A Redundancy Protocol for Service-Oriented Architectures

Nicholas R. May

RMIT University, Melbourne, Australia
Nicholas.May@rmit.edu.au

Abstract. Achieving high-availability in service-oriented systems is a challenge due to the distributed nature of the architecture. Redundancy, using replicated services, is a common software strategy for improving the availability of services. However, traditional replication strategies are not appropriate for service-oriented systems, where diverse services may be grouped together to provide redundancy. In this paper we describe the requirements for a redundancy protocol and propose a set of processes to manage redundant service providers.

1 Introduction

Service-Oriented Architecture (SOA) is a style of software architecture that promotes software reuse and inter-operability. This it achieves by distributing its functionality amongst services, which are loosely coupled software components. However, the distributed nature of SOA presents a serious challenge to a system's quality of service when services are spread across organizational boundaries.

The availability of a service is a quality that is difficult to manage when the service depends on inter-organizational services. It is usually improved by using redundancy, in the form of additional components that provide backup services in the event of a failure. Traditional redundancy strategies are principally concerned with the synchronization of state between identical components, but services in an SOA system are independent and autonomous and so do not need synchronizing. However, some service invocations incur a cost or result in a change in the shared state. Therefore, an SOA redundancy strategy is required to ensure that only one redundant service is executed per invocation.

We address this problem by identifying the requirements that a process must meet if it is to manage redundant services. These requirements are met by adapting the three-phase commit (3PC) fault tolerance protocol to SOA.

In the following sections we will cover some background, the protocol, and draw some conclusions about the protocol and its limitations.

2 Background

In order to define a protocol for redundancy in SOA, we must first provide some background in the relevant subject areas. In this section we discuss the areas of service-oriented architecture, fault tolerance, redundancy, and related work.

2.1 Service-Oriented Architecture

Software architecture is a discipline of software engineering. It can also be viewed as an abstraction of a software system, in terms of its functional components, the properties of the components, and the relationships between them [1].

The software components provide the systems functionality and can range from a primitive computational unit to a whole composite systems. The relationships between components are called connectors. They represent the means of communication between components, and can represent a simple association or a more complex interaction. The properties of a component are its externally visible, non-functional qualities. They influence the quality by which the functionality is provided by the component, but do not affect the functionality provided. Availability is an example of a non-functional property.

Another feature of architecture is the use of styles. These are patterns of software composition that have well known quality consequences. Styles are defined by the components types, their relations and the rules by which they may be combined [18].

Service-Oriented Architecture (SOA) is a style of software architecture designed to utilize distributed components that may be located across organizational boundaries. Its goals are to promote the reuse, evolution, scalability and interoperability of software components [12]. SOA components are called services, which interact through the publish and subscribe connection pattern.

The principles of SOA provide guidelines for implementing systems so that the aims of SOA can be achieved [2,5,12,15,20]. The important principles for this study are as follows;

- **Discoverability:** Services should be visible, such that they can be found and accessed via a discovery mechanism. This is accomplished by publishing descriptions, to some form of repository, in a widely accessible and understandable format.
- **Composability:** Services can be composed into composite services, either by static definitions or by the dynamic discovery of services at run time.
- **Statelessness:** The purpose of invoking a service is to realize an effect. This may be a response message or a change in the shared state of the participating services. However, a service provider is stateless in that it need not retain information about the state of a service consumer between invocations.

Other features of a service include; defined service contract, loose coupling, autonomy, abstraction, and reusability.

SOA is an abstract architecture in that it only describes the principles to which service-oriented systems should adhere. A common set of standards by which these systems can be implemented, collective known as Web Services, are published by the Organization for the Advancement of Structured Information Standards (OASIS) and the World Wide Web Consortium (W3C). The core standards provide for the description, publication, discovery, and consumption of services using the publish-subscribe communication paradigm.

Additional specifications, commonly known as the WS-* extensions, define standards for managing the quality of service consumption. Some of the Web Service extensions include; WS-ReliableMessaging (OASIS), WS-Coordination (OASIS), WS-AtomicTransaction (OASIS), and WS-Policy (W3C). These provide frameworks for managing messages, coordinating actions, implementing fault tolerance features, and defining policies for quality of service constraints. Two additional languages have been defined to allow the specification of orchestrations, BPEL4WS (OASIS), and choreographies, CDL4WS (W3C), of interacting services.

2.2 Fault Tolerance

A fault tolerant system aims to avoid system failure even if faults are present. In terms of SOA, a failure occurs if a service is unable to respond to a request as defined by its published definition. In an inter-organization SOA the most important phase of fault tolerance is error detection. Once an error is detected, a service consumer must take the appropriate action to find another provider to fulfill its request. This form of error recovery is the only guaranteed option available, where failed service providers may be in external domains. If no alternatives are available for a critical service provider then the fault cannot be recovered and the consumer must propagate the failure. The service will remain in a failure state until it can discover a working service provider to satisfy its critical functions.

Many techniques have been proposed to improve the fault tolerance of distributed systems. These can be characterized as either optimistic or conservative approaches. Optimistic techniques make assumptions from the properties of the system in order to improve the performance of fault tolerance. However, when the assumptions fail the technique requires additional work to undo operations. A discussion of optimistic approaches is provided by Jiménez-Peris and Patiño-Martínez [8].

Conservative techniques, such as atomic commitment, involve a greater number of messages under normal operation than an optimistic technique, and hence a worse performance. Two examples of atomic commitment are the two-phase (2PC) and three-phase (3PC) commit protocols. Both protocols assume that the communications network is reliable and detection of service failures is identified by timeout actions initiated by the non arrival of expected messages. However, the 3PC contains additional operations and states that ensure that it is a non-blocking protocol.

A finite state automata (FSA) of a 3PC protocol, adapted from Jalote [7, p239], is shown in Fig. 1. This FSA shows the input and output messages associated with state transitions, which synchronize a COORDINATOR with any number of PARTICIPANT components. It can seen that this is a non-blocking protocol because there is no commit state (c) adjacent to an abort state (a) or non-committable state.

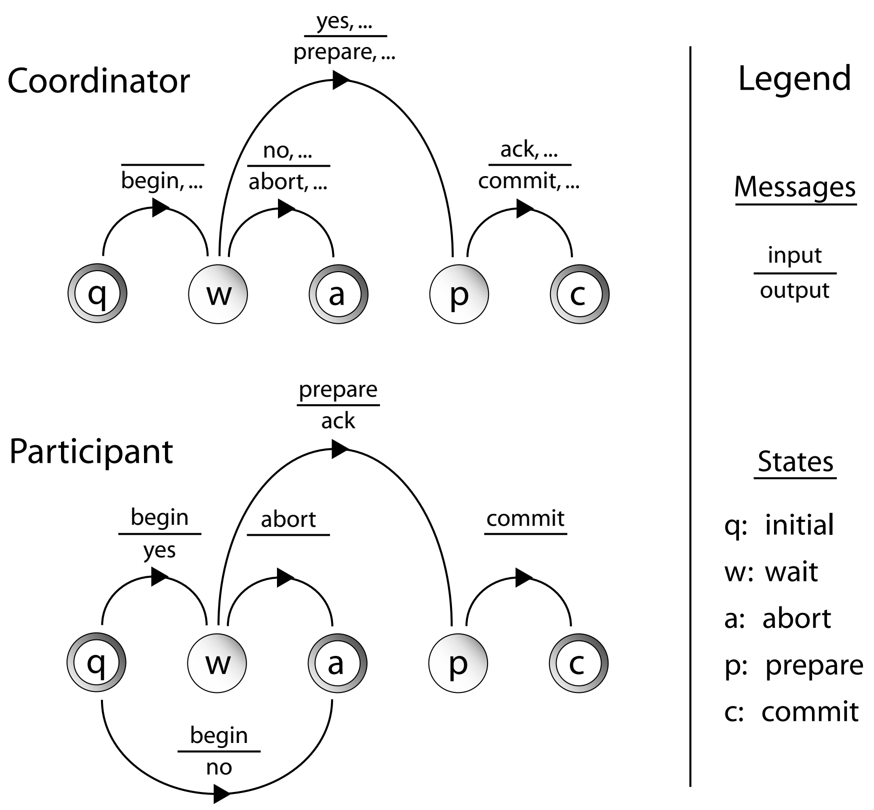

Fig. 1. FSA of a Three-Phase Commit protocol

2.3 Redundancy

An availability of less than 100% can result in a failure of a system if it means that a critical call cannot be serviced. If the required functionality is available in another component then a system failure may be avoided. A common strategy for improving the availability of a software component involves adding redundancy into the functionality on which the component depends [17]. Redundancy can be defined as the introduction of components that are not needed for the correct operation of the system if no failures occur [7]. This can be achieved by replicating copies of critical components.

In SOA, replication strategies must be assessed in terms of the service definition and the dynamic nature of binding. Service discovery allows a consumer to build a list of providers that are not identical but can still satisfy the required contract. These can be considered replicas for the sake of redundancy in SOA. However, these replicas may exist across organizational boundaries, so any strategy that relies on communication between replicas must be excluded, which means that passive replication [7] is not feasible in an SOA system. Furthermore, we can assume that any request to a group of replicas is atomic, stateless and satisfied if “at-least-one” response is returned. Finally, the distributed nature of an inter-organization SOA means that functional error detection is not guaranteed. Therefore, the best detection strategy is a time check failure, such as failing on timeout of a response.

2.4 Related Work

Studies of service redundancy have been focused on traditional solutions using service replication. They can generally be divided into replication strategies, replication architectures, and implementation frameworks.

Several studies that focus on the various replication strategies, such as ‘active’ and ‘passive’ techniques, are presented by Maamar et al. [11], Guerraoui and Shipper [6], and Chan et al. [3]. These papers discuss replication communities in Web Services, survey replication techniques, and evaluate temporal and spacial redundancy techniques.

Architectures are described by Osrael et al. [14], who propose a generalized architecture for a service replication middleware, and Juszczyk et al. [9], who describe a modular replication architecture.

Among the frameworks are those described by Salas et al. [16], Engelmann et al. [4], and Laranjeiro and Vieira [10]. They propose an active replication framework for Web Services, a virtual communication layer for transparent service replication, and a mechanism for specifying fault tolerant compositions of web services using diverse redundant services, respectively.

The focus of these studies is on describing and implementing various strategies for invoking and maintaining replicated services. However, most do not treat services as autonomous components, and so their applicability is limited for an SOA system.

3 Protocol

A protocol is a set of rules governing the exchange of data between devices [19]. In this instance, it can be described by the processes, states and allowable actions, that satisfy the protocol’s requirements. In order to deduce a protocol we must first define the requirements and assumptions for redundancy in an SOA system.

We can make the following assumptions of services included in a redundancy group. Service are stateless outside of the protocol. Each conversation with an available provider is assumed to be reliable because there are existing protocols, such as WS-ReliableMessaging and WS-AtomicTransaction, to manage a conversation once it is established. Finally, operations are not guaranteed to be idempotent, in that they may have an associated cost or state change for each invocation. However, any operation that is idempotent, such as a simple query, may be invoked many times without consequence and so will not require a redundancy protocol.

```
CANDIDATE = (begin   -> WAIT),
WAIT      = (no      -> END    | yes     -> READY),
READY     = (abort   -> END    | prepare -> PREPARED),
PREPARED  = (execute -> END).
```

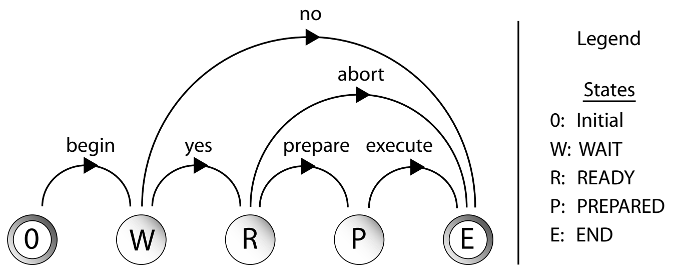

Fig. 2. FSP and state machine of the CANDIDATE process

The requirements of the protocol are as follows;

1. Provider services are autonomous (SOA).
2. Provider services in a redundancy group may have different contracts, but each must satisfy a common sub-contract (SOA).
3. Consumers must be able to conduct multiple, simultaneous conversations with provider services (Fault Tolerance).
4. Any invocation of an operation that has a cost must result in the execution of only one redundant service (SOA).
5. Providers are selected by voting or time ordering (Fault Tolerance).

These requirements lead to the following attributes of the protocol. Firstly, each redundant service must be modeled as an independent concurrent process (Req. 1). Secondly, the protocol must be independent of a particular service contract (Req. 2). Thirdly, the protocol must be non-blocking (Req. 3). Fourthly, the protocol must support 'at most once' execution (Req. 4). Finally, the protocol must include a controlling action that can select which redundant service to invoke (Req. 5). A solution to these requirements is to adapt the 3PC protocol. This must be modified to ensure that only one process is executed, rather than all committed. To reflect this change in emphasis the name 'Participant' is replaced by the name 'Candidate' to represent the redundant processes.

The constituent processes of the protocol are modeled as Finite State Processes (FSPs) [13]. FSP is a language especially suited to modeling synchronized, concurrent processes. Processes are defined in a textual language that represents the states and the actions that trigger state transitions. Concurrency is modeled in FSP with interleaved actions. However, actions that must be performed simultaneously can be defined with shared action pairs. FSP does not show message exchange explicitly, but messages are the normal mechanism used to implement shared actions. The Labeled Transition System Analyzer (LTSA) [13] is a tool that can be used to generate state machines from FSP definitions and to check

```
COORDINATOR = (begin    -> WAIT),
WAIT        = (timeout  -> END    | found   -> READY),
READY       = (timeout  -> END    | select  -> PREPARED),
PREPARED    = (invoke   -> END).
```

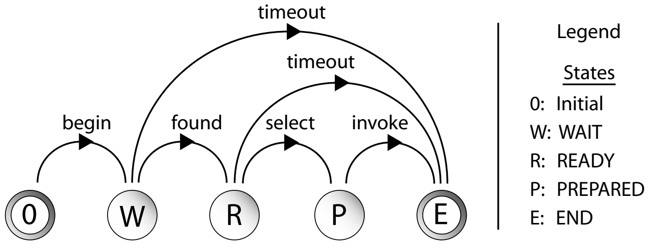

Fig. 3. FSP and state machine of the COORDINATOR process

their properties. The protocol definition using FSP consists of a three processes; CANDIDATE, COORDINATOR and REDUNDANCY.

The CANDIDATE process is initialized with a *begin* action, after which it will respond with a *yes* or *no* action to indicate whether it is able to perform its service. If it is able, it will wait until it receives a *prepare* action. If none is received before a specified timeout period then it will *abort* the process. A CANDIDATE that is PREPARED will then be *executed.* This process satisfies the requirements for a non-blocking protocol because the *execute* and *abort* actions are not available from the same state. The FSP and State Machine for the CANDIDATE process are shown in Fig. 2.

The COORDINATOR process is also initialized with a *begin* action. It will then wait until it receives a message to indicate that a CANDIDATE has been *found.* If none are found before a specified period then the COORDINATOR will *timeout* and the process will end. If a CANDIDATE is found then the COORDINATOR will *select* an appropriate CANDIDATE using a specified election method, for instance the first response. Finally, the COORDINATOR will *invoke* the CANDIDATE in the PREPARED state and accept the response. Similarly to the CANDIDATE process, this process also satisfies the requirements for a non-blocking protocol. The FSP and State Machine for the COORDINATOR process are shown in Fig. 3.

The REDUNDANCY process consists of a COORDINATOR and two CANDIDATE processes, labelled *a* and *b*. In addition, the FSP includes the following shared actions pairs; *(begin,begin)*, *(yes,found)*, *(select,prepare)*, and *(invoke,execute).* This process ensures that the COORDINATOR will select and invoke a single CANDIDATE, and the other CANDIDATE processes will abort if available but not selected. The FSP and state machine for a REDUNDANCY process with two CANDIDATE processes is shown in Fig. 4. The state machine is shown only to give an impression of its scale. This process satisfies the requirements for a redundancy protocol in an SOA system because;

– Each service provider is modeled by a separate CANDIDATE process.
– The protocol is independent of any particular service contract

```
||REDUNDANCY = (COORDINATOR || {a,b}:CANDIDATE)
               /{ begin/{a,b}.begin,
                  {a,b}.yes/found,
                  {a,b}.prepare/select,
                  {a,b}.execute/invoke }.
```

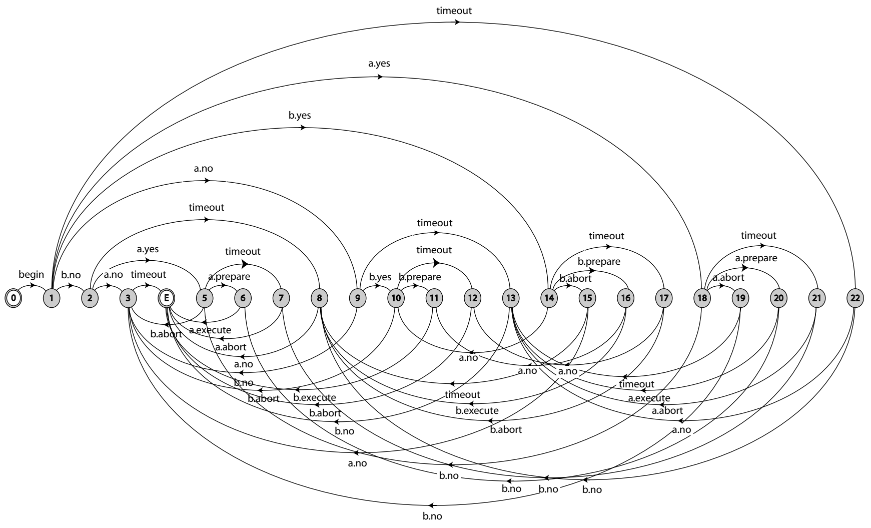

Fig. 4. FSP and state machine of a REDUNDANCY process with two CANDIDATES

- The processes are non-blocking.
- "At most one" CANDIDATE is executed.
- The 'select' action provides a mechanism for the COORDINATOR to determine which CANDIDATE to invoke.

The state machine covers all available paths of the interleaved actions, synchronized by the shared actions. The safety and liveness properties of the state machine can be validated using the LTSA. Safety properties include the absence of deadlocks and mutual exclusion of participant execution. The LTSA reports no deadlocks for the state machine. In addition, the LTSA animator allows individual paths to be traced through the state machine. This shows that each path that includes an *execute* action includes only one. Liveness properties include path progress and successful termination. The LTSA shows that all states, except the END state, have at least one out action, and that all paths eventually terminate at the END state.

4 Conclusions

In this paper we discuss the background to redundancy in SOA systems and identified the requirements for a protocol to manage redundant services which are not idempotent. These requirements have been satisfied by adapting the three-phase commit protocol to ensure that 'at most one' redundant service is executed per invocation. The protocol consists of two non-blocking processes (CANDIDATE and COORDINATOR) and a synchronizing process (REDUNDANCY), all of which have been modeled as finite state processes.

The protocol is a conservative, passive, fault detection mechanism, in that it cannot predict where a fault will occur or actively exclude unavailable services before the invocation process begins. This will reduce the performance and increase the number of messages at invocation compared to an active or optimistic protocol. However, a conservative protocol can be adapted to combine the fault detection with the negotiation of quality attributes for the provided service. Therefore, this protocol would be more efficient in a dynamic, quality negotiation scenario.

Only a simple redundancy process has been modeled, with two redundant candidates and selection by first response. This protocol would benefit by modeling more complex redundancy groups and investigation of how to integrate different selection strategies. In addition, the modeling of fault recovery has not been considered. For instance, how would the protocol recover from a failure whilst in the *prepared* state?

The processes of this protocol have not yet been implemented. Further investigation will be required to determine the most appropriate method to communicate and implement different redundancy strategies. The various Web Service specifications may provide a basis for the protocol and a means to communicate implementation parameters, such as whether a service invocation incurs a cost.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
2. Bastani, F., Ma, H., Gao, T., Tsai, W.-T., Yen, I.-L.: Toward qos analysis of adaptive service-oriented architecture. In: IEEE International Workshop on Service-Oriented System Engineering (SOSE 2005), pp. 219–226 (2005)
3. Chan, P.P.W., Lyu, M.R., Malek, M.: Making services fault tolerant. In: Penkler, D., Reitenspiess, M., Tam, F. (eds.) ISAS 2006. LNCS, vol. 4328, pp. 43–61. Springer, Heidelberg (2006)
4. Engelmann, C., Scott, S.L., Leangsuksun, C., He, X.: Transparent Symmetric Active/Active Replication for Service-Level High Availability. In: 7th IEEE International Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil, May 2007, pp. 14–17 (2007)
5. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. The Prentice Hall Service-Oriented Computing Series from Thomal Erl. Prentice Hall, Upper Saddle River (2005)
6. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. Computer 30(4), 68–74 (1997)
7. Jalote, P.: Fault Tolerance in Distributed Systems. PTR Prentice Hall, Englewood Cliffs (1994)

8. Jiménez-Peris, R., Patiño-Martínez, M.: Towards Robust Optimistic Approaches. In: Schiper, A., Shvartsman, M.M.A.A., Weatherspoon, H., Zhao, B.Y. (eds.) Future Directions in Distributed Computing. LNCS, vol. 2584, pp. 45–50. Springer, Heidelberg (2003)
9. Juszczyk, L., Lazowski, J., Dustdar, S.: Web service discovery, replication, and synchronization in ad-hoc networks. In: 1st International Conference on Availability, Reliability and Security (ARES 2006), pp. 847–854. IEEE Computer Society, Washington (2006)
10. Laranjeiro, N., Vieira, M.: Towards fault tolerance in web services compositions. In: 2007 workshop on Engineering fault tolerant systems (EFTS 2007), p. 2. ACM, New York (2007)
11. Maamar, Z., Sheng, Q.Z., Benslimane, D.: Sustaining web services high-availability using communities. In: Third International Conference on Availability, Reliability and Security (ARES 2008), Barcelona, Spain, pp. 834–841 (March 2008)
12. MacKenzie, C.M., et al.: Reference Model for Service Oriented Architecture 1.0. Organization for the Advancement of Structured Information Standards (October 2006) (October 5, 2007), `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html`
13. Magee, J., Kramer, J.: Concurrency: State Models and Java Programming, 2nd edn. John Wiley and Sons, Chicester (2006)
14. Osrael, J., Froihofer, L., Goeschka, K.M.: What service replication middleware can learn from object replication middleware. In: 1st Workshop on Middleware for Service Oriented Computing (MW4SOC 2006), pp. 18–23. ACM, New York (2006)
15. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-oriented computing: A research roadmap. In: Cubera, F., Krämer, B.J., Papazoglou, M.P. (eds.) Service Oriented Computing (SOC), Schloss Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 05462, Internationales Begegnungs und Forschungszentrum fuer Informatik (IBFI) (2006)
16. Salas, J., Pérez-Sorrosal, F., Patiño Martínez, M., Jiménez-Peris, R.: WS-Replication: a framework for highly available web services. In: 15th International Conference on World Wide Web (WWW 2006), pp. 357–366. ACM, New York (2006)
17. Schmidt, K.: High Availability and Disaster Recovery: Concepts, Design, Implementation. Springer, Berlin (2006)
18. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Upper Saddle River (1995)
19. Thompson, D. (ed.): The Concise Oxford English Dictionary of Current English, 9th edn. Oxford University Press, Oxford (1995)
20. Tsai, W.T., Malek, M., Chen, Y., Bastani, F.: Perspectives on service-oriented computing and service-oriented system engineering. In: Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering, Washington, DC, USA, pp. 3–10. IEEE Computer Society, Los Alamitos (2006)