FAULT TOLERANT APPROACHES FOR DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS*

Paul Rubel
Matthew Gillen
Joseph Loyall
Richard Schantz
BBN Technologies
Cambridge, MA

and

Aniruddha Gokhale
Jaiganesh Balasubramanian
Vanderbilt University
Nashville, TN

and

Aaron Paulos
Priya Narasimhan
Carnegie Mellon University
Pittsburgh, PA

ABSTRACT

Fault tolerance (FT) is a crucial design consideration for mission-critical distributed real-time and embedded (DRE) systems, which combine the real-time characteristics of embedded platforms with the dynamic characteristics of distributed platforms. Traditional FT approaches do not address features that are common in DRE systems, such as scale, heterogeneity, real-time requirements, and other characteristics. Most previous R&D efforts in FT have focused on client-server object systems, whereas DRE systems are increasingly based on component-oriented architectures, which support more complex interaction patterns, such as peer-to-peer. This paper describes our current applied R&D efforts to develop FT technology for DRE systems. First, we describe three enhanced FT techniques that support the needs of DRE systems: a transparent approach to mixed-mode communication, auto-configuration of dynamic systems, and duplicate management for peer-to-peer interactions. Second, we describe an integrated FT capability for a real-world component-based DRE system that uses off-the-shelf FT middleware integrated with our enhanced FT techniques. We present experimental results that show that our integrated FT capability meets the DRE system's real-time performance requirements for both the responsiveness of failure recovery and the minimal amount of overhead introduced into the fault-free case.

INTRODUCTION

Distributed Real-time Embedded (DRE) systems are a growing class of systems that combine the strict real-time characteristics of embedded platforms with the dynamic, unpredictable characteristics of distributed platforms. As these DRE systems increasingly become part of critical domains, such as defense, aerospace, telecommunications, and healthcare, *fault tolerance (FT)* becomes a critical requirement that must coexist with their real-time performance requirements. DRE systems have several characteristics affecting their fault tolerance:

DRE systems typically consist of many independently developed elements, with different fault tolerance requirements. This means that any fault tolerance approach must support mixed-mode fault tolerance (i.e., the coexistence of different strategies) and the coexistence of fault tolerance infrastructure (e.g., group communication) and non-fault tolerance infrastructure (e.g., TCP/IP).

DRE systems' stringent real-time requirements mean that any fault tolerance strategy must meet real-time requirements with respect to recovery and availability of elements and the overhead imposed by any specific fault tolerance strategy on real-time elements must be weighed as part of the selection of a fault tolerance strategy for those elements.

DRE applications are increasingly component-oriented, so that fault tolerance solutions must support component infrastructure and their patterns of interaction.

DRE applications are frequently long-lived and deployed in highly dynamic environments. Fault tolerance solutions should be evolvable at runtime to handle new elements.

This paper makes two major contributions. First, it describes the particular characteristics and challenges of component-oriented DRE systems and describes three advances we have made in the state of the art in fault tolerance for DRE systems:

1) A new approach to communicating with replicas that supports the coexistence of non-replicated and replicated elements for DRE systems with varying FT requirements, with no extra elements and no extra overhead on non-replicated elements that only communicate with other non-replicated elements.
2) An approach to self-configuration of replica communication, which enables replicas, non-replicas, and groups to discover one another automatically as the number of, and fault tolerance requirements of, elements change dynamically.
3) An approach to duplicate management that supports replicated clients and replicated servers, necessary to support the complicated calling patterns of DRE applications.

A second contribution of this paper is that we demonstrate these advances in the context of an integrated fault tolerance capability for a real-world DRE system with strict real-time and fault tolerance requirements, a multi-layered resource manager (MLRM) used in shipboard computing systems. The fault tolerance we developed for this context utilizes off-the-shelf fault tolerance and component middleware with the above enhancements; and supports a mixture of fault tolerance strategies and large numbers of inter-operating elements, with varying degrees of fault

* This work was supported by the Defense Advanced Research Projects Agency (DARPA) under contract NBCHC030119.

tolerance. We then evaluate the performance of the replicated MLRM to meet its real-time and fault tolerance requirements and present analysis of the performance overhead of our fault tolerance approach.

CHALLENGES IN PROVIDING FAULT TOLERANCE IN DRE SYSTEMS

We first motivate our work by describing the fault-model and general approach under which our system operates. The next three sections introduce particular challenges with applying existing fault tolerance solutions to the needs of DRE systems, specifically:

- Communicating with replicas in large scale, mixed mode systems
- Handling dynamic system reconfigurations
- Handling peer-to-peer communications and replicated clients and servers.

A. FAULT-MODEL AND FAULT-TOLERANCE APPROACH

A fault model describes the types of failures we expect our system to deal with. By being specific about our fault model, we make clear the types of failures the system is designed to handle.

For our solution, we assume that all faults are fail-stop at the process level. When an application process fails, it stops communicating and does not obstruct the normal functioning of other unrelated applications. Network and host failures can be seen as a collection of process failures on the element that has failed.

We tolerate faults using both active [15] and passive [3] replication strategies. In these schemes we use multiple copies of an application, called replicas, to deal with failures of the applications. In active replication all replicas need to be deterministic in their message output, and each replica responds to every input message. Our software takes care of ensuring that only one request or response is seen regardless of how many actual replicas are used. In passive replication one *leader* replica responds to messages and shares its state with any non-leader replicas so they can take the leader's place in case of a failure. These passive replicas do not need to be deterministic but do need to be able to save and restore their state when responding to a message. Using these schemes, if a replica fails, there is another ready to act in its place and we can replace failed replicas if or when resources allow.

B. COMMUNICATION WITH GROUPS OF REPLICAS

Providing fault tolerance using replication requires a means to communicate with groups of replicas. A common approach is the use of a group communication system (GCS), to ensure consistency between and among replicas. DRE systems provide several challenges for using a GCS. DRE systems can contain large numbers of elements with varying fault tolerance and real-time requirements. These requirements range from not needing FT or RT to having very strict requirements. The following paragraphs describe approaches to group communication and its applicability to DRE systems.

Pervasive GCS. Some approaches [11] use GCS for communication throughout the entire system. This approach provides strict guarantees and ensures that interactions between applications and replicas are always done in the correct manner. In very large DRE systems, non-replica communication can be the more common case and using GCS everywhere can severely impact performance (as we show in a later section).

Pervasive GCS is particularly problematic in component-oriented systems due to features of component deployment. These deployment tools need to interact with a newly started application while existing replica continue to run. Unfortunately, the use of pervasive GCS would result in deployment messages going to existing replicas (which were previously deployed and are not prepared for additional deployment commands). Thus, replicating components requires the coexistence of non-group communications (during deployment) and group communications (once all replicas have been fully deployed).

In general, being able to do some initial work before all the requirements of replication are enforced is a very useful capability and can be used in other situations such as secure bootstrapping, registration, and other situations where initial non-replica processing or communication is required at start-up time.

Gateways. Other systems [4], [12] make use of gateways on the client-side that change interactions into GCS messages. This limits group communication to communication with replicas and provides the option to use non-GCS communication paths where necessary. The gateway approach does come with tradeoffs, however. First, it is less transparent than the pure GCS approach because the gateway itself has a reference that has to be explicitly called. Second, gateways typically introduce extra overhead (since messages need to traverse extra process boundaries before reaching their final destination) and extra elements that need to be made fault tolerant to avoid single points of failure. Other gateway-like strategies [6], [16] have also been explored, similar to the "fault-tolerance domain" specified in FT-CORBA.

Other projects [13] take a hybrid approach where GCS is only used to communicate between replicas and not to get messages to the replicas. This places the gateway functionality on the server-side of a client-server interaction, which limits the interactions between replicated clients and replicated servers but has implications for replicating both clients and servers at the same time. It introduces the possibility that lost messages may need to be dealt with at the application level as they cannot use the guarantees provided by the GCS.

ORB-provided transports. Some service-based approaches [7] completely remove GCS from the fault-tolerance infrastructure and use ORB-provided transports instead, which limits them to using passive replication.

C. CONFIGURING FT SOLUTIONS

A recurring problem with using GCS in dynamic systems like DRE systems is keeping track of groups, replicas, their references, and their supporting infrastructure as elements come and go during the life of a large system. Many existing fault tolerance solutions make use of static configuration files or environment variables [4], [11]. The DRE systems that we are working with are highly dynamic, with elements and replicated groups that can come and go and need to make runtime decisions about things such as fault tolerance strategy, level of replication, and replica placement. Static configuration strategies lack the flexibility needed to handle these runtime dynamics. Eternal [10] does support dynamic fault tolerance configurations. Greater flexibility is also available in some agent-based systems [9], but for more common non-agent infrastructures dynamically adding additional FT elements to a running system is not common.

D. REPLICATED CLIENT AND SERVERS AND PEER-TO-PEER INTERACTIONS

Support for replicated servers is ubiquitous in fault tolerance replication solutions, whereas support for replicated clients is not as common. Many CORBA-based fault tolerant solutions concentrate on *single-tier* replication semantics, in which an unreplicated client calls a replicated server, which then returns a reply to the client without making additional calls. Multi-tiered or peer-to-peer invocations are possible but the FT-CORBA standard does not provide sufficient guarantees or infrastructure to ensure that failures, especially on the client-side, during these invocations can be recovered from. A similar situation exists in some service-based approaches [2], [7] where peer-to-peer interactions are possible but care must be taken by developers using the functionality.

In contrast, component-oriented applications routinely exhibit peer-to-peer communication patterns, in which components can be clients, servers, or even both simultaneously. Many emerging DRE systems are developed based on component models and exhibit peer-to-peer calling structure, making solutions based on strict server replication of limited applicability.

Since components can be both clients and servers, component-oriented DRE systems can have chains of nested calls, where a client calls (or sends an event to) a server, which in turn calls another server, and so on. This leads to a need to consider replication of multiple tiers of servers. Research into supporting fault-tolerance in *multi-tiered* applications is still ongoing. Some of the most promising recent work has concentrated on *two-tier replication*, specifically addressing applications with a non-replicated client, a replicated server, and a replicated database [8].

General, unrestricted calling patterns, such as asynchronous calls, nested client-server calls, and even callbacks (where clients also act as servers and can have messages arrive via the callback mechanism while replies from sequential request-reply messages are pending), present tremendous challenges for fault tolerance solutions. This is partially due to the need for fault tolerance to maintain message ordering, reliable delivery, and state consistency, which is harder to do in asynchronous, multi-threaded, and unconstrained calling patterns. It is also due to the fact that the semantics of such calling patterns in the face of replication are more difficult to define.

FAULT TOLERANCE SOLUTIONS TO THE CHALLENGES FOR DRE SYSTEMS

In this section, we describe three new fault tolerance advances that we have developed, each of which addresses one of the challenges described in the following section. First, we describe a *Replica Communicator (RC)* that enables the seamless and transparent coexistence of group communication and non-group communication while providing guarantees essential for consistent replicas. Next, we describe a self-configuration layer for the RC that enables dynamic auto-discovery of new applications and replicas. Finally, we describe an approach and implementation of duplicate message management for both the client- and server-side message handling code in order to deal with peer-to-peer interactions.

Once we've described each of these solutions, we discuss how we integrated them with other off-the-shelf fault tolerance software solutions to create a flexible and generally applicable fault tolerance solution which we then demonstrated and evaluated using a specific DRE system.

A. THE REPLICA COMMUNICATOR

Limiting the use of group communication provides a way to separate concerns and limit resource usage and complexity in a large system. Where group communication is necessary for maintaining consistent replicas it needs to be available. In other areas, where group communication is not needed, we want to remove it. This separation allows us to not disturb the delicate tuning necessary for real-time applications when group communication is not needed. Analysis of our replication schemes shows that the only places where GCS communication is necessary is when interacting with a replica. That is, only replicas and those components that interact directly with them need the guarantees provided by group communication. Other applications can use TCP without having to unnecessarily accept the consequences of using group communication.

There are several advantages to limiting the use of GCS. The first reason is that GCS introduces a certain amount of extra latency, overhead, and message traffic that is undesirable in the non-replica case and, in fact, can jeopardize real-time requirements. Second, many off-the-shelf GCS packages, such as Spread [1], have built-in limits on their scalability and simply do not work with the large-scale DRE systems that we are targeting. Finally, as described earlier, many of the components of our targeted DRE systems are developed independently. Since the non-replicated case is the prevalent one (most components are not replicated), retrofitting these components onto GCS, with the subsequent testing and verification, would be a tremendous extra added effort for no perceived benefit.

Therefore, we developed a new capability, called a Replica Communicator, with the following benefits:

- The RC supports the seamless co-existence of mixed mode communications, i.e., group communication and non-group communication.
- It introduces no new elements in the system.
- It can be implemented in a manner transparent to applications.

The RC can be seen as the introduction of a new *role* in an application, along with the corresponding code and functionality to support it. That is, the application now has three communication patterns:

1) Replica to replica communication, which uses GCS
2) Non-replica to non-replica communication, which uses TCP
3) Replica to non-replica communication, in which the replica always uses GCS and the non-replicas make use of an RC to route the communication to a replica over GCS while using TCP for communicating with other non-replicas

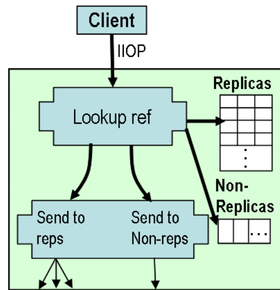

Fig. 1. Generalized pattern of the Replica Communicator

An abstract view of the RC is illustrated in Fig. 1. Its basic functionality consists of the following pieces:

- Interception of client calls (In this case calls used to send messages formatted using CORBA IIOP)
- A lookup table to hold references to replicas and to non-replicas
- A decision branch that determines whether a call is destined for a non-replica or a replica and treats it accordingly
- A means to send a message to all replicas, e.g., using multicast, looping over all replica references, or using GCS
- A default behavior, treating a message by default as one of the branches
- A configuration interface to add references to new servers, to add new replicas to an existing group, or to remove a replica (if it has failed)

Documented in the above pattern, the RC can be realized with multiple implementations, from application specific implementations to easier to integrate solutions using standard insertion techniques and library code.

The RC functionality resides in the same process space as the application. This improves over traditional gateway approaches, because it introduces no extra elements into the system. Notice that the RC does not need to be made fault tolerant, since it is only used by non-replicas.

We have realized a prototype of the RC pattern in the system described in the next section and have implemented it using the MEAD framework [11] and its system call interception layer, as illustrated in Fig. 2. CORBA calls are intercepted by MEAD. The RC code maintains a lookup table associating IP addresses and port numbers with the appropriate transport and group name if GCS is used. The default transport is TCP; if there is no entry in the lookup table, the destination is assumed to be a non-replicated entity. For replicated entities, the RC sends the request using the Spread GCS, which provides totally-ordered reliable multicasting. For replies, the RC remembers the transport used for the call, and returns the reply in the same manner.

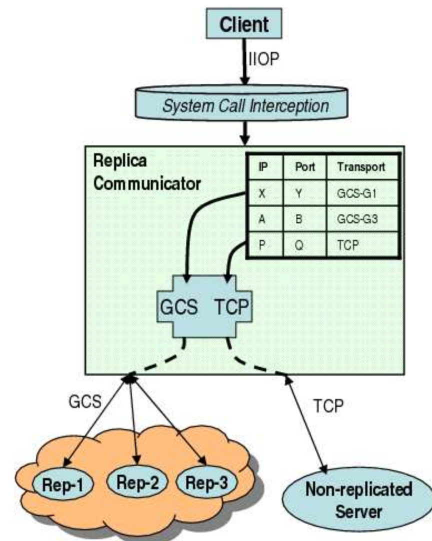

Fig. 2. The Replica Communicator instantiated at the system call layer

The Replica Communicator was crucial for resolving the problem outlined previously, namely that the CCM deployment infrastructure needs a way to communicate with exactly one replica during bootstrapping so that start-up messages are not sent to already running and processing replicas. We used the RC with our CCM-based active and passive replicas to allow a replica to be bootstrapped while not disturbing the existing replicas.

B. A SELF-CONFIGURING REPLICA COMMUNICATOR

Populating the table distinguishing GCS and TCP endpoints can be done in multiple ways. One way is to set all the values statically at application start-up time using configuration files. However, this leads to static configurations in which groups are defined a priori and supporting dynamic groups and configurations is difficult and error prone. To better support the dynamic characteristics of DRE systems and to simplify configuration and replica component deployment, we developed a self-configuring capability for the RC.

When a GCS-using element (i.e., a replica or non-replica RC) is started, we have it join a group used solely for distributing reference information. The new element announces itself to the

other members of the system, which add an entry to their lookup table for the new element. An existing member, chosen and made fault-tolerant in the same way that a leader is chosen in warm-passive replication, responds to this notification with a complete list of system elements in the form of an RC lookup table. The new element blocks until the start-up information is received, to ensure that the necessary information is available when a connection needs to be established (i.e., when the element makes a call). Since GCS-using elements always register and are blocked at start-up until they are finished registering, the RC will always have all the information it needs to initiate any connection. If there is no entry for a given endpoint it means that TCP should be used for that connection.

One complexity that does not affect users, but needs to be taken into account while developing the self-configuring RC, is that the relationships between elements are not necessarily transitive. Simply because RC_1 interacts with replica R via GCS and R also interacts with RC_2 via GCS, this does not mean that RC_1 should use GCS to interact with RC_2. In the case of manual configuration this is handled by having a configuration specific for each application. However, in our automated solution it is necessary to do more than note that a given endpoint can be contacted via a given GCS group name. We also need to distinguish the circumstances where GCS is necessary and those where it is not. We accomplish this by noting whether a reference refers to a replica or non-replica. Given that interacting with a replica or being a replica are the only two times GCS is necessary, an RC knows to use GCS when it is interacting with a replica (and TCP elsewhere) and replicas always use GCS.

C. CLIENT- AND SERVER-SIDE DUPLICATE MANAGEMENT

One step towards a solution for replication in multi-tiered systems is the ability for each side of an interaction to perform both client and server roles, at the same time. They also need to detect and suppress duplicate messages while allowing nested calls to be made. All this needs to be done without locking up an entire tier waiting for a response, which can guarantee consistency, but is very limiting.

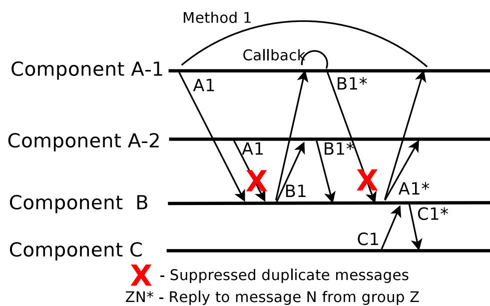

Fig. 3. Duplicate management during peer-to-peer interactions

One characteristic necessary to support duplicate management is that messages need to be globally distinguishable, both within an interaction and between multiple interactions. When multiple senders independently interact with a shared receiver, it is important to differentiate messages based not only on message ID, but to use a combination of message ID and source. In Fig. 3 both A (replicated as A-1 and A-2) and C use sequence number 1 to send a message to B, but since suppression uses both the sequence number and the sender there is no confusion. A-2's duplicate message is suppressed while C's non-duplicate is allowed.

Our solution enables duplicate management in the highly dynamic situations typical of DRE and component-based software. Requests and replies can be dealt with in parallel and are unaffected by failures that could reset application-level sequence numbers. We replace the ORB supplied request ID with a unique and consistent value for each request or reply and distinguish messages upon receipt using both the ID as well as the sending group. This allows replicas to come and go without introducing any extra messages at the application layer.

IMPLEMENTATION OF AN INTEGRATED FT CAPABILITY

As part of a case study we performed on providing fault tolerance in a real-world DRE system with stringent real-time requirements [14], we implemented a fault tolerance architecture integrating the techniques described previously with other off-the-shelf fault tolerance software to make a pre-existing software base called the Multi-Layer Resource Manager (MLRM) fault tolerant. The MLRM is a critical piece of system functionality because it deploys mission-critical applications and enables them to continue functioning after failures by redeploying them.

There are three distinct hierarchical layers of the MLRM, each corresponding to a physical division:

- *Node*: Provides services (such as *Start Application* or *Stop Application*) for a specific node. One of this layer's tasks is to start execution of applications. There are many (hundreds to thousands) nodes in the system.
- *Pool*: Provides services and an abstraction layer for physically clustered groups of nodes called pools. One of this layer's responsibilities is to designate nodes for applications to run on.
- *Infrastructure*: Provides the control interface to operators and coordinates the pool-layer services. One of this layer's tasks is to designate pools for applications to run in.

The MLRM is implemented using a number of base technologies including DAnCE [5]; CIAO,which is a C++ implementation of CCM; a real-time CORBA ORB, TAO; and JacORB, a Java ORB.

The MLRM takes care of the fault tolerance of individual applications, by restarting them if they fail (on a different node if the cause of the failure is a node failure). However, our design requirements state that pool failures (entire clusters of nodes) are a possibility. Since the infrastructure-level components are hosted on nodes within one of the pools, the failure of an arbitrary pool could lead to the failure of the infrastructure-level components, rendering the entire system unusable. Therefore, our fault tolerance focus is on the availability of the infrastructure-level MLRM components and recovery from catastrophic pool failures.

The goal was to have an instance of the infrastructure-level components in every pool, so that the failure of any pool would not bring down the infrastructure layer MLRM functionality.

MLRM is representative of classic DRE systems, and exhibits many of the characteristics outlined previously, including the following:

- The infrastructure layer functionality, since it is critical functionality, needs to be nearly continuously available, motivating fault masking or very rapid recovery.
- The MLRM has to deploy many hundreds or thousands of application components, and there are hundreds or thousands of node level service components, most of which do not need to be fault tolerant to the same degree. Yet some of these need to communicate with the infrastructure layer components. Existing off-the-shelf fault tolerance software used GCS systems for the group management; reliable, ordered multicasting; and consistency guarantees that we needed. However, they did not handle mixed-mode communications and re-hosting the entire system over GCS was out of the question due to scalability and performance concerns. This motivated using the RC approach described earlier.
- The infrastructure components are implemented as CCM components, many of which are both clients and servers, and with multiple tiers needing replication. Because of this, we needed both client-side and server-side duplicate management, as described earlier.
- The infrastructure components differed in their amount of state and their use of non-determinism. Despite our need for continuous availability, some components simply could not be actively replicated. We had to use mixed-mode replication, with active and passive schemes applied where they met the requirements and matched the component characteristics.

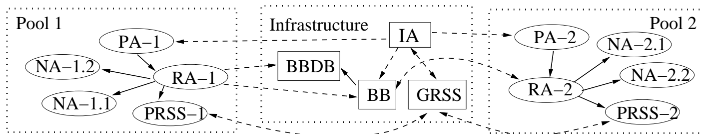

Fig. 4. Target system architecture

The relevant MLRM components and their communications paths are shown in Fig. 4. Replicated components are shown in boxes, non-replicated components are ellipses. Communication paths that need to go over GCS are shown as dotted lines, and TCP connections are shown as solid lines.

The *Infrastructure Allocator (IA)* component makes top-level application deployment decisions. This component has significant state, but is largely deterministic. We determined that *active* replication was most appropriate for this component. Active replication provides fault masking, so that there is always a replica available for processing messages, and allows us to avoid state transfers, which in the case of the IA (with significant state, but infrequent message traffic) reduced the impact of fault tolerance on system performance.

While active replication would seem the appropriate choice to use wherever possible, the characteristics of other MLRM components made it infeasible. The *Global Resource Status Service (GRSS)* (which monitors resource usage and system health) and *Bandwidth Broker (BB)* (which manages bandwidth allocation) elements were written in Java and used JacORB. The fact that JacORB is inherently multi-threaded and the fact that the application logic uses internal timers means that the GRSS and BB elements are non-deterministic. The GRSS has a very small amount of state, and the BB element is stateless (it uses a back-end database to store all state). We determined that *passive* replication was the best choice for the GRSS and BB elements, as long as we could implement the passive recovery within our real-time requirements (the next section shows that we did). The performance trade-off is favorable. With passive replication, overhead on ordinary message traffic is lower, but periodic state transfers are necessary. The state being transferred for these two elements was much smaller (when compared to the state of the IA), but the GRSS receives frequent messages reporting on the health of nodes, processes, and pools.

We used the MEAD fault tolerance framework [11], extended with our new Replica Communicator and client-side duplicate suppression, and the Spread GCS to implement our active and passive fault tolerance. Spread provides group membership management and total ordered, reliable multicast of messages to group members. We used Spread's group membership features to detect failures.

While the GRSS element used a standard passive replication scheme that broadcasts the primary's state on every state change, the BB element used a custom passive scheme. The BB element kept all its state within a MySQL database. We used MySQL's clustering mechanisms (with some customizations by our collaborators from Telcordia) to achieve a replicated database. Since the BB element itself had no state, and the MySQL back-end replicated itself using the built-in clustering mechanisms, we were able to use an optimized passive scheme for the BB element that did not transfer state from the primaries to the secondaries.

None of the pool-level components are replicated, but several must communicate with replicated infrastructure-level components, and therefore use the RC pattern (i.e., PA-1, RA-1, and PRSS-1). Note also that the Node Application (NA) components are not replicated, and use regular TCP connections to communicate with the Resource Allocator (RA) component.

EVALUATION OF THE INTEGRATED FT SOLUTION

We measured the performance of our fault tolerance solution both in terms of meeting the real-time recovery requirements and in terms of the impact of the solution on the fault-free performance of the system.

First, we measured the failure recovery time (i.e., downtime during a failure) using two failure scenarios. Second, we measured the fault-free overhead (i.e., the extra latency during normal operation introduced by our fault tolerance software) by comparing the "raw TCP" performance of a simple client-server configuration

against the same client-server using our fault tolerant software.

A. SINGLE POOL FAILURE SCENARIO

For our first scenario, we considered a single pool failure, in which a whole pool fails instantaneously. This kind of failure might result from a major power failure or destruction of a hosting facility. We simulated this failure by creating a network partition so that packets sent to the failed pool would be dropped at a router. This is an accurate simulation of failure and has the advantage that we are able to determine the time the failure occurred to the millisecond.

The experiment evaluated two things, (1) that the mission-critical functionality (i.e., the infrastructure layer MLRM) could recover from the failure and (2) the speed of recovery, with the goal being that a failure should not inhibit processing for more than one second. We measured the recovery times for the actively replicated and passively replicated elements using instrumentation we inserted in the fault tolerance software. We measured the recovery time for the database replication using a program that made constant (as fast as possible) queries on the databases. When the database failover occurs, there is a slight increase in the time it takes to do the query, since the database blocks until it determines that one of the replicated instances is gone (and is never coming back).

As illustrated in Fig. 5A, the MLRM recovered all its functionality well within our real-time requirement. The MLRM elements made fault tolerant using MEAD, Spread, and our enhancements recovered on average in under 130 ms, with a worst case recovery in less than 140 ms. The database, made fault tolerant using MySQL's clustering technique, recovered on average within 140 ms, with a worst case recovery time under 170 ms.

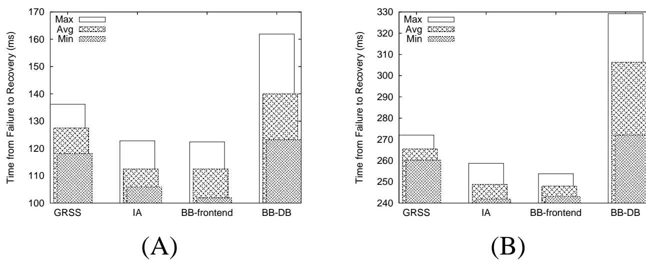

Fig. 5. Experimental Results for Scenarios A and B

B. TWO CASCADING FAILURES SCENARIO

Our second scenario evaluated that the fault tolerant MLRM could recover from cascading failures. We used three pools and induced failure on one pool. Before the recovery was complete, we induced a failure on another pool.

As with the single-pool failure's results, the mission-critical MLRM functionality survived the cascading failures. Also, as expected, the recovery times (from the time of the first failure) are about twice those of the single failure, but still well within a few hundreds of milliseconds, as shown in Fig. 5B. The recovery times are higher than for the first scenario because we induced the second (cascading) fault when recovery was nearly complete, the worst possible time with respect to recovery from the first failure.

C. OVERHEAD OF THE FAULT TOLERANT SOFTWARE

We measured the fault-free overhead of C++/TAO and Java/JacORB versions of our fault-tolerance. These tests did not involve the MRLM system, but instead used a simple client-server configuration.

Our goal was to compare the latency of using CORBA with raw TCP against the latency of using CORBA with our fault-tolerant middleware.

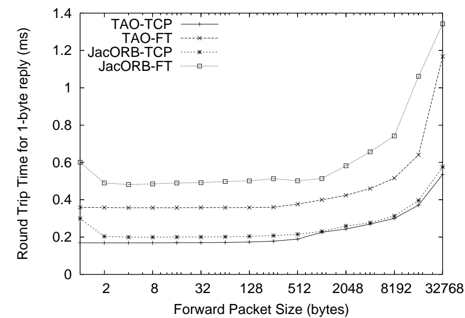

Fig. 6. Latency of Transport Mechanisms

The results shown in Fig. 6 show that our fault tolerance software adds approximately a factor of two to the latency compared to CORBA over TCP. However, if we didn't need replicated servers, then we wouldn't use anything but regular TCP (the whole point of the Replica Communicator). So we also ran the same tests, but with an actively replicated server. To implement the replicated server in the TCP version, we constructed a simple sequential invocation scheme where in order to make a single logical call the client would make serial invocations on each server instance. The results from two and three replicas are shown in Fig. 7. While this implementation may be simplistic in terms of not making parallel invocations, it also does not deal with multi-phase commit protocols which would be used to provide guarantees needed for replicas and is a reasonable first-order stand-in for such protocols.

In the two replica case, the results show that the fault tolerance software using GCS performs nearly as well as TCP, introducing very little extra latency for its total order and consensus capabilities. In the three replica case, the fault tolerance with GCS performs better than raw TCP.

D. ANALYSIS OF EXPERIMENTAL EVALUATION

Our results indicate that our FT solution enables fast recovery both in a single failure and cascaded failure scenarios.

The fault-free overhead experiments highlight the importance of the Replica Communicator. If every component in the system was required to use our fault tolerance software, the cumulative effect would adversely affect the real-time applications in our

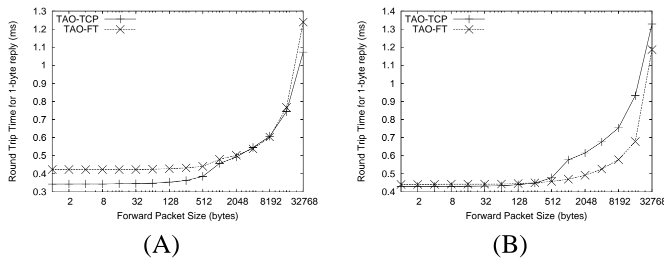

Fig. 7. Latency of Transport Mechanisms with 2(A) and 3(B) replicas

system. This supports our claim that only components that require fault tolerance infrastructure (i.e., replicas and components that communicate with replicas) should use it, and using the RC to limit the fault tolerance infrastructure to where it is needed improves real-time performance, while at the same time enabling total ordered messages, consistency among replicas and group management.

The experimentation we have done gives us confidence that our software fault tolerance solution handles failures sufficiently rapidly and within acceptable overhead parameters for soft-real time systems as exemplified by the requirements for our evaluation context. However, this may prove insufficient where hard real-time guarantees are needed.

CONCLUSIONS

This paper has described advances we have made in software support for fault tolerance for DRE systems. Our approach – very successful in this project – was to utilize off-the-shelf fault tolerance software where it was applicable for our needs, customize it where necessary, and develop new reusable capabilities where none existed.

The three techniques that we presented in this paper – the Replica Communicator, self-configuration for replica communication, and client- and server-side duplicate management – extend existing fault tolerance techniques to make them suitable for component-oriented DRE applications. Yet, they are complementary to, and interoperable with, other existing fault tolerance services. To illustrate this, we have instantiated them and applied them to a real-world DRE example application. Our experiments show that these solutions provide suitable real-time performance in both failure recovery and fault-free cases.

REFERENCES

[1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Center for Networking and Distributed Systems, Johns Hopkins University, 1998.

[2] R. Baldoni, C. Marchetti, and A. Virgillito. Design of an Interoperable FT-CORBA Compliant Infrastructure. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, 5 2001.

[3] N. Budhiraja, K. Marzullo, F. Scneider, and S. Toueg. *The Primary-Backup Approach*, chapter 8. ACM Press, Frontier Series. (S.J. Mullender Ed.), 1993.

[4] M. Cukier, J. Ren, C. Sabnis, 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 *Proc. of the IEEE Symposium on Reliable and Distributed Systems (SRDS)*, pages 245–253, West Lafayette, IN, October 1998.

[5] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, November 2005.

[6] P. Felber. Lightweight Fault Tolerance in CORBA. In *Proc. of the International Conference on Distributed Objects and Applications*, pages 239–250, Rome, Italy, September 2001.

[7] C. D. Gill, D. L. Levine, and D. C. Schmidt. Towards Real-time Adaptive QoS Management in Middleware for Embedded Computing Systems. In *Proc. of the 4th Annual Workshop on High Performance Embedded Computing*, Lexington, MA, September 2000. MIT Lincoln Laboratory.

[8] B. Kemme, M. Patino-Martinez, R. Jimenez-Peris, and J. Salas. Exactly-once interaction in a multi-tier architecture. In *Proc. of the VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, August 2005.

[9] O. Marin, M. Bertier, and P. Sens. Darx - a framework for the fault-tolerant support of agent software. In *Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)*, pages 406–417, November 2003.

[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, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. MEAD: Support for Real-time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 2005.

[12] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Gateways for Accessing Fault Tolerance Domains. In *Middleware 2000, LNCS 1795*, pages 88–103, New York, NY, April 2000.

[13] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck. Fault-Tolerant Replication Based on Fragmented Objects. In *Proc. of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - DAIS 2006*, pages 256–271, June 2006.

[14] P. Rubel, J. Loyall, R. Schantz, and M. Gillen. Fault Tolerance in a Multi-layered DRE System: a Case Study. *Journal of Computers (JCP)*, 1(6):43–52, 2006.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[16] A. Vaysburd and S. Yajnik. Exactly-once End-to-end Semantics in CORBA invocations across heterogeneous fault-tolerant ORBs. In *IEEE Symposium on Reliable Distributed Systems*, pages 296–297, Lausanne, Switzerland, October 1999.