# Component Replication based on Failover Units

Friedhelm Wolf, Jaiganesh Balasubramanian, Aniruddha Gokhale, and Douglas C. Schmidt
Department of EECS, Vanderbilt University, Nashville, TN, USA
{fwolf,jai,gokhale,schmidt}@dre.vanderbilt.edu

## 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 two contributions to R&D on component-based fault-tolerance. First, we present the structure and functionality of our COmponent Replication based on Failover Units (CORFU) middleware, which provides fail-stop behavior and fault correlation across groups of components in DRE systems. Second, we empirically evaluate CORFU 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 and (2) eases the burden of application development by providing middleware support for fault-tolerance at the component level.*

## 1 Introduction

Fault-tolerance is a key requirement for mission-critical distributed real-time and embedded (DRE) systems, such as air traffic management, total-shipboard computing, and fractionated spacecraft. Software for DRE systems increasingly uses component middleware to reduce application development time and effort. Component-based DRE systems incur a new set of fault-tolerance challenges compared with DRE systems based on distributed object computing that operate at the granularity of individual objects.

For example, DRE system functionality may be obtained by assembling a group of components procured from different providers. Supporting fault-tolerance for this functionality requires treating the group of components as a single failure and recovery unit. Other challenges include the need to maintain consistent state across components and component groups, as well as the tedious and error-prone process of configuring fault-tolerance properties into middleware and application components.

This paper presents the *COmponent Replication based on Failover Units* (CORFU) component middleware, which supports fault-tolerance of component-based DRE systems, specifically for groups of components. CORFU extends FLARe [1] middleware, which operates at the level of distributed objects to provide passive replication and fast client-side failover mechanisms. It is specifically targeting real-time systems with stringent resource constraints and short response time requirements. By capturing component dependencies in specific component groups (called failover units), CORFU reduces error reaction time at run-time and therefore ensures immediate and deterministic behavior as required by real-time applications.

CORFU implements algorithms that provide efficient fail-stop behavior of component groups. Instead of reactively providing failover capabilities for a sequence of consecutive failures of single components, CORFU can restore system consistency in a single execution step. Failover operations can thus be more deterministic through directly dealing with the original error and not allowing error propagation.

This paper evaluates several capabilities of CORFU qualitatively and quantitatively. It presents a 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. The paper also presents experiments that quantify the latencies involved in a failover operation of component based fault-tolerant applications and the timing characteristics of the fail-stop behavior of CORFU component groups.

The remainder of this paper is organized as follows: Section 2 uses a space system case study to motivate the need for component middleware and dependency-based component groupings; Section 3 summarizes the structure and functionality of CORFU; Section 4 analyzes experimental results to evaluate CORFU's performance; Section 5 compares CORFU with related work; Section 6 presents concluding remarks.

## 2 System Model and Case Study

CORFU enhances the OMG's Lightweight CORBA Component Model (LwCCM) [9] to support component-based fault-tolerance. This section briefly describes key characteristics of LwCCM and highlights the need for component-based fault-tolerance using a space system case study.

### 2.1 Overview of the Lightweight CORBA Component Model (LwCCM)

*Components* in LwCCM are implemented by *executors* 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.

LwCCM component deployment and configuration (D&C) is performed by the actors shown in Figure 1. The central entity 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. Each deployment plan will be represented by a *DomainApplicationManager* that is the administration interface to start and stop the application. The *ExecutionManager* splits a deployment plan into partial deployment plans that is processed by each associated *NodeManager*.

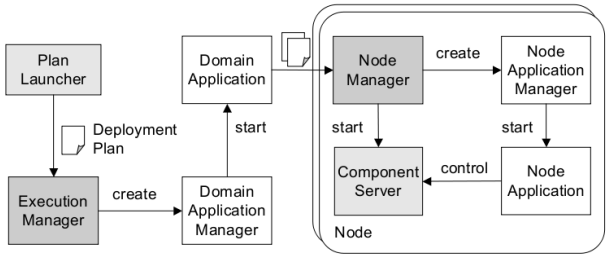


Figure 1: System Model for Component-based Deployment and Configuration

Each node deployment plan is represented by a *NodeApplicationManager* that starts and stops *NodeApplications* and *component servers*. A *component server* hosts containers and provides the run-time process context. A *NodeApplication* is a management entity that controls the life-cycle of components hosted in a component server.

### 2.2 DRE Component-based Case Study

The domain of space systems has stringent requirements for real-timeliness as well as for fault tolerance. To showcase the challenges confronting component-based DRE systems, we describe 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.

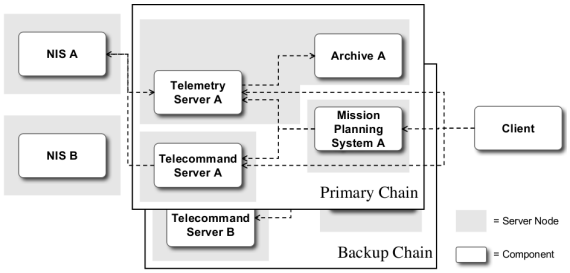


Figure 2: Component-Based Mission Control System

#### 2.2.1 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 2 shows the structure of a component-based MCS.

The time windows for active connections to satellites can be very short due to their orbit and visibility to ground stations, so 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 closely together.

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.

Together, 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 2. If an error occurs in the primary chain, the complete chain must be passivated and a backup chain must assume operation through a warm passive failover. All components of the backup chain are already deployed to assume operation 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.

#### 2.2.2 Fault-Tolerance Requirements of the MCS Case Study

The MCS chain forms a unit of failover and recovery. Providing replication and recovery semantics for component groups must support the following requirements: (1) fault isolation, (2) ensure fail-stop behavior of failed groups and (3) service recovery at the group level.

**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 failover of all components within the same chain.

**Requirement 2: Ensure fail-stop behavior.** If a failure of a component has been detected and its chain has been identified, this chain must be shutdown immediately so the system state is not corrupted.

**Requirement 3: Service recovery.** When components of the primary chain fail and are deactivated, all components in the backup chain must become active 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 end up having components failing over to replicas in different chains, which can cause performance problems or even malfunctions due the way components are deployed on a given infrastructure.

## 3 The Structure and Functionality of CORFU

This section describes the structure and functionality of CORFU, focusing on how it provides DRE systems with passive replication of a group of components treated as a logical *failover unit* [12]. A failover unit contains a set of components have dependencies with respect to failure propagation. Failover units allow for failure reaction times suitable for real-time systems by proactively failing over dependent components instead of reacting on slow failure propagations.

CORFU's layered architecture is shown in Figure 3. 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 three fundamental dimensions of fault-tolerance, including (1) *replica grouping,* which defines which replicas form one logical entity for group failover, recovery and synchronization of internal state, (2) *error detection,* which detects and reports failures to initiate failover operations for the group, and (3) *failover mechanism*, which redirects processing of client requests in case of a detected failure.

The rest of this section describes how CORFU's layered architecture provides these three dimensions of fault-tolerance. We describe our approach starting at the lowest layer working our way up the layers.

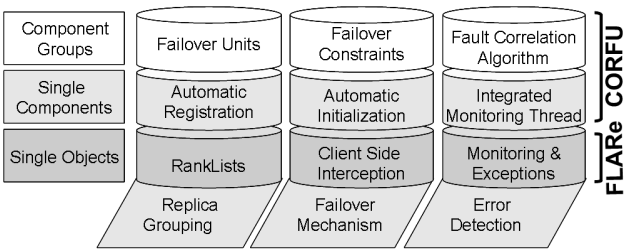


Figure 3: CORFU contributions

### 3.1 Fault-Tolerance for Individual Objects

CORFU's lowest layer of support for fault-tolerance at the level of individual objects is based on FLARe [1], which is a middleware framework that achieves real-time fault-tolerance through passive replication of distributed objects, as shown in Figure 4. We base our implementation on top of FLARe because of its capabilities to assure real-time performance even in the presence of failures through its adaptive and predictable failure recovery mechanisms. The three dimensions of fault-tolerance are obtained in FLARe as follows:

**Replica grouping.** FLARe's *middleware replication manager* (Label A in Figure 4) provides the replication needs of applications hosted in the system. FLARe's *state transfer agent* (label D in Figure 4) allows server objects within one group to synchronize their application states. Due to space restrictions we do not discuss state transfer in this paper but refer to a technical report [15].

**Error detection.** FLARe's *client request interceptor* (label C in Figure 4) catches failure exceptions, observes process and host liveliness via a *monitor* deployed on each processor and provides application transparent failover for clients.

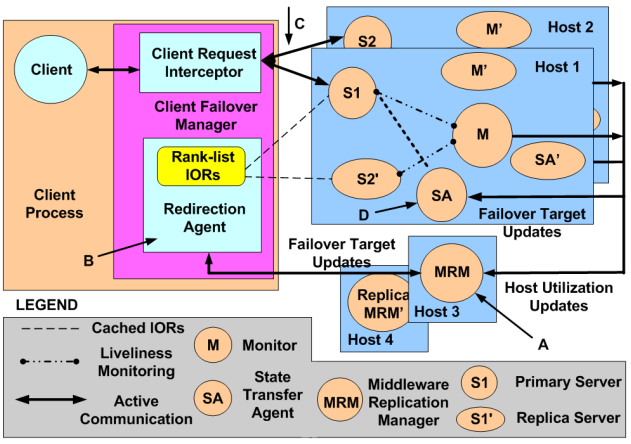


Figure 4: The FLARe Middleware Architecture

**Failover mechanism.** FLARe's *client failover manager* (label B in Figure 4) contains a *redirection agent* that is updated with failover and redirection targets by the *middleware replication manager* as it tracks group membership changes.

### 3.2 Fault-Tolerance for Individual Components

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 objects. The three dimensions of fault-tolerance at this layer are provided as follows:

**Replica grouping.** Since components can consist of several objects, replica objects must not only be grouped according to which object they represent, but also 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. Interfaces for state transfer between replicated components as described in [15] are now integrated into the component container and into required component interfaces.

**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 and thus ensures that each fault-tolerant component server is observed automatically.

**Failover mechanism.** CORFU automates server-side and client-side initialization of FLARe's mechanisms for failover, 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-side all IORInterceptors are automatically registered and allow for transparent enhancement of IORs to contain the replica object id necessary for failover operations.

### 3.3 Fault-Tolerance for 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 thus explained in depth below along the three dimensions of fault-tolerance.

#### 3.3.1 Replica Grouping for Component Groups

**Challenge.** The MCS chain in our case study in Section 2.2 requires a fault-tolerance solution to treat each chain as a logical failover unit. LwCCM does not provide first class support for treating components as part of a group. Addressing these limitations while remaining spec-compliant 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 run-time level, failover units must be realized within a management service. At the D&C level, CORFU realizes each failover unit as a separate 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 failover targets. Doing so enables CORFU to seamlessly use existing D&C actors (see Figure 1) to start and shutdown a failover unit when necessary.

The run-time aspects of failover units are realized by a management service called the *FaultCorrelationManager* (FCM), which manages failover units belonging to a system. To integrate the FCM into the existing D&C infrastructure, the Decorator pattern [5] is applied. The FaultCorrelationManager implements the ExecutionManager interface and can therefore be accessed by any service that uses the ExecutionManager interface.

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

The FCM 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 FCM enhances 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 this algorithm hinges on actions the FCM takes during the deployment phase and how it populates different data structures.

**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 ∈ 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 ∈ P do
    look up DomainApplicationManager m with key p in M;
    retrieve list of ApplicationManagers A through
    m.getApplications () ;
    for each NodeApplication a ∈ A do
        call m.destroyApplication ( a);
    end
end
```

CORFU's FCM uses 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 is done in two phases. In phase one, all affected failover units, represented as deployment plans, are determined based on the failure information. This phase uses the internal maps. In phase two, existing D&C actors (namely the DomainApplicationManagers) stop all component applications that belong to these deployment plans.

The run-time 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 failover to a replica failover unit. Since CORFU builds upon the object-level failover 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 → Failover constraints.** CORFU handles this challenge by modifying the ReplicationManager's RankList ordering algorithm such that it can process failover constraints. Figure 5 shows an example system infrastructure with three replicated components grouped into a failover unit with two backup units. The FCM transforms this information into failover constraints that define an order of objects per replica object id. An ordered sequence of host names defines the failover 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 ob-

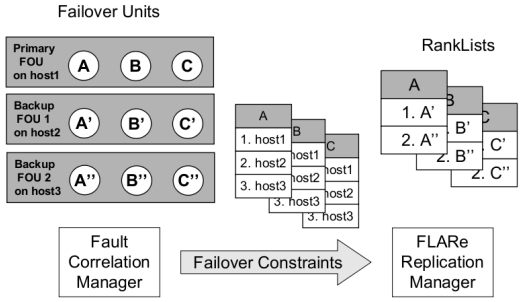


Figure 5: Interaction between FaultCorrelationManager and ReplicationManager through Failover Constraints

ject id uniquely identifies a replica.

The FCM provides another algorithm called 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.

**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
```

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.

# 4 Qualitative and Quantitative Analysis of CORFU

Section 3 describes how CORFU provides advanced fault-tolerance capabilities for DRE systems. This section evaluates CORFU using two different approaches. First, we conduct a conceptual analysis of the development effort by comparing object-based development of fault-tolerant applications with development using the CORFU infrastructure. Second, we present measurements of CORFU's timing behavior to show its suitability for real-time systems. This includes measurements of client-side failover latency and of the round-trip latency of failover unit fail-stop events.

## 4.1 Benefits of Component-based Fault-Tolerance compared to Object Level Fault-Tolerance

Developing applications that support distributed object-oriented fault-tolerance as provided by FLARe involves additional effort with respect to application development. This evaluation qualifies those efforts and contrasts them with the component-based fault-tolerance approach CORFU provides.

**Development obligations of object-oriented fault-tolerance.** FLARe requires different means to implement fault-tolerance on the server-side, where the object to be replicated resides, and on the client-side, which contains the failover mechanisms.

Figure 6 gives an overview of all obligations related to server-side development. These obligations can be

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

Figure 6: Responsibilities for Server-Side Fault-Tolerance

grouped into (1) object implementation obligations that each CORBA servant needs to 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 fault-tolerant aspects of the application.

Some initialization steps, such as HostMonitor thread instantiation and registration, must be performed only once per process. Other steps, such as the object implementation obligations, application configuration and registration of objects with the ReplicationManager, must be done for each object in the process. The client-side initialization is not as complex, but still involves some 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 meta-data.

**Benefits of CORFU's component-based approach.** By integrating FLARe functionality into a fault-tolerant component server, CORFU overcomes many of these limitations of traditional object-oriented fault-tolerance approaches. Server- and client-side capabilities are available within the same component server. Since CORBA objects often play both roles of server and client at the same time this is a suitable architectural decision. We present the benefits of the component server approach by relating them to the three different types of obligations as presented earlier.

a. **Object Implementation.** CCM provides code generation functionality in the form of the IDL and CIDL compilers that automatically can create necessary code artifacts.

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

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

**Summary of analysis.** CORFU increases the transparency of using fault-tolerance mechanisms for both client and server development. This transparency allows application developers to focus on business logic implementation 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

This section presents experiments that evaluate the timing behavior of CORFU. These experiments allow a better understanding of latencies involved in the failover mechanisms and clarifies for which timing requirements CORFU is sufficient. The first experiment evaluates failover latency as experienced by a client application. The second one focuses on timing latency of the coordinated shutdown of a failover unit.

#### 4.2.1 Testbed

All experiments have been conducted on ISISLab[1], a LAN virtualization environment with identical blades connected through 4 Gbps switches that allow for dedicated links per experiment. The blades each have 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 the CORBA component model. CORFU and all testing applications have been built using the GNU compiler collection gcc version 3.4.6.

[1] http://www.isislab.vanderbilt.edu

#### 4.2.2 Failover Latency

**Experiment setup.** This experiment compares the failover latency a client experiences for CORBA 2.x applications and component-based applications. A client application periodically calls 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[8]. After 10 calls a fault is injected that causes the server to shut down. This 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 as well as all CORFU infrastructure entities to not interfere with the timing measurements. The experiment is implemented in two variants. Variant 1 is object-oriented and consists of a client and a server executable that directly use FLARe functionality. Variant two 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. Each measurement configuration is repeated 100 times to gain representative results.

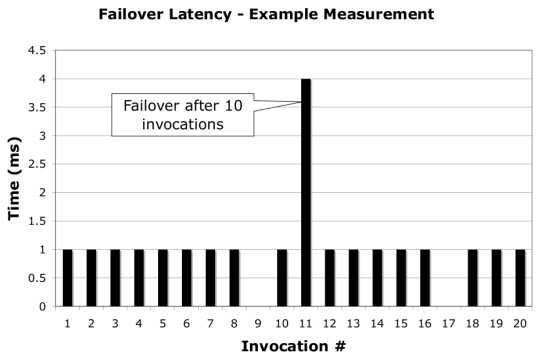


Figure 7: Single Failover Latency Measurement

**Measurement results.** An example for a single measurement for failover latency is given in figure 7, which represents the component-based case with one application set running. The ten invocations before and after a failure event are recorded. The first 10 invocations show a communication overhead between zero and one millisecond, which represents failure free communication with the primary server.

The client experiences an increased response time on the eleventh request, since the primary server is no longer responding. This results in a client side failover that involves the interception of a CORBA exception and the forward-

ing to a backup replica. As the diagram shows, this incurs latency increase from one millisecond to four milliseconds for the client.

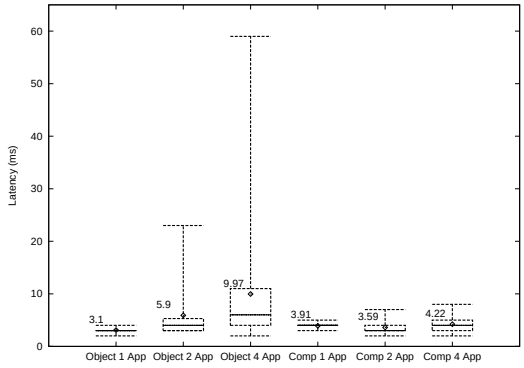


Figure 8: Results for Failover Latency Measurements

Figure 8 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 the component-based fault-tolerance with 25 percent additional overhead is relatively small.

Looking at the configurations with two and four applications, we can see that the component-based experiments have a much lower jitter and have a similar average of four milliseconds, while 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 failover-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 component based fault-tolerance.

#### 4.2.3 Failover Unit Shutdown Latency

**Experiment setup.** The second experiment is designed to give insight into the latency involved in the process of shutting down a failover unit.

The structure of the experiment and its logical sequence of events is shown in figure 9. The setup includes six processing nodes of which one node is dedicated for the CORFU management entities, such as the ReplicationManager, the FCM, the ExecutionManager and other elements of the D&C run-time. The other five nodes have a HostMonitor deployed to observer 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

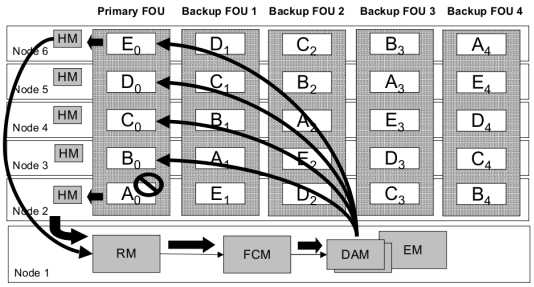


Figure 9: Setup for FOU Shutdown Latency Measurement

failover unit is replicated four times through the backup failover units one through four. Each of the backup units contains replica components $A_n$ to $E_n$ of each component in the primary unit. The failover order of the units corresponds to their number.

The experiment will inject failures in the currently active component, leading to a failover sequence of primary FOU, backup FOU 1, backup FOU 2, backup FOU 3 and finally backup FOU 4. Each experiment run therefore allows us to measure four failover latencies. Due to the need for consistent time, all measurements are taken on node-1 in the ReplicationManager and the FaultCorrelationManager, which alleviates the need for synchronized clocks. The measurements are done in the following sequence:

a. A failure is provoked in component $A_n$ of the active FOU.
b. The failure is detected by the HostMonitor and reported to the ReplicationManager.
c. The ReplicationManager takes a time-stamp at time $t_1$ when it receives the failure notification and notifies the FCM about the occurred failure. The FCM takes a times tamp at time $t_2$ when it is notified about a failure.
d. The FCM performs the FAILURE-REACTION algorithm and takes a time-stamp $t_3$ after the affected failover units have been identified in phase 1.
e. The FCM will then access the DomainApplicationManager to retrieve all node applications for the corresponding deployment plans and then will iterate through them to shut them down. After the last call is returning, a time-stamp at $t_4$ is taken to indicate the finishing of the shutdown request.
f. The ReplicationManager will be notified about all the shutdowns of the affected components by the HostMonitors. On reception of the last shutdown notification, a time-stamp for $t_5$ is taken that represents the time when the FOU is completely shut down and a client would failover to a backup replica no matter which component in the FOU it tries to access.

**Measurement results.** As summarized in table 1, we can determine three essential durations from our experi-

| Time | Formula | MIN | AVG | MAX |
|---|---|---|---|---|
| $t_{\text{round-trip}}$ | $t_5 - t_1$ | 25.87 | 70.59 | 260.06 |
| $t_{\text{reaction}}$ | $t_3 - t_2$ | 42.37 | 56.04 | 73.84 |
| $t_{\text{shutdown}}$ | $t_4 - t_3$ | 0.11 | 0.24 | 0.86 |

Table 1: Measurement results for fail-stop latencies (ms)

ment. First, the round-trip time is the sum of all latencies involved in the shutdown of a failover unit, which includes failure detection, reaction time within the FCM, and shutdown time by the D&C run-time. Second, the reaction time is the time spent within the FCM between the failure notification and the start of the shutdown process, which is the time needed to perform the FAILURE-REACTION algorithm 1 and to serialize incoming notifications into a thread-safe queue to ensure correct processing of parallely detected errors. Third, the shutdown time as measured by the FCM allows us to understand which proportion of $t_{\text{round-trip}}$ is not related to the D&C shutdown mechanism, which cannot be changed without breaking the LwCCM standard.

#### 4.2.4 Summary of the Analysis

Based on the experiments described above, several characteristics of CORFU are exposed. Using a client-side failover mechanism allows for short failover 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 systems.

As shown by the first experiment, the client-side failover latency is relatively small, being three milliseconds for the object 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 failover in CORFU is comparable in performance and occurs only minimal overhead, having an average response time of four milliseconds.

Compared to the client failover latency the failover unit shutdown latency with 70 milliseconds in 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 FCM. The internal reaction time of the FCM to determine deployments that are affected by faults is already optimized through the use of hash maps with close to constant access times. With an average beneath 0.25 milliseconds it does not substantially contribute to the overall processing time.

Our results show the need for further optimizations. Possible approaches to do so are (1) parallelized shutdown of all node application parts using asynchronous invocation, (2) collocation of the ReplicationManager, the FaultCorrelationManager and the ExecutionManager to reduce communication time and (3) application of RTCORBA to improve determinism in network communication. Although there still is potential for performance improvement, the measurements show that CORFU is suitable for DRE systems, such as MCS, and offers comparable performance to distributed object computing fault-tolerance.

## 5 Related Work

This section compares our work on CORFU with related work in the areas of fault-tolerance dependency analysis, frameworks for fault-tolerance, and modeling techniques for dependability aspects.

*Fault-Tolerance dependency analysis.* 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. The component based dependency model [14], Cadenas dependency model [7] and event correlation based on dependency graphs [6] use different type of system models to achieve a high-level understanding of component dependencies. 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. Active dependency discovery [2] and active failure-path inference [3] are observation based approaches. This work on dependency analysis relates to CORFU since it provides methodologies to define groups of depended components. CORFU provides a mechanism to use the dependency information gathered by these techniques for efficient failover operations.

*Frameworks for fault-tolerance.* A framework for fault-tolerance integrates different aspects of dependability. AQuA [11], an adaptive architecture for dependable distributed objects focuses on providing redundancy for distributed objects. It uses CORBA to define and implement objects, but maps them to an underlying group communication mechanism. AQuA supports fault-tolerance on the granularity of objects, while CORFU provides component-based fault-tolerance. JAGR [4] builds on a component-based infrastructure for the domain of three tier web applications with permanent data storage. JAGR uses automatic failure-path inference and escalating micro-reboots. CORFU's approach is more suited for DRE systems since it focuses on passive replication rather than simple reboots of system parts.

*Modeling techniques for dependability aspects.* R&D in modeling fault-tolerance aspects on a higher level of ab-

straction in form of modeling languages resulted in several solutions. Cadena [7] focuses on the modeling of component behavior early in the design process based on property specifications that capture external and internal component dependencies. Cadena provides a domain specific modeling tool suite for system modeling and a simulation environment for model verification. CORFU provides fault-tolerance at a higher level of abstraction by grouping components into failover units.

MDDPro[12] and GRAFT [13] are two modeling environments that support failover units. MDDPro focuses on algorithms that automatically place components and their replicas while minimizing the chances of their simultaneous failure (necessary deployment meta-data is also auto-generated). CORFU's approach is complementary to MDDPro by providing a run-time infrastructure that can process and instantiate a system modeled by MDDPro.

Like CORFU, the GRAFT project identifies the lack of first class support for fault-tolerance in component middleware. GRAFT relies on an aspect-oriented approach to weave in fault-tolerance, which may however become a limiting factor when fault management and recovery results in a need for interactions with complex semantics (*e.g.*, timing, consistency). In these circumstances, a first class support within the middleware is preferable. Moreover GRAFT uses exceptions to detect critical errors, while CORFU provides a monitoring framework that allows for advanced error detection.

## 6 Concluding Remarks

Prior research on fault-tolerant DRE systems has not accounted for application development effort, application lifecycles, and system evolution. Moreover, many middleware-based solutions provide relatively low-level abstractions, *e.g.*, on the level of objects. Our work on CORFU presented in this paper shows that component middleware can provide advanced error reaction behavior for real-time systems, while also improving transparency of fault-tolerance aspects in the application development process. CORFU is thereby enhancing system flexibility, evolvability, and quality. Our measurements of CORFU performance showed that component-based fault-tolerance can be provided without undue overhead.

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

## References


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

[2] A. Brown, G. Car, and 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*, pages 377–390, 2001.

[3] G. Candea, M. Delgado, M. Chen, and 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*, page 132, Washington, DC, USA, 2003. IEEE Computer Society.

[4] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: an autonomous self-recovering application server. *Autonomic Computing Workshop, 2003*, pages 168–177, June 2003.

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

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

[7] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. *Software Engineering, International Conference on*, 0:160, 2003.

[8] J. Hill, J. Slaby, S. Baker, and 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, August 2006.

[9] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

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

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

[12] S. Tambe, J. Balasubramanian, A. Gokhale, and T. Damiano. MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems. In *Proceedings of the International Service Availability Symposium (ISAS)*, volume 4526, pages 127–144, Durham, New Hampshire, USA, 2007. Springer.

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

[14] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 241–244, 2002.

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