University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Fault Tolerace Support for SOFA

PhD Report

Jan Rovner

Technical Report No. DCSE/TR-2001-03
November, 2001

Distribution: public

Technical Report No. DCSE/TR-2001-03
November 2001

Fault Tolerace Support for SOFA

Jan Rovner

Abstract

Several approaches for software fault tolerance in component oriented systems have been proposed, with varying level of reliability. This paper gives a proposal to enhance SOFA framework with replication based fault tolerance. With added fault tolerance support, SOFA framework allows to create reliable, non-stop running applications without needs to modify source code of components.

Keywords: SOFA, components, fault-tolerance, replication

This work is partly supported by the Grant Agency of the Czech Republic, project No. 201/99/0244 "Developing software components for distributed environment".

Contents

1 Introduction

The main goal of the article is to research and provide sketch to enhance SOFA [1] component framework with fault tolerance support that it currently lacks. The designed extensions of SOFA allow to the developers of SOFA components to create applications that require a high level of reliability. The fault tolerance support is provided transparently by SOFA runtime, i.e without needs to modify the source code of SOFA components.

Additional features in SOFA framework appear together with the added fault tolerance support - automated component's state saving/restoring, the possibility of component activity tracing, and the possibility of runtime component determinism checking.

2 SOFA framework

2.1 General overview of SOFA

The SOFA framework is based on the concept of component oriented programming. SOFA application is composed of many items called software components cooperating together to achieve the application's functionality. Each software component provides to the other ones sets of services, and to make theses services available, usually also requires the services of other components. SOFA's components can be nested. In these aspects, SOFA is similar to the well-known middleware systems like OMG's CORBA, Sun's Enterprise JavaBeans and Microsoft's COM.

The innovative SOFA DCUP (Dynamic Component Updating) [1] architecture gives the component providers the possibility of updating their components at runtime without manual intervention on the enduser side.

2.2 SOFA component model

In analogy with the classical concepts of an object as an instance of a class, SOFA introduce a software component as an instance of a component template.

Basically, a component template is a framework which contains definitions of implementation objects and nested components. The component interfaces are named sets of method signatures, possibly with an attached specification of semantics (behavior).

SOFA template defines two views of a software component. The black box view called component frame defines the set of component's provided and required interfaces plus its behavior. The gray-box view is called component architecture and shows the organization of the first level of components' nested subcomponents.

Every component template is determined by its interface (set of services either provided and/or required), and by the definitions and bindings of implementation objects and nested components.

For the specification of a component's interfaces and its architecture (in terms of nested components and their interconnections), a language called SOFA Component Description Language (CDL) [2] is used.

3 Fault tolerance in component oriented systems

In component oriented fault tolerant systems, fault tolerance is dependent on component redundancy, fault detection and recovery. The redundancy is achieved by replication of components onto different machines (processors), the replicated components are called replicas.

Replication-based fault tolerant systems replicate each component of the application. If an component or its hosting processor fails, another instance of the component is available on another processor that is ready to provide services. There are basically two basic kinds of component replication - active and passive [9, 4, 10].

3.1 Active replication

With active replication, all of the component replicas execute the methods invoked on the replicated component simultaneously.

Fault tolerance infrastructure must ensure that each replica receives the same sequence of method invocations in the same order and executes those invocations in the same order (totally ordered invocations) and exactly once. Thus, it maintains the consistency of the state of the replicas.

To enable active replication, replication mechanism multicast the invocation (response) from every client (server) replica to the server (client) component group. This invocation is usually performed via reliable totally ordered multicast protocol to ensure that the states of both the client and the server replicas are consistent at the end of the operation.

3.2 Passive replication

With passive replication, only one of the replicas, called the primary replica, executes the invoked methods. All other replicas do not perform any operation.

Fault tolerance infrastructure must ensure that the method invocations and responses are not delivered to the backup replicas but, instead of execution, they are logged at the backup replicas. The system periodically captures the state of the primary replica and logs that state as a checkpoint at the backup replicas.

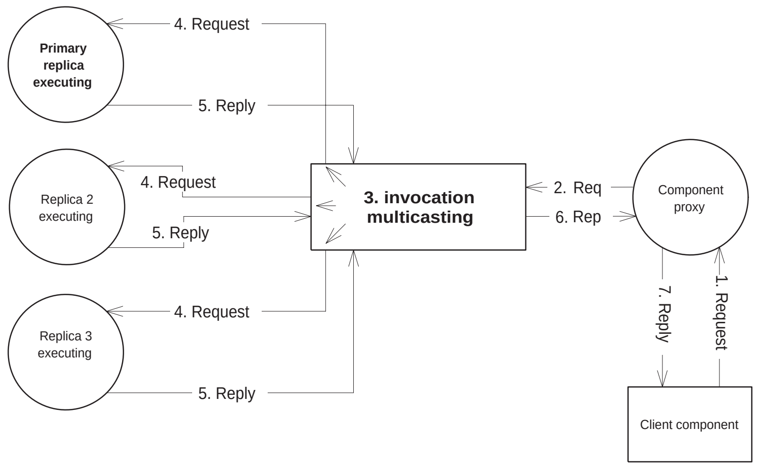

Figure 1: Active replication

If the primary replica fails, fault tolerance infrastructure chooses a backup replica to replace the failed primary, retrieves the most recent checkpoint from the log and loads it into the backup replica. The backup replica then processes subsequent invocations from the log to repeat the work that the replica did before it failed.

The requirements (total ordering) for the multicast mechanism are the same as for active replication.

There exist different styles of passive replication that differ in the degree to which the states of the backup replicas lag behind the state of the primary replica.

3.2.1 Cold passive replication

In the case of a cold passively replicated components, the backup replicas are not even loaded into memory, and thus do not come into existence until the primary replica fails.

In the event that the primary replica fails, one of the cold backup replicas is loaded into memory, and assumes the role of the new primary replica. The state of the new primary replica is recovered using the checkpoint information and invoking methods that were logged after the previous checkpoint.

3.2.2 Warm passive replication

Unlike cold passive replication, the warm passive backup replicas are loaded into memory and are running.

As long as the primary replica is running, the only messages that the replication mechanisms deliver to the backup replicas are the messages that contain the state of the primary replica. If the primary replica fails, a new primary replica is chosen from the backup replicas. The lag in the state of the new primary replica (formerly a backup

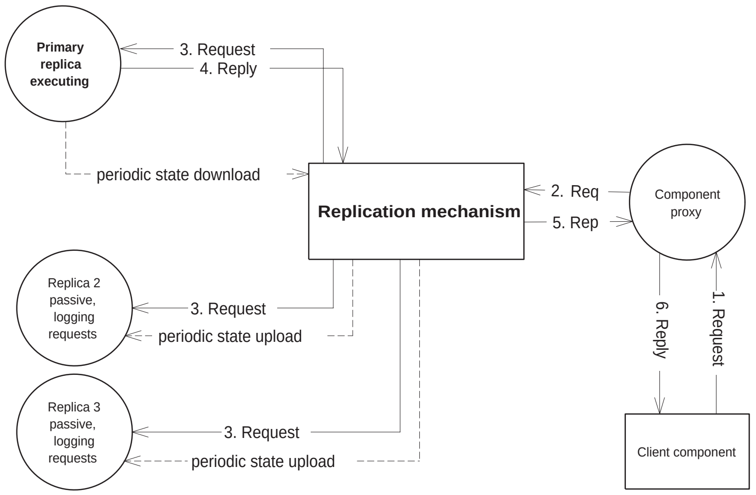

Figure 2: Passive replication

replica) and the old primary replica depends on the frequency of state transfer.

3.2.3 Hot passive replication

Hot passive replication is a variant of warm passive replication, with the state transfer occurring at the end of every operation on the primary replica. Thus, while the states of the backup and primary replicas may differ while the primary replica performs an operation, their states are consistent at the end of each state transfer.

3.3 Current state of the art

Most of currently specified/designed component oriented fault tolerant systems lack simplicity and transparentness for software developers. Almost all of them do not still provide fault tolerance support transparently in the design time stage of software development process. Software designers and developers must be familiar with the fault tolerant support provided by given systems and must explicitly provide some ”plugging points” to allow the fault tolerant subsystems to communicate with the components during component replication or fault recovery. Implementing such ”plugging points” always requires additional knowledge and component’s source code modifications[9, 4, 10].

These ”plugging points” can be provided very differently, but most common techniques require the developer to either implement some vendor specific interface(s) or to inherit the class definition (in case of object oriented systems, of course) from some well-known superclass that itself partially implements fault tolerance.

Supporting such ”plugging points” at all events leads to the needs of system redesign and reimplementation. This makes complicated and time consuming to add fault tolerant support to already designed and implemented systems with the possibility to infect modified system with new potential bugs.

3.3.1 Describing component's state

The very primary task of replication mechanisms is the ability to obtain the component's state, which (in case of today's common OO development) reside in member variables of implementation objects. However, not all member variables contain the information that is critical for correct operation of a component, e.g. private variables used for temporary results, etc.

In most cases, the code added by developer for fault tolerance support must manually implement some method of transformation of actual component state to a form that can be understood by fault tolerant system - and this is usually a code for bidirectional transfer of the object's state from/to member variables.

In addition to problem of providing the state can arise the problem of determining whether actual component's state is "stable", because the request for state providing can appear asynchronously during the component execution.

4 Fault tolerance for SOFA

As a solution of the issues discussed above, aim of this work is to enhance SOFA framework to support fault tolerance mechanisms. The source code of SOFA components will not have to be modified to take advantages of fault tolerant support. This will enable possibility to run "truly 24x7" applications using existing DCUP infrastructure for component updating and possibility to stop SOFANode[1] for cases of hardware upgrades and repairs.

4.1 Basic concepts

SOFA framework will provide fault tolerant support and recovery using entity replication pattern. SOFA runtime will manage replicating running component instances, and distributing these replicas across different SOFANodes over the SOFANet[1].

4.1.1 Describing component via CDL

SOFA components, their bindings and other properties are described using CDL. The CDL is a ideal facility to declaratively describe component's state as well. The purpose of the CDL compiler in case of DCUP and replication operations is to automatically generate code that reads and writes complete component state from/to its member variables.

This problem could be solved introducing new section to CDL, starting with keyword `state`. `State` keyword should declare state variables and should appear at interface level, following method declarations.

The task of CDL compiler is then to produce appropriate source code for:

- member variable declarations in class implementing an interface
- state transfer from/to these member variables

However, several constraints must be put to the data types of member variables, depending on the target programming language which is produced by CDL compiler. The only restricting condition is the ability of the particular language to handle assignments with these data types in means of actual content copying, not only copying variable's reference. In case of most common languages - like C, Java, Pascal etc. assignment operator would restrict component's state only to the primitive data types.

As a solution, CDL compiler should have knowledge of assignment semantics for different data types, i.e. how to generate code for proper data assignments. In case of most object oriented environments, some special support must be provided by CDL compiler for non primitive data types of a particular language. For example - in case of object references in Java, Java's native serialization mechanism can be used to achieve desired state transfer.

4.1.2 DCUP reuse for replication

To save developers from state transfer issues and to create a really transparent framework SOFA's DCUP principles will be reused. The basic idea is that the state of a SOFA component is also transferred during DCUP operation and the mechanisms introduced in DCUP will be used to implement component replication [9].

Unfortunately, DCUP itself does not solve the problem of state definition, which is needed to achieve transparent state transfers. In fact, up to current version of SOFA, the artifacts of component state were undefined and the state transferred was left on developer.

5 Component replication framework in SOFA

5.1 SOFA architecture modifications

One of the most important goals in developing every fault tolerant system is to eliminate single point of system failures. To make SOFA fault tolerant, SOFA subsystems residing only on a single SOFANode must be made fault tolerant as well - the solution is to make these services distributed over the SOFANet.

5.1.1 Template repository

The task of current Template repository is to provide binary images of components. The task of new, distributed version, is to replicate Template repository to SOFANodes participating on fault tolerant framework and ensure that local copies of Template repository are replicated, consistent and properly synchronized.

5.1.2 Interface wrappers

Interface wrappers are generated modules acting as a component's interface proxies, bringing to SOFA a layer of indirection that is required for DCUP architecture. The most radical changes must be made here, because interface wrappers are the key points for the component replication technology. The very primary idea of whole concept - component replication - will be in fact executed here.

The task of modified interface wrappers is to enable multi-point component connection (1:n instead of 1:1).

In case of replicated server objects, incoming calls to interface wrappers will be multicasted via multiple method calls to all server replicas and vice versa - incoming results from method calls will be delivered to replicated client objects.

Another very important change to interface wrappers is to change the moment of target object's method invocation, depending on chosen replication method. In case of active replication, the method is executed immediately as usual. However, in case of passive replication, the method call is not invoked at all. Instead of invocation, the incoming call is stored in a log.

5.2 Application components

5.2.1 Component determinism issues

In the interests of strong replica consistency, it is necessary to ensure deterministic behavior of components.

An example of non-deterministic component's behavior is using date/time API functions in component's code. When these API functions are called by a replicated object, different replicas may obtain different results, depending on a value of real-time clock on a machine where a replicated component runs.

Unfortunately, the replicated component may use this information to update its internal state, and also to invoke other replicated objects in the system. The condition for determinism must be fulfilled in all possible replication strategies (active and passive replication). There is no simple way how to generally detect nondeterminism in component's code and how to force developers to create deterministic code. But at least some partial checking can be done by the SOFA runtime services.

The first choice is comparing component's internal state after finishing operation against state on all other replicas. The second possibility is logging all outgoing invocations (with method parameters and return values) and after the operation is finished, comparing created log against logs on all other replicas. Both these methods should be implemented as efficient as possible - for example using hash values, etc.

One can expect that introduced determinism checking mechanism will be most probably very resource consuming and should be turned on only in case of application testing and debugging.

5.2.2 State transfer - backup

It is not always easy to decide, where the component's state is stable; i.e. whether the component after restore captured state will perform its functions properly.

This naturally determines the only moment, where we can implicitly consider that component's state is stable. It's the time, when the component is passive - and this is generally only after the return from an executed method, i.e. when no thread is executing component's code.

To satisfy the condition of determinism, the component must not internally execute any threads that could nondeterministically manipulate its state. Secondly, only one client (thread) must use the component at a time (parallel method's invocations are prohibited). This condition may be satisfied either by simple - instance per thread (client) threading model or by some kind of serialization mechanism.

5.2.3 State transfer - restore

The component's recovery is divided into two key parts - first one is the re-instantiation of the component followed by loading the component's state. The second phase is the restoring of failed component's bindings - references to other components. The problem is, that the state of referenced (or owned) components must be considered as the part of failed component's state. So to properly recover failed components, the state of referenced components must be set to satisfy "parent's" component expectations.

This operation will cause flood effect on the all levels of component's reference tree. Fortunately, at least some kind of optimization for this case may be done. We can check, whether the component's state have been modified (the case the component was used by its "owner" before the "owner" died and its state has changed) and restore the component only in positive case.

5.3 Method invocations

In the current, not fault-tolerant version of SOFA, method invocations are delivered via standard Java RMI mechanism. Because the implementation of reliable and totally ordered multicast protocol is a difficult task and is not subject of the research, single

(1:1) RMI calls can be replaced by multithreaded unicasting or by using third-party software, e.g. by FilterFresh for Java [4]

5.4 Fault detection

Fault detection will be based on the concept of Java exceptions. Therefore, any "unwanted" exception is considered to be a signal of some failure. Thus, any fault will be detected by standard Java exception handling mechanism using try-catch block.

This approach, in general, allows detection of all kinds faults. Most Java API classes raise exceptions when some kind of fatal error occur. Among these fatal errors belongs all network, memory or CPU related faults.

Other faults that do not cause exception rising - such application faults as when different function results occur on different replicas or when a replica state differ from the state of other replicas - will be detected and the proper exception will be created and raised and then passed to the standard mechanism of the fault detection.

6 Summary

The article describes a proposal to enhance SOFA component framework with fault tolerance support. The designed extensions of SOFA should allow the developers to create applications that require a high level of reliability with transparently provided fault tolerance support, without needs to modify the source code of SOFA components. All required information for the fault tolerant subsystem will be provided purely declaratively, via CDL.

Together with the fault tolerant support, several new interesting features coming from fault tolerance technology reuse appear.

Automated State Transfer for DCUP - the code required for DCUP operation can be automatically generated from the state information declared in CDL. This allows real usage of DCUP, which is not implemented in SOFA yet.

Component Activity Tracing - the log files gained in the during the passive replication can be used for component activity tracing and analysis.

Component Determinism Checking - in case of active replication, not only the methods' return values will be checked, but components internal state as well. When the internal state of two replicas differ, it is the sign of nondeterminism in a component.

Acknowledgement
This work was supported by the Grant Agency of the Czech Republic (GACR),

project No. 201/99/0244 ”Developing software components for distributed environment”.

References

[1] F. Plasil, D.Balek, R.Janecek, *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Charles University, Faculty of Mathematics and Physics Department of Software Engineering, Prague.

[2] V.Mencl, *Component Definition Language*, Dep. of SW Engineering, Charles University, Prague.

[3] C. Szyperski, *Component-Oriented Programming A Refined Variation on Object-Oriented Programming*, The Oberon Tribune, Vol 1, No 2, December 1995.

[4] A.Baratloo, P. E. Chung, Y.Huang, S. Rangarajan, S. Yajnik, *FilterFresh: Hot Replication of Java RMI Server Objects*, Lucent Technologies, Bell Laboratories.

[5] P.Narasimhan, *Transparent Fault Tolerance for CORBA*, University of California, Santa Barbara.

[6] J.Rovner, P.Brada, *Methods of SOFA Component Behavior Description*, DCSE, University of West Bohemia, Pilsen.

[7] J.Rovner, J.Valdman, *SOFA Review: Experiences from Implementation*, DCSE, University of West Bohemia, Pilsen.

[8] J.Valdman, *Documentation Of SOFA Implementation*, DCSE, University of West Bohemia, Pilsen.

[9] *Fault Tolerant CORBA Specification, V1.0*, OMG.

[10] J.C. Fabre, T. Perennou, *A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach*, LAAS-CNRS, Toulouse.