End-to-End Consistency of Multi-Tier Operations Across Mixed Replicated and Unreplicated Components

Priya Narasimhan and Aaron M. Paulos

July 20, 2006
CMU-CyLab-06-013

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

End-to-End Consistency of Multi-Tier Operations Across Mixed Replicated and Unreplicated Components [*]

Aaron M. Paulos
ECE Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213
apaulos@ece.cmu.edu

Priya Narasimhan
ECE Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213
priya@cs.cmu.edu

ABSTRACT

While distributed applications need replication for the purposes of fault-tolerance, realistic and feasible deployments cannot afford to replicate every single component within the system. Potentially, over the lifecycle of such deployments, the consistency and fault-tolerant properties might be compromised when replicated and unreplicated components interact. We describe some of the challenges in providing end-to-end fault-tolerance under the mixed semantics. Our approach facilitates communication between the unreplicated and replicated components of a distributed client-server application, without compromising the consistency of the replicated servers and without restricting any concurrent TCP semantics that unreplicated clients expect. We describe the resulting architectural and implementation enhancements to the MEAD system and provide an empirical evaluation of our new mechanisms.

1. INTRODUCTION

Replication is a common technique used for providing fault-tolerance to distributed applications. With replication, the idea is to provide multiple copies, or replicas, of various application components so that even if one replica crashes or fails, another replica can take over and continue the operation. The fundamental property underlying replication is *consistency*, i.e., the replicas of every component must produce the same output and undergo the same state changes when processing a given operation; otherwise, the replicas might produce different results and diverge in their internal states. Inconsistent replicas defeat the purpose of replication as a fault-tolerance mechanism because they cannot serve as replacements for each other in the event of failures.

To achieve consistent replication, most replicated systems require a number of mechanisms, such as the identical ordering of messages delivered to replicas, the detection and suppression of duplicate messages, etc. Frequently, the fault-tolerance infrastructure that supports such replicated applications employs an underlying totally ordered, reliable group communication system (GCS) to convey all of the messages to and from the replicated components.

For real-world distributed applications, fault-tolerance might be required only for specific, critical parts of the application, and it would be overkill to replicate all of the application's components and host them atop a fault-tolerance infrastructure. There are several reasons for this, including: (i) the presence of components that necessarily exist/run outside the purview of the fault-tolerance infrastructure because of logical, physical or administrative constraints, (ii) the requirement to support non-GCS protocols for communication between some of the application's components, and (iii) the fact that fault-tolerance comes at a price, which means that not all of the components can afford the runtime overheads that inevitably accompany a useful, but complex, concept such as replication.

Recognizing this need to support *interacting* replicated and unreplicated components simultaneously within a distributed system, the Fault-Tolerant CORBA standard [8] introduced the concept of a fault-tolerance, or replication, domain. This is a logical decomposition of the application where all of the components assigned to the fault-tolerance domain are replicated and hosted over a fault-tolerance infrastructure. Components that are relegated to being the fault-tolerance domain are unreplicated, and obviously, not fault-tolerant.

If the unreplicated components kept to themselves, i.e., they did not communicate with the replicated components, no interesting or difficult issues arise. The challenges emerge when we consider the possibility that the unreplicated and replicated components might communicate with each other, as an intrinsic part of end-to-end operations that span both the fault-tolerant and non-fault-tolerant domains. The problem is partly due to the fact that, on one hand, the unreplicated components communicate using TCP/IP with its FIFO delivery guarantees, while the replicated components communicate using GCS messages with their totally ordered delivery guarantees. Furthermore, to ensure deterministic operation, operations that are delivered to replicated components via GCS are often serialized (i.e., with blocking semantics for end-to-end synchronous operations) while TCP/IP-supported entities can have concurrent access (i.e.,

[*]This work has been partially supported by the NSF CAREER grant CCR-0238381 and the DARPA Adaptive and Reflective Middleware Systems (ARMS).

connections are independent of each other and do not need to block each other).

We have previously explored the idea of connecting unreplicated clients to replicated CORBA [7] servers through the use of gateways [6]. In this case, a gateway is a non-CORBA, unreplicated, system-level process executing on some node in the distributed system; its sole purpose is to forward messages back and forth between a single unreplicted CORBA client and a replicated CORBA server. While the gateway was useful in serving as a proof-of-concept of the Fault Tolerant CORBA (FT-CORBA) standard's gateway support for unreplicated clients, it merely scratched the surface of the issues that can arise when unreplicated and replicated components are allowed to interact in a distributed system.

In this paper, we go beyond the simple concept of a FT-CORBA gateway. Our objective is to enable reliable communication between unreplicated and replicated components for a distributed CORBA application, *without compromising the consistency of the replicated components and without restricting concurrent TCP semantics that unreplicated components expect.* We introduce the concept of a *singleton*, an application-level CORBA component whose underlying infrastructure functions as a natural bridge for end-to-end synchronous CORBA operations where the origin of the operation is an unreplicated client and the target is a replicated server. This infrastructural-bridging enhancement to our MEAD fault-tolerant middleware [5] enable us to host singletons that can service *both* TCP and GCS connections.

Concretely, the contributions of our approach are (i) the design and implementation of the singleton-based infrastructural bridging approach for integrating replicated and unreplicated semantics in end-to-end operations, (ii) the description of the challenges involved in combining the two different (TCP/unreplicated and GCS/replicated) semantics in distributed client-server operation, and (iii) an empirical evaluation of our singleton-based approach in order to evaluate its performance overheads, under both fault-free and faulty conditions.

2. BACKGROUND AND RELATED WORK

The Common Object Request Broker Architecture (CORBA) [7] middleware supports applications that consist of objects distributed across a system, with client objects invoking server objects that return responses to the client objects after performing the requested operations. CORBA's TCP/IP-based protocol, the Internet Inter-ORB Protocol (IIOP), allows client and server objects to communicate regardless of differences in their operating systems, byte orders, hardware architectures, etc.

CORBA client-server applications can be multi-tiered in structure. We use the term tier to refer to a pure client, a pure server or a client+server component in an end-to-end CORBA operation. For example, consider the chain of requests when a client A invokes a server $S1$ that, in turn, invokes another server $S2$; assume that, after $S2$ returns a reply to $S1$, the latter, in turn, sends a reply to A. We consider this to be a three-tier application, with A, $S1$ and $S2$ representing tier 1, tier 2 and tier 3, respectively; the end-to-end synchronous operation, $A \rightleftarrows S1 \rightleftarrows S2$, spans all three tiers, as shown in Figure 1.

The Fault Tolerant CORBA standard [8] specifies reliability support through the replication of the CORBA servers, and the subsequent distribution of the replicas of every server across the nodes in the system. The idea is that, even if a replica (or a node hosting a replica) crashes, one of the other server replicas can continue to uphold the server's availability. Replication does not alter the number of tiers in the application, i.e., replicating $S2$ is a logical feature that does not change how many tiers the end-to-end operation must pass through. Also, as shown in Figure 1, in the multi-tiered CORBA example, a pure client C might remain unreplicated but still continue to use GCS to communicate with a replicated server such as $S2$. We refer to such unreplicated clients as *GCS clients* to differentiate them from unreplicated *TCP clients* such as A (that reside outside the GCS domain and that communicate via TCP/IP). While our mechanisms target support for both GCS and TCP clients, the consistency issues differ across the two kinds of clients because the GCS clients can exploit totally ordered messaging guarantees while the TCP clients cannot, making them more complicated to handle.

The Fault Tolerant CORBA standard outlines a list of interfaces that constitute a fault-tolerance service and emphasizes that strong replica consistency should be provided in order to ensure effective fault-tolerance. Several fault-tolerant CORBA systems (Eternal, OGS, IRL, FTS, AQuA, DOORS, Orbix+Isis, Electra) [3] have emerged over the past decade. These systems have tended to require all of the objects of the CORBA application to be replicated and hosted over the fault-tolerant system. The Fault Tolerant CORBA standard, driven in part by the needs of vendors and real-world customers, incorporated the notion of fault-tolerance domains for scalability and ease of adminstration. Each fault-tolerance domain handled the replication of only its constituent objects, thereby making it possible to manage the large fault-tolerant applications by decomposing them into appropriate fault-tolerance domains.

The standard introduced the idea of a gateway to allow unreplicated clients running over non-fault-tolerant CORBA systems to invoke replicated objects that were located within fault-tolerance domains. However, the standard remains sketchy in its description of the gateway's implementation details, merely mentioning that the gateway's addressing information needs to be supplied to unreplicated clients outside the domain. Gateways [2, 6] and (predecessor) gateway-like client-side support [10] represent successful designs for supporting two-tier CORBA applications with an unreplicated client. However, to the best of our knowledge, this previous work does not include an in-depth critical look at the end-to-end consistency issues in mingling unreplicated/TCP and replicated/GCS semantics, and has also not included any empirical assessment of the attendant scalability and performance issues. The approach that we describe in this paper was born out of the lessons that we learned first-hand in implementing such a gateway [6] – thus, we aim to understand, articulate and resolve the underlying consistency challenges, as well as to quantify the actual performance overheads of the resulting system.

3. APPROACH

The MEAD system [5] that we have developed at Carnegie Mellon University is a fault-tolerance infrastructure that supports the transparent replication and recovery of components in stateful CORBA applications. In order to ensure the system-wide consistency of replicas of every CORBA

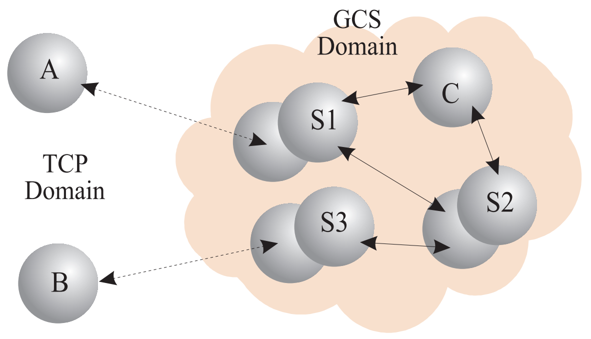


Figure 1: Example scenario used to show the mingling of TCP and GCS across unreplicated (A, B and C) and replicated ($S1$, $S2$ and $S3$) components.

component, MEAD exploits the guarantees of the underlying Spread group communication system [1] to convey the replicated application's messages. In addition, to ensure exactly-once semantics for processing requests and replies in the presence of replication, MEAD perfoms the detection and suppression of duplicate requests and replies before delivering messages to the application.

MEAD goes beyond previous fault-tolerant CORBA systems in aiming for adaptive fault-tolerance (where the key replication properties of the system can be modified at runtime based on various system conditions), proactive fault-recovery (where pre-failure symptoms can be extracted from the system to provide advance notice of an impending failure, thereby paving the way for preemptive recovery action) and support for nondeterministic applications (where the application is analyzed for nondeterministic actions and appropriately handled, rather than forbidding programming practices such as multithreading).

This paper focuses on MEAD's support for the bridging of unreplicated (TCP) and replicated (GCS) semantics in multi-tier CORBA applications. As straightforward as it sounds to add support for TCP connection establishment and communication into a system, the interesting technical issues emerge because of the need to maintain replica consistency in the face of the dichotomy of (communication, ordering, connection and consistency) semantics in the resulting hybrid replicated+unreplicated system.

3.1 Challenges

Several challenges arise in the design and implementation of a solution that aims to keep the mingling of replicated and unreplicated semantics as transparent and as consistent as possible to the application.

Transparency. The TCP/IP-based components must continue to experience behavior that is consistent with their expected TCP/IP semantics. For example, when a client establishes a TCP connection, there might be various socket-related options, flow-control mechanisms, timeouts and error-reporting behavior that it expects. None of these mechanisms should be invalidated by the presence of a GCS connection somewhere in the end-to-end path of the operation that originates at the TCP client. While metrics such as the client-side response time will likely change, the client-side communication behavior (e.g., expectations of specific TCP exceptions under error conditions) should remain the same. In addition, a component should be unaware, at runtime, of whether it has been contacted via GCS or TCP. The idea is to divorce the application programmer from worrying/knowing about mixing the different semantics and to let the MEAD infrastructure handle them transparently instead.

Consistency. Point-to-point TCP client or server connections should not be allowed to communicate directly with an individual replica, e.g., in Figure 1, A should never be allowed to communicate with an individual replica of $S1$. Such out-of-band non-GCS communication is prohibited because that replica's state might be modified unilaterally and subsequently start to diverge from the states of its fellow replicas. In the interests of consistency, any communication with a replicated component should occur atomically across all of its replicas so that they all receive the same ordered sequence of messages. When the intended recipient of a TCP message is a replicated component, the TCP message must be conveyed over the GCS protocol. Of course, on the outbound path from a replicated component to a TCP recipient, a TCP message must be constructed out of the corresponding GCS messages. However, additional care is required – when GCS requests or replies originate from a replicated component, multiple identical copies (one from each replica) of a message might be generated and sent. If a receiving component were to process all of these duplicate messages, its state would likely be incorrect. Therefore, such duplicate messages must be detected and suppressed in order to deliver a single, non-duplicate message to the intended TCP recipient.

Concurrency. TCP messages are ordered on a first-in-first-out (FIFO) basis while the GCS messages that support component replication are totally ordered. The mixing of the different ordering schemes (FIFO vs. total order) can cause interesting side-effects when multiple unreplicated TCP components are involved.

Any mechanism that we insert to bridge between TCP and GCS semantics needs to support concurrent operation because multiple, independent TCP components might wish to communicate with various replicated components simultaneously. Within a replicated domain, common approaches to maintaining consistency include the serialization of requests with client-side blocking, which means that every participating tier in an end-to-end operation can effectively not service any other invocation as long as the end-to-end operation is ongoing at some other tier. This is the quiescence property that fault-tolerant systems require in order to ensure that nondeterministic multithreading does not lead to inconsistent replication. A request can only be delivered to a server that is quiescent, which means that the server is not waiting on a response and is also not processing any request[1]. We could require that TCP clients also be serialized this way, i.e., every TCP client must wait its turn before accessing *any* replicated component, for fear that concurrent, dependent TCP invocations might result in state inconsistencies within the replicated domain. Not only is this requirement restrictive, but it forces independent TCP clients to synchronize their actions and block on each other's operation, when the

[1]The anticipation is that a quiescent server is idle and not modifying its internal state; clearly this might be voided if we consider servers that might actively and asynchronously change their states even while not processing any request.

need for this might not exist. Thus, if we insert a mechanism such as a singleton to serve as the entry-point into the GCS domain, we should permit the singleton's bridging infrastructure to allow multiple, distinct TCP clients to pass through it simultaneously.

Of course, this only works under the assumption that all TCP clients are independent of each other. If dependencies arise, e.g., if clients A and B in Figure 1 dispatch invocations to $S1$ and these invocations modify some shared piece of state at $S1$, then, these operations cannot be allowed to execute concurrently and serialization of all TCP messages would be warranted. Note that it does not matter whether $S1$ is a singleton or a replicated server; as long as $S1$ resides within the GCS domain, it has the potential to influence the consistency of replicated components. Thus, the assumption of independent TCP clients and independent GCS clients described in Section 3.2.

Note that this assumption can be relaxed if state consistency can be guaranteed at the application level, i.e., if the application was programmed such that concurrent invocations do not share state or they share it safely without inconsistency side-effects. In that case, we could support concurrent *and* dependent TCP clients. However, in the absence of application-level state consistency guarantees for dependent invocations, our infrastructure needs to assume the worst-case scenario and prohibit dependencies.

Reconnection and failure scenarios. As an extension of our transparency objective, failures within the replicated domain should ideally be hidden from the unreplicated TCP components (because the latter have no concept of replicated semantics). This means that, if a replica of a component fails, the reconfiguration should be internal to the replicated domain and not be made visible to any TCP clients or servers.

3.2 Assumptions

In order to make progress towards addressing some of these challenges in a practical system implementation, we make some assumptions. We assume that the operations for both the replicated and unreplicated components of the applications takes place in a distributed asynchronous system supporting non-blocking server semantics such as multiplexed I/O[2]. This inherently results in the interleaving of requests and replies when using non-serialized communication patterns. MEAD's fault model for the replicated part of our distributed system comprises communication faults such as message losses, node-crash faults and process-crash faults. Arbitrary faults are outside the scope of the work described in this paper.

We also make specific assumptions about the CORBA applications that we support. We assume strictly synchronous request-reply operation on the part of the application, i.e., when a pure client sends a request to a server, the client blocks, waiting for a reply from the server, and unblocks only when the reply has been received. Note that the blocking semantics only refer to pure (tier 1) clients, and not to the client-side roles that a middle tier (tier ≥ 2) client+server component assumes for a multi-tier operation. While a pure client always blocks as a part of an end-to-end, multi-tier, synchronous operation, intermediate client-side tiers do not block because they might be processing multiple, simultaneous end-to-end operations. Focusing only on synchronous requests, we do not consider CORBA `oneway` operations, where a client dispatches a fire-and-forget message to the server, expecting only unreliable, best-effort semantics for message delivery.

For the singleton's infrastructural support in MEAD, we place maximum, yet adequate, bound on the number of incoming client connections. This is used to ensure a resource-effective server-side duplicate detection and suppression mechanism for concurrent messaging systems. To preserve state consistency for communication through the singleton, we also require TCP clients to be independent of each other, and of GCS clients.

We place no bounds on the number of replicated GCS-based servers or the number of tiers involved in the application. We expect that any kind of client, TCP-based or GCS-based, might need to communicate with the singleton, depending upon various deployment scenarios. In our current implementation, the singleton is not replicated (Section 4.4 discusses how to overcome this limitation).

[2]A non-blocking server can support simultaneous communication with multiple clients. At the communication level, this effectively causes the multiplexing of the server's dispatching and processing across I/O that it receives from multiple clients on a `select` system call. In contrast, a blocking server serializes requests that it receives, using a one-client-at-a-time and a one-request-at-a-time approach.

4. SYSTEM IMPLEMENTATION

Figure 2 shows the singleton, an application-level component that, through infrastructural support from the underlying MEAD system, can communicate transparently using both TCP and GCS messages. MEAD's infrastructural support for the singleton includes the capability for encapsulating (extracting) TCP messages into (from) GCS ones, and for performing duplicate detection and suppression.

4.1 The MEAD System

MEAD's various features, such as adaptive and proactive fault-tolerance and compensation for nondeterminism, are described elsewhere [5]. Only the details of MEAD that are relevant to the singleton concept are covered here. Section 4.3 discusses the specific architectural enhancements to MEAD in order to support the singleton.

MEAD exploits library interpositioning [4] to intercept a process' network system calls by using the dynamic linker's run-time support. Using the `LD_PRELOAD` environment variable, we load the MEAD system as a shared object library into the address space of each GCS-hosted component and of the singleton, ahead of all of the other dynamically linked system and application libraries. The MEAD library overrides some socket and network functions (such as `socket`, `connect`, `bind`, `read`, `write`, etc.) to perform the transparent re-routing of the CORBA application's IIOP messages over the Spread GCS [1].

4.2 The Singleton Concept

Our infrastructural-bridging approach differs from the previous work on gateways in that our singleton is not a separate process that adds one more layer of indirection. Instead, we identify the first application-level entry-point of a TCP component's communication into a replicated system and exploit that entry-point as an application-level bridge, without its knowledge. Returning to the example of Figure 1, an ideal candidate for the singleton in the invocation

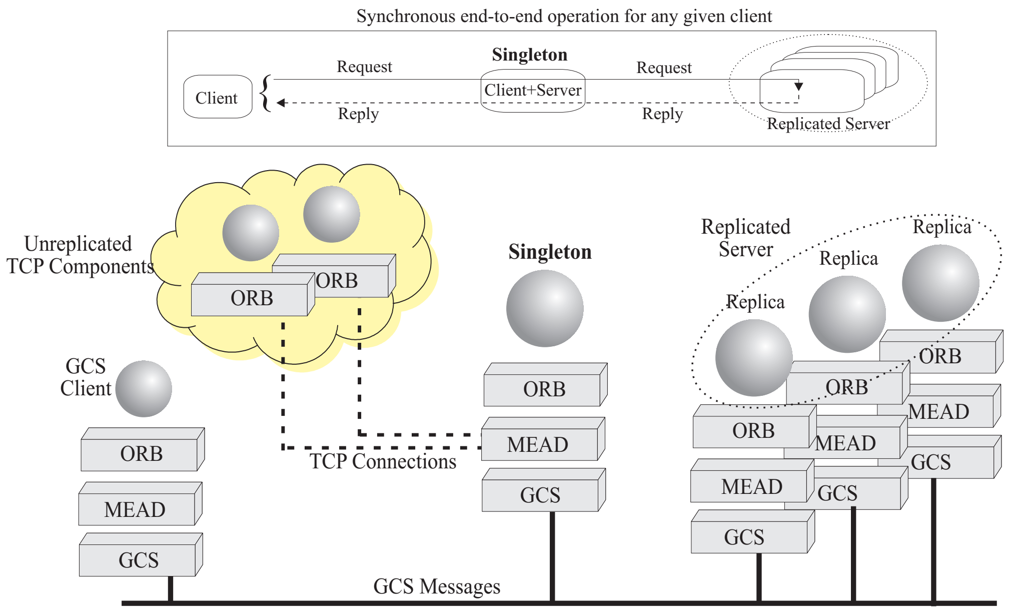


Figure 2: The singleton's bridging infrastructure spanning unreplicated and replicated components.

path $A \rightleftarrows S1 \rightleftarrows S2$ would be server $S1$, particularly if $S2$ is the critical component that must be replicated. Clearly, one issue is how to protect $S1$ against failures if $S1$ is just as critical as $S2$; we elaborate on this further in Section 4.4.

For the purposes of describing the singleton bridging, assume that we have selected $S1$ to be a singleton. We can consider $S1$ to be two-sided[3] because in its client role (when invoking the replicated server $S2$) it uses GCS, and in its server role it uses either TCP (when invoked by client A) or GCS (when invoked by client C). We can host a single, unreplicated instance (hence the term, *singleton*) of $S1$ over the MEAD system, and use its underlying MEAD infrastructure to perform the bridging between TCP and GCS semantics. Note that $S1$ is unaware of the fact that it is a singleton (just as any component within the GCS domain is unaware of the fact that it is replicated), and simply participates in end-to-end operations (such as $A \rightleftarrows S1 \rightleftarrows S2$ and $C \rightleftarrows S1 \rightleftarrows S2$) without any knowledge of which connection is TCP and which is GCS; in fact, due to MEAD's transparency, $S1$ will still continue to believe that all of its communication still occurs over CORBA's standard TCP-based protocol.

The MEAD way of performing the TCP-GCS bridging introduces additional considerations for the implementation of its infrastructure. The singleton that forms the bridge between unreplicated TCP clients and replicated servers must also support GCS clients, albeit in a mutually exclusive manner. It is perfectly possible for the end-to-end operations passing through the singleton to arise from either TCP or GCS clients. Returning to Figure 1, singleton $S1$ interacts with both TCP client A and GCS client C.

[3]To visualize this in Figure 1, pretend that $S1$ is unreplicated, and then move it to the left so that it is neither completely in the TCP domain nor in the GCS domain, but is located at the boundary between the two domains.

4.3 Bridging Mechanisms

Because MEAD operates at the network/socket level of the CORBA application (in user space, underneath the ORB, but not within the kernel), its operation hinges upon the sequence of network-level interactions that CORBA clients and server undergo during connection establishment, during normal communication of requests and replies, and during connection teardown. For synchronous CORBA client-server communication, the MEAD library underneath either the GCS client or server is typically driven off interrupt-driven I/O (through a `select` system-call with a timeout value set to `NULL`, indicating that call blocks until interrupted by a signal indicating incoming I/O) whenever it receives notifications of received GCS messages that are waiting for processing.

The singleton infrastructure is assigned to a separate group of its own, which allows it to communicate over the GCS protocols within the replicated domain. The singleton's addressing information (a group identifier for GCS access and an IP address and port number for TCP access) is made available to its GCS and TCP clients.

Because MEAD conveys the application's messages through GCS, any component supported by MEAD will have its connections funneled/mapped by MEAD onto a single socket connection to the local Spread GCS daemon. Effectively, when a CORBA client/server/singleton is supported over MEAD, there is only way into and out of that component through MEAD, and this "GCS channel" (accessed via a `SP_receive` blocking call on the Spread GCS API) supplies only totally ordered messages. As long as it delivers (to its hosted clients and server) only the ordered sequence of messages that it receives off this GCS channel, MEAD can ensure replica consistency because every MEAD-hosted server replica will "see" the same requests and replies.

Any connection from the TCP domain to the singleton will take the form of a separate socket connection at the MEAD infrastructure. These sockets are channels of communication that are distinct from MEAD's preferred GCS channel, into and out of the singleton. In fact, from MEAD's viewpoint, every TCP connection represents out-of-band communication with the singleton that can potentially compromise its consistency because the TCP messages (especially when there exist multiple TCP connections) are not totally ordered across the system. While we could opt to handle the TCP communication also in an interrupt-driven fashion, as we did with the GCS communication, the two sets of interrupts can become arbitrarily interleaved and make for tricky concurrent programming.

We chose the simpler and more controllable solution of using polling-based I/O for the TCP connections. Although this keeps the MEAD infrastructure in a "busy-wait" state, polling periodically for TCP-based I/O, this allows for the careful scheduling of when TCP communication can interleave with the GCS communication, through the appropriate adjustment of the polling frequency. Effectively, the file-descriptors associated with the TCP connections are inserted into a `select` system-call with a fixed timeout value; we note that this does change application semantics because a standard CORBA application (without replication or GCS involved) would use interrupt-driven I/O.

Because the GCS channel and the TCP sockets are managed separately by MEAD, the infrastructure underlying the singleton can differentiate between the two kinds of messages and handle the bridging seamlessly. Because the singleton must handle multiple TCP or GCS clients simultaneously, MEAD needs to perform concurrent duplicate detection-and-suppression across all of the various virtual end-to-end connections that pass through the singleton. For each incoming connection, MEAD maintains some book-keeping information, such as the last-seen request identifiers on that connection, in order to ensure state consistency.

4.4 Critique

Our current approach has limitations that are artifacts of our implementation and our transparent approach. As next steps in this research, we intend to investigate how we might overcome these limitations.

It is easy to see how the polling mechanism for the TCP connection (see Section 4.3) makes the singleton's performance sensitive to the value of the polling timeout. It is important to select effective timeouts when mixing interrupt-driven and polling-based I/O within the same infrastructure. We also need to take care to implement the polling-mechanism to prevent starvation and deadlock. We avoid this in our current implementation by requiring that all TCP and GCS clients remain independent of each other and not share resources; this is not likely to be realistic for many applications. In fact, carrying this further, it is also essential that all singletons in the system be independent of each other. Relaxing the strict requirement of independence of the various entities is an issue for further investigation.

All of the TCP connections must be polled by MEAD in turn to ensure that all of them are fairly serviced. We might, in the process, run the risk that any timing-sensitive TCP messages might be delayed and need to wait for their corresponding file descriptors to be polled before they can be serviced. Thus, it might be inappropriate to use the same timeout value across all TCP connections.

Concurrent duplicate detection-and-suppression comes at a cost; the amount of book-keeping information increases with the number of connected clients. There is an intrinsic performance vs. resource trade-off here: supporting concurrent clients requires more resources at the infrastructural level, but provides increased performance (unless the resource cost becomes so prohibitive that the singleton becomes a bottleneck).

The singleton, as described here, is admittedly a single point of failure in our current architecture. In future versions of our system, we intend to exploit cold passive replication and TCP failover to support fault-tolerant singletons. We also need to consider the state of the singletons to ensure consistent recovery.

5. EMPIRICAL EVALUATION

5.1 Configuration

Testbed. Our empirical evaluation was conducted in the Emulab distributed environment [11]. In our experiments, we used upto 10 nodes (850MHz processor, 256KB cache, 512MB RAM, running RedHat Linux kernel 2.6.11-1.27_FC3) to host the various components of the application and our MEAD infrastructure (version 1.5) with its underlying Spread group communication protocol (version 3.17.1) [1]. Our CORBA test application was built by using the TAO ORB (ACE version 5.4.6 and TAO version 1.4.6) [9].

Test Application. We used a simple three-tier distributed CORBA application. The pure client in tier 1 invokes the server+client in tier 2 (also the singleton tier), passing in a `long` integer as a part of the request. The singleton tier 2 invokes the pure server in tier 3, again passing in a `long` integer. In processing this request, the pure server in tier 3 increments a local counter and sends the counter's value in its reply to tier 2. In processing the reply received from tier 3, the tier 2 singleton adds the return value in the reply to its own incremented local counter; it then returns the value of its local counter in its reply to the client in tier 1. While MEAD supports both active and passive replication styles, we chose to actively replicate the back-end tier 3 – this means that all of the replicas of tier 3 simultaneously service all of the requests and responses intended for tier 3. Because the back-end tier is a pure server in our test application, in our graphs and the text below, "x servers" implies "x replicas of server in tier 3".

While this test application is relatively simple, it serves its purpose in exercising the three tiers for our experiments.

Metrics. We focus on the end-to-end round-trip time (RTT) under both fault-free and faulty conditions. By comparing RTT across various scenarios, we gain some idea of the overhead of our infrastructure for those scenarios. The piecewise decomposition of the RTT across the tiers allows us to understand which tier contributes most to the overhead.

5.2 Experiments

Scalability. In this case, we perform separate experiments for TCP and GCS clients, increasing the number of clients from 1 to 5 in each case. We also increase the number of tier 3 replicas from 2 to 4. This examines the singleton's scalability to the number of TCP or GCS clients and the number of server replicas. We compute the average RTTs over 2500 end-to-end invocations, discarding the warm-up period (the first 100 invocations) and focusing on steady-

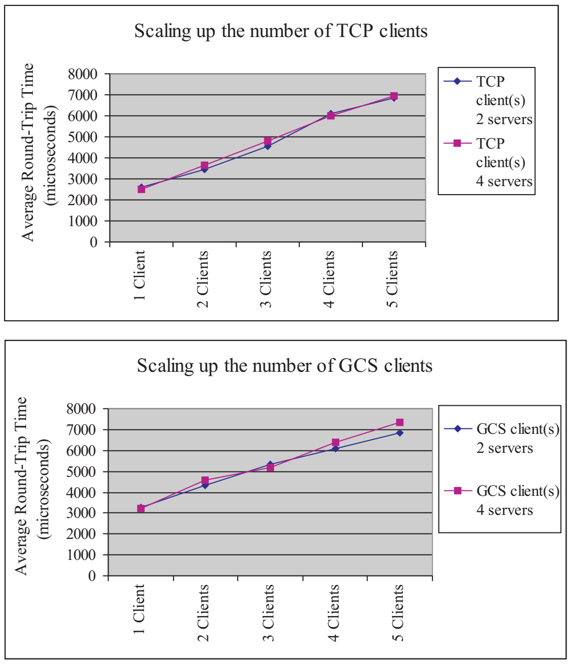


Figure 3: Scalability of the singleton's performance in supporting varying numbers of TCP clients, GCS clients, and server replicas.

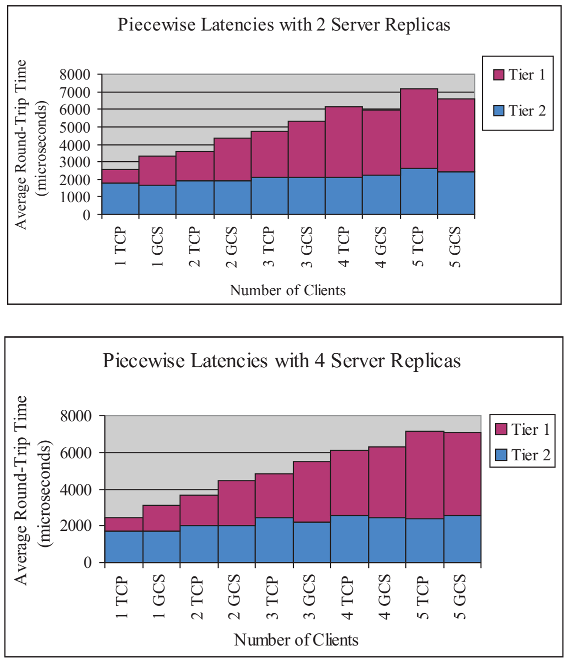


Figure 4: Piecewise decomposition of the RTT across tier 1 (client-to-singleton) and tier 2 (singleton-to-server) for varying numbers of TCP clients, GCS clients, and server replicas.

state performance. These results are shown in Figure 3.

Piecewise behavior. We perform the scalability experiments again to measure the breakdown of the RTT in terms of its contributions from tier 1 latencies (client-to-singleton traffic) and tier 2 latencies (singleton-to-server traffic). We measure the RTT from the client and we also instrument the singleton to measure the round-trip latency for the request-reply pair between the singleton and the replicated server. While our measurements are slightly skewed by the presence of the timing instrumentation at the singleton, the overall trends are still visible, as shown in Figure 4.

Fault injection. These experiments measure how faults in the replicated domain impact the performance of the singleton, with TCP and GCS clients. In all cases, the back-end tier 3 server has two replicas. We inject replica-crash failures at various points in the end-to-end path of an invocation from either a single TCP or GCS client. As shown in Figure 5 and Figure 6, the fault-injection points include: (i) case 1: application failure, i.e., before the server returns a response, (ii) case 2: when MEAD executes a `writev` call, (iii) case 3: when MEAD executes a `read` call, (iv) case 4: before MEAD executes an `SP_receive` to extract a message from the Spread GCS, and (v) case 5: after MEAD has executed a `SP_receive` call.

We take the average of the RTTs over 10 fault-injection runs of 500 consecutive end-to-end invocations, discarding the warm-up oeriod (the first 100 invocations), and injecting one of the failures in (i)-(v) at the 250th invocation.

5.3 Observations

As we add clients, early performance gains from TCP clients seem to diminish marginally, leading to similar times and trends observed for increasing numbers of TCP and GCS clients. It is not clear whether this trend will continue to manifest for very large numbers of clients. For one, GCS clients will be limited by the scalability of their underlying group communication protocols; frequently, as in the case of the Spread GCS, these protocols are based on logical token rings and their internal timeouts and configuration parameters are carefully tuned for specific bounds on the number of nodes involved in the protocol. TCP clients will not be exposed to the GCS subtleties directly, but might face them indirectly if their end-to-end invocations have a GCS component in their paths; however, the effect is likely to be more pronounced for GCS, rather than TCP, clients of the singleton. We note that increasing the number of server replicas does not significantly affect the RTTs perceived by the GCS and TCP clients; again, we have not tested the system for a high degree of server replication, where GCS scalability characteristics might manifest themselves.

From a piecewise-RTT perspective, the tier 2 (singleton-to-GCS) costs seem to be fairly constant, for both TCP and GCS clients and for increased degree of server replication, as seen in Figure 4. For increased number of TCP or GCS clients, the dominant increasing factor appears to be the tier 1 (client-to-singleton) costs. This contribution of the tier 1 cost to the overall RTT appears to increase proportional to the number of GCS or TCP clients.

From the fault-injection experiments, the TCP clients seem not to be as affected by the replica-crash failures as the GCS clients. This is likely due to the fact that the GCS clients participate in the GCS protocol and experience, indirectly, the effects of any GCS reconfiguration due to membership changes (i.e., a replica leaving the group upon failing). Look-

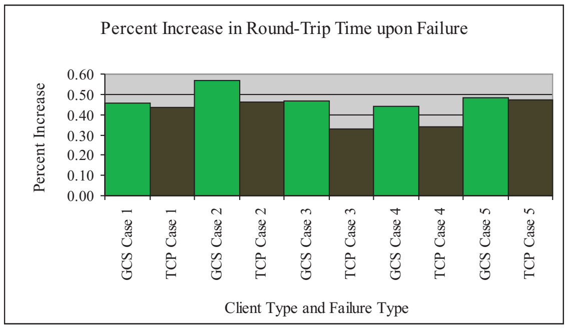
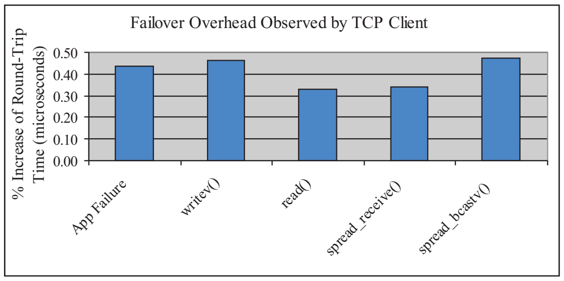


Figure 5: Percentage increase in RTT upon fault injection at various points (labeled cases 1-5) in the end-to-end execution.



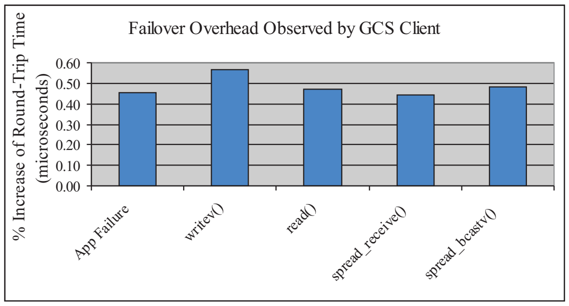


Figure 6: Impact of replica-crash failures at various fault-injection points, for TCP and GCS clients.

ing across the various fault-injection points, although there is some variation across the overheads, there is not one single fault-injection point that stands out as being the worst case in terms of failover overhead.

We note that we did observe a few significantly high outliers in our experimental runs, and that we eliminated these in computing the average RTT numbers reported in the graphs. Fortunately, the outliers constituted a small fraction of the measurements – the percentage of outliers was less than 0.5% across all experimental runs.

6. CONCLUSION

This paper describes the challenges that exist in supporting true end-to-end consistency of multi-tier CORBA applications that span replicated and unreplicated components. We introduce the concept of a singleton and the MEAD bridging infrastructure that exploits the multi-tiered nature of these applications to combine unreplicated and replicated components. From the empirical evaluation of the performance of the singleton, we demonstrate reasonable overheads under both fault-free and faulty conditions at the replicated servers, and also under scaling of the numbers of TCP and GCS clients that the singleton can support. We outline in Section 4.4 how we propose to overcome some of the limitations and challenges that remain.

7. ACKNOWLEDGEMENTS

We acknowledge Joe Loyall, Paul Rubel and Matt Gillen of BBN Technologies for their valuable suggestions and discussions in developing this concept.

8. REFERENCES

[1] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 327–336, New York, NY, June 2000.

[2] P. Felber. Lightweight fault tolerance in CORBA. In *International Conference on Distributed Objects and Applications*, pages 239–250, Rome, Italy, September 2001.

[3] P. Felber and P. Narasimhan. Experiences, approaches and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 54(5):497–511, May 2004.

[4] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[5] P. Narasimhan, T. A. Dumitraş, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[6] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Gateways for accessing fault tolerance domains. In *Proceedings of Middleware 2000, Lecture Notes in Computer Science 1795*, pages 88–103, New York, NY, April 2000.

[7] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. OMG Technical Committee Document formal/2001-12-01, December 2001.

[8] Object Management Group. Fault Tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.

[9] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998.

[10] 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.

[11] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.