Supporting Component-based Failover Units in Middleware for Distributed Real-time and Embedded Systems[☆]

Friedhelm Wolf[a], Jaiganesh Balasubramanian[a], Sumant Tambe[a], Aniruddha Gokhale[*,a], Douglas C. Schmidt[a]

[a]*Department of EECS, Vanderbilt University, Nashville, TN 37235, USA*

Abstract

Although component middleware is increasingly used to develop distributed, real-time and embedded (DRE) systems, it poses new fault-tolerance challenges, such as the need for efficient synchronization of internal component state, failure correlation across groups of components, and configuration of fault-tolerance properties at the component granularity level. This paper makes three contributions to R&D on component-based fault-tolerance. First, it describes the COmponent Replication based on Failover Units (CORFU) component middleware, which provides fail-stop behavior and fault correlation across groups of components treated as an atomic unit in DRE systems. Second, it describes how CORFU's Components with HEterogeneous State Synchronization (CHESS) module provides mechanisms for real-time aware state transfer and synchronization in CORFU. Third, we empirically evaluate the client fail over and group shutdown capabilities of CORFU and its CHESS module and compare/contrast it with existing object-oriented fault-tolerance methods. Our results show that component middleware (1) has acceptable fault-tolerance performance for DRE systems, (2) allows timely recovery while considering failure location, size, and functional topology of the group, and finally (3) eases the burden of application development by providing middleware support for fault-tolerance at the component level.

Key words: Failover units, component middleware, real-time and embedded systems

[☆]This work was supported in part by NSF CAREER Award CNS #0845789

[*]Corresponding Author

Email addresses: fwolf@dre.vanderbilt.edu (Friedhelm Wolf), jai@dre.vanderbilt.edu (Jaiganesh Balasubramanian), sutambe@dre.vanderbilt.edu (Sumant Tambe), gokhale@dre.vanderbilt.edu (Aniruddha Gokhale), schmidt@dre.vanderbilt.edu (Douglas C. Schmidt)

1. Introduction

Ensuring the fault-tolerance for mission-critical distributed real-time and embedded (DRE) systems, such as air traffic management, total ship computing environments, and fractionated spacecraft, is essential to meet end-to-end system requirements. Fault-tolerance in the context of DRE systems is defined as the property that masks failures of servers from clients by transparently redirecting clients to backups of servers that have a consistent state as the failed server. Moreover, for DRE systems, this redirection must be achieved in a timely manner so that response time of clients is maintained. A recent study [1] indicates that many system crashes and downtime stem from undependable components and that fault-tolerance mechanisms fail to work correctly.

Software for DRE systems increasingly uses component middleware [2, 3, 4, 5], which is middleware that support the component programming model [6, 7]. A component provides a higher level of abstraction than traditional objects in object-oriented programming by providing capabilities to encapsulate the application "business" logic, as well as the means of grouping related interfaces to offer a service family. Individual components can be assembled together to form applications.

Component middleware helps to reduce application development time and effort [5] by enabling the rapid realization of large-scale applications by procuring third-party components. It also enables the packaging and assembly of components into reusable units of functionality that can be deployed in the target distributed environment. Despite these advantages, however, component-based DRE systems incur the following fault-tolerance challenges compared to DRE systems that use distributed object computing and operate at the granularity of individual objects:
1. Support for fail over and recovery of a group of components. Although DRE system functionality may be obtained rapidly by assembling a group of components procured from multiple providers, supporting fault-tolerance for this overall functionality requires treating the group of components as a single unit of failure and recovery.

For example, a real-time analytic stock application may consist of a group of interacting components that together provide the analytic capabilities. Since the group of components together serve a single request (*i.e.*, stock analytics in this case), they share a common state of the client request. As a result, a failure of even a single component from the group should result in a fail over to a backup group of components.

Unfortunately, contemporary component middleware, such as Lightweight CORBA Component Model (LwCCM) [8], do not provide group-based fail over semantics out-of-the-box. What is needed, therefore, is first-class support for the notion of a *failover* group, which is a logical grouping of components that are treated as an atomic unit for failure detection and recovery.

2. Efficient state dissemination mechanisms for a failover group. DRE systems have to maintain a wide variety of application state ranging from scalar types (*e.g.*, int, float) to complex types (*e.g.*, large structures and sequences). To provide fault-tolerance using passive replication [9], middleware must provide mechanisms to disseminate this state to application replicas in a timely manner. Moreover, depending on the size, timing requirements, and degree of replication different communication mechanisms are needed to provide optimal performance.

The small-sized state of applications with high reliability requirements should typically be transferred through synchronous, point-to-point protocols with error correction capabilities. Conversely, large-sized state (particularly when transmitted to a large number of replicas) needs efficient protocols, such as group communication protocols and multicast messages. Directly encoding the type of communication mechanism into the applications' implementation results in a tight coupling between business logic and transport mechanism, and therefore complicates development and adaptation of the application.

3. Resource-aware deployment and configuration. The benefits gained by using component abstractions (*e.g.*, reduced application developer effort) are often nullified by inefficient use of DRE system resources, such as memory, CPU, and network bandwidth. In particular, without first-class support to manage the semantics of a failover group, it becomes tedious and error-prone to deploy failover groups and their replicas such that resources are utilized efficiently. Similar problems arise in configuring fault-tolerance properties in the middleware for these failover groups.

Achieving fault-tolerant DRE systems involves moving from point solutions for specific scenarios towards solutions that integrate all aspects of fault-tolerance (detection, recovery and state consistency) through well-understood models, metrics, development processes and tools. This paper makes three contributions to address the three challenges described above and provides a component middleware-based fault tolerance solution that supports the fault-tolerance aspects presented above.

1. First-class support for group fail over and recovery based on the *COmponent Replication*

based on Failover Units (CORFU) component middleware. CORFU supports fault-tolerance of component-based DRE systems that use passive replication, where backup replicas take over processing quickly when a failure occurs. CORFU implements algorithms that provide efficient fail-stop behavior of component groups. Rather than reactively providing fail over capabilities for each component in a group as they fail sequentially, CORFU restores system operation by activating a fresh group of components while discarding the faulty group in a single logical execution step. Fail over operations can thus be more deterministic since they deal directly with the original error and do not propagate errors.

2. First-class support for real-time state dissemination among the group based on CORFU's *Components with HEterogeneous State Synchronization* (CHESS) module. CHESS provides mechanisms for state dissemination in passively replicated failover groups managed by CORFU. State can change through client invocations, other system events, or timed signals. CHESS synchronizes the internal states of replicas—particularly replicas of failover groups. It also makes state synchronization as transparent to developers as possible, while providing the flexibility to handle varying internal state types (such as scalar types, structures of scalar types, and their sequences) and assuring timely state transfers.

3. Resource-aware, automated deployment and configuration for group semantics. CORFU adopts a static deployment approach that captures the complete system structure and the placement of components in a standardized XML format. CORFU uses this declarative system metadata to optimize runtime behavior for both individual components and groups of dependent components. By capturing component dependencies in specific component groups (called *failover units*), CORFU reduces error reaction time at runtime and helps ensure timely response required by DRE systems. CORFU also provides declarative mechanisms that help automate the deployment and configuration of components in DRE systems.

This paper evaluates several capabilities of CORFU and its CHESS module qualitatively and quantitatively. It presents qualitative analysis that compares the effort involved in applying conventional object-level fault-tolerance versus CORFU's component level fault-tolerance. This analysis shows how CORFU improves efficiency and reliability of system development by making fault-tolerance aspects orthogonal to application development. It also presents experiments that quantify the latencies involved in a fail over operation of component based fault-tolerant applications and the timing characteristics of the fail-stop behavior of CORFU component groups

both for stateless and stateful groups of components.

The remainder of this paper is organized as follows: Section 2 uses a DRE system case study to motivate the need for component middleware and dependency-based component groupings; Section 3 summarizes the structure and functionality of CORFU and its CHESS state synchronization module; Section 4 analyzes experimental results to evaluate the performance of CORFU during failover of groups and the impact of CHESS in state synchronization; Section 5 compares CORFU with related work; and Section 6 presents concluding remarks.

2. Motivating Component Group Fault Tolerance via a Case Study

We use a representative DRE system from the domain of space systems as a case study to highlight the need for component-based, group failover and recovery requirements that are also QoS-enabled, *i.e.*, aware of real-time performance requirements. This case study has stringent requirements for real-time and fault-tolerant behavior. To showcase the challenges confronting component-based DRE systems, we focus on the Mission Control System (MCS) being developed by the European Space Agency [10] to control satellites that perform missions, such as earth observation or deep-space exploration.

2.1. Overview of the Mission Control System (MCS)

An MCS controls satellites and processes data they gather. It is deployed in a central control station and communicates with a network of ground stations that provide communication links to the satellites. Figure 1 shows the structure of a component-based MCS. Since the time windows for active connections to satellites can be short due to their orbit and visibility to ground stations, the availability of the MCS during such phases is crucial. The MCS therefore uses redundant hardware and software. Each entity is deployed twice and some are grouped into chains of functionality, which are groups of components working together as a unit.

For example, an MCS must be tailored to specific missions and reconfigured for different mission phases. The *Mission Planning System* is responsible for configuring and observing the other system entities based on the mission specific characteristics. Likewise, the *Telemetry Server* analyzes telemetry data and preprocesses it for the mission operators. The *Archive* stores telemetry data permanently and is fed by the Telemetry Server. The *Telecommand Server* is responsible for creating and sending new commands issued by the mission operators.

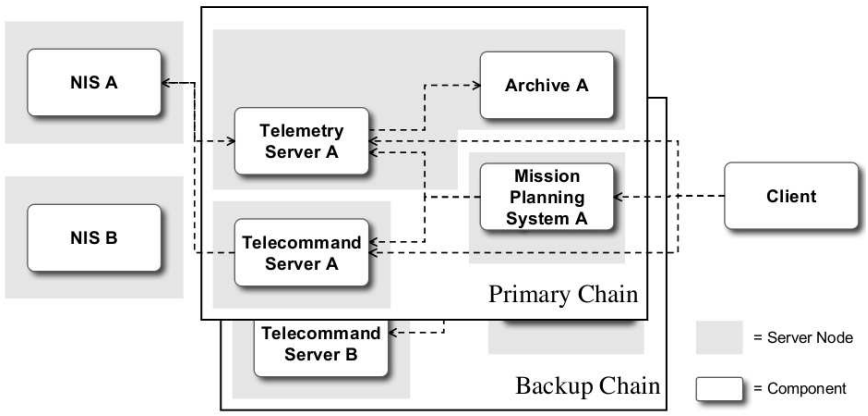

Figure 1: **Component-Based Mission Control System**

These four entities form a task chain that provides the main MCS functionality. To avoid single points of failure, this chain is replicated. A primary chain is active during normal operation, as shown in Figure 1. If a fault occurs in the primary chain, the complete chain must be passivated and a backup chain must assume operation through a warm-passive fail over. All components of the backup chain are already deployed to assume responsibility as quickly as possible.

The *Network Interface System* serves as a gateway from the ground stations to the MCS through a wide area network. It uses the space link extension protocol to process and transmit all mission relevant data to and from the MCS. The Network Interface System is not part of the MCS chain and is replicated separately.

2.2. Fault-Tolerance Requirements of the MCS Case Study

The MCS chain forms a unit of fail over and recovery. Providing replication and recovery semantics for component groups must therefore support the following requirements:

Requirement 1: Fault isolation. In the MCS scenario the components within one chain depend on each other. A failure of one component must lead to the automated shutdown and fail over of all components within the same chain. If the telemetry server, for example, crashes, all other components in chain A should be deactivated and the four components in chain B should resume operation.

MCS therefore needs timely detection of the fault and correlation of this fault to the group

as a whole so that subsequent group-wide shutdown and recovery actions can be initiated in real-time. Section 3.3.1 describes how CORFU provides timely detection of faults and correlates them to group-wide semantics.

Requirement 2: Ensure fail-stop behavior. After a fault has been isolated by determining affected components, it is necessary to regard these components as inconsistent, which may carry transient faults. Subsequently, this chain must be shutdown to prevent further propagation of the fault. All affected components must therefore be stopped as soon as possible, *i.e.*, the time from error detection to the complete stop of all affected components must be minimized. Section 3.3.2 describes CORFU's support for fail-stop semantics.

Requirement 3: Service recovery. When components of the primary chain fail and are deactivated, all components in the backup chain must be promoted to serve as the primary, and process incoming requests. Although the MCS scenario presented here only contains one backup failover unit (and thus only one backup replica per component), the mechanism generally must account for any number of backups. With more than one backup, however, the system could have components failing over to replicas in different chains, thereby causing performance problems or even malfunctions due to the way components are deployed on a given infrastructure.

To achieve successful fail over it is necessary to synchronize the promotion of backup components. In passive replication this involves ensuring that the correct backup replicas become primaries. To ensure consistent system state after fail over the middleware must ensure that all backups that become primaries belong to the same failover group. Section 3.3.3 describes how CORFU provides support for group-wide failure recovery.

Requirement 4: State dissemination. As the MCS scenario uses passive replication, state dissemination mechanisms must be provided to keep the state of the replica components consistent with that of the primary component. Moreover, state dissemination overhead should not adversely affect the client's response time. This indicates that as the size of the application state and number of replicas increase, reliable group communication based mechanisms must be favored over reliable point-to-point (multiple unicast) communciation. Section 3.3.1 describes how CORFU supports mechanisms for timely and consistent transfer of state to replicas.

2.3. DRE System and Deployment Model

The MCS is being developed and deployed using the Lightweight CORBA Component Model (LwCCM) [11] shown in Figure 2 and described below to show how DRE systems like

MCS are developed using component abstractions, as well as how component-based middleware like CORFU are designed and implemented. *Components* in LwCCM are implemented by *ex-*

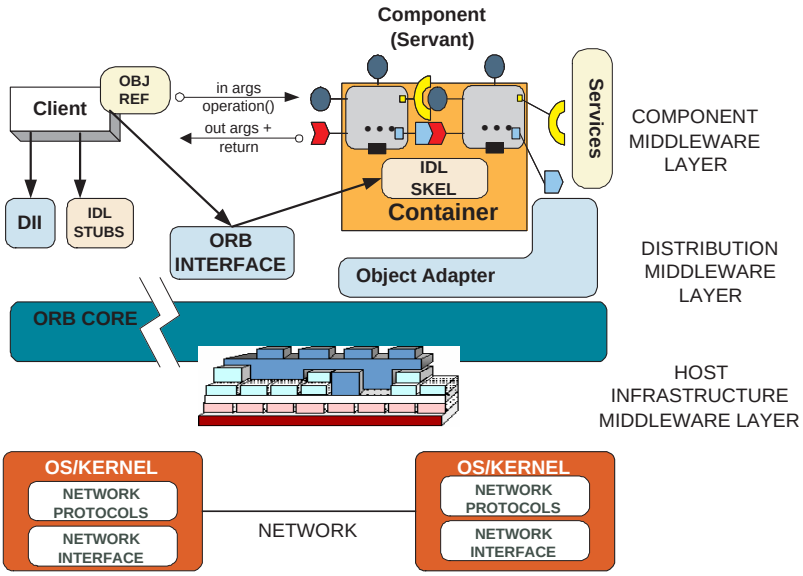

Figure 2: **The LwCCM Architecture**

ecutors and collaborate with other components via *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on a point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

There are two general types of components in LwCCM: (1) *monolithic components*, which are executable binaries, and (2) *assembly-based components*, which are a set of interconnected components that can either be monolithic or assembly-based.

A *container* in LwCCM provides the run-time execution environment for the component(s) it manages. Each container is responsible for initializing instances of the components it manages and mediating their access to other components and common middleware services.

The deployment and configuration of DRE system components is performed by the actors defined in the OMG LwCCM deployment and configuration (D&C) specification [12] and are shown in Figure 3. The specification comprises two parts: (1) a management model that defines the roles of the various actors and (2) a data model that defines the metadata format for the

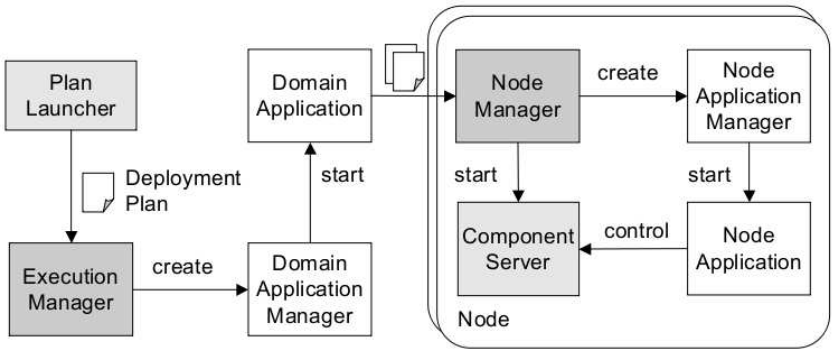

Figure 3: **System Model for Component-based Deployment and Configuration**

deployment and configuration properties of the DRE components that are used by the actors in deploying and configuring the components.

The central entity in the management model of the LwCCM D&C spec is the *ExecutionManager*, which is responsible for instantiating *DomainApplication*s as defined in deployment plans. Every node is represented by a *NodeManager* in the management layer.

A deployment plan is part of the data model of the LwCCM D&C specification. It contains information about which component *implementations* and corresponding *artifacts* are used and which component *instances* are present in the system. Each entity can also contain configuration properties that allows tailoring of components to the specific deployment. The target infrastructure is represented in form of *nodes* corresponding to server machines on which a component instance will run. Each instance is associated with a node on which to run. The deployment plan captures component interdependencies through *connections* between their ports.

Each deployment plan is handled by a *DomainApplicationManager* that provides the administration interface to start and stop the application. The *ExecutionManager* splits a deployment plan into partial deployment plans that are processed by each associated *NodeManager*. Each node-specific deployment plan is handled by a *NodeApplicationManager* that starts and stops *NodeApplications* and *component servers*. A *component server* hosts containers and provides the runtime process context. A *NodeApplication* is a management entity that controls the lifecycle of components hosted in a component server.

The D&C actors can be leveraged to build additional capabilities for group shutdown and fail over in which node-specific actors can actuate fine-grained control in activating and shutting

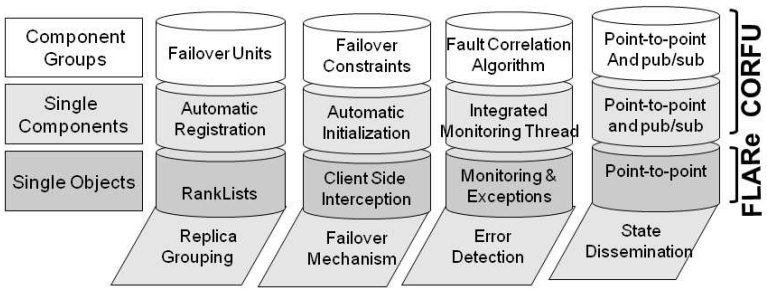

Figure 4: **CORFU Layered Architecture**

down a component, as well as triggering state synchronization. Section 3 delves into the details of CORFU's design that leverages the capabilities of the LwCCM D&C specification.

3. The Structure and Functionality of CORFU

Addressing the fault-tolerance and timeliness requirements of DRE systems described in Section 2.2 requires innovative solutions that can leverage and integrate with the middleware platforms used by DRE systems. This section describes how the structure and functionality of CORFU provides DRE systems with passive replication of a group of components treated as a logical *failover unit* [13], which contains a set of components that have dependencies with respect to failure propagation. Failover units allow for failure reaction times suitable for DRE systems by terminating a group of components with a single logical execution step, rather than reacting on slow failure propagations.

CORFU's layered architecture is shown in Figure 4.

This architecture enables CORFU to provide sophisticated fault-tolerance capabilities, including support for component group replication and failover. Each layer of fault-tolerance functionality is provided along four fundamental dimensions of fault-tolerance, including (1) *replica grouping,* which defines the replicas that form one logical entity for group failover, recovery, and synchronization of internal state, (2) *error detection,* which detects and reports failures to initiate fail over operations for the group, (3) *fail over mechanism,* which redirects processing of client requests in case of a detected failure, and (4) *state dissemination,* for maintaining the state of replicas consistent with the primary.

The rest of this section describes how CORFU's layered architecture provides these four

dimensions of fault-tolerance, starting at the lowest layer working up through the layers.

3.1. Architectural Foundation of CORFU: Fault-Tolerance for Individual Objects

CORFU's lowest layer of support for fault-tolerance at the level of individual objects is based on our earlier work on FLARe [14], which is a middleware framework that achieves real-time fault-tolerance through passive replication of distributed objects, as shown in Figure 5. FLARe is

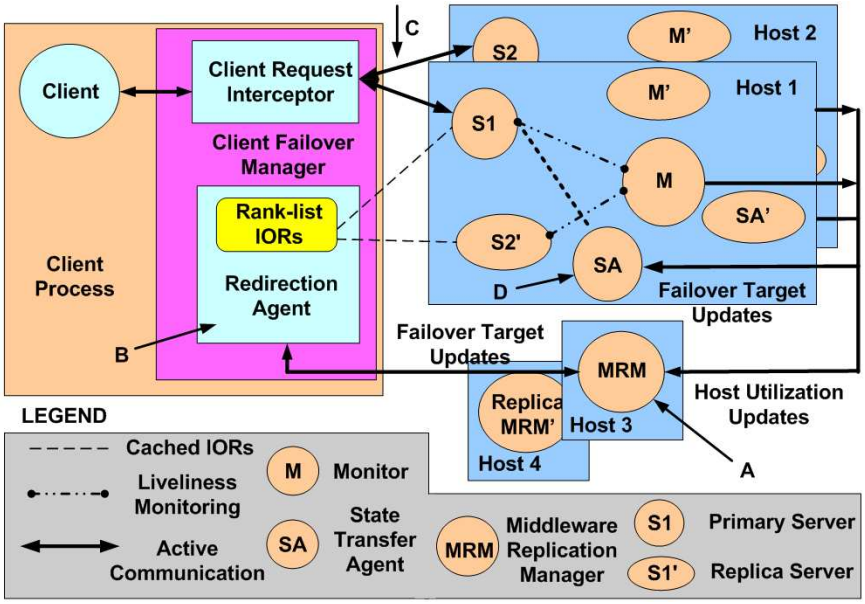

Figure 5: **The FLARe Middleware Architecture**

based on the CORBA architecture and provides lightweight fault-tolerance for CORBA *objects* (*i.e.*, *not* LwCCM *components*). We base CORFU on FLARe for the following reasons that make it suitable for DRE systems: (1) FLARe uses passive replication and allows failover mechanisms that are suitable for resource constrained DRE systems and (2) its adaptive and predictable failure recovery mechanisms provide algorithms and mechanisms to assure real-time performance even in the presence of failures.

FLARe supports the four dimensions of fault-tolerance outlined above as follows:

1. Replica grouping. The primary entity for replica grouping is the ReplicationManager (Label A in Figure 5), which is a standalone server that registers all existing replicas of a DRE system. Replicas representing one logical object are registered with the ReplicationManager by

the server application using a shared replica object ID string. The ReplicationManager builds a list of replica object references per object ID, which is known as the *RankList*.

A RankList defines the order in which replicas are activated in case of their predecessors failing. This mechanism allows FLARe to change the order at system runtime based on monitored load on hosts. The replica object ID is added to each fault-tolerant object reference on the server side through an IORInterceptor that allows modifications of Interoperable Object References (IORs).

2. Error detection. FLARe detects errors in two ways. Clients detect errors based on CORBA exceptions that occur when establishing or using a connection. The *ClientRequestInterceptor* (Label C in Figure 5) gets notified and can analyze which exception was thrown. If the exception indicates a server failure, a fail over is initiated (as described next). The ReplicationManager obtains information about failed hosts and failed processes through the *HostMonitor* applications that run on each host.

A host failure is detected by receiving heart-beat messages from the HostMonitors. If a message is not received within a given time, a host failure is assumed. Each server application on a host registers with the local HostMonitor, which then observes the liveliness of the process through a TCP/IP connection. If the connection is closed without previous deregistration of the application, a process failure is assumed and reported to the ReplicationManager.

Fail over mechanism. Fail overs are performed transparently on the client. For this purpose, a *ClientFailoverManager* (Label B in Figure 5) contains a *RedirectionAgent* that is updated periodically and proactively with fail over and redirection targets by the ReplicationManager as it tracks group membership and load changes.

The ClientRequestInterceptor is a part of the ClientFailoverManager and is registered with CORBA's object request broker. Whenever a connection to a server object fails, the request is forwarded automatically to a backup replica. The client application contains a RedirectionAgent that receives and stores up-to-date RankLists from the ReplicationManager on a regular basis. In case of a connection error, the request interceptor communicates with the redirection agent to obtain the next known reference to a server replica. A fail over can thus be performed decentrally in every client without the need to communicate with the central ReplicationManager during fail over, thereby enhancing FLARe's scalability and preventing the ReplicationManager from becoming a bottleneck.

4. State dissemination. With every application, FLARe provides a *StateTransferAgent* (Label D in Figure 5) to allow server objects within one group to synchronize their application states. To minimize impact on client-perceived response times, the StateTransferAgent supports two strategies of state dissemination. First, point-to-point communication like in FLARe using CORBA and second, group communication using anonymous publish-subscribe substrate. Unlike FLARe, however, CORFU hides the complexity of using these strategies by exposing only a choice to be made by the application developers using declarative meta-data.

3.2. First-Class Support for Component-level Fault-Tolerance

The next layer in CORFU's architecture provides fault-tolerance to individual components. This layer adds no new fault-tolerance capability, but instead raises the level of fault-tolerance abstraction provided by FLARe to encompass *components* rather than (just) objects. The four dimensions of CORFU fault-tolerance supported at this layer include the following:

1. Replica grouping. Since components can consist of several objects, replica objects must be grouped according to which object they represent and by the component to which they belong. CORFU's component server helps automate the registration of replicas within the FLARe infrastructure. The component server will create as many names as there are objects implementing a component, using the component name as a prefix of the replica object ID name. This design allows grouping the replicas according to their component and also preserving the object ID scheme of the basic FLARe mechanism.

2. Error detection. The component server not only automatically initializes the ClientRequestInterceptor that detects connection failures as described earlier, but also automatically starts a thread for communication with the local HostMonitor. It also establishes the necessary connection to the HostMonitor, thereby ensuring that each fault-tolerant component server is monitored automatically.

3. Fail over mechanism. CORFU automates server- and client-side initialization of FLARe's mechanisms for fail over, including the redirection agent on the client-side that allows all component servers to automatically receive the rank lists from the ReplicationManager. On the server all IORInterceptors are automatically registered and transparently modify IORs to contain the replica object ID necessary for fail over operations.

4. State dissemination. State dissemination process in CORFU is divided in two distinct steps: state extraction and state transfer. For state extraction, CORFU exposes FLARe's State-

TransferAgent to components. To achieve platform- and language-neutrality for the state extraction mechanism and integration, the CORFU's state transfer mechanism uses the CORBA IDL `any` data type, which allows dynamic insertion of any data type and preserves type-safety through type code annotation and support for type checking and (de)marshaling. The interfaces for state transfer between replicated components are described in [15], which are integrated into the component container and into required component interfaces.

For state transfer, CORFU not only provides FLARe's point-to-point communication using CORBA but also supports anonymous publish-subscribe communication using Data Distribution Service (DDS) [16]. CORFU provides XML annotations in the deployment plan to allow application developers to choose the transport mechanism they desire. Section 3.3.4 describes this mechanism in detail.

3.3. First-Class Support for Fault-Tolerant Component Groups

The topmost layer in CORFU is responsible for providing fault-tolerance to groups of components that are designated as failover units. This capability is a significant contribution of CORFU and is explained below in accordance with the four dimensions of fault-tolerance described earlier.

3.3.1. Replica Grouping for Component Groups

Challenge. The MCS chain in our case study in Section 2 requires a fault-tolerance solution that treats each chain as a logical failover unit. The challenge of grouping replicas results from the need to group dependent components into entities that expose fail-stop behavior as a whole and provide the basis for fail overs. Meeting this challenge is hard since the component-based DRE system description given by a deployment plan defines neither hierarchies of components nor fault-tolerance related properties. Likewise, the LwCCM runtime management interfaces do not support operations on groups of components or support fault-tolerance configuration. Addressing these limitations while remaining compliant to the standard is necessary since it ensures that the standard LwCCM programming model and existing application code is not impacted.

Solution → Failover units managed by a FaultCorrelationManager. Adding support for failover units involves two steps: (1) the notion of a failover unit must be integrated into existing LwCCM D&C system descriptions and (2) at the runtime level, failover units must be realized within a management service. At the D&C level, CORFU realizes each failover unit as a separate

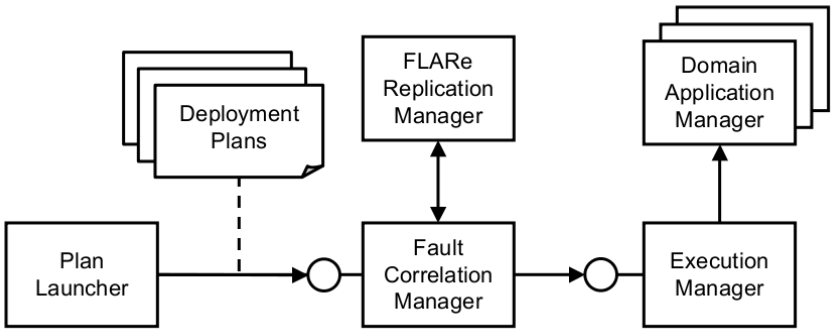

Figure 6: **FaultCorrelationManager Integration into the D&C Infrastructure**

deployment plan. Additional standard compliant properties are added to the D&C descriptors in the form of *infoProperties,* which indicate the ID of the failover unit and its rank in the list of fail over targets. These additions enable CORFU to seamlessly use existing D&C actors (see Figure 3) to start and shutdown a failover unit when necessary.

The runtime aspects of failover units are realized by a management service called the FaultCorrelationManager, which manages failover units belonging to a system. To integrate the FaultCorrelationManager into the existing D&C infrastructure, the Decorator pattern [17] is applied. The FaultCorrelationManager implements the ExecutionManager interface and can therefore be accessed by any service that uses the ExecutionManager interface. The resulting deployment system structure is depicted in Figure 6.

The benefit of CORFU's FaultCorrelationManager design is that for a client (*i.e.*, the PlanLauncher) it is indistinguishable whether it interacts with the ExecutionManager directly or with a FaultCorrelationManager. The FaultCorrelationManager will forward all requests to the ExecutionManager, but will also perform additional actions prior to delegating to the ExecutionManager.

The FaultCorrelationManager design ensures that all computation-intensive operations are performed at system start-up, which optimizes reaction times after a system is activated. To accomplish this, the FaultCorrelationManager enhances the implementation of the interface methods `preparePlan()` , `getManagers()` , and `destroyManagers()` . The main tasks are performed at start-up of the system through the `preparePlan()` method, as discussed next.

3.3.2. Efficient Error Detection at Component Group Level

Challenge. If any component of a failover unit fails, the entire component group must fail. In the MCS case study this applies to components within the primary chain, where the failure of one component leads to the shutdown of the complete chain. The challenge for error detection is that failover units can be large. Despite the size, it is necessary that errors be detected quickly and correlated with the failover unit semantics since otherwise it may adversely impact the QoS requirements of DRE systems.

Solution → A fast fault correlation algorithm. CORFU relies on the underlying FLARe layer to detect a fault in a single object, and hence in a single component. CORFU provides a fast fault correlation algorithm to correlate these detected errors with the failover unit so that shutdown operations for the unit can be initiated. Algorithm 1 depicts the fault correlation algorithm. The efficiency of Algorithm 1 hinges on the actions of the FaultCorrelationManager during the

Algorithm 1: **FAILURE-REACTION** (*h*,*F*)

Input: host name *h*
Input: list of failed object ids *F*
Data: Component Instance Map *I*
Data: Node Map *N*
Data: DomainApplicationManager Map *M*

/* phase 1 - determining affected failover units */;
look up object_id map *O* with key *h* in *N*;
create empty set *P* of deployment plan names;
for *each* $F_i \in F$ **do**
    look up instance name *i* with key F_i in *O*;
    look up plan name *p* with key *i* in *I*;
    if *p is not in P* **then**
        add *p* to *P*;
    end
end
/* phase 2 - shutting down all affected components */;
for *each* $p \in P$ **do**
    look up DomainApplicationManager *m* with key *p* in *M*;
    retrieve list of ApplicationManagers *A* through `m.getApplications ()` ;
    for *each NodeApplication* $a \in A$ **do**
        call `m.destroyApplication ( a);`
    end
end

deployment phase and on how it populates the following data structures in its fault correlation algorithm:

a. A hash map I uses component instance names as keys and associates them with the ID of the deployment plan they are hosted in.
b. A map O is maintained for each node that uses the object_id as a key to find the component instance name that represents a replica for this object_id on that node. The object_id of the incoming failure notification can therefore be associated with a concrete component instance. The node maps themselves are stored within a hash map N that allows to find them by using the node name as a key.
c. Each created DomainApplicationManager is stored in a map M with its deployment plan ID as key.

Algorithm 1 operates on these maps to process fault notifications during system operation. This processing occurs in two phases. In phase one, all affected failover units (represented as deployment plans) are determined based on the failure information using the internal maps. In phase two, existing D&C actors (namely the DomainApplicationManagers) stop all component applications that belong to these deployment plans.

The runtime complexity of this algorithm is proportional to the number of affected node applications, which can maximally be $O(m*n)$, where m is the number of deployment plans in the system and n the number of nodes in the system. This complexity stems from the fact that each NodeApplication of each affected deployment plan must be shut down separately according to the D&C interfaces. The complexity of the part that determines which plans are affected is proportional only to the number of received failure entities and is optimized by using hash maps.

3.3.3. Failover of Component Groups

Challenge. Supporting failover units as a first-class attribute in the CORFU middleware implies that after failure, all components within the group must fail over to a replica failover unit. Since CORFU enhances the object-level fail over capabilities provided by FLARe, it is necessary to map the semantics of the group to a collection of objects. Moreover, since FLARe uses the notion of a ranked ordering for objects, this concept should carry over to the semantics of the failover unit. Adding these semantics directly within the ReplicationManager would break the abstraction layering, since the ReplicationManager operates on the object level.

Solution → Fail over constraints. CORFU handles this challenge by modifying the ReplicationManager's RankList ordering algorithm such that it can process fail over constraints. Figure 7 shows an example system infrastructure with three replicated components grouped into a failover unit with two backup units.

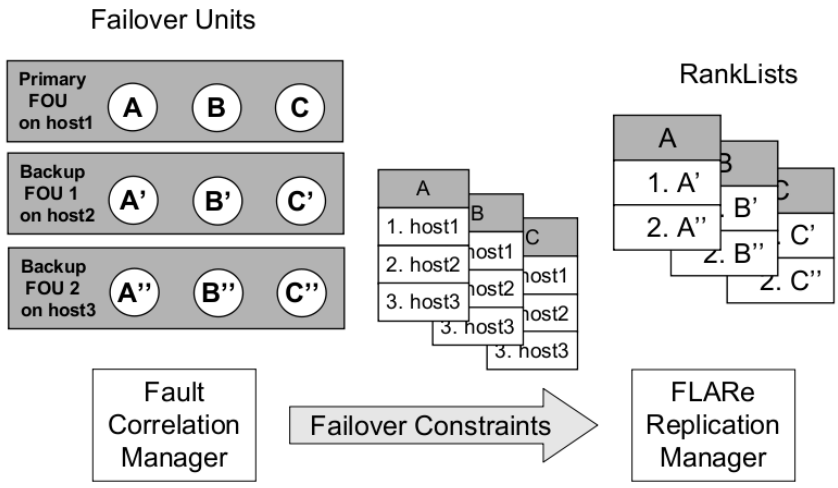

Figure 7: **Interaction between FaultCorrelationManager and ReplicationManager through Failover Constraints**

The FaultCorrelationManager transforms this information into fail over constraints that define an order of objects per replica object ID. An ordered sequence of host names defines the fail over order of each replica. The first host list entry indicates where the primary is hosted and the following hosts contain backup component replicas. Since every host has only one replica of the same group, this object ID uniquely identifies a replica.

The FaultCorrelationManager provides another algorithm called Failover Unit (FoU)-Ordering to create constraints based on information from the deployment plan. Each deployment plan representing a failover unit has an assigned rank within its group of failover unit replicas. Algorithm 2 describes how the failover unit-based replica ordering is done. All known plans are processed in the order of their failover unit rank. Each component entity results in one host name entry in the corresponding object replica group. Constraints are updated using this algorithm whenever the system structure changes. These changes occur when new deployment plans are loaded or when failures occur and deployments are removed.

Algorithm 2: **FoU-Ordering**

```
Data: List of deployment plans D
Output: A constraint list L
partially sort plans in D by their ranks;
for each plan d ∈ D do
    for each instance i ∈ d do
        get object_id o property from i;
        get host name n property from i;
        append n to list entry of L with object_id o;
    end
end
```

3.3.4. Real-time Aware Group-wide State Dissemination

Challenge. To address the timeliness requirements, applications may dictate *when* snapshots shall be distributed from the primary replica to backup replicas. There are two main types of timing behavior: (1) *cyclic timing*, where state is updated based on a given time interval, and (2) *acyclic timing*, where specific events (such as a client request handling in the primary) triggers state synchronization. Since the timing cannot be predicted in the acyclic case, active involvement of applications is needed to disseminate state at the right time. Combining both cases into a general framework mechanism is thus needed to ease the burden of the application developer without restricting timing schemes.

Solution→CHESS framework. We designed the Components with HEterogeneous State Synchronization (CHESS) framework within CORFU that treats both cases in a uniform way. This approach includes several steps of interaction between an application and a StateSynchronizationAgent, which is a CORFU-supplied agent for state synchronization. Each process containing server object replicas also hosts a StateSynchronizationAgent that is responsible for all replication related functionality and therefore removes this obligation from the application developer.

The sequence of interactions described in Figure 8 provides a mechanism for flexible and generic state dissemination, as described below:

a. *Registration of components* with the StateSynchronizationAgent through a unique application ID allows the manager to retrieve state from the application when needed. The registration is performed during the start-up phase of the component.

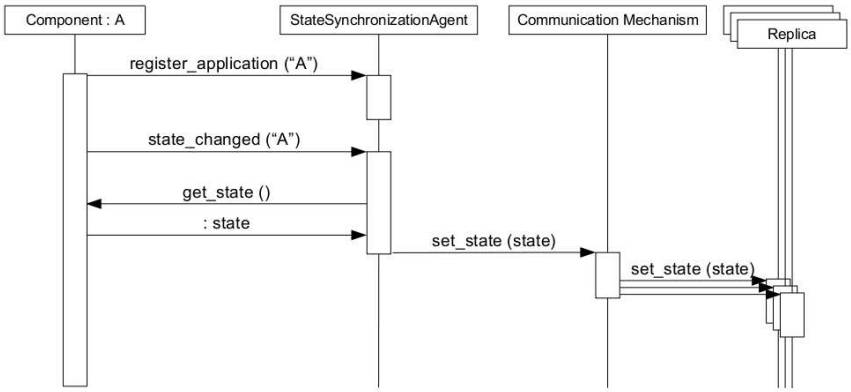

Figure 8: **State transmission sequence based on a common interface**

b. The StateSynchronizationAgent exposes the interface method `state_changed (in string id)` that allows the component to indicate a change of its internal state, which then triggers state synchronization. The id parameter is needed by the agent to identify the component among all locally deployed components managed by this agent.

c. It is the agent's responsibility to react to the notification of a state change and retrieve the component state from the component that issued the notification. Agents retrieve component state by calling back the `get_state()` method, which is an upcall method invocation on the application object to retrieve application state that is serialized for distribution.

d. As the final step, the StateSynchronizationAgent will then distribute component state to backup replicas in form of the CORBA `any` instance, which stores the data type information together with the value in a serialized format.

To allow for variability in state size and transport mechanisms used, CHESS uses the Strategy pattern [17, pp.315f] so applications can flexibly select the desired protocol at runtime. The state dissemination mechanism is represented by an object interface that provides a generic way to access all variants of state dissemination in a uniform manner. This pattern shields component developers from the concrete protocol for state dissemination. The Strategy pattern implementation is available as part of the StateSynchronizationAgent.

When applications create and register component replicas they can set a policy to determine which mechanism will be used by the agent. The agent then will instantiate the appropriate concrete strategy object instance and associate it with the application to use with every dissemination of state information.

Figure 9 shows how CHESS supports two different communication mechanisms using the Strategy pattern: (1) synchronous CORBA calls and (2) multicast communication based on OMGs Data Distribution Service (DDS) [16]. By providing a strategized approach to state

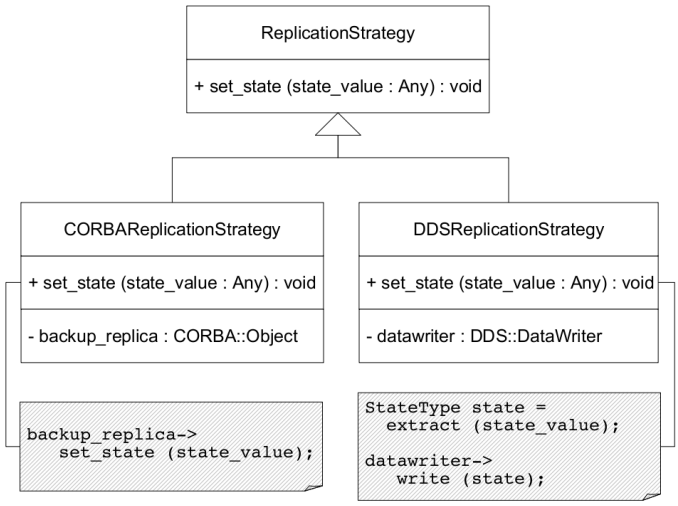

Figure 9: **The Strategy Pattern Applied to State Synchronization**

synchronization, applications can choose to configure the mechanism best suited to their performance and fault-tolerance requirements.

4. Qualitative and Quantitative Analysis of CORFU

This section evaluates CORFU using two different approaches. First, we conduct a qualitative analysis of development effort by comparing object-oriented development of fault-tolerant applications with development using the CORFU component-based infrastructure. Second, we evaluate CORFU's timing behavior to show its suitability for DRE systems by measuring failover

unit shutdown latency and client perceived fail over latency (which includes the state transfer capabilities provided by CHESS).

4.1. Evaluating Component-based Fault-Tolerance with Object-Oriented Fault-Tolerance

Developing applications based on distributed object-oriented fault-tolerance (*e.g.*, using the mechanisms provided by FLARe) incurs additional effort relative to component-based fault-tolerance (*e.g.*, using the mechanisms provided by CORFU). Below, we qualify this additional effort and contrast it with CORFU.

Development obligations of object-oriented fault-tolerance. FLARe requires different means to implement fault-tolerance on the server than on the client. The difference stems from variations in the infrastructure on the client and server. Figure 10 describes the obligations related to server development. These obligations can be grouped into (1) object implementation

CORBA 2.x Server Obligations		
Object Implementation	Initialization	Configuration
1. Implementation of get_state/set_state methods 2. Triggering state synchronization through state_changed calls 3. Getter & setter methods for object id & state synchronization agent attributes	1. Registration of IORInterceptor 2. HostMonitor thread instantiation 3. Registration of thread with HostMonitor 4. State Transfer Agent instantiation 5. Registration of State Transfer Agent with Replication Manager 6. Registration with State Transfer Agent for each object 7. Registration with Replication Manager for each object	1. ReplicationManager reference 2. HostMonitor reference 3. Replication object id 4. Replica role (Primary/Backup)

Figure 10: **Responsibilities for Server-Side Fault-Tolerance**

obligations that each CORBA servant must implement to integrate into the fault-tolerance infrastructure, (2) initialization obligations an application needs to perform to use FLARe functionality and (3) configuration obligations at start-up that configure application fault-tolerant aspects.

Some initialization steps (such as HostMonitor thread instantiation and registration) are just performed once per process. Other steps (such as object implementation obligations, application configuration, and registration of objects with the ReplicationManager), must be done for each object in a process. Client initialization is not as complex, but still involves several process-wide initialization steps (such as creating and registering the redirection agent and the request interceptor).

Consequences for application development. The presented obligations result in considerable effort for application development. Manually implementing these initialization steps in clients and servers increases the risk of accidentally omitting or confusing steps. It also limits software reuse for different deployment scenarios since the number and types of object replicas per-server process are hard coded. Collocating objects within one process require recompilation of the server application and changes of configuration metadata.

Benefits of CORFU's component-based approach. By integrating FLARe functionality into a fault-tolerant component server, CORFU overcomes many limitations with traditional object-oriented fault-tolerance approaches. For example, CORFU's client and server capabilities are available within the same component server, which is an important architectural capability since CORBA objects often play both client and server roles simultaneously. Below we present the benefits of CORFU's component server approach by evaluating them in terms of the three different types of obligations outlined above:

a. **Component-based application business logic.** The application business logic in a component is provided by the servant executor, which is similar to an object implementation in CORBA 2.x. Unlike CORBA 2.x, however, LwCCM provides code generation functionality in the form of the IDL and CIDL compilers that can automatically create necessary code artifacts thereby completely decoupling a servant executor from having to provide additional code to integrate with the fault tolerance mechanisms.

b. **Initialization.** Most client and server initialization states can be done automatically. CORFU's fault-tolerant component server hides the complexity of initializing FLARe entities from component developers. The registration of individual components with the framework are also done automatically by CORFU's fault-tolerance-aware session container.

c. **Configuration.** Instead of using proprietary mechanisms on a per-application level, CORFU's component server approach enables the use of standardized configuration mechanism provided by the LwCCM D&C specification. Special fault-tolerant component attributes are used in the context of CORFU's automated configuration framework, so that no proprietary solutions that differ from application-to-application are needed.

Summary of analysis. CORFU increases the transparency of using fault-tolerance mecha-

nisms for both client and server development. This transparency allows application developers to focus on implementing their application business logic, while fault-tolerance aspects can be added and configured orthogonally. It is possible to collocate fault-tolerant components without changing their implementation code. CORFU therefore also substantially improves the flexibility of system deployment and system evolution. Moreover, there are fewer possibilities for accidental faults in application development since initialization is performed in a standard way by the component server.

4.2. Experimental Results

Below we present experiments that evaluate the timing behavior of CORFU and quantify the overhead and latencies involved in its fail over mechanisms. First, we evaluate the overhead in client's call-path due to client-side interceptor and its capability to fail over to the next server replica in the order of RankList. Second, we evaluate the overall time taken by CORFU's FaultCorrelationManager to react to a failure and shutdown the primary failover unit. Third, we evaluate the client perceived latency with the increasing size of the failover unit and where the failure occurs within a failover unit. Finally, we evaluate how CORFU reacts in the case of processor failure. All experiments except the last one assume process failures only.

4.2.1. Testbed

All experiments have been conducted on ISISLab (`www.isislab.vanderbilt.edu`), which is a LAN virtualization environment containing upto 56 identical blades connected through 4 Gbps switches that allow for dedicated links per experiment. Each blade has two 2.8GHz Xeon CPUs and 1 gigabyte RAM. The Fedora Core 6 Linux distribution with rt11 real-time kernel patches is used as operating system. The enhancements to FLARe and the CORFU implementation are based on TAO version 1.6.8, a real-time CORBA implementation and CIAO version 0.6.8, which is an implementation of LwCCM. CORFU and all testing applications have been built using the GNU compiler collection (gcc) version 3.4.6.

4.2.2. Overhead Measurements

Experiment setup. This experiment compares the overhead a client experiences for CORBA 2.x object-oriented applications and LwCCM component-based applications. A client application periodically invokes an operation on a replicated server application. For each call the server

processing time and the response time on the client side are measured. The communication latency is calculated by subtraction of the processing time from the response time.

Requests are made with a period of 200 milliseconds. A defined execution time of 20 milliseconds is realized through the CPU worker component of the system execution modeling tool CUTS[18]. On the eleventh invocation, a fault is injected in the server process to shut it down, which then causes the client to fail over to the server's backup replica.

All primary servers are hosted on one host, the backup servers are hosted on a separate host. The clients are deployed on an additional host. CHESS state synchronization module is disabled in this experiment because it is needed only when the primary servers modify their state.[1]

The experiment is implemented in two variants. The first variant is object-oriented and consists of a client and a server executable that directly use FLARe functionality. The second variant is component-based and uses CORFU's fault-tolerant component server. Each variant has three different experiment configurations with one, two, and four client server groups running simultaneously. We repeat each measurement configuration 100 times to obtain representative results.

Analysis of results. An example for a single measurement for fail over latency given in Figure 11 represents the component-based case with one running component. Ten invocations before and after the failure event are recorded. The first ten invocations show a communication overhead between zero and one millisecond, which represents failure-free communication with the primary server component.

The client experiences an increased response time on the eleventh request, since the primary server is no longer responding. This results in a client fail over that involves the interception of a CORBA exception and the forwarding to a backup replica. As shown in Figure 11, latency increases from one millisecond to four milliseconds for the client.

Figure 12 shows the latency averages and jitter minima and maxima as measured in all six configurations. The CORBA 2.x-based object-oriented experiment with one application shows a communication overhead of approximately three milliseconds, while the corresponding component-based experiment has a latency of four milliseconds. This result shows that the extra cost for CORFU's LwCCM component-based fault-tolerance with 25 percent additional overhead is relatively small.

[1] Since state synchronization overhead is highly application dependent and can skew overhead measurements, we take state synchronization time into account in the third experiment, where we measure client perceived fail over latency.

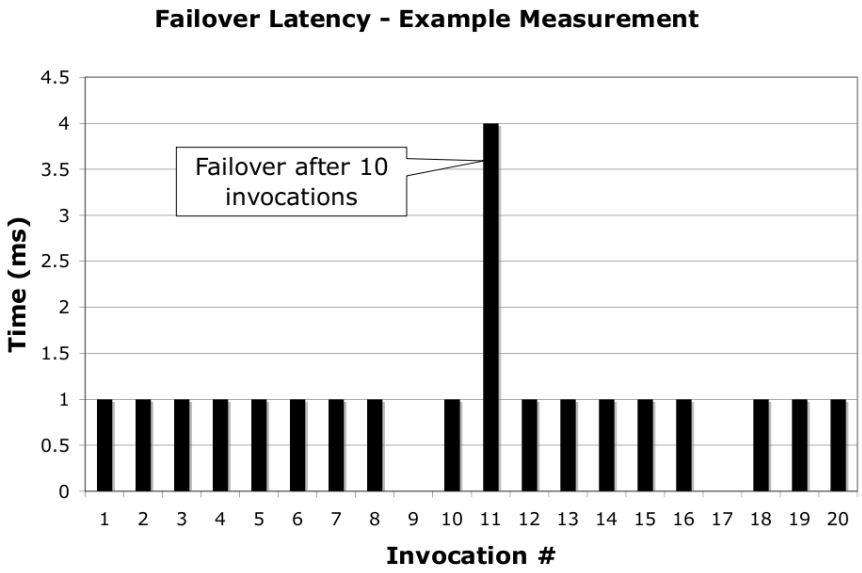

Figure 11: **Single Fail over Latency Measurement**

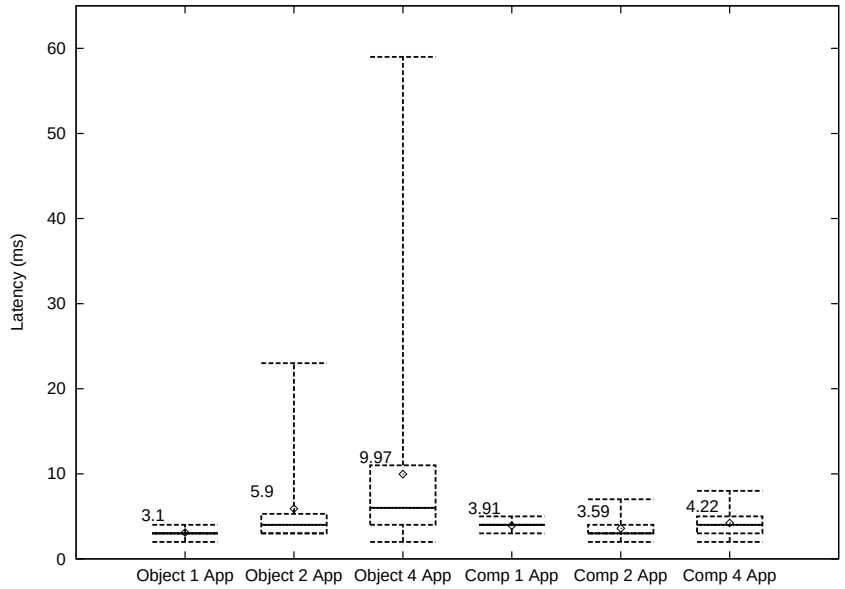

Figure 12: **Results for Failover Latency Measurements**

The component-based experiments with configurations of two and four applications have a much lower jitter and a similar average of four milliseconds, whereas the object-oriented examples have growing latencies. This latency and jitter increase—which is proportional to the number of applications—is not directly related to the fail over mechanism but reflects the implicit differences between the experiment variants. In the object-based case, executables start processing right away while a component is first loaded into the container and then triggered later on to start processing. Nevertheless, the results show that there is no unreasonably high overhead for CORFU's component-based fault-tolerance.

4.2.3. Failover Unit Shutdown Latency

Experiment setup. This second experiment measures the latency involved in the process of shutting down failover units of various sizes. The following factors contribute to shutdown latency:

a. *Error detection and notification delay* from the failure of a component to the beginning of the notification processing in the FaultCorrelationManager.

b. *Reaction delay* within the FaultCorrelationManager to determine which components are affected and which deployments therefore need to shutdown. This latency is dependent on the size of the failover unit.

c. *Shutdown time* using the LwCCM D&C services (DomainApplicationManager) and its interfaces to destroy the affected NodeApplications. Depending upon how D&C services are implemented, this latency may or may not depend on the size of the failover unit.

To observe the impact of the size of the failover unit, we increased the number of components in a failover unit from two to five. The structure of the five component experiment and the sequence of events are shown in Figure 13. The setup includes seven processing nodes of which one node is dedicated for the CORFU management entities, such as the ReplicationManager, the FaultCorrelationManager, the ExecutionManager, and other elements of the LwCCM D&C run-time. Five other nodes have a HostMonitor deployed to observe the system state per node.

Each node hosts one component for each of the five deployed failover units. There is one primary failover unit that includes one component per node, named A_0 to E_0. This failover unit is replicated four times ensuring that no two replicas of the same component are collocated on the same node. Each backup unit contains replica components A_n to E_n of each component in

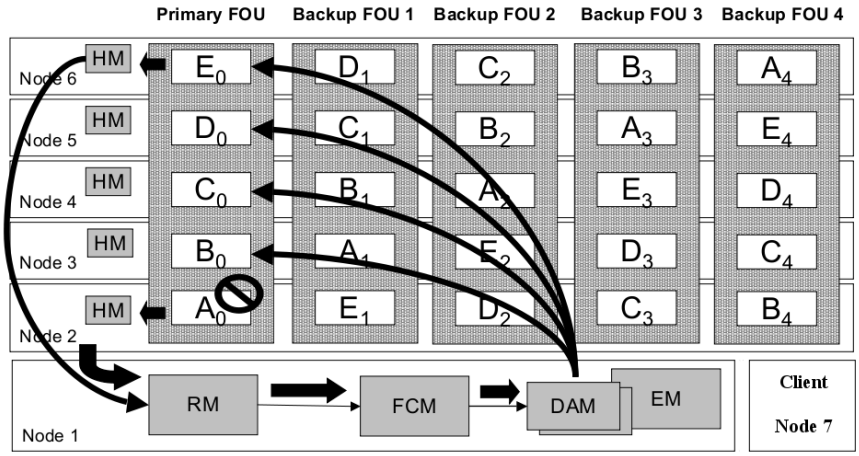

Figure 13: **Setup for FOU Shutdown Latency Measurement**

the primary unit. The fail over order of the units corresponds to their number.

The client is deployed on node-7 as shown in Figure 13. At the beginning of the experiment, the client initiates communication with component A_0. The failures are triggered at the predetermined iterations in the active server component. Subsequently, client fails over to the next replica of component A_n, which belongs to the next failover unit.

As the experiment progresses, failures are triggered leading to a failover sequence of primary FOU, backup FOU 1, backup FOU 2, and so on. Each experiment run therefore allows us to measure four fail over latencies. Due to the need for consistent time, all measurements are taken on node-1 in the FaultCorrelationManager, which alleviates the need for synchronized clocks. The events are timestamped in the following sequence:

a. A failure is provoked in component A_n of the active FOU, which is detected by the HostMonitor and reported to the ReplicationManager.

b. The ReplicationManager notifies the FaultCorrelationManager about the failure and the FaultCorrelationManager takes a timestamp t_1.

c. The FaultCorrelationManager performs the FAILURE-REACTION algorithm and takes a timestamp t_2 after the affected failover units have been identified in phase 1.

d. The FaultCorrelationManager accesses the DomainApplicationManager to retrieve all node applications for the corresponding deployment plans and iterates through them to shut them down. After the last call is returning, a third timestamp, t_3, is taken to indicate the

Size of failover unit	$t_{\text{round-trip}}$ (t_4 - t_1) (ms)			t_{shutdown} (t_3 - t_2) (ms)		
	min	avg	max	min	avg	max
2	192	215.6	265	13	14.4	16
3	138	225.4	283	18	21.9	26
4	130	263.1	313	23	26.9	31
5	252	267.4	310	32	32.9	35

Table 1: **Shutdown Latencies (Min and Max) of Failover Units of Various Sizes**

end of the shutdown request.

e. The HostMonitors notify ReplicationManager about all the shutdowns of the affected components, which in turn notifies FaultCorrelationManager. Upon reception of the last shutdown notification, a time-stamp t_4 is taken in FaultCorrelationManager that represents the time when the failover unit is completely shutdown.

Analysis of results. The results are summarized in Table 1. Three essential components of the latency from this experiment include the following:

- **Reaction time** (t_{reaction}), which is the time spent within the FaultCorrelationManager between the arrival of the first failure notification and the beginning of the shutdown process. The reaction time constitutes the time needed to execute the FAILURE-REACTION algorithm and to serialize incoming notifications into a thread-safe queue to ensure correct processing of potentially concurrent error notifications. We observed that t_{reaction} was consistently under one millisecond in the experiment and therefore we do not consider it further in the experiment.
- **Shutdown time** (t_{shutdown}), which is the time needed to invoke D&C services provided by the DomainApplicationManager to terminate the remaining live components in the failover unit where the failure occurred. Table 1 shows that minimum, average, and maximum values of t_{shutdown} increase linearly with the increasing size of the failover unit. This result is clearly an artifact of the implementation detail because an iterative construct is used in the FaultCorrelationManager to invoke the LwCCM D&C services.
- **Round-trip time** ($t_{\text{round-trip}}$), which is the time difference between the notifications of the first component failure and the last component shutdown (t_{reaction} and t_{shutdown} are both subcomponents of $t_{\text{round-trip}}$). In spite of the predictability of prior two sub-components,

Table 1 indicates that $t_{\text{round-trip}}$ is not predictable (but bounded at 313 milliseconds). This result is expected due to the inherent non-determinism in three timing subcomponents: (1) the time taken by component server processes to shutdown gracefully, (2) the subsequent failure notification going from the HostMonitor to the ReplicationManager, and (3) concurrent handling of these notifications in the FaultCorrelationManager.

4.2.4. Impact of Failover unit Size on Client Perceived Shutdown Latency

Experiment setup. The third experiment measures client perceived fail over latency and how the location of the failing component in a failover unit affects it. The deployment of components and failover units in this experiment is identical to the previous one, though we make two important changes in the functionality of the application:

- We enabled CORFU's state synchronization module to allow server components to synchronize their internal state with their replicas. Synchronization occurs synchronously using CORBA at the end of the remote method call. We did not use DDS in these experiments.
- We allow nested synchronous invocations from component A_i to B_i, B_i to C_i, C_i to D_i, and D_i to E_i for every invocation made by the client, where i indicates then active failover unit. Upon invocation, every component (except E_n) immediately invokes its following component and waits for it to return. Upon returning, it consumes CPU for a predetermined (20 milliseconds) amount of time. These nested invocations allow us to trigger failures in different components instead of the just the *head* component client talks to. In this experiment, we trigger failures in the *head* component (A_i) as well as in the *tail* component (E_n) and observe its effect on the client perceived fail over letency.

When a *tail* component fails, the client remains unaware of such a failure and continues waiting for the *head* component to return. Client interceptors detect failure of the *head* component only after FaultCorrelationManager shuts it down as described in the previous experiment. In such a case, failover unit shutdown latency and the order in which components are shutdown affect the client perceived failover latency.

Analysis of results.

Figure 14 and Figure 15 summarize the effects of location of the failing component on client perceived fail over latency with increasing size of failover unit.

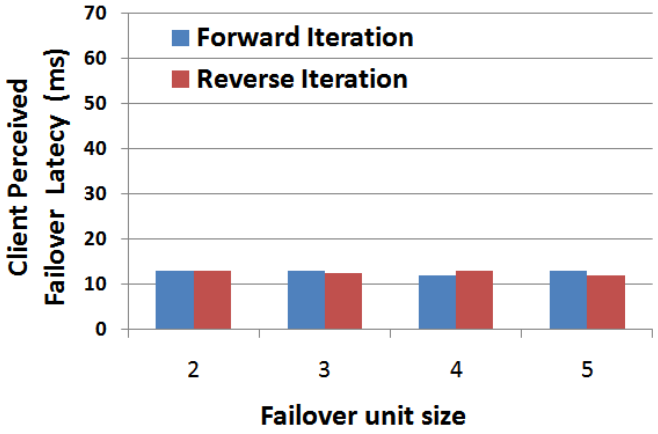

Figure 14: **Client Perceived Fail over Latency When the Head Component Fails**

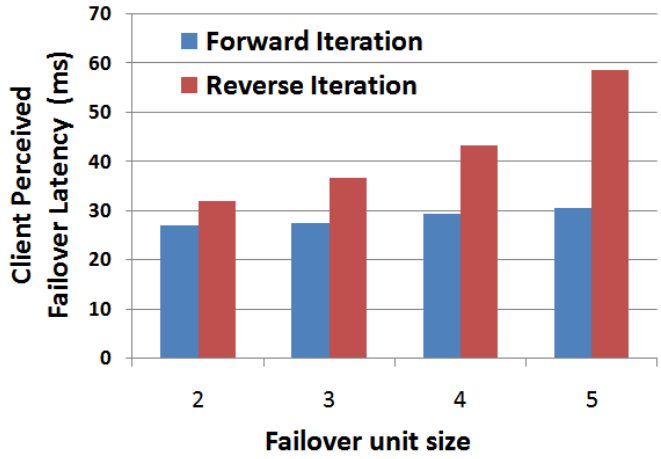

Figure 15: **Client Perceived Fail over Latency When the Tail Component Fails**

When the *head* component (A_n) of a failover unit dies, client recognizes the failure immediately, and CORFU's interceptors redirect the client to the next available failover unit. We therefore do not observe a linear increase in the client perceived fail over latency, even with the increasing size of failover unit in Figure 14.

The observed latency also remains unaffected by the order in which D&C services shutdown the components. In forward (A_n to E_n) and reverse (E_n to A_n) iteration order, the fail over latency remains tightly bounded at 13 milliseconds. The increase in the client perceived fail over latency compared to the first experiment is due to the state synchronization overhead at the end of each nested invocation.

In contrast to the *head* component failure, client perceived fail over latency is sensitive to the varying sizes of failover units when the *tail* component fails. Figure 15 shows that when LwCCM D&C services shutdown components in the forward (A_n to E_n) order, client perceived fail over latency remains relatively constant because as soon as the *head* component is shutdown, the client fails over to the backup failover unit.

In the case of reverse (E_n to A_n) iteration, conversely, the *head* component is the last one to shutdown causing client to wait longer in proportion to the size of the failover unit. Moreover, t_{shutdown} component described in the earlier experiment also gets added to the overall client perceived latency.

4.2.5. Impact of Processor Failures.

Experiment setup. The fourth experiment measures the impact on client perceived fail over latency when processors fails. We maintained the same experimental setup as before to compare process and processor failures. We ran two variants of the experiment: the first simulates a processor failure of the node hosting the *head* component and the second simulates a processor failure of the node hosting the *tail* component.

To simulate processor failure we used the *iptables* utility, which allows us to setup, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel. We insert a new filter rule in the *OUTPUT* chain of the kernel's iptables that drops every outgoing TCP packet to the nodes involved in the experiment. The filtering rule has the following two effects on the experiment:

- ReplicationManager ceases to obtain the periodic heart-beat beacon from the HostMonitor running on the node that fails. When the ReplicationManager fails to receive 3 consecutive heart-beats from the same node, it declares node failure and informs the FaultCorrelation-

Manager.

- The component that invoked the remote method call on the component running on failed node, waits till the method call returns. In fact, the method call never returns because all the outgoing TCP traffic is dropped at the failed node. The client will remain blocked till the remote method call returns from the *head* component if the node hosting that component fails.

Analysis of results. In case of the *tail* node failure, the client perceived fail over latency was identical to that of Figure 15 since FaultCorrelationManager begins shutting down the primary failover unit as soon as it receives a failure notification from the ReplicationManager. The *head* component the client is talking to is also shutdown by D&C services and subsequently, the client fails over to its next available failover unit.

In the case of *head* node failure, however, we observed that client perceived fail over latency is much higher than before, although FaultCorrelationManager shuts the failover unit down. In fact, it is bounded by TCP's retry timeout, which is at least 100 seconds as specified in RFC 1122 [19]. We verified this result by modifying the default value, 15, of *tcp_retries2* variable of kernel's IPv4 module using *ipsysctl* utility. When the value of *tcp_retries2* variable was low, the client TCP connection times out much earlier.

This experiment indicates that current CORFU's infrastructure requires enhancements, particularly in the RedirectionAgent, to force clients fail over in a timely fashion in the case of *head* processor failure. In the following section we describe how we plan to improve client perceived latency and address other limitations.

4.2.6. Summary of the Analysis

The experiment results described above showcase several benefits of CORFU. Using a client-side fail over mechanism allows for short fail over latencies since communication with the central replication manager in the instant of a failure is avoided. This interaction with the ReplicationManager would be a bottleneck in performance of large-scale DRE systems.

As shown by the first experiment, the client-side fail over latency is relatively small, being four milliseconds for the component variant. Having evaluated the benefits for CORFU concerning application development and system deployment we also needed to ensure that this does not drastically degrade performance and therefore render the solution unusable for DRE applications. As our experiment shows, client fail over in CORFU is comparable in performance and

incurs only minimal overhead, having an average response time of four milliseconds.

Compared to the client fail over latency the failover unit shutdown latency of more than 200 milliseconds on average is relatively high. The reason for this is partly to be found in the iterative way a deployment has to be shutdown based on the domain application and node application interfaces. Another source of high response times is the communication time between the different entities, such as the HostMonitors, the ReplicationManager and the FaultCorrelationManager. The internal reaction time of the FaultCorrelationManager to determine deployments that are affected by faults is already optimized through the use of hash maps with close to constant access times.

Based on these sources of overhead, we envision the following four approaches to reduce the round-trip latency for failover unit shutdown and improve client perceived fail over latency:

- **Concurrent shutdown**. To reduce the shutdown latency, the calls initializing shutdowns for affected node applications can be parallelized instead of being done in sequential order. A suitable mechanism is the CORBA Asynchronous Method Invocation (AMI) [20, 21] specification. AMI allows the FaultCorrelationManager as a client to issue all shutdown requests without having to wait for their response in between, which would significantly reduce shutdown time, especially in large deployments.
- **Collocation of management entities**. Some communication paths, especially between ReplicationManager, FaultCorrelationManager, and ExecutionManager can be optimized by collocating these entities into the same process space. This optimization greatly reduces communication times since the network stack can be avoided and in-process (*i.e.*, loopback) communication mechanisms are used instead.
- **Real-time CORBA**. For the communication paths that need to go through the network, communication can be made more reliable and deterministic by using Real-time CORBA [22] features, such as the real-time scheduling service, private connections, pre-allocation of connections, and end-to-end priority preservation.
- **Enhanced redirection agent**. CORFU's existing Redirection Agent reacts only when system-level exceptions are raised on the client-side. In the case of the *head* processor failure, these exceptions are thrown when the TCP connection times out, which is much later than the actual event of processor failure. The ReplicationManager must therefore communicate to the Redirection Agent not only the RankList of IORs but also whether a

processor has failed. The Redirection Agent must perform extra steps to close the soon-to-be-dead TCP connection and force the client to fail over.

Although there is still potential for performance improvement, the measurements show that CORFU is suitable for DRE systems, such as our MCS case study described in Section 2, and offers comparable performance to the distributed object-oriented computing fault-tolerance provided by FLARe.

5. Related Work

This section compares our work on CORFU with related research in DRE systems along the following three dimensions:

- **Frameworks for fault-tolerance.** Since CORFU provides fault-tolerance support to component-based DRE systems we compare and contrast it with related fault-tolerance frameworks.
- **Dependency analysis for fault correlation.** Since CORFU provides fault-tolerance for a group of components treated as a failover unit we compare and contrast it with related work on fault correlation frameworks.
- **Modeling dependability aspects.** Since CORFU is geared to integrate with and leverage the analysis of model-based frameworks (such as MDDPro [13]) we compare it with related modeling efforts.

In each dimension we also compare CORFU with our prior work and summarize the novel contributions made by CORFU relative to this earlier work.

5.1. Frameworks for Fault-Tolerance

A framework for fault-tolerance integrates different aspects of dependability, including error detection, fault diagnosis, fault isolation, error recovery including state consistency, and system reconfiguration. Other forms of dependability, such as fault prevention, fault removal, and fault forecasting, can also benefit from fault-tolerance frameworks. Below we compare CORFU with prior work that covers a wide range of fault-tolerance mechanisms provided by different frameworks and different domain scope.

Related work that is most similar to CORFU are AQUA [23, 24] and JAGR [25]. AQUA uses ACTIVE replication to provide both availability and timeliness capabilities for applications, and optimizes the response times for applications by dynamically deciding on the number of

replicas executing the request. AQuA objects are contained in replication groups that provide a variety of replication schemes realized by a message-based group communication mechanism. AQuA uses CORBA to define and implement objects but maps them to the underlying group communication mechanism. The mapping layer includes mechanisms for error detection and failover. The fault model includes process failures that are detected through heartbeat messages and data value failures. A centralized dependability manager coordinates groups and manages the fault tolerance infrastructure.

While AQuA supports fault-tolerance at the granularity of objects, component-based systems often includes additional levels of granularity. For example, components themselves can comprise objects and hence dependencies between components can result in their need to failover together. Component-based frameworks such as CORFU go beyond the general framework approach provided by mechanisms like Aqua by additionally defining a component life-cycle and development process. The benefit of such a process is that it allows to formalize other aspects of dependability, such as fault prevention through offline analysis or methodologies for fault removal and system validation.

JAGR [25] builds on a component-based infrastructure for the domain of three tier web applications with permanent data storage. It focuses on intelligent fail over mechanisms based on dependency information gained through automatic failure path interference as described earlier. JAGR's main components are a modular monitoring structure that allows to plug in different monitors for different error types. An intelligent recovery manager gathers this information and applies micro-reboots to restart parts of the system that are affected. Based on the result it can escalate the reboot scope from single components to the whole system.

Mechanisms such as JAGR apply predominantly to enterprise systems where persistence of data is important and for which large storage capacities are available. In DRE systems, however, persistent data storage and stateless components cannot be applied in all cases due to limited storage and processing resources. The CHESS state management mechanism provided by CORFU therefore takes into account state replication of individual components and provides fail over mechanisms as a major means for fault-tolerance instead of micro-reboots provided in Jagr.

There are also other frameworks for fault tolerance. For example, Delta-4/XPA [26] provides real-time fault-tolerance to distributed systems using semi-active replication. MEAD [27] and its proactive recovery strategy for distributed CORBA applications can minimize the recovery time

for DRE systems. The Time-triggered Message-triggered Objects (TMO) project [28] considers replication schemes such as the primary-shadow TMO replication (PSTR) scheme, for which recovery time bounds can be quantitatively established, and real-time fault tolerance guarantees can be provided to applications. DARX [29] provides adaptive fault-tolerance for multi-agent software platforms by dynamically changing replication styles in response to changing resource availabilities and application performance.

Many mechanisms for providing fault tolerance and assuring timeliness properties in these related works are orthogonal to the focus of CORFU. It is conceivable to design a different container mechanism in CORFU that can support active replication as in MEAD, or support the dynamic changing of replication styles as in DARX.

Our earlier work on the DOORS framework [30, 31] provides warm-passive replication to CORBA objects. Efforts such as DOORS and Eternal [32] led to the standardization of the Fault-tolerant CORBA specification [33]. These efforts focus on object-based fault-tolerance, and did not address real-time requirements of component-based DRE systems.

Our recent work called GRAFT [34] provides a generative approach to deal with group fail over. Like CORFU, the GRAFT project identifies the lack of first-class support for fault-tolerance in component middleware. Unlike CORFU (which provides a first-class middleware support for group fail over), however, GRAFT relies on an aspect-oriented approach to weave in group-based fault-tolerance.

The GRAFT approach may become a limiting factor when fault management and recovery yields interactions with complex semantics (*e.g.*, timing and state consistency). In these circumstances, first-class support within the middleware is preferable. Moreover GRAFT uses exceptions to detect critical errors, whereas CORFU provides a monitoring framework that supports advanced error detection, such as failure of component server processes, and failure of the nodes hosting these servers.

5.2. Dependency Analysis for Fault Correlation

A key challenge for effective failure handling is to gain comprehensive knowledge about which parts of a system are affected by a fault. Faults cannot be detected directly but only through the resulting errors they cause. Pinpointing the cause of faults allows reasoning about system parts that are affected by the original fault. This allows fast reaction to errors before they can cause subsequent errors in other parts of the system.

Obtaining knowledge of error propagation dependencies between system elements is therefore crucial to realize dependable systems. This information can be used to determine which system parts will eventually be compromised. This enables comprehensive failure handling as opposed to simple reactive approaches that provide only monitoring for the basic elements of the system. Fault correlation is particularly important to support group fail over since any single fault within any of the participant of a failover group must quickly be correlated as a group failure thereby effecting a rapid group fail over.

Research on detection and expression of failure dependencies between system components can be categorized into (1) static modeling and (2) observation-based techniques. Static modeling follows a white box approach that allows system developers to explicitly specify different types of dependencies and then reasons on fault propagation based on this information. Observation-based modeling treats systems as a black box and uses fault injection and monitoring to analyse which errors cause which parts of the systems to fail. This information is then used to build a system model.

Viera et. al. [35] present an approach that automates dependency analysis in component-based systems. The *Component-Based Dependency Model* allows incorporating diverse types of dependencies that are categorized into intra-component dependencies that define execution and error propagation paths within one component implementation, and inter-component dependencies that define dependencies on external component or hardware and software infrastructure elements.

This approach is powerful because it integrates different sources of information about the system, such as deployment information, additional component metadata and metadata about component connection, all of which help in the correlation. A concrete example is event correlation [36] in the domain of event based systems, where dependencies between different event sources are used to identify the original fault. This approach is static, however, since it builds its dependency information based on statically known metadata. It therefore cannot react to unforeseen or emergent failures, and error propagation paths.

To address the limitations of static dependency information the *Automatic Failure-Path Inference* [37] and Active Dependency Discovery [38] approaches rely on system behavior analysis at runtime. For example, the automatic failure-path inference approach focuses on component-based web applications implemented in Java and assumes that errors express themselves as ex-

ceptions. Fault dependencies are captured as a directed graph called failure-propagation map. This graph is populated through direct interaction with the system.

Fault injection and monitoring of resulting component crashes is used to build up an initial graph for a system. Subsequently, this graph is corrected based on non-intrusive monitoring of the system under nominal operation. While this approach is very flexible in adopting the dependency information to the system it is limited in its support of different fault types due to its focus on exceptions and the Java programming language.

This related work on dependency analysis relates to our research on CORFU since they provide methodologies to define groups of dependent components. The related work is generally orthogonal to CORFU, however, since they provide algorithms for dependency analysis, whereas CORFU provides mechanisms to honor the dependencies. It is conceivable that to improve the fail over shutdown latencies shown in Section 4.2, CORFU's component lifecycle mechanisms may include the pre-determined dependencies to expedite the group shutdown process.

A related approach known as *Rx* [39] handles deterministic software faults. Rx treats software faults as allergies correlating the exact cause of the fault to its operating environment. When Rx encounters a fault, it rolls back the application to an earlier checkpointed state and reexecutes the application in a different operating environment. The premise behind this approach is that the operating environment and parameters are the likely cause for the software fault, so a change in the conditions may help eliminate the fault.

While Rx is a mechanism to handle deterministic faults (which arise predominantly from coding mistakes or imperfect assumptions of the operating environment), CORFU focuses on process and processor crash faults. If the component implementation is imperfect, therefore, CORFU can benefit from Rx by forcibly failing over the imperfect group to a backup group that is deployed in a different operating environment.

5.3. Modeling Dependability Aspects

CORFU provides runtime middleware mechanisms for component-based fault-tolerance, particularly for a group of components treated as a single unit. The deployment and runtime mechanisms in CORFU can benefit from offline analysis tools for dependability. For example, such an analysis could include decisions on where to deploy the components so that the overall reliability of the system improves. Below we provide some examples of related work in this area.

Cadena [40] is a model-driven engineering tool that supports modeling of component behavior early in the design process based on property specifications that capture high-level component information. This information includes inter-dependencies between ports of other components and intra-dependencies that capture relationships between ports of the same component. The properties also capture behavioral specifications that allow reasoning of temporal behavior and control-flows within components.

Based on this information, interface definitions and assembly descriptions of the system model can be constructed to allow reasoning of various system aspects, such as event rate assignment, effective component distribution that minimizes network traffic, and schedulability analysis. These aspects can consider replicas of system components to support fault-tolerance. Cadena not only encompasses a runtime framework, but also a domain-specific modeling tool suite for system modeling and a simulation environment for model checking and verification.

Our earlier work on MDDPro [13] focused on modeling dependability QoS requirements. MDDPro's domain-specific modeling language provides an orthogonal view to the deployment structure of a system and allows the annotation of fault-tolerance attributes to components. It introduces three concepts to explicitly model component replication:

a. *Failover units*, which define a group of system entities as an atomic unit of failover. Different parameters can be defined on the group that characterize the type of failure recovery strategy used (*e.g.*, number of replicas, heartbeat frequency, among others).

b. *Replication groups*, which define what components replicate the same logical object. Replication groups can be used to configure state synchronization policies, such as synchronizing the state every N requests, where N is configurable; or defining the levels of consistency including weak or eventual consistency.

c. *Shared risk groups*, which can model the probability of a failure propagating from one processing node to other nodes. This model is realized as a tree where edges represent neighboring nodes and distances in number of edges serve as a measure for how likely a failure is to propagate.

The MDDPro modeling framework can be strategized with different placement algorithms that determine the mapping of replicas to nodes. In particular, MDDPro focuses on algorithms that automatically place components and their replicas to minimize the chances of simultaneous failures. It also generates the necessary deployment metadata via model interpreters. CORFU

complements Cadena and MDDPro by providing a runtime middleware infrastructure that can process and instantiate systems modeled with Cadena or MDDPro.

6. Concluding Remarks

The state-of-the-art in fault-tolerant DRE systems has not accounted for application development effort, application lifecycles, and system evolution simultaneously. Moreover, many middleware-based solutions provide relatively low-level abstractions, *e.g.*, function-based or object-oriented. In contrast, component-based middleware can provide sophisticated fault detection and recovery that is suitable for DRE systems, while also improving transparency of fault-tolerance aspects in the application development process, thereby enhancing DRE system flexibility, evolvability, and quality. This paper describes how our CORFU middleware addresses key challenges of component-based fault-tolerance, including the need for efficient synchronization of internal component state, failure correlation across groups of components, and configuration of fault-tolerance properties at the component granularity level.

We learned the following lessons from our work on CORFU presented in this paper:

a. Fault-tolerance affects all aspects of a system and introduces a new dimension of complexity. It is therefore hard to capture all fault-tolerance aspects in a comprehensive middleware framework. Application characteristics differ greatly even within the DRE domain, which impacts decisions on what protocols are used, architectural concepts applied and technologies chosen. Each of these choices might require different approaches to fault-tolerance.

b. Component-based middleware allows for greater fault-tolerance transparency. As demonstrated by CORFU's fault-tolerant component server, the component-based development paradigm and lightweight fault-tolerance integrate well, thereby hiding key sources of complexity in this domain.

c. Layering and separation of concerns foster flexible and extensible architectures. This lesson became clear in the design of CORFU's FaultCorrelationManager. By building the failover units on top of the existing object based approach and separating concerns through fail over constraints, the FaultCorrelationManager design and implementation could be kept small and focused on its main task to analyze the system infrastructure and react on failures using other existing software, namely the deployment and configuration infrastructure.

d. Although separation of concerns is needed to foster flexibility, care must be taken to ensure that it does not impact performance. For example, the clean separation of the fault-tolerance capabilities within the container model of LwCCM and the reliance on the D&C actors for startup and shutdown may impact performance, particularly when a failover unit must be shutdown in a timely manner. Leveraging the D&C process in a traditional manner leads to sequential invocation of D&C actors to shutdown the necessary components. As the size and distribution of failover units increase, the shutdown latency will increase linearly, which impacts client-perceived response times.

e. Performance of fault-tolerance is hard to measure due to the singular nature of failures and non-determinism in networks, operating system and middleware. Since faults are not periodic events in systems expecting fail-stop behavior, the setup of experiments is complex. Each measurement can only measure a limited number of faults before the complete system has to be restarted. The nature of DRE systems also makes it hard to gather reliable timing information due to network jitter, operating system scheduling and other sources of non-determinism. Experiments and testing scripts must be automated to allow a sufficient number of single measurements.

CORFU is available in open-source form as part of the CIAO LwCCM distribution available from `www.dre.vanderbilt.edu/CIAO` .

References

[1] A. Romanovsky, A looming fault tolerance software crisis?, SIGSOFT Softw. Eng. Notes 32 (2) (2007) 1–4. doi:http://doi.acm.org/10.1145/1234741.1234767.

[2] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, E. Böde, Boosting Re-use of Embedded Automotive Applications Through Rich Components, Proceedings of Foundations of Interface Technologies 2005.

[3] M. A. D. Miguel, Qos-aware component frameworks, in: IWQOS '02: Proceedings of the Tenth International Workshop on Quality of Service, IEEE Computer Society, Washington, DC, USA, 2002, p. 51.

[4] G. Ahlforn, E. Örnulf, Ericsson's Family of Carrier-class Technologies, Ericsson Review 4 (2001) 190–195.

[5] D. C. Sharp, Reducing Avionics Software Cost Through Component Based Product Line Development, in: Software Product Lines: Experience and Research Directions, Vol. 576, 2000, pp. 353–370.

[6] G. T. Heineman, B. T. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, Reading, Massachusetts, 2001.

[7] Clemens Szyperski, Component Software — Beyond Object-Oriented Programming - Second Edition, Addison-Wesley, Reading, Massachusetts, 2002.

[8] Object Management Group, Lightweight CCM FTF Convenience Document, ptc/04-06-10 Edition (Jun. 2004).

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

[10] N. Peccia, Egos: Esa/esoc ground operations software system, 2005, pp. 3988–3995.

[11] Object Management Group, Light Weight CORBA Component Model Revised Submission, OMG Document realtime/03-05-05 Edition (May 2003).

[12] OMG, Deployment and Configuration of Component-based Distributed Applications, v4.0, Document formal/2006-04-02 Edition (Apr. 2006).

[13] S. Tambe, J. Balasubramanian, A. Gokhale, T. Damiano, MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems, in: Proceedings of the International Service Availability Symposium (ISAS), Vol. 4526 of Lecture Notes in Computer Science, Springer, Durham, New Hampshire, USA, 2007, pp. 127–144.

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

[15] F. Wolf, J. Balasubramanian, A. Gokhale, D. C. Schmidt, A State Transfer Framework for Object Oriented Fault-Tolerance, Tech. Rep. ISIS-09-106, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN (October 2009).

[16] Object Management Group, Data Distribution Service for Real-time Systems Specification, 1.2 Edition (Jan. 2007).

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[18] J. Hill, J. Slaby, S. Baker, D. Schmidt, Applying system execution modeling tools to enterprise distributed real-time and embedded system qos, in: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia, 2006.

[19] R. Braden, Requirements for Internet Hosts, Network Information Center RFC 1122 (1989) 1–116.

[20] Object Management Group, CORBA Messaging Specification, Object Management Group, OMG Document orbos/98-05-05 Edition (May 1998).

[21] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, J. Parsons, The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging, in: Proceedings of the Middleware 2000 Conference, ACM/IFIP, 2000.

[22] Object Management Group, Real-time CORBA Specification, 1.2 Edition (Jan. 2005).

[23] S. Krishnamurthy, W. Sanders, M. Cukier, A Dynamic Replica Selection Algorithm for Tolerating Timing Faults, DSN' 01 (2001) 107–116.

[24] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, M. Seri, Aqua: An adaptive architecture that provides dependable distributed objects, IEEE Transactions on Computers 52 (1) (2003) 31–50.

[25] G. Candea, E. Kiciman, S. Zhang, P. Keyani, A. Fox, Jagr: an autonomous self-recovering application server, Autonomic Computing Workshop, 2003 (2003) 168–177.

[26] D. Powell, Distributed Fault Tolerance: Lessons from Delta-4, IEEE Micro 14 (1) (1994) 36–47.

doi:dx.doi.org/10.1109/40.259898.

[27] S. Pertet, P. Narasimhan, Proactive recovery in distributed corba applications, in: DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, IEEE Computer Society, Washington, DC, USA, 2004, p. 357.

[28] K. H. K. Kim, C. Subbaraman, The pstr/sns scheme for real-time fault tolerance via active object replication and network surveillance, IEEE Trans. on Know. and Data Engg. 12 (2). doi:dx.doi.org/10.1109/69.842258.

[29] O. Marin, M. Bertier, P. Sens, Darx: A framework for the fault-tolerant support of agent software, in: ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering, IEEE Computer Society, Washington, DC, USA, 2003, p. 406.

[30] B. Natarajan, A. Gokhale, D. C. Schmidt, S. Yajnik, DOORS: Towards High-performance Fault-Tolerant CORBA, in: Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000), OMG, Antwerp, Belgium, 2000.

[31] B. Natarajan, A. Gokhale, D. C. Schmidt, S. Yajnik, Applying Patterns to Improve the Performance of Fault-Tolerant CORBA, in: Proceedings of the 7^{th} International Conference on High Performance Computing (HiPC 2000), ACM/IEEE, Bangalore, India, 2000.

[32] L. Moser, P. Melliar-Smith, P. Narasimhan, A Fault Tolerance Framework for CORBA, in: International Symposium on Fault Tolerant Computing, Madison, WI, 1999, pp. 150–157.

[33] Object Management Group, Fault Tolerant CORBA Specification, OMG Document orbos/99-12-08 Edition (Dec. 1999).

[34] S. Tambe, A. Dabholkar, 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), IEEE Computer Society, Tokyo, Japan, 2009.

[35] M. Vieira, D. Richardson, Analyzing dependencies in large component-based systems, Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on (2002) 241–244doi:10.1109/ASE.2002.1115020.

[36] B. Gruschke, A new approach for event correlation based on dependency graphs, in: In 5th Workshop of the OpenView University Association, 1998.

[37] G. Candea, M. Delgado, M. Chen, A. Fox, Automatic failure-path inference: A generic introspection technique for internet applications, in: WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications, IEEE Computer Society, Washington, DC, USA, 2003, p. 132.

[38] A. Brown, G. Car, A. Keller, An active approach to characterizing dynamic dependencies for problem determination in a distributed application environment, IEEE/IFIP International Symposium on Integrated Network Management (2001) 377–390.

[39] F. Qin, J. Tucek, Y. Zhou, J. Sundaresan, Rx: Treating Bugs as Allergies: A Safe Method to Survive Software Failures, ACM Transactions on Computer Systems (TOCS) 25 (3) (2007) 7.

[40] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, V. P. Ranganath, Cadena: An integrated development, analysis, and verification environment for component-based systems, Software Engineering, International Conference on 0 (2003) 160. doi:http://doi.ieeecomputersociety.org/10.1109/ICSE.2003.1201197.