Service Redundancy Strategies in Service-Oriented Architectures

Nicholas R. May, Heinz W. Schmidt, Ian E. Thomas

RMIT University, Melbourne, Australia

Abstract

Redundancy can improve the availability of components in service-oriented systems. However, predicting and quantifying the effects of different redundancy strategies can be a complex task. In our work, we have taken an architecture based approach to the modeling, predicting and monitoring of properties in distributed software systems.

This paper proposes redundancy strategies for service-oriented systems and models services with their associated protocols. We derive formal models from these high-level descriptions that are embedded in our fault-tolerance testing framework.

We describe the general framework of our approach, develop two service redundancy strategies and report about the preliminary evaluation results in measuring performance and availability of such services. While the assumptions for the chosen case study are limiting, our evaluation is promising and encourages the extension of our testing framework to cater for more complex, hybrid, fault-tolerance strategies and architectural compositions.

Index Terms—availability; redundancy; service-oriented architecture

I. Introduction

Redundancy is a technique for improving the fault tolerance of software systems. This is achieved by replacing individual components with groups of components that, together, are less susceptible to failures. Common strategies focus on the problem of maintaining state between component replicas and synchronizing changes. However, each strategy impacts differently on global system properties such as performance, availability, and maintainability.

Service-Oriented Architecture (SOA) is a style of software architecture, composed of loosely coupled, largely stateless components called services [2, p37]. In SOA, existing redundancy strategies are principally concerned with synchronizing state between replicas. However, simpler strategies can improve the service availability whilst, at the same time, providing more flexibility to optimize the properties of the redundancy group.

In this paper, we propose several redundancy strategies for SOA and model them formally as Finite State Processes (FSPs) [7]. These models allow us to specify the strategies as concurrent processes and check the safety and liveness properties of the generated state machines. In addition, the FSP allows us to implement the separate processes in an evaluation framework so that we may verify the operation of the strategies.

The rest of this paper consist of the following sections: background to the appropriate areas and related work, a discussion of service redundancy in SOA, an overview of the evaluation framework, and some conclusions.

II. Background

In this section we describe some of the context for this study including: software architecture, SOA, fault tolerance, availability, redundancy, common replication strategies, and some related work.

Software architecture is a discipline of software engineering, which can be used to gain a better understanding of the qualities of software systems. SOA is an architectural style which aims to increase the reuse and interoperability of software components. The architectural components in SOA are called services, which publish well-defined interfaces and can be composed across organizational boundaries. SOA is an abstract architecture in that it describes the principles for service-oriented systems. A common set of standards by which these principles can be applied are known as Web Services. However, SOA systems can be implemented with any distributed technology and a system developed with Web Services may not necessarily conform to SOA principles [2, p104].

A fault tolerant system aims to avoid system failure even when faults are present. The phases of a fault toler-
ant process include: error detection, damage confinement, error recovery, and fault treatment [4]. Availability is a non-functional property of a service which provides one measure of its fault tolerance. A typical definition of availability is the probability that a system will be able to deliver its services when requested [11, p359]. Reliability is another non-functional property. However, reliability of software is associated with the correctness of the response received [11, p373], as compared to availability which is concerned with whether the response is returned.

A common strategy for improving the availability of a software component involves introducing redundancy into the functionality on which it depends [10]. If the components of a system that provide this functionality are always available then redundancy is unnecessary. However, in deployed systems, where the availability of components and that of the networks connecting them is not perfect, redundancy can be achieved by replicating components, and network infrastructure, with components that perform the same function. There are several common strategies for error recovery with redundant software components. These are distinguished by the number of calls, the order of calls made, and the communication between the replicas. A principle concern is that the replicas should be indistinguishable, such that the response from one replica should be indistinguishable from any other.

Passive replication, also called primary site replication [4], is based on the principle that all calls are made to a designated primary component which returns the response. Simultaneous to the response, the state of the primary is synchronized with its backup replicas. On primary failure, the call is redirected to a backup, which then assumes the role as the new primary.

Active replication involves the duplication of the client’s call [4]. This is sent simultaneously to the entire set of replicas. The call will succeed if at least one replica responds, but only one response is retained for use. Replicas are not ordered and so have no relative priority. The third replication strategy is a hybrid of the passive and active strategies, which can be called active client replication [1]. It requires that the client maintains an ordered list of replicas. Calls are sent primarily to the first replica in the list. On error detection the same request is sent to the next replica in the list, and so on until the response is received or all replicas fail.

Service redundancy studies that include multiple redundancy strategies are principally concerned with techniques, implementation frameworks and architectures. Two such studies are: a survey of software replication techniques [3], and an architecture based on communities of Web Services [6]. These describe examples of active, passive and hybrid strategies. However, both these studies assume that service state must be synchronized between replicas and, therefore, the frameworks may be too restrictive when forming redundancy groups from stateless services.

A third study, an implementation framework [5], assembles redundancy groups from diverse services using strategies they call simultaneous and sequential. In addition, they describe a testing framework to determine the total availability for replica groups using the different strategies. However, test invocations were executed at intervals of one minute over a period of two hours. This limits the number of invocations that can be performed and so restricts the accuracy of availability metrics that can be verified.

A fourth study proposes customizable connectors that add fault tolerance to services [8]. The connectors can be used to implement various passive and active strategies, that provide more options for error recovery than those described in the other studies. However, an underlying assumption is that service providers are required to collaborate in order to manage state, which may not be possible with providers in multiple domains.

III. SERVICE REDUNDANCY

The attributes of services in SOA allow for the simplification of redundancy strategies and, therefore, more flexibility in the invocation of services in a replica group. In this study, we are principally concerned with increasing the availability of services and so shall confine ourselves to the strategies that are appropriate to that non-functional property. An advantage of SOA is that it aids the reuse of services across domains of control. However, this means a service provider cannot be guaranteed to be in the same domain as its consumer, which leads to several limitations.

Firstly, when providers do reside in an external domain, the owner of the consumer cannot replicate additional copies of the provider. Therefore, the strategies can only be concerned with the management of invocations between a consumer and a group of existing providers.

Secondly, the availability of a service is a property determined from a consumers perspective, which cannot distinguish between failure types, such as hardware or software failures. This limits our strategies to actions in the error detection and recovery fault tolerance phases. Thus, the only guaranteed method of detecting the failure of a provider is to place a time limit on its response and to initiate error recovery on the non-arrival of the response within this timeout period. Failures due to corrupt or invalid responses can be ignored as they are not associated with availability, but with reliability. Once an error has been detected, a service consumer must find an alternate provider to fulfill its request. If no alternatives are available, for a critical service provider, then the fault cannot be recovered and the consumer must propagate the failure to its service consumers.
In SOA, services are autonomous, independent components that do not normally retain functional state between invocations [2]. However, a service’s operations may not be independent and, in such cases, state can be retained for the duration of a number of invocations, also called a session [9]. To determine the availability of a service we are only concerned with the establishment of a session because failures that occur during a session are issues with the reliability of a service, which can be handled by alternative fault tolerance mechanisms. As a consequence, service invocations do not need to be deterministic and, therefore, replication strategies for availability do not need to synchronize state between the services in a replica group. This removes one limitation that restricts existing replication strategies.

A. Redundancy Strategies

These assumptions and limitations leave us with three strategies for replication in SOA. These we have called parallel, serial and composite redundancy, to distinguish the invocation and recovery modes.

1) Parallel Redundancy: This is an active strategy, where the consumer maintains a set of replica services for a required contract and each invocation involves a request to every replica. The first response received is processed immediately and the send-receive protocol is synchronized to ensure that ‘at-most-one’ response is accepted. The process will fail if no responses are received within the time constraint and, therefore, no error recovery phase is required. An example of parallel redundancy with three replicas is shown on the left hand side in Fig. 1. The consumer sends simultaneous requests to all replicas, the first response is received from Replica3, which is then processed. No further requests are sent and any additional responses are ignored.

2) Serial Redundancy: This is an active client strategy, where the service consumer maintains an ordered list of replicas. Initially, a request is sent only to the first replica in the list. If the time constraint fails then error recovery is handled by sending a duplicate request to the next replica on the list and resetting the time constraint. The cycle continues until a response is received or no more replicas are left to invoke. This process is also synchronized on the first response received, because a response may be received from a replica even after its invocation has exceeded the time check. Serial redundancy has the advantage of reduced communication overhead and workload for service providers, but can be slower as it only invokes one replica at a time. An example of serial redundancy with three replicas is shown on the right hand side in Fig. 1. The consumer sends an initial request to the first replica in its list, Replica1. However, no response is received within the time constraint, so the consumer sends a duplicate request to Replica2. This elicits a successful response which is then processed. No further requests are sent and any response received from Replica1 is ignored.

3) Composite Redundancy: The serial and parallel strategies are primitive strategies, in that they group services at a single level of composition. They provide a limited mechanism to manipulate the properties of a replica group. For instance, if the minimization of network traffic is required then a serial strategy would be preferred. However, if performance is of greater importance then a parallel strategy should be used. In more complex scenarios, the simplification of invocation in SOA allows us to replace any replica in a primitive strategy with a manager, which represents the consumer of another replica group. This enables the assembly of composite, hierarchical structures using replica groups of different strategies.

An example of a composite replication structure is shown in Fig. 2. This structure has a serial group with three children at its root. The first child in the list has been replaced with a manager for a parallel replication group. The operation of this structure results first in simultaneous requests to Replica1 and Replica2. If neither of these respond within the timeout period of the SerialManager, then the SerialManager will send a request to Replica3, and if Replica3 does not respond, to Replica4. This structure could apply where response time and network traffic are to be reduced, but Replica1 and Replica2 are local and so incur no network traffic.
REPLICA = ( initialise -> READY ),
READY = ( destroy -> END
| invoke -> CHECK ),
CHECK = ( available -> HANDLE
| unavailable -> READY ),
HANDLE = ( {callback,done,cancel}
| BLOCKED
-> READY ).

Fig. 3: FSP and state machine of the Replica process.

**B. StrategyProtocols**

In order to implement the strategies it is necessary to construct formal models of them. These models should be flexible, so as to enable the management of replication groups from a variable number of replicas, and easily translated into a programming language. Processes with these characteristics can be modeled using FSP, a language designed specifically for modeling concurrent processes, and a tool called the Labelled Transition System Analysers [7]. The redundancy strategies have been modeled with four process types: a Replica, a Monitor, and two Manager processes, one each for the parallel and serial strategies.

The Replica process represents a service provider. It will respond to a request with a callback action only if it is available. The synchronization monitor is represented by the Monitor process. After creation, this process will wait until it is called back or canceled. The callback and cancel actions are synchronized so that only one process at a time may call either of them. The FSP and state machines for the Replica and Monitor processes are shown in Fig. 3 and Fig. 4 respectively.

The Manager processes represent the service consumer and are composed with a Monitor and a number of Replica processes, executing concurrently. The actions in the processes are synchronized to ensure that only one Replica is able to execute the Monitor’s ‘callback’ action, and the Manager can ‘cancel’ the Monitor if it fails its time check. The ParallelManager and SerialManager processes differ based on the sequence of invoke and timeout actions. The number of replicas associated with a manager is coded as a variable in the

MONITOR = ( start -> WAIT ),
WAIT = ( destroy -> END
| monitor -> READY ),
READY = ( cancel -> WAIT
| callback -> BLOCKED ),
BLOCKED = ( done -> WAIT ).

Fig. 4: FSP and state machine of the Monitor process.

FSP specification. The complexity of the associated state machines will grow as more replicas are added, however, this is only an issue with the visualization and not with the manager processes themselves, and so the process definitions will not suffer from scalability issues. Examples of the FSP and generated state machines for parallel and serial managers, both for two replicas, are shown in Fig. 5 and Fig. 6 respectively.

**IV. Evaluation Framework**

The parallel and serial strategies have been implemented in an evaluation framework so that their properties can be investigated. The framework allows us to control the availability of individual replicas and investigate the influence of the strategies on the availability and response time of the redundancy group. Finally, the framework is validated with some simple scenarios to ensure that the availability of replicas can be accurately specified and that the total execution times are reported correctly.

The FSPs and the test cases were implemented using the Java language. However, several limiting assumptions were made so that the expected results would be easier to calculate. For each test case the assumptions were: all replicas had the same availability, only one redundancy group was used, replica response time was kept constant, and the group was invoked 10 000 times.

The difference between the test results and the calculated values for mean availability and execution time can be compared with the standard error to give a measure of the accuracy of the testing framework. For both properties, 95.5% of the test runs satisfy the 95% confidence interval, which confirms that the testing framework generates availabilities that follow a normal distribution and, hence,
PARALLEL MANAGER (N=2) =
( start -> READY ),
READY = ( finish -> END |
receive -> monitor ->
CALL[0] ),
CALL[i:0..N] =
( when(i<N) invoke[i] ->
CALL[i+1] |
when(i==N) check ->
CHECK ),
CHECK = ( done -> READY |
timeout -> {cancel,done} ->
READY ).

SERIAL MANAGER (N=2) =
( start -> READY ),
READY = ( finish -> END |
receive -> monitor ->
CALL[0] ),
CALL[i:0..N] =
( when(i<N) invoke[i] ->
CHECK[i] |
when(i==N) {cancel,done} ->
READY ),
CHECK[i:0..N] = ( done -> READY |
timeout -> CALL[i+1] ).

Fig. 5: FSP and state machine of a parallel Manager for two replicas.

its accuracy has a direct relationship with the sample size of the test runs. This will allow us to increase the accuracy by increasing the number of invocations in each test run.

The assumptions made in implementing and validating the replication processes limit the scope for which the framework can be used. We are currently working on extensions towards more realistic assumptions to study the following areas; optimization of composites structures, optimization of serial groups, dynamic assembly of replica groups, temporal replication strategies, and the management of services with shared resources.

V. Conclusion

In this paper we model two primitive redundancy strategies, that are applicable for SOA systems, and describe how they can be composed into complex hierarchical structures. In addition, we introduce a framework for testing the impact of choices in fault-tolerance design. The testing framework is modeled on a composite pattern which allows for the easy integration of new fault-tolerance strategies and the assembly of complex redundancy structures. This will allow for the analysis of more complex fault-tolerance strategies by comparing predicted availability and performance with simulated behaviour.

References