Rectifying Orphan Components using Group-Failover in Distributed Real-time and Embedded Systems.*

Sumant Tambe
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
sutambe@dre.vanderbilt.edu

Aniruddha Gokhale
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
a.gokhale@vanderbilt.edu

ABSTRACT

Orphan requests are a significant problem for multi-tier distributed systems since they adversely impact system correctness by violating the exactly-once semantics of applications and may waste resources. Orphan requests stem from the failure(s) of non-deterministic components involved in nested invocations of replicated components. Resolving this problem in the context of resource constrained, component-based, distributed real-time and embedded (DRE) systems that form end-to-end task chains is challenging because conventional transaction-based solutions cannot assure real-time properties of the DRE applications. To address these challenges, this paper presents a group-failover protocol that comprises three key capabilities: real-time failure detection and client failover, timely mitigation of orphan requests, and two novel application state consistency strategies to ensure the correctness of DRE systems by maintaining the exactly-once semantics even during failures. Our solution is implemented in the context of the CIAO real-time CORBA Component Model middleware. Empirical evaluations of the group-failover protocol in both fault-free and failure recovery scenarios for DRE task chains of different sizes demonstrates a low overhead and predictable performance.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*soft real-time, fault-tolerance, availability, components*

General Terms

Algorithms, Reliability, Performance

*This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

Keywords

Soft Real-time and Fault tolerance, Component-based systems

1. INTRODUCTION

Component-based technologies are gaining prominence in distributed, real-time, and embedded (DRE) systems found in domains, such as shipboard computing [1] and avionics mission computing [2], since they facilitate reuse and expedite system development. The end-to-end service specifications of these applications include mostly soft real-time requirements in the form of predictable response times. The system structure can be modeled as *end-to-end task chains* [3] of software components where the application behavior and state is divided among these components. The mission criticality of such systems require them to be reliable and highly available, which imposes the need for replication of the end-to-end task chains.

The componentized structure of DRE systems often results in nested invocations among the components to service client requests. The combination of replication and nested invocations within the end-to-end task chains, however, may result in potential undesired side-effects of *replicated invocations,* which are (nested) requests from a replicated server to another (possibly replicated) server [4]. These undesired side-effects arise when the components are stateful, and one or more components in the end-to-end task chain are non-deterministic. In DRE systems it is reasonable to expect the presence of stateful components that admit several forms of non-determinism [5] stemming from local information (*e.g.,* sensors and clocks), timers and timeouts, multi-threading (*e.g.,* dynamic scheduling, preemption), load-balancing, time-dependent sensor calibration, and program constructs, such as true random number generators.

Consequently, a key undesired side effect called the *orphan request* [4, 6] problem arises when one or more of the non-deterministic, stateful components in the replicated invocations fail. Intuitively, an orphan request is a request received by a component that is no longer valid due to failure of the invoking, non-deterministic component. In such a case, it is not guaranteed that the reinvocation of the request by a replica of the failed non-deterministic component will lead to the same nested invocation thereby leaving the earlier partially completed request an orphan.

Figure 1 shows how failure of a non-deterministic component causes an orphan request in another component. Assuming passive (*i.e.*, primary-backup) replication [7], the client invokes an operation on the primary replica of com-

ponent A (shown as Replica 1), which in turn calls another component B. The primary replica of the component A, however, crashes before returning the reply to the client. The client reinvokes the request on the backup replica of component A (shown as Replica 2), which is promoted to primary.

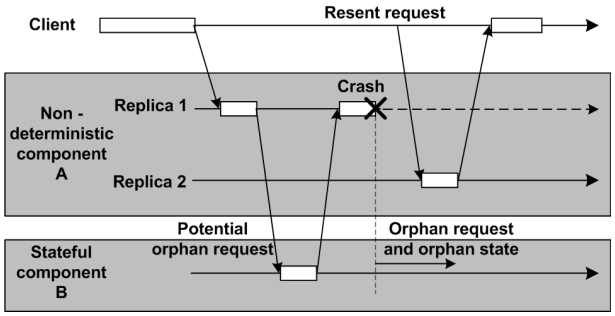

Figure 1: An orphan request caused by the failure of non-deterministic component A

Since the backup replica of A is also non-deterministic, there are three possible behaviors when the backup replica of A is promoted to a primary: (1) it makes no invocation, (2) it makes an invocation with different parameters, and (3) it makes an identical invocation on component B. The lattermost behavior, however, is not guaranteed at all times. If the promoted backup of component A makes no invocation on component B, the earlier request that is continuing in the system (as shown in the figure) is considered an orphan, and the affected components become orphan components. If an invocation with different parameters is made, then the state of component B may become inconsistent with respect to the rest of the system.

Neither of these conditions is permissible in DRE systems because system correctness is adversely affected. The root cause of this problem is the violation of the exactly-once semantics of request execution by the non-deterministic component, which is not allowed for mission-critical DRE systems. Simple caching [8] of the request and reply may not be applicable since the re-invocation of the request after failure recovery by the non deterministic component may not be identical thereby rendering the cached values useless.

The orphan request problem has been addressed [4,6,9–11] for enterprise systems by integrating replication and transactions. However, the use of transactions becomes a significant source of overhead in the critical path for most real-time systems causing degradation in response times to clients. Solutions based on enforcing determinism [12–15] target only a small subset of the sources of non-determinism out of many that are possible in contemporary real-time systems. Moreover, their use is limited since they require access to the source code.

In this paper we present a novel approach to resolving the orphan request (and hence the orphan component) problem for DRE systems that simultaneously meets their real-time, fault-tolerance, and system correctness requirements. We present a *group-failover* protocol that supports exactly-once execution semantics despite the presence of replicated invocations and non-deterministic stateful components. The protocol ensures that replicated data is both consistent (strong state consistency) and timely. We present two variations of the protocol for state consistency that are optimized for either the fault-free or failure scenarios. DRE system designers can easily program the group-failover protocol semantics in their applications since they do not need to implement complex application-specific *prepare*, *commit*, and *rollback* methods found in conventional transactional systems. Instead, our middleware implementation that is built on top of the CIAO [16] Lightweight CORBA Component Model (LwCCM) [17] middleware supports these capabilities out-of-the-box.

The work presented in this paper differs from our earlier work on FLARe [18] and CORFU [19]. FLARe provides real-time, fault-tolerance to client-server applications and not to end-to-end component-based systems. Moreover, FLARe provides only weak state consistency. CORFU exposes FLARe's capabilities at the component abstraction level thereby allowing end-to-end task chains. However, CORFU does not handle the orphan request problem. Nevertheless, our work leverages the infrastructure provided in FLARe and CORFU, which themselves are part of the CIAO framework.

The rest of the paper is organized as follows. Section 2 presents related work describing why existing approaches to solving the orphan request problem are not suitable for real-time systems; Section 3 presents the group-failover protocol and its two variations for strong state consistency; Section 4 describes our implementation of the group-failover protocol; Section 5 evaluates the protocol and shows its suitability for DRE systems, and finally Section 6 concludes the paper.

2. RELATED RESEARCH

We categorize existing research on addressing the issues of non-determinism in replicated invocation in two categories. The first set of work admits non-determinism but compensates for the side effects of replicated invocation using transactions. The second set of work focuses on enforcing determinism that avoids the side effects.

2.1 Integrated Transaction and Replication

Replication and transactions are two separate techniques for achieving fault-tolerance in reliable systems. Replication represents *roll-forward* recovery where an incomplete request is re-executed at another replica. Conversely, transactions represent *roll-back* recovery where a failed parent transaction forces undo of all the sub-transactions no matter their outcome. The two reliability mechanisms are different in that the former protects processing whereas the latter protects data to ensure system consistency.

Note that transactions provide *all-or-nothing* (atomicity) guarantees whereas replication provides *at-least-once* guarantees as long as there are fewer failures than the available replicas. Neither provides *exactly-once* guarantee, which is stronger than both and is needed in DRE systems for system correctness. Therefore, the solutions that depend on both replication and transactions to provide *exactly-once* semantics must integrate the two services in non-trivial ways.

Felber *et al.* [10] reconcile CORBA's transaction service (OTS) [20] and the replication service (FT-CORBA) [21] to protect both data and processing to provide consistent end-to-end reliable operation. Their approach restarts execution of a failed sub-transaction on a backup and aborts sub-transactions where a parent has failed. This reconciliation, however, does not handle the intricate details of transaction completion in failure scenarios.

Pleisch *et al.* [4] address the shortcomings in [10] by providing two alternatives to handle non-determinism: one pessimistic and one optimistic. The pessimistic approach forces the subtransaction to wait for the commit of the parent, while the optimistic approach allows subtransactions to commit before its parent. Information about how to undo the changes is sent to the backups before making the nested invocation. In the pessimistic case, orphan subtransactions are aborted whereas in the optimistic case they are compensated by *undo* transactions.

Frølund *et al.* [9] present an approach to integrate replication and transactions for three-tier applications. However, their approach is limited to stateless middle-tier servers, where all state is forced to the end-tier databases. ITRA [11] handles the side effects of replication by propagating the result of each non-deterministic operation to the backups. ITRA supports replicated transactions by replicating the start, join, prepare, commit, and abort operations.

Kolltveit *et al.* [6] present an approach where a passively replicated transaction manager is allowed to break replication transparency to abort orphan requests thus handling non-determinism. The transaction manager must be aware of the replication and be able to see the individual replicas of the transaction participants instead of treating them as an opaque group.

Despite the elegant solutions described above, a combination of transactions with replication causes significant overhead in the critical path for most DRE systems thereby adversely impacting their real-time properties. A detailed description of the overhead is presented in Section 3.2.

2.2 Enforcing Determinism

Considerable research efforts have been expended in designing strategies to enforce replica determinism and to circumvent certain sources of non-determinism. Slember *et al.* [12, 13] apply program analysis to discover the sources of non-determinism. They target the instances of non-determinism that can be compensated automatically and highlight the remaining instances that must be manually rectified. The work on deterministic scheduling algorithms [14, 15] handles the non-determinism of multi-threading. A deterministic schedule is ensured either by multicasting the scheduling decisions to the replicas or by assuming shared state between all threads of the same replica. The fault-tolerant real-time MARS system [22] requires deterministic behavior in highly responsive automotive applications which exhibit non-determinism due to time-triggered event activation and preemptive scheduling. Replica determinism is enforced using a combination of timed messages and a communication protocol for agreement on external events.

Despite resolving the challenges stemming from non-determinism, the approaches outlined above target only a small subset of the sources of non-determinism out of many that are possible in contemporary real-time systems. The automated source code analysis approach may not be applicable when only component binaries are available. Some approaches cannot be completely automated whereas others add significant overhead in the critical path due to the need to communicate the non-deterministic decisions.

Finally, timestamp-based orphan elimination techniques [23, 24] have been developed to remove crash and abort orphans from the system. Closely synchronized real-time clocks are required for timely elimination of orphans. These algorithms also incur additional messaging overhead of periodic system-wide *refresh*. This overhead is similar to the periodic garbage collection phase in [4].

3. RECTIFYING ORPHAN COMPONENTS USING GROUP-FAILOVER

We now present our approach based on a group-failover protocol to rectifying orphan components in DRE systems. Note that orphan computations are an instantaneous consequence of a failure in a non-deterministic middle tier of an end-to-end task chain. Before the failure, the requests sent by middle tier components are *potential* orphan requests (as shown in Figure 1). Upon failure of the non-deterministic component making the nested invocation, however, they instantaneously become orphan. Therefore, orphan computations cannot be prevented; instead they must be rectified because they waste resources, may hold locks thereby delaying other computations, and may lead to system inconsistencies.

3.1 System and Fault Models

To set the context to describe our contribution, we first describe the system and fault model assumed in this work.

System Model. We consider systems in which applications are composed of multiple tiers (*n-tier*) of components/processes that communicate over a network of computing nodes. All or a (non-null) subset of the components may be non-deterministic and may maintain volatile state across multiple client invocations (*i.e.*, a session). The services in the system are invoked by clients periodically via remote operation requests. Every client expects bounded response times from its services, *i.e.*, services have soft real-time deadlines, which if missed, reduces the value to the client gradually down to zero. If a deadline is missed (say, due to failures), earlier completion of the request has higher value to the client than later completion.

More formally, the nested invocation of components can be modeled as end-to-end task chains [3]. To satisfy the end-to-end response time of clients, the end-to-end schedulability [25] of the computing resources is ensured based on the deployment of components. Further, we do not allow component sharing across multiple services of DRE systems. Component sharing complicates system scheduling and deployment planning. Moreover, failures of a shared non-deterministic component may result in orphan state in more than one service at a time, which complicates recovery and correctness in mission-critical DRE systems.

We assume that the application hosting middleware has access to the deployment and composition metadata of the components, and that the deployment and configuration of the end-to-end task chains in the DRE systems is handled through our prior work on CORFU [19].

Communication Semantics. We assume that networks provide bounded communication latencies and do not fail or partition (synchronous environment). This assumption is reasonable for many soft real-time systems, such as Supervisory Control and Data Acquisition (SCADA) systems, where nodes are connected by highly redundant high-speed networks. Due to the synchronous environment, perfect failure detection is possible that bounds the delay for failure detection and eliminates false suspicions as described in [26]. Moreover, we also assume that requests made by the client

are received in FIFO order. This assumption is justified given the periodic model of client requests.

Fault Model. Processors and processes hosting the components may experience fail-stop [27] failures. The passive replication (primary-backup) [28] approach is used for high-availability and roll-forward recovery because it tolerates non-determinism better than active replication and consumes less resources compared to other strategies. All these properties are highly suited for DRE systems.

The state updates in the primary are *transmitted* to the backups upon completion of each request. In contrast to traditional primary-backup schemes where a backup replica immediately updates its state on receipt of state from a primary replica, in our scheme the state updates are *pushed* in the backups only when dictated by the group-failover protocol, which is described in Section 3.3.

A minimum of $f + 1$ replicas will be required in a given tier to tolerate f replica failures. We leverage the implementation of failure detectors based on per-node daemons and periodic *heart-beat* beacons from our prior work in FLARe [18]. If a replica crashes and restarts, it joins the group of backups. Maintaining the necessary quorum of replicas is not the focus of this paper, and we assume external mechanisms are available that address this requirement.

3.2 Understanding the Sources of Overhead in Transaction-based Techniques

Although conventional transaction-based approaches resolve the orphan request problem, they are unsuitable for DRE systems because they cause significant messaging and synchronization overhead in both fault-free and failure recovery scenarios. Empirical evaluations in [6] confirm that the response time may suffer up to 200% increase in the fault-free case whereas client-perceived failover delay could vary from 200 to 400 ms. Contemporary DRE systems often have more stringent performance requirements. Below we describe the sources of overhead in the solutions that combine replication with transactions.

(1) A server must initiate a transaction by sending a *create* message to the transaction manager;
(2) Every object that participates in the transaction must register itself with the transaction manager using a *join* message;
(3) Non-deterministic components must transfer undo information to their replicas to either *abort* or *commit* the subtransactions in case of a failure;
(4) The server must finish the transaction using a *commit* message;
(5) While finishing a transaction, the transaction manager initiates a two phase commit (2PC) protocol that sends *prepare* messages to all the transaction participants in the first phase, which if acknowledged positively, sends *commit* messages to all the participants in the second phase;
(6) Each participant object sends its vote to the replica before sending it to the transaction manager; and finally,
(7) the reply is sent to the client only when the transaction manager indicates successful completion of the 2PC to the initiating server.

In case of a failure, the orphan components are eliminated by sending *abort* messages (or compensating transactions) to every orphan. Moreover, the steps for fault-free scenario must be repeated to re-execute the aborted sub-transactions. Note that with the increase in the number of tiers in the system, the number of orphans that must be eliminated increases. Clearly, the client has to block during this time and may miss the deadline during recovery.

Some optimizations are possible in the related work cited earlier. For instance, in the optimistic approach of [4], 2PC is not required whereas in the pessimistic case, only the second phase of 2PC is sufficient. In [6], there is no need to send undo information to the replicas, instead, votes must be synchronized with the replicas. The transaction manager must be extended to support join messages with *view-id* of the underlying group communication system thereby losing the transparency of replication. Additionally, all the above approaches also require objects to implement *prepare*, *commit*, and *rollback* methods to participate in the 2PC.

3.3 The Group-Failover Protocol

We now develop the intuition behind our group-failover protocol. The key observation we make is that the orphan components resulting from a failure of a non-deterministic component often form a group. For instance, consider a nested invocation among stateful components A, B, C, and D as shown in Figure 2. Just before the non-deterministic component A returns the reply to the client, suppose A fails, which renders components B, C, and D orphans and they form a group. Intuitively, the orphan request problem can be rectified in real-time by discarding the group of orphan components, and letting the client failover to the replica group. Note that because of the non-deterministic behavior of one or more components in the nested invocation, the resulting state updates in the components are unique to that particular execution, and hence application state belonging to the orphan group too must be discarded. Such an intuition lies at the core of our group-failover protocol.

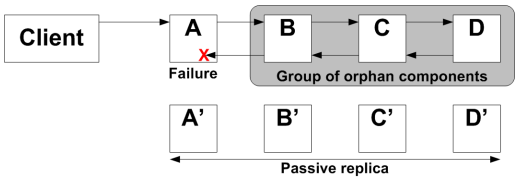

Figure 2: A group of orphan components (B, C, and D) due to failure of a non-deterministic component A

To rectify and eliminate the group of orphan components (and requests and state) while simultaneously ensuring the exactly-once execution semantics, three actions are required to be performed. Moreover, all of these actions must be performed in bounded time to suit DRE systems.

1. **Accurate and real-time failure detection and transparent failover.** To ensure forward recovery, the client must detect the failure in real-time, transparently failover to the replica of component A (A'), and must quickly resume execution of the original request to deliver acceptable response times to clients.

2. **Real-time identification and elimination of orphans:** The orphan state (and any unfinished orphan computations) must be eliminated to avoid inconsistencies in the system and conserve resources. In Figure 2 the orphan state is bounded in the group of primary components B, C, and D.

3. **Ensuring state consistency in bounded time:** Exactly-once semantics require that the state of all the components that participate in the re-execution of the client's request (*e.g.,* A', B', C', and D') must be the same as the state of the respective primary components before the execution of the request that caused the failure. These assurances must be provided in a timely manner.

The rest of this section describes how our group-failover protocol meets these requirements. For bounded-time state consistency, we present two novel state synchronization strategies: *eager* and *lag-by-one* that are tailored to suit different operating conditions and DRE system requirements. The eager state synchronization strategy performs extra work during fault-free execution so that failure recovery is instantaneous. On the other hand, in the lag-by-one state synchronization strategy no additional overhead exists for state synchronization during fault-free execution, however, recovery now takes longer than the eager strategy.

3.3.1 Failure Detection and Transparent Failover

To detect process/processor failures and to initiate recovery actions, we use a monitoring infrastructure consisting of dedicated *host-monitor* daemons in every node where the components are deployed. The daemons send a periodic heart-beat to a central *replication manager* (RM) process. In case of a server process failure, the daemons notify the RM about the failure whereas processor failures can accurately be detected by RM when periodic heart-beat from a host-monitor ceases. Upon detection of a failure, RM initiates the process of identifying orphans.

Most contemporary middleware, such as CORBA support a standard, client-side request interceptor, which enables interception of the call-path at the client-side. User-supplied code can be executed in response to various events such as remote function call return or exceptional return (*e.g.,* standard CORBA exceptions, such as OBJECT_NOT_EXIST, COMM_FAILURE). The code may also invoke other remote services to obtain a failover replica reference. A standard CORBA exception called LOCATION_FORWARD is raised locally, which then redirects the client to the failover target replica. Client-side request interceptors are used for the group-failover protocol because they can be portably implemented for standard-compliant CORBA clients. Note that programmers do not need to implement the interceptors. Even the logic can very well be generated as shown by our prior work in GRAFT [29]. The only requirement is to integrate them in the build process of the client executable.

3.3.2 Identifying Orphans

A failure of the non-deterministic component orphans the components it has directly or indirectly communicated with during the execution of a remote invocation. The number of orphan components (and in turn the spread of the orphan request and orphan state) varies depending upon where the non-deterministic component lies in the nested invocation and how many components have executed computations in response to the direct/indirect requests from the non-deterministic component.

Note that the group-failover protocol does not use *join* messages as in the conventional transactional approach to avoid their overhead in the critical path. Consequently, the group-failover protocol has no run-time information about the stage of a nested invocation. For instance, when components A, B, C and D in Figure 2 communicate in response to the client's invocation, an independent observer cannot pinpoint how far the execution has progressed (among A, B, C, and D) without instrumenting[1] each component and incurring additional messaging overhead. Therefore, when the non-deterministic component A fails, the span of the orphan state cannot be determined accurately at run-time. The possibilities vary from no orphans at all to all three (B, C, and D) rendered orphan. To overcome this inherent difficulty, we provide two static strategies to identify the span of orphan components and thereby confine the spread of the orphan request.

(A) The entire end-to-end task chain strategy: This strategy is the most pessimistic of all and designates the entire end-to-end task chain as orphan in case of a failure. The strategy does not need component-specific knowledge of whether they are deterministic or non-deterministic. Pessimistically, it considers the entire group of components participating in an end-to-end task chain as an atomic *failover unit* (FOU). If any one of them fails, the whole unit is considered failed.

These failure atomicity semantics are desirable in systems that employ N-version programming [30] to achieve reliability though diversity. N-version programming is an effective technique to avoid *Bohr-bugs*, which predictably repeat themselves when the same set of conditions reappear in the system. To avoid Bohr-bugs, end-to-end task chains often use non-identical replicas of the fault-tolerant, multi-tier applications. The replicas could be dissimilar in many ways, such as structure, implementation, deployment, resource and QoS requirements, end-to-end deadlines, and priorities. Due to the non-identical replication, the client fails over to a replica of the entire end-to-end task chain while the failed end-to-end task chain recovers from the failure.

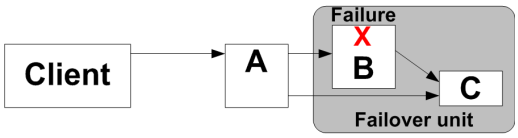

Figure 3: A failover unit spanning two components (B and C).

(B) Dataflow-aware component grouping strategy: Accurate meta-information about system composition, such as dataflow and component behavior (deterministic/non-deterministic) can easily be leveraged to optimize orphan elimination. For instance, consider Figure 3, which shows the dataflow between components A, B, and C where component B is non-deterministic. Failure of component B may make component C an orphan but not component A because the data in component A is not dependent on component B. Therefore, the span of the failover unit should include components B and C only.

[1]Network-level packet sniffers such as Wireshark can detect messages without intrusive modifications to the components but cannot guarantee that the captured messages are in response to a specific invocation from the client.

3.3.3 Eliminating Orphans through Component Lifecycle Operations

Orphan components (and requests and state) must be eliminated from the system to prevent inconsistencies and to conserve resources. As noted earlier, the orphan state is bounded inside a group of orphan components. In the group-failover protocol, this group of orphan components is simply *discarded* to achieve predictable recovery time. The group-failover protocol uses component passivation mechanisms supported by conventional component middleware for component lifecycle management as the way to eliminate orphan state and computations from the system. For instance, *activate* and *passivate* are two life-cycle operations supported by all LwCCM session components. Passivating a component discards all its application-specific state as well as middleware state (it is no longer remotely addressable and can no longer initiate remote invocations).

3.3.4 The Eager State Synchronization Strategy

The eager strategy is a variation of the atomicity achieved using the Two-Phase Commit (2PC) algorithm, however, it is optimized to support the QoS demands of DRE systems. In the eager strategy, the state of all the nested components is synchronized with their respective replicas only after the client-side interceptor has received the result of the non-deterministic computation. The state that builds up at the server-side during the execution of the client's request is a *potential orphan* state because any subsequent failure in the *upstream*[2] component renders these state changes orphan. Therefore, it is only after the client-side interceptor has received the reply, that the server-side state changes can be made permanent in the system.

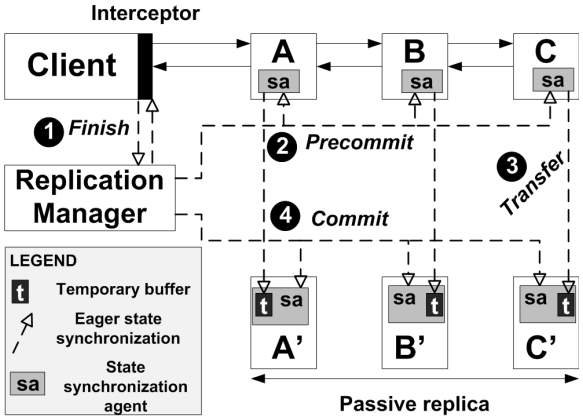

Figure 4: The eager state synchronization strategy

Figure 4 shows how the eager strategy works. A client is shown requesting a service from a non-deterministic set of components A, B, and C. Consider that the k^{th} periodic request from the client has completed and that the reply has arrived at the client-side interceptor. Although the execution has completed, replica components (A', B', and C') do not yet have the state updates from the k^{th} execution. The client-side request interceptor detects successful execution and before returning to the client application (so that it can make the next periodic request), initiates the eager state synchronization by sending a *finish* message to the RM.

In response to the *finish* message, the RM initiates a two phase process to synchronize the state atomically. The RM first sends a *precommit* message to each *state synchronization agent* (*sa*) located in the server processes hosting the primary components using asynchronous messaging (*e.g.*, AMI in CORBA). The intent of the precommit message is to persist[3] the application-specific state by sending it to the replicas. The state is retrieved by using predefined standardized interfaces, such as *get_state* implemented by the primary components. The state synchronization agents collocated with the primary components *transfer* the state to the state synchronization agents in the process hosting the replica components. Instead of accepting the state permanently at the replica, the state is maintained instead in a temporary buffer at the *sa* until the second phase.

The RM then initiates the second phase only if the first phase completes successfully. In the second phase, RM sends a *commit* message to every state synchronization agent collocated with the replica components using AMI. In response to the commit message, the state in the temporary buffers is pushed in the replica components by means of a predefined standardized interface, such as *set_state*. When both the phases complete, the RM returns a status indicating success to the client-side interceptor thereby allowing the client's application logic to process the reply.

In case of a failure, the orphan state is eliminated as described in Section 3.3.3. Naturally, a client's request must be reexecuted on replica components with the state of the last successful execution. The eager state synchronization strategy ensures that the state in the replicas is indeed from the last successful execution. The RM has the responsibility of maintaining state consistency. State from the last successful execution is maintained at the replicas by ensuring atomic state synchronization. Either all the primary components synchronize their state with respective replicas or none at all. This atomicity is guaranteed by the variation of the optimized 2PC protocol we designed.

If any primary component is unable to send the state to its replicas, the RM detects the failure during the first phase, skips the second phase, eliminates the orphans as described in Section 3.3.3, and returns an error value indicating failure to the client-side interceptor. The client-side interceptor does a transparent failover to the replica group and reinvokes the same request to ensure that there is exactly-one non-deterministic execution of the request. Note that a failure of the second phase (say, due to a failure of a replica) does not affect state consistency because primary components can serve the subsequent requests. Failed replicas may be restarted to maintain a desired replica quorum.

3.3.5 The Lag-by-one State Synchronization Strategy

Our second strategy called the *Lag-by-one* is an optimization of the eager strategy, wherein no overhead is incurred during the fault-free execution but pays a slight penalty in terms of recovery messaging overhead in the failure scenario. We call this technique lag-by-one state synchronization strategy because the state in the replicas is always lagging by exactly one state update compared to the pri-

[2]Upstream with respect to the dataflow.

[3]Enterprise systems often use *write-ahead* logging to achieve the same effect.

mary components. A schematic of the lag-by-one strategy is shown in Figure 5.

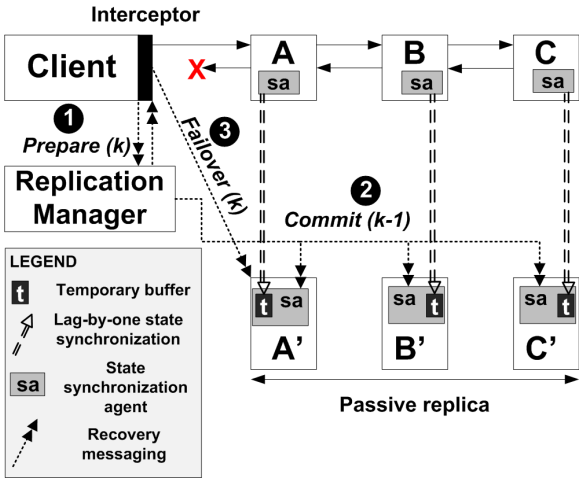

Figure 5: The lag-by-one state synchronization strategy

The lag-by-one strategy eliminates the need for explicit two phases of state synchronization found in the eager approach. Instead, the potential orphan state is transferred to temporary buffers of the state synchronization agents collocated with the replica components immediately after the completion of a request at every primary component. This feature is shown in Figure 5 by double-dashed lines. The state synchronization in this approach is initiated lazily (using AMI) immediately after sending the reply back to the calling component thereby eliminating the overhead in the critical path. In effect, the propagation of the reply back to the client via the primary task chain occurs concurrently with the AMI-based state synchronization activity.

Note that the transferred state is considered a *potential orphan* until the client receives the reply and therefore, it is maintained in the temporary buffer at the replica side as was the case with the eager strategy. Unlike the eager approach, however, no *finish* message is sent to the RM at the end of successful invocation. Instead, the client is allowed to proceed with its subsequent periodic request invocation and execution. Execution of the next request at the primary components and the subsequent generation of new state is interpreted as the successful completion of the earlier request.

In more formal terms, when the k^{th} state update arrives at the replica state synchronization agent, the $(k-1)^{th}$ state update (which must have been stored in the temporary buffer at each agent in the backup components) is permanently accepted into each of the replica component. In effect, the explicit *commit* phase of the eager strategy is replaced by an implicit arrival of potential orphan state for the succeeding (*i.e.*, k^{th}) invocation from the primary components. Such an approach eliminates the need for explicit messaging as well as complex interactions among the RM, client-side interceptors, and the server components during fault-free conditions.

This strategy, however, pays a small price in case of a failure. Notice that the replica components are not ready for an instantaneous client failover because the state that client expects lies in the temporary buffers of the state synchronization agents. More formally, when the k^{th} invocation completes successfully, the temporary buffers at the replica side contain the corresponding state but the replicas themselves have the state at the end of the $(k-1)^{th}$ execution. If the k^{th} request fails with partial success in a subset of primary components, some temporary buffers at the replica side may contain the state from the k^{th} execution whereas others may still have the state from the $(k-1)^{th}$ execution. Therefore, a clean-up phase is necessary that *prepares* the replica components to accept the client's reinvocation.

Upon detection of failure, the client-side interceptor sends a *prepare(k)* message (Step #1 in Figure 5) to the RM requesting it to prepare the replicas for the k^{th} invocation. Arrival of a prepare message to the RM in this strategy is an indication that there was a failure and that the RM must proceed to eliminate the orphans. Subsequently, the RM sends (using AMI) the *commit(k-1)* message (Step 2) to the state synchronization agents collocated in the backup replicas, which take one of the following two actions depending upon the state in the temporary buffers:
(1) If the state in the temporary buffer is from the k^{th} invocation, it is discarded because the orphan was created during the k^{th} invocation;
(2) If the state in the temporary buffer is from the $(k-1)^{th}$ invocation, it is permanently accepted by the backup replica components because the fact that the k^{th} invocation was in progress is an indication that the previous invocation was successful (the lag-by-one property).

The RM returns to the client-side interceptor indicating success using an appropriate return value. The client-side interceptor thereafter transparently fails over to a backup replica group and reinvokes the k^{th} request (Step #3 in Figure 5).

Although lazy state transfer implemented using AMI in the lag-by-one strategy eliminates the overhead in the critical path, dynamic workloads or failure of the replicas may delay/prevent state transfer. To avoid the replicas lagging behind more than one state update, the primary state synchronization agents maintain a count of AMI callback handlers that are missing notification of successful state transfers. If the count increases beyond a specific threshold value, pre-configured recovery actions, such as notifying a human operator are initiated. The primary components continue to operate while the system is restored to the desired level of replication.

4. MIDDLEWARE IMPLEMENTATION

We now present the details of the implementation of our group-failover protocol alluding to the three requirements. The group-failover protocol is implemented using the open-source Component Integrated ACE ORB (CIAO) [16] middleware, which is an implementation of Lightweight CORBA Component Model [17] (LwCCM). We leveraged a number of infrastructure elements from our CIAO-based prior work called FLARe [18] and CORFU [19], however, a number of extensions and new capabilities were implemented for this research explained below.

(1) Failure Detection and Transparent Failover: For failure detection and failover, we leveraged the fault monitoring infrastructure from FLARe [18] and CORFU [19]. Faults are monitored using periodic heart-beats. A loss of heart-beat triggers recovery actions at the RM. Although

FLARe handles fault detection for only single server processes, CORFU extended the capability to end-to-end task chains of components. CORFU, however, takes linearly increasing amount of time (shutdown latency) for client failover depending on the tier where the failure occurred.

For the group-failover protocol we optimized the algorithm in the RM and made the client-perceived failover latency insensitive to (1) the location of the failure and (2) the number of components in the end-to-end task chain. For instance, irrespective of which component in Figure 2 fails, then failover-latency experienced by the client remains unchanged. Our technique uses the system structure described in the deployment meta-data to identify the orphans and CORBA asynchronous messaging interface (AMI) to eliminate them concurrently. The concurrent elimination of the orphans allows the client-perceived failover latency to be independent of the number of components. The details of our technique are described below.

(2) Identifying and Eliminating Orphans: Since our approach to orphan identification is statically defined, we obtain the necessary static meta-information about the dataflow between components in the system from annotated system composition models. For this we leverage and extend our earlier work [29] on domain-specific modeling languages that allow system developers to specify composition of the system and the component behavioral properties using intuitive higher-level abstractions.

Using the dataflow dependencies between components, graph reachability algorithms are used to statically determine the extent of orphan state due to failure of non-deterministic components. Ad-hoc grouping of components is also supported using FOU annotations. Such application-specific meta-information is generated and embedded in standards-based deployment metadata for the CIAO middleware by our modeling tools, which in turn is leveraged by the RM to eliminate orphans at run-time.

This metadata contains remote references (*e.g.,* CORBA-compliant URLs) to the server processes hosting the server components. The RM instructs host processes to passivate the *suspected* orphan components thereby eliminating them from the system. Unlike CORFU, remote invocations for component passivation are implemented using CORBA asynchronous messaging interface (AMI), which exploits concurrency to make the shutdown latency independent of the size of the end-to-end task chain and the location of the failure.

Since passivating a component discards all its application-specific state and middleware state, such as open TCP/IP connections, the loss of connection is recognized by the client-side middleware and a system-level exception is raised in response to that. The client-side request interceptor detects the exceptional return of the remote invocation and allows the client to transparently fail over to the replica.

Passivated components can be activated again and may join the group of replicas to maintain the quorum and receive state updates from the then-active primary. A subsequent activation phase after recovery is conceivable to maintain the desired level of replication in the system. All the deployment and configuration activities, including activating and passivating the components are seamlessly handled by the existing CIAO middleware [16].

(3) Strategies for Bounded-time State Consistency: To support the state consistency strategies, the state synchronization agent in FLARe was refined to include a temporary buffer. The RM design was improved using the Strategy design pattern to support the two state synchronization strategies. When using the eager strategy, the RM is equipped with the messaging logic for the optimized 2PC approach. When using the lag-by-one strategy, the RM gets involved only during failure recovery and orphan elimination. The client-side interceptor is also strategizable. For the eager state synchronization strategy, the interceptor is refined to include the additional logic for ensuring atomicity. In the lag-by-one strategy, the client interceptor invokes RM during failure recovery only. Host monitors were used directly from prior work. All these elements were implemented as CORBA objects and integrated into the deployment process of components.

Our implementation of the eager synchronization strategy uses CORBA AMI in both phases to exploit parallelism during state transfer (Step #2 and #4 in Figure 4). RM uses AMI to send the *precommit* to all the primary state synchronization agents in parallel and waits for all of them to complete state transfer in parallel. The use of AMI makes the eager state synchronization strategy (nearly) insensitive to the number of components involved in the group. For lag-by-one, AMI is used to lazily transfer state (see double dashed lines in Figure 5) thereby eliminating the overhead of state transfer in the critical path. With every asynchronous call a callback handler is registered, which is invoked upon completion of the remote request. It is also possible to abort waiting for a callback based on a timeout.

5. EVALUATING THE GROUP-FAILOVER PROTOCOL

We empirically evaluated our implementation of the group-failover protocol at ISISlab (www.isislab.vanderbilt.edu) on a testbed of up to 6 blades. Each blade has two 2.8 GHz CPUs, 1GB memory and are connected by a Gigabit LAN.

Methodology and Rationale. We evaluated the overhead and the client-perceived failover latency in fault-free and failure scenarios, respectively. In every experiment we varied the nesting depth of components from 2 to 5 to show that both state synchronization strategies have bounded overhead independent of the nesting depth. Dummy computations, calibrated to consume 25ms of CPU time, were performed on every component in response to the client request. As a result, the server-side execution time increases from 50ms to 125ms as the number of server-side components increase from 2 to 5. We executed every experiment 20 times and averaged the data over all the runs. We deployed host-monitors on every node and a single instance of replication manager.[4] Our future work will validate our approach on real-world DRE systems.

5.1 Overhead Measurements in Fault-free Scenarios

In the fault-free scenario experiments, we measured the response time, which includes (1) the actual server-side execution time, (2) overhead of the state synchronization phases (if any), and (3) the communication latency between client and the (front) server and the server components themselves. The evaluations are presented in Figure 6 and Table 1.

[4]Fault tolerance of the replication manager can be achieved through replication and is outside the scope of the paper.

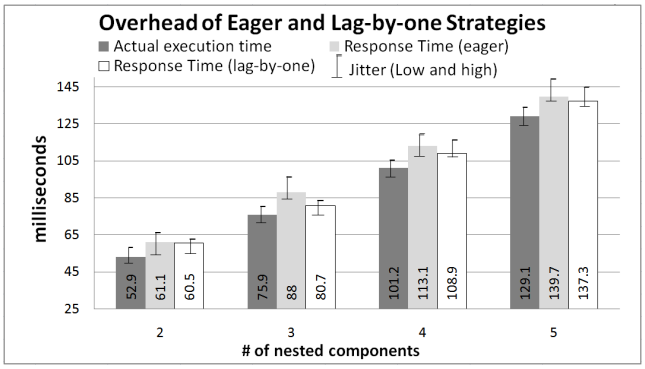

Figure 6: Overhead of the eager and lag-by-one strategies during fault-free scenarios (Jitter up to 12ms)

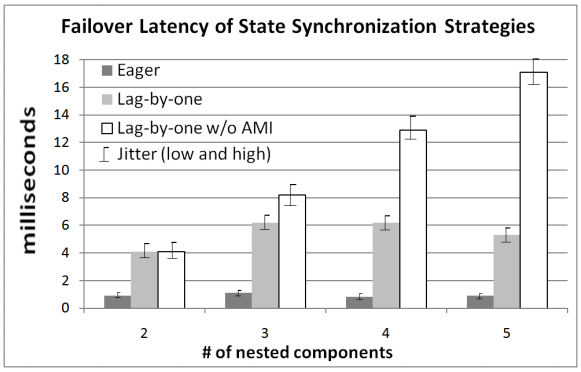

Figure 7: Client-perceived failover latency of the state synchronization strategies

Nesting level	2	3	4	5
Eager strategy (ms)	8.2	12.1	11.9	10.6
Eager strategy w/o AMI (ms)	13	23	26.1	32.7
Lag-by-one strategy (ms)	7.6	4.8	9.7	8.2

Table 1: Overhead of the eager and lag-by-one strategies (fault-free)

Figure 6 shows the overhead of the eager and lag-by-one strategies along with the actual execution time during fault-free executions. Table 1 shows that the overhead was between 8.2 ms and 12.1 ms for both phases of the eager strategy. Because of the two phases in the eager strategy, it has higher overhead than the lag-by-one strategy. Although the lag-by-one strategy does not execute any additional steps during fault-free scenario, in practice, the overhead of the lag-by-one strategy was observed between 4.8 ms and 9.7 ms. This is primarily due to the communication latency between client and the (front) server and between server components themselves. The overhead in both the strategies is independent of the nesting level because of asynchronous messaging interface (AMI). To better understand the benefits of AMI, we implemented the two phases of the eager strategy without AMI. From Table 1 it is evident that without AMI the overhead is sensitive to the nested depth.

These results indicate that the overhead of both the strategies is low compared to the protocols that integrate transactions and replication, and that the additional work performed is bounded irrespective of the size of the task chain.

5.2 Client-perceived failover latency in failure scenarios

The objective of this experiment is to evaluate the failover latency that a client will experience when the server component that the client is communicating with (*i.e.*, head of the chain) fails. Upon detecting the failure we measure the time needed to begin execution at the backup replica components.

Figure 7 shows the client-perceived failover latency for different nesting levels (2 to 5). The failover latency of the eager strategy is about 1 ms because the only step necessary in this case is to reinvoke the request on the replica components. The observed failover latency is the sum of time needed for failure detection and subsequent reinvocation. We also measured the overhead of the *prepare* phase of the lag-by-one strategy using a high resolution timer in the client-side interceptor. We evaluated the implementations of *prepare* phase with and without AMI. When the replication manager exploits concurrency by sending the *commit* messages using AMI, we see that the overhead of the prepare phase is not dependent on the nesting level. On the other hand, when *commit* messages do not use AMI, the overhead is clearly dependent on the number of components in the task chain. These results indicate that the performance of group-failover protocols is acceptable in failure scenarios, and that the latencies are bounded irrespective of the task chain size (*i.e.*, nesting level).

6. CONCLUDING REMARKS

Orphan requests (and components) are an instantaneous consequence of a failure in DRE systems where multiple tiers of non-deterministic stateful components are replicated for high availability. While soft real-time systems require timely elimination of orphans, equally important is the need to maintain state consistency and to deliver acceptable response times to clients. This paper presented the *group-failover* protocol, which is a novel approach to ensure consistent and timely data for multi-tiered DRE systems (modeled as end-to-end task chains) in the presence of failures of non-deterministic stateful components. Unlike previous approaches, the group-failover protocol has negligible messaging overhead depending upon the choice of the state synchronization strategy. State consistency is ensured using the *eager* and *lag-by-one* state synchronization strategies, which present different tradeoffs during fault-free and failure scenarios. Empirical evaluations of the protocol demonstrate that the performance is suitable for soft real-time systems and the overhead is largely insensitive to the number of tiers.

Failing over groups of components may appear too pessimistic, however, for a class of DRE systems where fast and predictable failover is important while maintaining strong state consistency, our approach provides an acceptable tradeoff. Moreover, using model-based techniques to annotate the groups provides intuitive mechanisms in addition to the benefits of automation. The middleware artifacts resulting from this research are part of the open source CIAO middleware.

7. REFERENCES

[1] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and

Reflective Middleware for Network-Centric Combat Systems," in *CrossTalk - The Journal of Defense Software Engineering*. Hill AFB, Utah, USA: Software Technology Support Center, nov 2001, pp. 10–16.

[2] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003, Washington, DC, May 2003.

[3] J. W. S. Liu, *Real-time Systems*. New Jersey: Prentice Hall, 2000.

[4] S. Pleisch, A. Kupsys, and A. Schiper, "Preventing Orphan Requests in the Context of Replicated Invocation," in *Proceedings of 22nd Symposium on Reliable Distributed Systems*, 2003, pp. 119–129.

[5] S. Poledna, "Replica Determinism in Distributed Real-time Systems: A Brief Survey," *Real-Time Syst.*, vol. 6, no. 3, pp. 289–316, 1994.

[6] H. Kolltveit and S. olaf Hvasshovd, "Preventing orphan requests by integrating replication and transactions," in *11th East-European Conference on Advances in Databases and Information Systems, ADBIS*. Springer, 2007.

[7] R. Guerraoui and A. Schiper, "Software-Based Replication for Fault Tolerance," *IEEE Computer*, vol. 30, no. 4, pp. 68–74, Apr. 1997.

[8] R. B. David, D. Lomet, S. Paparizos, H. Yu, and S. Ch, "Persistent Applications via Automatic Recovery," in *7th International Database Engineering and Applications Symposium (IDEAS 2002)*, 2002, pp. 258–267.

[9] S. Frolund and R. Guerraoui, "Transactional Exactly-Once," *Technical report, Hewlett-Packard Laboratories*, 1999.

[10] P. Felber and P. Narasimhan, "Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'02)*, 2002, pp. 737–754.

[11] E. Dekel and G. Goft, "ITRA: Inter-Tier Relationship Architecture for End-to-end QoS," *Journal of Supercomputing*, vol. 28, no. 1, pp. 43–70, 2004.

[12] J. Slember and P. Narasimhan, "Using Program Analysis to Identify and Compensate for Nondeterminism in Fault-Tolerant, Replicated Systems," in *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, 2004, pp. 251–263.

[13] ——, "Living with Nondeterminism in Replicated Middleware Applications," in *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, 2006, pp. 81–100.

[14] R. Jimenez-peris, M. Patino-Martinez, S. Arevalo, and J. Carlos, "Deterministic Scheduling for Transactional Multithreaded Replicas," in *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, 2000, pp. 164–173.

[15] C. Basile, Z. Kalbarczyk, and R. Iyer, "A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas," *International Conference on Dependable Systems and Networks*, vol. 0, p. 149, 2003.

[16] Institute for Software Integrated Systems, "Component-Integrated ACE ORB (CIAO)," www.dre.vanderbilt.edu/CIAO, Vanderbilt University.

[17] *Lightweight CORBA Component Model RFP*, realtime/02-11-27 ed., Object Management Group, Nov. 2002.

[18] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," in *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS '09)*, San Francisco, CA, Apr. 2009, pp. 118–127.

[19] F. Wolf, J. Balasubramanian, S. Tambe, A. Gokhale, and D. C. Schmidt, "Supporting Component-based Failover Units in Middleware for Distributed Real-time and Embedded Systems," *Journal of Software Architectures: Embedded Software Design, Special Issue on Embedded and Real-time (in print)*, Nov. 2010.

[20] *Transaction Services Specification*, OMG Document formal/97-12-17 ed., Object Management Group, Dec. 1997.

[21] *Fault Tolerant CORBA, Chapter 23, CORBA v3.0.3*, OMG Document formal/04-03-10 ed., Object Management Group, Mar. 2004.

[22] S. Poledna, "Replica Determinism in Fault-Tolerant Real-Time Systems," Ph.D. dissertation, Technical University of Vienna, Vienna, Austria, 1994.

[23] M. Herlihy and M. McKendry, "Timestamp-Based Orphan Elimination," *IEEE Transaction on Software Engineering*, vol. 15, no. 7, pp. 825–831, 1989.

[24] B. Liskov, R. Scheifler, E. Walker, and W. Weihl, "Orphan Detection," *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pp. 2–7, 1987.

[25] J. Sun, "Fixed-Priority End-to-End Scheduling in Distribured Real-time Systems," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champagne, 1997.

[26] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, pp. 225–267, 1995.

[27] R. D. Schlichting and F. B. Schneider, "Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems," *ACM Transaction on Computer Systems*, vol. 1, no. 3, pp. 222–238, 1983.

[28] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-backup Approach," in *Distributed systems (2nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216.

[29] S. Tambe, A. Dabholkar, and A. Gokhale, "Generative Techniques to Specialize Middleware for Fault Tolerance," in *Proceedings of the* 12^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2009)*. Tokyo, Japan: IEEE Computer Society, Mar. 2009.

[30] A. Avizienis and C. Liming, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," *Compsac*, pp. 149–155, 1977.