ACTIVE REPLICATION IN AN EVENT RULE FRAMEWORK FOR DISTRIBUTED SYSTEMS

Hillary Caituiro-Monge (hcaituiro@ieee.org), Javier Arroyo-Figueroa (jarroyo@ece.uprm.edu)
Department of Electrical and Computer Engineering
University of Puerto Rico, Mayagüez Campus

ABSTRACT

In Distributed Systems (DSs) with Asynchronous, Non-Deterministic Reactive Components (ANDRCs), the timing assumptions are not valid and the output could be different, even if the same sequences of stimuli are input with the same initial state. In such systems, when the availability and reliability are critical, so fault tolerance can be achieved by replicating the ANDRCs. This research uses active replication with a middle-tier component that multicasts incoming-events to all replicas, and detects and suppresses duplicated outgoing-events that were posted by each replica. The Event-Rule Framework (ERF), which is a framework for developing DSs with ANDRCs, was used as a test bed for this research. The performance analysis shows linear execution-time curves; therefore, proposed solutions were proven to be feasible, and their performance results were proven to be acceptable.

1. INTRODUCTION

A Distributed System (DS) is a collection of software components distributed among processors of heterogeneous platforms, with the purpose of sharing resources and workload, and maximizing availability. The main goals that designers pursue in the design of DSs are transparency, scalability, reliability and performance. The most challenging ones are scalability and reliability. Fault tolerance (FT) is one means of achieving reliability and it is the ability of a system to continue operating as expected, despite internal or external failures. The FT of a DS can be improved through the replication of its components.

There are three different types of components in a DS: passive, proactive and reactive. Passive components limit their functionality to replying to invocations made by other distributed components (DC). Proactive components continually monitor the system state and initiate interactions with other DCs. Reactive components (RC), initiate such interactions as a reaction to an external stimulus. A RC is asynchronous if any other component whose stimuli cause a reaction in the RC does not have to wait for such reaction to continue its execution. An RC is non-deterministic if, given one type of stimulus, there is more than one possible type of reaction. An Asynchronous, Non-deterministic Reactive Component (ANDRC) is an RC that is both asynchronous and non-deterministic.

An example of a DS framework having ANDRCs is ERF (Event/Rule Framework). ERF is an Event-Rule Framework for developing distributed systems [1]. Among the leading features of ERF are: (i) a high-level definition language to specify system behavior by using rules that handle events; (ii) an object-oriented model in which system components, events and rules are treated as objects; (iii) the use of a Rule-Based Intelligent Event Service (RUBIES) for processing events; and (iv) a visual development environment.

There is a challenge to achieve fault tolerance in ANDRCs. Since components are non-deterministic, the output could be different even if the same sequence of stimuli is input with the same initial state. Also, since components are asynchronous, timing assumptions are not valid. Existing fault-tolerance techniques have been implemented using components such as failure detectors, state transfer protocols, and duplicates detection/suppression mechanisms, which rely in timing assumptions, determinism, and synchronism respectively.

This research is about the use of Active Replication (AR) techniques for achieving fault tolerance in ERF. In AR all replicated components accept third-party incoming events, and the consistency problem is solved by using a middle-tier component that is in charge of the event multicasting and detecting and suppressing duplicated events.

2. SURVEY OF RELATED WORKS

The OMG (Object Management Group), which standardized CORBA, has recently published specifications for Fault-tolerant CORBA. The framework is based replications of objects, fault detection, and recovery. This standard has been adopted in our work.

Baldoni et al. [2] address fault tolerance for asynchronous deterministic distributed systems with a three-tier architecture for software replication. Baldoni et al. work has many similarities to our approach. However, it

does not address non-determinism. It only works with method invocations. And, it relies on sequencers that limit the system to determinism.

Collet et al. [3] present a flexible infrastructure to create component-based distributed applications using rules based on event and rule services. The fault tolerance is supported by means of the adaptable replication framework RS2.7. This approach has the disadvantage that protocols consume time and bandwidth in the communication process. Our approach reduces and eliminates such overhead by avoiding the use of such protocols.

Yeast (Yet another Event-Action Specification Tool) [4] is a general-purpose event-action system for building distributed applications using high level specifications. It incorporates two special-purpose fault tolerance components. Watchd is a watchdog daemon for detecting failures and recovering from crashes in a primary replica fashion. Libft is a library with operations to checkpointing internal state of processes. YEAST is not useful for tight time constrain systems due to its high time consumed in the recovery process.

FATOMAS is a fault-tolerant mobile agent system. Pleisch's thesis [5] proposes a solution in the case of non-deterministic client replicas, which is based on the idea of sharing and recording enough "undo" information to enable replicas to cancel multiple executions caused by the failure of a replica. It does not work on ANDRCs, since operations could not be canceled.

3. ERF OVERVIEW

ERF [1] is a framework for developing DSs, which provides a set of abstractions for specifying and modeling the behavior of DSs in terms of events and rules. ERF is designed following an object oriented model in which rules and events are treated as objects. In ERF, the specification of DSs is made at a high level, with an environment that provides high level abstractions in which rules are used for specifying behavior in terms of events, conditions, actions and alternative actions (ECAA). The main component of ERF is RUBIES, a Ruled Based Intelligent Event Service that provides services for rule specification, service registering, and event handling through rules.

3.2. Model

The ERF overall model consist of: (i) an event model, which defines the structure of events; (ii) a rule model, which defines the structure of rules; and (iii) a behavioral model, which defines how the rules are triggered and evaluated upon the occurrence of events. ERF provides the event abstraction to represent significant occurrences in a distributed system. In ERF, the behavior of a DS is defined in terms of how the system reacts to event occurrences. Such behavior is defined in terms of rules. A rule is an algorithm that is triggered and executed upon the occurrence of events satisfying an event pattern.

3.2. Components

The Event Channel is a middleware DC that allows sending and receiving events from producers and consumers. It supports events treated as objects, represented by event classes. Such classes must inherit from the Event class of ERF. RUBIES is an engine that handles events through the evaluation of rules. RUBIES is an ANDRC registered to the event channel both as a consumer and as a producer. Events consumed by RUBIES from the event channel cause the triggering and execution of rules.

4. FT-ERF ARCHITECTURE

One approach for achieving scalability and fault tolerance in ERF is distribution and replication. Scalability can be achieved by distributing rules over several replicated RUBIES instances. Fault tolerance can be achieved by means of RUBIES replication.

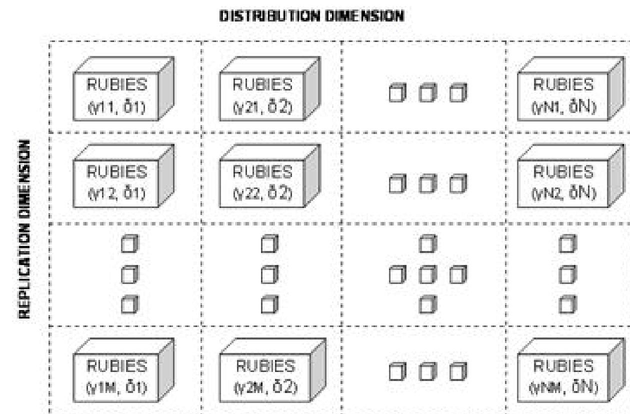

Figure 4.1 Scalable and fault-tolerant ERF Architecture.

In the distribution (horizontal) dimension, N RUBIES instances are distributed, forming a distribution group, while in the replication (vertical) dimension, each distributed RUBIES has M replicas, where all replicas form a replication group. Let ? be a distribution group {d1, d2, …, dN}, where each di is a RUBIES instance. Each di is replicated by a replication group Gi {?i1, ?i2, …, ?iM} where each ?ij is a replica of the ith RUBIES instance. A rule set from a given domain is partitioned into N disjoint rule subsets, and distributed over N RUBIES instances of ?.

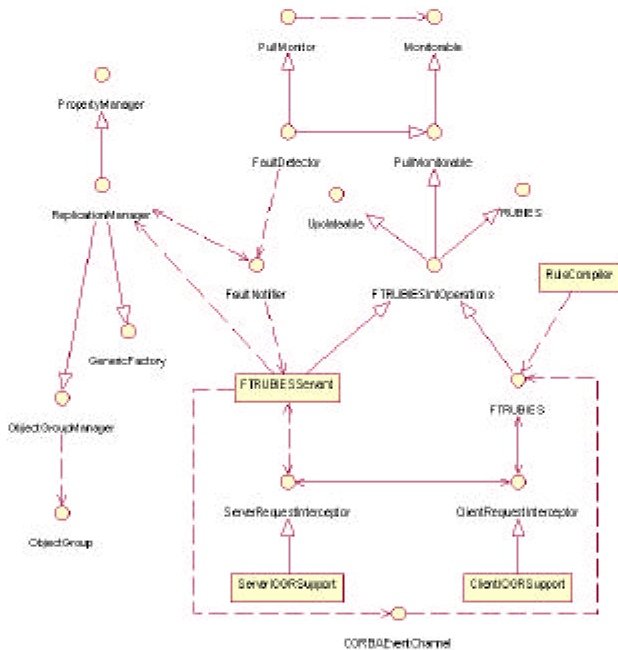

Figure 4.2 Architecture of Fault-Tolerant ERF-CORBA

Part of the architecture of OMG's Fault Tolerant CORBA and part of the architecture of ERF are integrated into the architecture of Figure 4.2.

The *ReplicationManager* combines functions of the *PropertyManager*, the *ObjectGroupManager*, and the *Generic Factory*. Its functions can be invoked directly by group members/servers, clients, and other services. To be either a replica or a member group, it is enough to inherit from the *Updateable* and *PullMonitoreable* interfaces. *FTRUBIESIntOperations* inherits from these two interfaces, as well as from the *RUBIESInt* interface. Clients (e.g., *RuleCompiler* and *CORBAEventChannel*) interact with servers transparently. Thus, clients get the CORBA object reference from the Name Server, narrow it to the interface, and invoke the desired operation.

The lower right brand part of the figure 4.2 helps understand how a client's method invocation takes place within the infrastructure. Two steps are followed, the first one for method invocations, and the second one for replies. First, the client (i.e. Rule Compiler or CORBA Event Channel), after begin connected, makes a request to the server through its interface (i.e. Fault Tolerant RUBIES interface). After that, the request is delegated to the *ClientIOGRSupport* (a client-request interceptor), which, based on the replication style, either multicasts the request to all replicas or forwards the request to the primary replica. Second, the *ServerIOGRSupport* (a server-request interceptor) receives the request and delivers the response through the same path. Some actions can be performed based on the replication style. For example, the ACTIVE replication style is enabled, it will be necessary to detect and suppress duplicated replies.

4.2. Pattern Management

Rules use a pattern management approach to prevent being triggered more than once for a given event pattern.

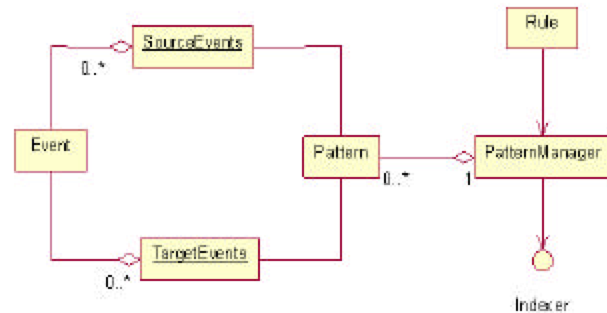

Figure 4.3 Architecture of Pattern Management

Figure 4.3 presents the architecture of Pattern Management, and its relation with the rule representation. The Rule class has complete access to the public interface of the Pattern Manager. The Pattern Manager class stores and arranges map of patterns, indexed in accordance with the specification provide through the Indexer interface. The Indexer interface provides a pluggable mechanism for customized ways to arrange patterns within the Pattern Manager class, thus it can be changed for performance purposes. The Pattern class stores a complete ordered list (Source Events instance) of the events contained in the pattern, since other events arrangements are different patterns, and it stores a complete unordered list (Target Event instance) of the events produced due to the pattern.

4.3. Active Replication

In active replication, all replicas are running and are active at the same time. All replicas reply to a given request, but only one reply is returned to the client and the others are stopped. Upon a crash, there is no recovery overhead, because it is not necessary switch to a new replica. The disadvantage is the number of computational resources involved; however, cost reduction in time is more important than cost reduction in resources. It is crucial to deliver a unique reply and at the same time maintain consistency.

4.3.1. Duplicated-Events Detection and Suppression Mechanism

It is addressed with a mid-tier component that provides a pluggable mechanism that, through an analysis of an event's history, detects if the event has already been delivered. Only events that do not match any of the events previously delivered are delivered and registered within

the mid-tier component, thus preventing the duplication of events.

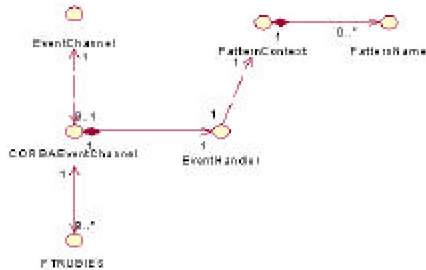

Figure 4.4 Architecture of a Pluggable Duplicated-Events Detection/Suppression Mechanism

In Figure 4.4 *EventHandler* is plugged into *CORBAEventChannel*, thus incoming events are intercepted to create a *PatternName* instance with the purpose of binding this instance to *PatternContext*. Also, outgoing events are intercepted to create a *PatternName* instance with the purpose of asking *PatternContext* if an equal event was already registered.

5. PERFORMANCE ANALYSIS

Performance analysis was made by measuring the execution time of FT-ERF for an increasing number of replicas and increasing number of failures. It was executed over fourteen computers connected by a 100 Mbps Ethernet arranged in to four different subnets, three of them were servers running Solaris 8 and the remaining ones were workstations running RedHat Linux.

The test scenario had six workstations, each one used to run a factory service, which is used for launching a replica. The ERF application was loaded with a rule set containing 193 rules. Failure occurrences were given as a power set, each of which subset was a test case consisting of a set of failure times, defined by the power-set function $F(n=1..\infty)=\{\{f(1)\},\{f(1),f(2)\},\ldots\{f(1),f(2),f(3),\ldots f(n-1),f(n)\}\}$, where n is the number of replicas, f(p=n) = 8 and f(p=1...n-1) = p*T/n determines the time of the failure, p is the position of the replica in the sub set, and T is the average of the execution time of ten failure-free runs with n replicas.

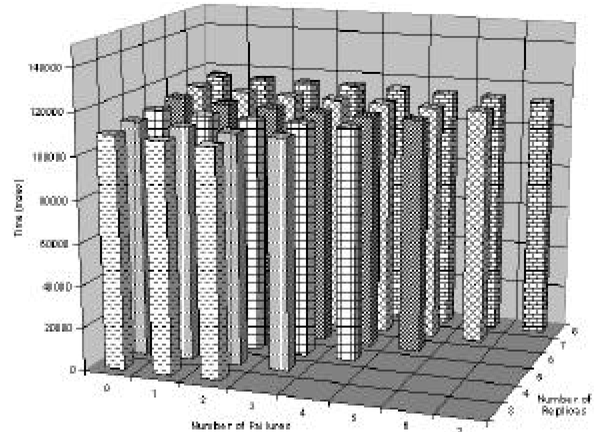

Figure 5.1 Execution time of FT-ERF

In Figure 5.1 the execution time grows since, when a replica is added, it increases the workload of the event channel and of the process for the detection and suppression of duplicated events. The execution time declines since, when a replica fails or is suppressed, it releases workload of the components that were previously charged.

6. CONCLUSIONS AND FUTURE WORK

In distributed systems with ANDRCs with tight time constraints is better to use an AR technique. There are three major advantages of this approach: (i) this is not significantly affected by either increasing the number of replicas or increasing the number of failures; (ii) both clients and replicas need not be aware of the replication mechanism; and (iii) this can be used for large distributed systems. The disadvantage of this approach is that it relies on a centralized mid-tier component. However, it can be replicated for FT with traditional replication techniques. In conclusion, this approach successfully covers the requirements for distributed systems with ANDRCs with tight time requirements, providing an intelligent detection and suppression of duplicates.

This research rise outstanding challenges that need to be resolved in a foreseeable future: (i) scalability for ANDRCs; and (ii) designing and implementation of a general purpose infrastructure for addressing simultaneously fault-tolerance and scalability using replication and load balancing respectively.

7. REFERENCES

[1] J. Arroyo-Figueroa, J. Borges, N. Rodríguez, A. Cuaresma-Zevallos, E. Moulier-Santiago, M. Rivas-Aviles, J. Yeckle-Sánchez, "An Event/Rule Framework for Specifying the Behavior of Distributed Systems", *Proceedings of the third International Workshop on Software Engineering and Middleware*, California, pp. 59-71, 2002.

[2] R. Baldoni, C. Marchetti, S. Tucci Piergiovanni, "Asynchronous Active Replication in Three-tier Distributed Systems", *Prooceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, Tsukuba, Japan, pp. 19-26, 16-18 December, 2002.

[3] C. Collet, , “The NODS Project: Networked Open Database Services”, *Procedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)-Symposium on Objects and Databases*, Cannes, France, June 2000.

[4] B. Krishnamurthy, D. Rosenblum, “Yeast: A General Purpose Event-Action System”, *Research Department at AT&T Bell Laboratories*, 1995.

[5] S. Pleisch, A. Schiper, “Fault-Tolerant Mobile Agent Execution”, *in IEEE Transaction on Computers*, vol. 52, nº 2, February 2003.