capabilities. It supports a push event delivery model with the following features that are not part of the standard OMG Event Service.

Event scheduling. The event channel subscriptions can supply different QoS parameters so that event delivery can be scheduled with fixed priority, earliest deadline first, least laxity first or maximum urgency first strategies [6].

Event filtering/correlation. Events can be filtered or correlated with other events by type or identifier.

Timer events. TAO's Real-Time Event Service can be configured to push timer events at specified rates.

Fault-Tolerant CORBA (FT-CORBA) [9] is a specification developed by the OMG to provide fault-tolerant infrastructure to CORBA systems. This infrastructure enables CORBA applications to control the creation of object replicas and supports different fault-tolerance strategies for data consistency between replicas. These strategies include request retry, redirection to different server objects, passive replication to minimize transmission overhead and active replication for faster response times. It also provides interfaces for fault detection, notification, and analysis.

The FT-CORBA specification is designed to give applications a high level of reliability. This reliability is achieved through entity redundancy, fault detection and recovery. Entity redundancy is provided by replication of objects. Several replicas of an object, which inhabit different processes or even different hosts, are managed as an *object group*. Clients treat the object group as a logical single object. The requests made by clients are routed transparently by the fault-tolerance infrastructure to members of the group.

In a CORBA system, an object is referenced by an Interoperable Object Reference (IOR). The IOR contains the object key as well as host information such as the address and port to which to connect. An Inter-operable Object Group Reference (IOGR) extends the IOR structure by allowing several profiles within an IOGR. Each profile contains a distinct object key and host information. Depending on replication styles, a client can communicate with the hosts in only one profile or in all profiles at a time.

To maintain state consistency between replicas in an object group, FT-CORBA defines three different replication styles. For *cold passive* and *warm passive* replication styles, only a single member, referred to as the *primary* member, executes the operation that has been invoked on the object group. If the system suspects the primary member has failed, a backup member is selected to become the primary member. In the cold passive style, a logging mechanism periodically invokes the get_state() operation, which must be implemented by every replicated object, to obtain the state of the object so that the state can be recorded. During recovery, a recovery mechanism invokes the set_state() operation of the new primary to synchronize its state with the recorded state. In the warm passive style, backup members periodically synchronize their states with the primary.

In the *active* replication style, the request issued by a client is multicast to all members of the object group and each replica executes the requested operation independently. The FT-CORBA ORB has to maintain a total order over the messages which arrive at all replicas and suppress the repeated replies to the client. Thus clients suffer limited delay for recovery during fail-over, but do so at a cost of greater message ordering overhead.

In addition to state consistency, the warm passive and active replication styles must maintain membership consistency. FT-CORBA specifies ReplicationManagers to control the membership of object groups as well as *fault detectors* to detect faults and generate and send fault reports to ReplicationManagers. There are again two models for fault monitoring: *push* and *pull*. For the pull model, the fault detector periodically interrogates the liveness of each monitored object. For the push model, the monitored objects report to the fault detector to indicate that they are alive.

3 Design and Implementation

Figure 1 shows our architecture for integrating fault-tolerance and real-time properties in event-mediated DRE systems, including: event suppliers and consumers that use a Fault-Tolerant Real-Time Event Service (FTRTES); a naming service where CORBA Interoperable Object References (IORs) can be registered, stored, retrieved; and primary and backup instances of a replicated Fault-Tolerant Real-Time Event Channel (FTRTEC) that implements the FTRTES. We now describe the challenges we faced in developing the FTRTES, and the solution approaches we used to address those challenges.

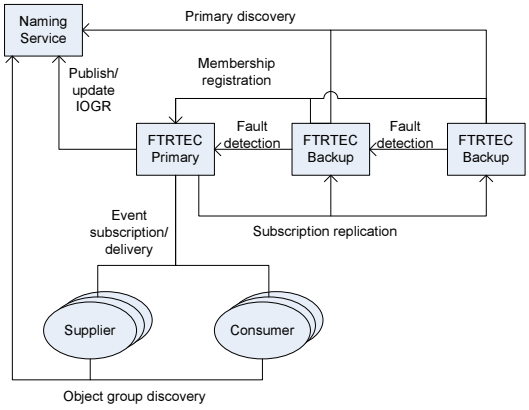

Figure 1. FTRTES Architecture

3.1. Replicate subscription state only

Context: There are two major kinds of operations in an event service: event subscription and event transmission. Subscription operations like connect_push_consumer and connect_push_supplier are used for registering a consumer or supplier with an event service to send or receive certain types of events, and setting up constraints to correlate or filter events by type or identifier. Event transmission operations like push are only used to transfer events from suppliers to the event service and from the event service to consumers.

Problem: Event subscription and event transmission have different timing and ordering constraints in an event service. Event transmission in DRE applications usually requires predictable low latency and high throughput. Their effect on the state of event service is also ephemeral as events enter and leave the event channel. Replication of events to event channel replicas can impose significant overhead and jitter for event delivery latency, and a better approach would be to replicate events at a lower (*e.g.*, SCTP [24]) architectural level.

Solution: Since the state change due to event delivery is ephemeral and expensive to maintain, we only replicate the subscription operations. The subscription information occurs at a more suitable time scale for replication and is in fact more essential for the delivery of events because it establishes the connectivity from suppliers to consumers. The loss of subscription state can affect the correctness of entire event delivery paths, while a limited number of lost events may be acceptable in many applications. As a consequence, we only replicate subscription operations, not events, at the middleware level.

3.2. Use semi-active replication

Context: Fault tolerance in FT-CORBA is achieved through entity redundancy and the replication of state or operations between replicas. As we described in Section 2, FT-CORBA specifies three different replication styles for providing state consistency: *cold passive*, *warm passive* and ,active. For both cold passive and warm passive, state consistency is only required at the time when a backup object takes over for a failed primary object. In active replication, the backup objects keep their states consistent with the primary at the end of each client invocation.

Problem: Both the cold passive and warm passive approaches suffer from long and unpredictable recovery times which are not suitable for DRE systems. In cold passive replication, a new primary has to replay every subscription operation performed since system initialization. In warm passive replication, the situation may be better because the new primary only has to replay subscription operations performed since the last time it synchronized with the failed primary. However, the time is still highly unpredictable. Although active replication has an assured recovery time after fail-over, it requires a multicast protocol to provide totally-ordered reliable message delivery. This kind of protocol usually require a lot of transport level communication and can significantly reduce system throughput, especially over low-bandwidth and/or high latency connections.

Solution: We adopt the semi-active replication style [7]. The object group is arranged as a linked list in which each member in the group maintains a link to its predecessor and/or successor. The head of the linked list is the primary

member of the object group. All clients (event consumers and suppliers) only interact with the primary. Moreover, only the primary can send a reply to a client. If the primary crashes, the successor of the crashed primary will be promoted to become the new primary. The linked list is maintained via the connection liveness of the transport layer (e.g., using TCP or SCTP based pluggable ORB protocols). Each replica establishes a transport connection to its predecessor and reports to a set of ReplicationManagers when and if the connection is down. At the end of each client invocation, the primary member synchronizes its state with the replicas using a reliable multicast protocol. The references in the linked list are packed into an IOGR that is passed to clients. A client must honor the order of that list when it retries a method invocation. No heart beats or poll messages are used to detect the failure of nodes (except perhaps internally within the transport layer, e.g., SCTP). Fault detection in the FTRTES only relies on the liveness of the transport connection, and no extra threads are required.

As described by Gokhale, *et al.* [7], the semi-active replication style offers three advantages over the other approaches. Compared to cold passive and warm passive replication, semi-active replication is more predicable and has a faster recovery time. Since there are no middleware-level heart beats or poll messages for fault detection, it reduces message transmission in the transport layer while maintaining a reasonable fault detection time. When used in conjunction with the prioritized method invocation semantics defined in RT-CORBA, it does not require a potentially expensive prioritized multicast protocol.

3.3. Customize state update strategies

Context: We consider two kinds of state update: *entire* and *incremental*. For entire state update, the primary sends the entire state to the replicas each time and the replicas replace their states with the information they received. For incremental state update, the primary only sends the state differences (or requests for the operations that need to be executed) since the last state update, and the replicas then update their states accordingly. Entire state update is more suitable for the case that the state does not grow in size or vary at fine granularity with time; otherwise, it is more suitable to use incremental state update. Our approach in the FTRTES was to use incremental state update because the subscription state may vary with time.

Problem: Without special design, incremental state update can suffer from state inconsistency problems if the primary crashes in the middle of replication. Suppose there are 3 members A, B and C in a object group where A is the primary member of the group. When A tries to replicate a state update to B and C, it crashes after B succeed in the replication but C did not. At the time B is promoted to be the new primary, B and C may not be in the same state.

Solution: Each incremental state update carries a sequence number which is used to detect missing state updates. The sequence number is incremented each time the primary sends a new update. If a replica receives an update with a non-contiguous number, it can request the missing incremental update(s) or an entire state update from the primary.

3.4. Allow reliability/timeliness trade-offs

Context: Semi-active replication uses reliable multicast to synchronize the state between primary and replicas.

Problem: In the semi-active replication style, the primary has to multicast its own state to all other members of the object group. However, IP-multicast is not applicable because it is neither reliable nor totally ordered. One intuitive way to implement this is to use two-way CORBA calls to transmit the state change from the primary to all backups. However, using this approach, the client waiting time will be proportional to the number of replicas.

Solution: To improve timeliness for fault-free operations as well as for fault recovery stages, we used the concept of *transaction depth*. If the transaction depth is n, a subscription method invocation has to be blocked until the first n replicas complete the operation, which is called assured-replication. Other replicas can get the state change via a so-called soft-replication which means the replication is not assured to complete before the request invocation returns to the client. So, if a soft-replication fails due to a crash of the primary or a replica, we are guaranteed only the assured depth of replication. Here as well the use of a replication sequence number can allow a recovery from an inconsistent soft-replicated state, but at a cost of a longer recovery time. FTRTES clients are allowed to specify the transaction depth using the service context mechanism in CORBA portable interceptor to trade off reliability and timeliness. If the transaction depth can not be met, the replicate operation has to be rolled back and the primary then throws an exception back to the client.

In the CORBA standard, there are 3 different kinds of method invocations : one-way, two-way, and asynchronous method invocation (AMI). For two-way method invocations, the clients will be blocked until servers finish execution and return the results back to the clients. For one-way method invocations, clients won't be blocked for the completion of the operations' execution on the server side, but return values and exceptions from the servers are not supported so no indication of the method invocation's success or failure is given to the client. AMI, on the other hand, allows clients to proceed without blocking on the method invocation but provides the capability to return results (e.g., via a callback object).

Our FTRTES solution supports two approaches for send-

ing replication messages. The first one uses two-way method invocations for assured-replicate operations and one-way method invocations for soft-replicate operations. When the primary member of the object group receives a subscription request from a client, it retrieves the transaction depth from the service context in the message. If the transaction depth is greater than 1, the primary will use two-way method invocation to replicate the request to its successor; otherwise, it uses one-way method invocation to replicate the operation. In the former case, each member will pass along a transaction depth that is one less than it received.

Our second approach for message replication is to use AMI for both assured and soft replication. In this case, the primary sends replication messages using AMI to all other members in the object group once it receives a subscription request. The primary waits for replies from the first n (equal to the transaction depth specified by the client) replicas before it sends a reply back to the client. The AMI replication strategy allows parallel replication operations in different replicas without sacrificing reliability. However, using AMI introduces some additional programming complexity to handle results that are returned asynchronously.

3.5. Collocate replication managers

Context: In FT-CORBA, ReplicationManagers are responsible for the management of the object group.
Problem: ReplicationManagers need to be replicated, to avoid a single point of failure. In turn, the ReplicationManagers form another object group which needs to be managed by some other entity. This poses a potential problem of recursive replication dependency.
Solution: To avoid the recursive replication dependency problem, we collocate ReplicationManagers with the replicated FTRTES objects. The primary of the ReplicationManager object group is also the primary of the replicated object group. Under the semi-active replication style, if the successor of the primary detects the failure of the primary, it directly promotes itself to become the primary for both the event service and the ReplicationManager groups, and the new primary will register a new IOGR with the naming service. This solves both the problems of a single point of failure and of recursive replication dependency.

3.6. Support priority-banded operations

Context: In our FTRTES solution, event push operations are not replicated and require higher throughput. On the other hand, subscription requests need to be replicated, requiring more network bandwidth and taking more time to process. To maintain a consistent view of the object group, the group management operations in the ReplicationManager also involve extensive communication between the primary and replicas and have non-trivial latency. Both subscription and group management operations are I/O bound, and their latency comes largely from waiting for the responses from other hosts.
Problem: As the context indicates, different kinds of operations have different characteristics. In the case of the FTRTES, the time to process the subscription and group management operations is longer than that of event pushes. If the FTRTES handles all operations in a reactive fashion, the subscription operations and group management operations can impede the throughput of event push operations.

One solution is to use a thread pool and the leader follower pattern [23] which allows a bounded number of threads to handle requests simultaneously. Thus if one thread has been waiting for the response from other members of the group, there is still another thread available to handle event pushes. However, this approach can lead to the following problems. First, the number of executing operations is bounded by the number of threads. If subscription operations have occupied all the available threads, no thread will be able to process event push operations until a subscription operation completes. Second, although increasing the number of threads will decrease the possibility that an event push operation can be stalled, that can increase system overhead due to extra context switching.
Solution: For the FTRTES, we consider the event push operation to be more urgent than the subscription and group management operations. Therefore, we assign higher priority to the push operation than to the other operations. In addition, we adopt the endpoint-per-priority model [22] in TAO where the server ORB uses multiple transport endpoints to accept connections from clients. Each transport endpoint has a priority, which is the priority of the thread(s) servicing the endpoint as well as of all the connections it accepts. When a server ORB creates an IOR for one of its objects, it embeds all of the server's acceptor [23] endpoints along with their priorities into the object's IOR. Then, a client ORB selects the priority that best matches the client's need (as specified by the Client Priority Policy) from those offered by the server, and uses the corresponding transport endpoint specified by the server to obtain the desired priority level.

Our FTRTES solution extends the FT-CORBA IOGR to incorporate the endpoint-per-priority model. Each IOGR contains several profiles which represent the primary and replicas. Each profile contains endpoints with specific priorities. When a client fails to communicate with the server using an endpoint in the active profile, the client ORB switches to using the endpoint given in the next profile. The endpoint-per-priority model offers the following benefit in our FTRTES design. It reduces delays to the push operation because push operations have a dedicated thread that that

runs at a higher priority than the thread(s) in which subscription operations run. Only two threads are strictly necessary to handle clients requests: one for push operations and one for the others.

3.7. Piggyback IOGR update onto reply

Context: Upon changes of membership in a object group, the clients' IOGRs must be updated. FT-CORBA defines a GROUP_VERSION service context that a client can send to the server along with requests. This include a version number that allows the server to check whether the IOGR used by the client is up to date. If the IOGR is obsolete, the server then sends a LOCATE_FORWARD_PERM exception to the client ORB with the new IOGR. After the client ORB updates its IOGR, it re-sends the request with the new service context.
Problem: When using semi-active replication, the request has to take one extra round trip even if the primary of the object group remains the same after the IOGR has been updated. If the membership of the object group has changed and the primary has also changed, it is necessary to redirect the client request to the new primary because only the primary can execute the request and send the reply. However, if the primary does not change, it is wasteful to require the client to re-send the request with the new GROUP_VERSION.
Solution: We use a service context piggybacked on a reply to update the IOGR when applicable. In our FTRTES implementation, when a primary receives a request with an obsolete GROUP_VERSION, it still processes the request and sends a reply. However, the reply contains another service context with the latest IOGR. This allows the client to update the IOGR without extra delay. If a non-primary replica receives a request, it still sends a LOCATE_FORWARD_PERM exception back to the client.

3.8. Present a common façade interface

Context: The CORBA event service specification includes ConsumerAdmin, SupplierAdmin, ProxyPushConsumer, and ProxyPushSupplier interfaces. The separation of the interfaces allows freedom to deploy different event service components on different hosts.
Problem: Multiple event service interfaces create extra overhead for IOGR management on the client, and are unnecessary because each replicated FTRTES object is contained within one host. To the client, each interface is represented by a different IOGR with no relationship assumed between different IOGRs. For example, if a client publishes two kinds of events and establishes two different logical connections with an event service, it gets two distinct IOGRs (a and b) to ProxyPushConsumers. Suppose that the primary crashes and the client ORB detects the failure because it fails to establish a transport connection with the primary profile in IOGR a. The client ORB can redirect the request to the host in the next profile of the IOGR a and the update the IOGR when it get the reply. When the client needs to push an event through IOGR b, the client ORB has to repeat the same procedure, which results in unnecessary delay.
Solution: Our FTRTES implementation uses the Façade pattern [5] to solve this problem. We create a single interface that combines all operations from the various interfaces of the event service. For operations that return object references in the original CosEvent model, opaque object handles are returned instead. All the invocations on the original object reference are replaced by invocations on the façade interface, with an object handle as a parameter. Therefore, the change of membership in an object group only needs to update one instead of many IOGRs on each client. This saves communication and overhead for extra IOGR updates.

3.9. Provide a client-side adapter

Context: Although the Façade pattern can remove the cost of updating multiple IOGRs and reduce unnecessary communication when object group membership changes, it changes the interface between the event service and the client. This may require source code modifications to the clients using the event service.
Problem: The use of a façade interface breaks backward compatibility of legacy applications using the event service.
Solution: To solve this problem, we use the Adapter pattern [5] to avoid the interface incompatibility problem introduced by the Façade pattern. For applications that require backward compatibility, we provide an object for adapting calls to the original TAO RTES interfaces into the new FTRTES interface given by the Façade pattern. For client applications written in C++ with source code, the adapter can be linked directly to the application with only minor source code modification and high run-time efficiency. To take advantage of the features provided by FT-CORBA, all the requests sent by clients should contain the service contexts defined in the specification. For client applications written for an ORB that is not FT-CORBA compliant, the adapter can be compiled into a binary executable and deployed in the same host with client. Client applications then interact with the adapter instead of the event service directly, and the adapter can then convert the request into FT-CORBA compliant messages. This allows the client applications to make immediate use of the FT-CORBA feature without source code modification. Using an adapter object on the client side also allows us to combine several stages of subscription operations into one. For example, in the RTES, the supplier subscription requires 3 CORBA method invo-

cations : for_suppliers(), obtain_proxy_consumer() and connect_proxy_supplier(). Our new interface for the FTRTES provides one operation, connect_proxy_supplier(), which combines the functionality of the other 3 operations and thus reduces the latency for subscription operations.

4. Empirical Evaluation

This section compares the performance of our Fault-Tolerant Real-Time Event Service described in Section 3 with the TAO Real-Time Event Service. We also examine the effect of node failures on the throughput of event push and subscription operations. The testbed we used to conduct our experiments consisted of 2 Pentium-IV 2.5 GHz machines and 2 Pentium-IV 2.8 GHz machines, each with 512MB RAM and 512KB cache and running KURT-Linux 2.4.18, connected by a 100 Base-T Ethernet isolated network. Our experiments used ACE/TAO version 5.4.5 / 1.4.5, and ran as root in the real-time scheduling class.

Our experiments assumed a single-failure fail-stop fault model with no nested failures. The methodology we adopted for each experiment, and our experimental results and analysis, are presented in the following subsections.

4.1. RT event latency with/without FT

We first describe benchmarks we conducted to measure end to end event latency in both our FTRTES implementation and the TAO RTES on which our implementation is based. The goal of these experiments was to quantify the additional overhead of the fault-tolerance features we added. Both event consumers and suppliers were located in one 2.8 GHz machine and the FTRTES or RTES was located on the other 2.8 GHz machine. We configured the FTRTES with between 0 and 3 backup replicas in addition to the primary. The measured latencies of event push operation are summarized in Figure 2. The standard deviations for all these cases were between 10.88 μsec and 13.12 μsec.

From Figure 2, we can see that the average latency was about 80 μsec higher, and the maximum latency was about 140 μsec higher (with 3 backup replicas), with the FTRTES than with the RTES. This additional latency stemmed from the extra service contexts attached to every message with the FTRTES. All FTRTES clients were required to inject these service contexts and the FTRTES was required to interpret them. These service contexts included the FT_GROUP_VERSION we discussed in Section 3 and the FT_REQUEST service context defined in FT-CORBA, which contained three fields: client_id, retention_id and expiration_time. These fields had two purposes in our experiments: the server could use the client_id and retention_id to

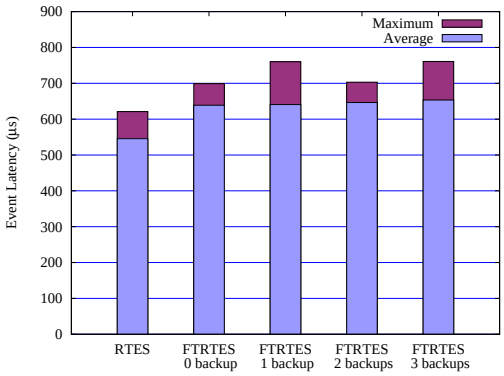

Figure 2. FTRTES/RTES Latency Comparison

detect duplicate requests in order to ensure at-most-one request delivery semantics, and the expiration_time field was used for evaluating the liveness of a request.

4.2 Effects of transaction depth

In this experiment, we configured the system with a primary on a 2.8 GHz machine, one replica and the event consumer and supplier on a 2.5 GHz machine, a second replica on the other 2.8 GHz machine, and a third replica on the other 2.5 GHz machine. We varied the transaction depth from one to four for both two-way/one-way and AMI replication. Since we do not replicate event state, we only measured the latency of subscription operations. The results are shown in Figure 3.

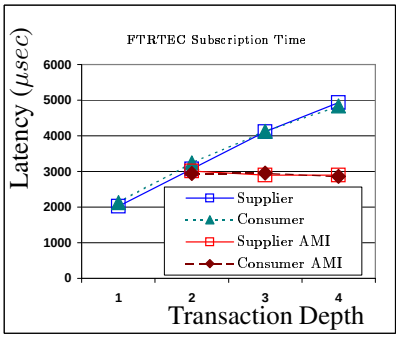

Figure 3. Subscription Time Scalability

As may be expected, for two-way/one-way call replication, the subscription latency grew linearly with the transaction depth. This was due to fact that the replication operation was serialized among the replicas: the state was not replicated to the next replica until the previous replica finished the operation. Soft replication then traded off reliability for response time by allowing replication to continue to other replicas without waiting for previous ones to finish.

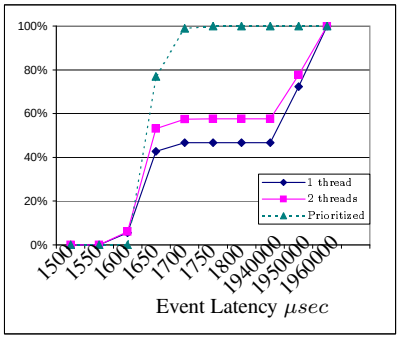

Figure 4. Cumulative Distribution (Fail-Over)

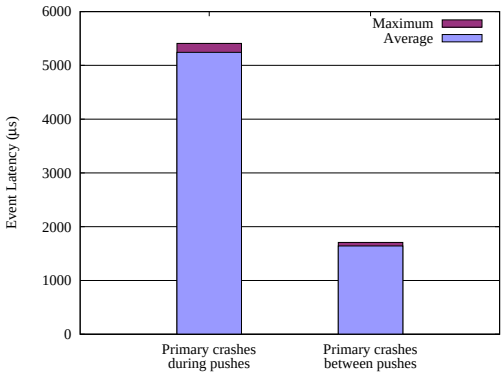

Figure 5. Event Latency and Failure Timing

With AMI replication, the subscription latency remained essentially constant as replicas after the first one were added, because the replicas perform the replication operations in parallel without waiting for the other replicas. Only the primary waits, until as many of the replicas finish as are specified by the transaction depth.

4.3 Event latency during fail-over

The experiments in this subsection examined the event push latency under fail-over conditions. Our experimental setup was the same as in Section 4.2 and the supplier sent events at 10 Hz frequency. We measured the latency of each event passing from the point it was sent by the event supplier until it reached the event consumer.

There are several factors that can affect performance during fail-over. The first factor to consider is the interference of group management operations with event push operations as we discussed in section 3. When the primary crashes, the backup will start to re-organize the object group to maintain group integrity which can delay event dispatching in the new primary if the event dispatching operation is not prioritized. We crashed the primary 50 msec after a certain number of events was handled and compared the cases where the server ORB used one versus two unprioritized threads to handle requests, as well as when event push operations are given higher priority than other operations. Figure 4 shows our results. All events were delivered between 1600 μsec and 1800 μsec when push operations were prioritized. In contrast, more than 50% (or 30%) of the events were delivered after 1.94 seconds when the server ORB used one (or two) unprioritized thread(s) to handle requests. Thus, prioritization with at least 2 threads greatly improved the predictability of event delivery.

The second factor to consider is the timing of when the fault occurs. If the primary crashes after it receives an event but before it replies to the event suppliers, the supplier has to wait until it detects the failure of the transport connection and re-route the event to the new primary. If the primary crashes when it is not dispatching any events, the supplier can save the time needed to re-route the event. Figure 5 shows the effect of the timing of the primary failure. The first column in Figure 5 summarizes the event latencies over 1000 samples during fail-over when the primary crashes in the middle of an event push. The average value in this case was 5242 μsec and the maximum value was 5408 μsec. The second column summarizes the the latencies when the primary crashed in between event push operations, which had an average value of 1642 μsec and a maximum value of 1707 μsec.

The third factor that affects event push performance under fail-over conditions is dynamic memory allocation along the event dispatching path. General purpose heap memory allocators can usually optimize memory allocation requests that show repeated patterns; therefore, the first memory allocation iteration of the pattern will take significantly more time than the rest. Under a fail-over situation, the first event that arrives at the newly elected primary may exhibit a longer processing time due to the memory allocation inside the ORB and its Portable Object Adapter, e.g., for decoding or encoding GIOP messages.

The fourth factor that can affect event push operation performance is the time for transport connection establishment. During an event push, the supplier Client ORB will check the availability of the connection to the primary. If the connection has been disconnected, the client ORB will try to re-connect to the primary and wait until it times out. After that, the client will connect to the first replica in the IOGR list and send the event if the connection can be established. In this case, the event could be delayed by waiting for a connection establishment time-out and a connection establishment time.

We examined the effect of each of these factors on fail-over event latency by mitigating each one in turn. To remove memory allocation variance, an initialization event was sent to every member of the FTRTES object group at

start up time. Similarly, we also modified the TAO ORB configuration to avoid reconnecting when the supplier detects the connection to the primary has failed.

With each of these factors having its worst impact, the average event latency was 616 $msec$, with a maximum of 1956 $msec$. With prioritization of event push operations, the average event latency dropped to 5242 μsec with a maximum value of 5408 μsec. When we also only triggered primary crashes *between* events, the average event latency was reduced to 1642 μsec with a maximum value of 1707 μsec. When we also performed initial memory allocation prior to the first event push, the average event latency was 1311 μsec with a maximum value of 1405 μsec. Finally, when we also avoided unnecessary re-connections the average event latency dropped to 806 μsec with a maximum value of 927 μsec.

These results show that the mitigation steps described in this section are essential to optimizing FTRTES performance. By applying them in our FTRTES implementation, we were able to bring the fail-over event latency when faults did not occur during an event push close to the event latency with no failures seen in Figure 2 in Section 4.1.

5. Related Work

RT-CORBA [10] provides support for application control of system resources to achieve end-to-end predictability. RT-CORBA provides end-to-end real-time QoS support via prioritized object method calls.

TAO's Real-Time Event Service [11] provides predicable anonymous message delivery. It also allows applications to specify QoS requirements explicitly, so events can be scheduled and delivered to their destinations with rigorous QoS assurances. The Real-Time Notification Service [8] extends the Real-Time Event Service with *structured event* support.

Electra [16, 15] and Orbix+Isis [1, 15] are based on specialized group communication toolkits (Horus and Isis respectively) to provide support for fault-tolerance by replicating CORBA objects. Both Electra and Orbix+Isis require modifications to the ORB in order to deliver CORBA messages using the group communication toolkits. The advantage of this approach is the ease of application development; however, this may result in proprietary systems with limited replication strategies. For example, both Electra and Orbix+Isis only support active replication.

The Eternal System [19, 18] applies the Interceptor pattern [23] to support fault-tolerance. It intercepts system calls made by CORBA clients to low-level OS I/O subsystems, and transforms point-to-point communication into the Totem [17] group communication protocol for replicating CORBA objects. This approach does not require modification of the ORB implementation.

AQuA [2] does not require ORB modification either. It uses a gateway for accepting calls from clients and translating the request messages into the group communication primitives of Ensemble/Maestro [12, 25] which allows it to replicate objects, and detect and filter duplicate messages.

The Object Group Service (OGS) [4] provides a set of CORBA services to support fault-tolerance, including a group service, a consensus service, a monitoring service and a messaging service. Unlike the previous approaches which provide transparency to the application, this approach exposes the replication of objects to the application program, but it thus allows programmers more easily to customize the services for their needs.

DOORS [20] also takes a service-based approach to fault-tolerance. Instead of using a particular group communication toolkit, it allows application developers to select suitable transport protocols via TAO's pluggable protocols framework. In [7], Gokhale, *et al.*, propose the use of semi-active replication to meet both real-time and fault-tolerance requirements. Our work presented in this paper extends this approach to an event-channel mediated publish-subscribe communication model.

6. Conclusions

FT-CORBA provides a framework for fault-tolerant point-to-point communication. However, many distributed real-time and embedded (DRE) systems require an asynchronous and publish-subscribe style of communication. The Fault-Tolerant Real-Time Event Service (FTRTES) presented in this paper provides DRE system an event based communication model that meets high reliability requirements as well as real-time requirements. Our FTRTES implementation is distributed with TAO as open-source software that is freely available for download from http://deuce.doc.wustl.edu/Download.html

Our development of the FTRTES focused on providing a reliable and fault-tolerant capability within the existing TAO Real-Time Event Service [11]. We used the FT-CORBA framework as the basis for new techniques discussed in Section 3 to provide a fault-tolerant and predictable system. That experience revealed several valuable lessons about building fault-tolerant and real-time CORBA applications and middleware. First, the exposure of multiple interfaces to the clients can lead to longer IOGR update times and proliferation of transport connection times during fail-over. It is better to use the Façade pattern to encapsulate the functionality of the entire server. If backward compatibility is an issue for legacy applications, the adapter pattern can be introduced. Second, the semi-active replication style is more suitable than active or passive replication styles in applications with real-time constraints. Third, although for two-way/one-way replication setting a trans-

action depth is an effective way of trading off consistency and performance, AMI replication offers better state replication performance overall at a cost of some additional programming complexity. Fourth, prioritizing event handling together with an endpoint-per-priority model can greatly reduce the duration and variability of fail-over event dispatching latency.

References

[1] K. Birman and R. van Renesse. Reliable distributed computing with the isis toolkit. IEEE Computer Society Press, 1994.

[2] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. Aqua: An adaptive architecture that provides dependable distributed objects. In *Symposium on Reliable Distributed Systems*, pages 245–253, 1998.

[3] M. Fan, J. Stallaert, and A. B. Whinston. A web-based financial trading system. *Computer*, 32(4):64–70, 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[6] C. D. Gill, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, 20(2):117–154, 2001.

[7] A. Gokhale, B. Natarajan, D. C. Schmidt, and J. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing: the Journal on Networks, Software, and Applications Special Issue on Dependable Distributed Systems, edited by Alan George*, 7(4), 2004.

[8] P. Gore, I. Pyarali, C. D. Gill, and D. C. Schmidt. The design and performance of a real-time notification service. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 112, Washington, DC, USA, 2004. IEEE Computer Society.

[9] O. M. Group. The Common Object Request Broker : Architecture and Specification, Version 3.0 , December 2002.

[10] O. M. Group. RealTime-CORBA Specification, Version 1.2, November 2003.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA*, pages 184–200, Atlanta, GA, 1997.

[12] M. G. Hayden. *The Ensemble System*. PhD thesis, Cornell Univerty, Ithaca, NY, 1998.

[13] M. Henning and S. Vinoski. *Advance CORBA Programming with C++*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1999.

[14] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

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

[16] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, 1995.

[17] 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. *Commun. ACM*, 39(4):54–63, 1996.

[18] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for corba. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 150–157, Washington, DC, USA, 1999. IEEE Computer Society.

[19] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The eternal system: an architecture for enterprise applications. In *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC '99)*, pages 214–222, Mannheim, Germany, 1999.

[20] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. Doors: Towards high-performance fault tolerant CORBA. In *DOA '00: Proceedings of the International Symposium on Distributed Objects and Applications*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[21] C. O'Ryan, D. Schmidt, and J. Noseworthy. Patterns and performance of a CORBA event service for large-scale distributed interactive simulations, 2001.

[22] D. J. L. Paunicka, D. D. E. Corman, and B. R. Mendel. A corba-based middleware solution for uavs. In *ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 261, Washington, DC, USA, 2001. IEEE Computer Society.

[23] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[24] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), Oct. 2000. Updated by RFC 3309.

[25] A. Vaysburd and K. Birman. The maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theor. Pract. Object Syst.*, 4(2):71–80, 1998.