Consistency for Web Services Applications

Paul Greenfield, Dean Kuo, Surya Nepal
CSIRO ICT Centre
Locked Bag 17
North Ryde, NSW 1670
Australia
firstname.lastname@csiro.au

Alan Fekete
School of Information Technologies
University of Sydney
NSW 2006
Australia
fekete@it.usyd.edu.au

Abstract

A key challenge facing the designers of service-oriented applications is ensuring that the autonomous services that make up these distributed applications always finish in consistent states despite application-level failures and other exceptional events. This paper addresses this problem by first describing the relationship between internal service states, messages and application protocols and then shows how this relationship transforms the problem of ensuring consistent outcomes into a correctness problem that can be addressed with established protocol verification tools.

1. Introduction

Web Services and service-oriented architectures are being promoted as the best way to build the next generation of Internet-scale distributed applications. These applications are made by gluing together opaque and autonomous services, possibly supplied by business partners and third party service providers, into loosely-coupled virtual applications that can span organisational boundaries and connect large-scale business processes.

Services are just applications that expose some of their functionality to other applications in a particularly simple and restricted way. Services are autonomous, opaque (and probably stateful) applications that communicate with each other solely by exchanging asynchronous messages. This services model is extremely simple but, unfortunately, this simplicity does not mean that large-scale service-based applications will prove to be easy to develop in practice or sufficiently reliable when they are deployed. Building robust large-scale stateful distributed systems will still prove to be hard for all the well-known reasons, but the wide-spread adoption of service-oriented architectures means that more programmers (and probably less skilful ones) will get to face these challenges with very little technology support or rigorous programming patterns and guidance without further fundamental research.

Some of the most critical problems faced by developers of service-oriented systems come from the independent and stateful nature of services and the potential for concurrency. The main focus of our work has been the consistency problem: ensuring that the set of autonomous services making up one of these applications always finish in consistent states despite failures, races and other such exceptional events. This problem can be solved through the use of traditional transaction technology but this solution depends on assumptions of trust and timeliness that no longer apply in the new loosely-coupled services-based world.

Rather than attempting to provide the equivalent of traditional distributed transactions for the loosely-coupled Web Services world, our approach has been focussed on the more modest goal of supporting the development of tools, programming models and protocols for the detection and avoidance of consistency faults, either at design time or at run-time.

The key to our work has been establishing a relationship between internal service states, messages and application-level protocols. This insight lets us transform the problem of ensuring consistent outcomes into a protocol problem, opening up the possibility of using proven techniques from the world of protocol verification as the basis for tools that can improve the robustness of service-based distributed applications.

These design-time protocol-based checking tools can detect the presence of consistency errors in a design, but they do not prevent inconsistent outcomes that result from implementation flaws. We are working on using the same message-based definitions of correctness and consistency as the basis for protocols that can dynamically check for

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

consistency failures at the termination of service-based applications, without requiring an overall coordinator or a global view of the entire application.

We have also been looking into the consequences of the lack of isolation in this services-based world, and our initial research indicates that this problem can also be resolved to some extent by taking a similar approach based on protocols and patterns.

2. Some Challenges

This paper uses a simple two-party eProcurement application as a source of examples. The application consists of two stateful services, Customer and Merchant, and these implement three activities: placing an order, payment and delivery. Fig. 1 shows the normal message flow between the customer and the merchant in this application.

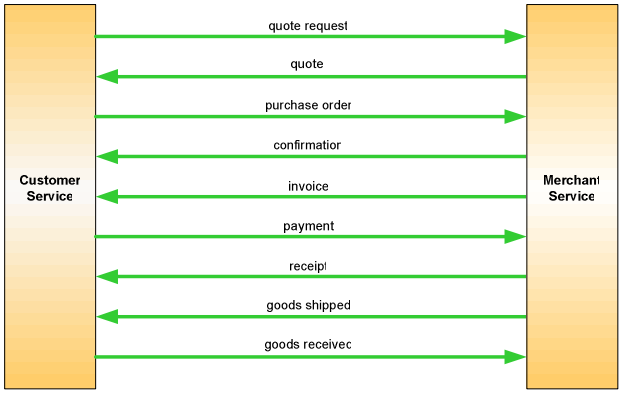

Fig. 1. Normal message flow in eProcurement

The application shown in Fig. 1 is conceptually simple to build when nothing can go wrong. The difficulties come from having to ensure that all the participating services always reach consistent outcomes despite application-level exceptions, asynchronous events and the effects of concurrency.

Our work has focussed on three major problems that could be encountered by such applications:

- **No Termination.** A distributed application may fail to terminate in some cases because of deadlocks, starvation and other such problems. For example, the application will deadlock if it can reach the state where the merchant is waiting for payment before delivering the ordered goods while the customer is waiting for the goods to arrive before paying for them.
- **Unprocessed Messages.** The termination of the participating services needs to be coordinated to ensure that no messages are left unprocessed after all of the participants have finished. For example, the customer may send its payment before the due date and then terminate. This payment message could be delayed in transit and not arrive at the merchant until after the due date has expired. As the merchant has not received the payment by the due date, it will send a late fee message to the customer, but this message can never be processed as the customer has already terminated.
- **Inconsistent outcomes**. The set of services making up the application must always finish in globally consistent states. For example, if the merchant's state records that they have received payment and delivered the goods, the customer's state must reflect the merchant's state and record that they have paid the invoice and received the goods they ordered.

Determining whether or not an application finishes in a consistent state can be quite difficult for service-oriented applications. We could evaluate a set of global consistency constraint expressions whenever the application terminates but this would require both access to the internal states of all the services, and a central coordinator to perform the consistency check. The opaque and peer-to-peer nature of service-oriented systems means that we cannot take this path and need to find an alternative solution.

Asynchronous messaging also greatly adds to the difficulty of always reaching consistent outcomes as it can introduce race conditions. A service can send multiple messages before receiving any responses and these messages may be notifications of application-level exceptions. The asynchronous nature of these messages means each service does not know what messages are currently in transit and potentially conflicting messages can pass by each other on the network.

3. Services and Transactions

The problems of ensuring consistency have natural solutions in the realm of on-line transaction processing, based on the key abstraction of ACID transactions [16]. This approach works well for short-duration applications that execute within a single trust domain. However, it is inappropriate for service-oriented systems with distributed applications that are loosely coupled across organisational boundaries, as implementations of ACID transactions rely on locking resources to achieve isolation and atomicity.

Over the past decade, numerous advanced transaction models have been proposed to address the problem of providing some of the benefits of ACID transactions for long-running and loosely-coupled systems [12, 14]. Sagas it is one notable example, which uses compensators to semantically undo completed operations, so reverting the system back to its original state when the application encounters a failure. This model has been accepted in several of the standards proposed for service-oriented computing, such as BPEL4WS [4] and WSCI [7]. The drawback of this model is that the application may not want revert to the original state in response to an exceptional event; rather it may want to handle the

problem and continue making forward progress. The limitations of Sagas are described in one of our earlier papers [15].

Recently service-based transaction frameworks such as WS-Coordination and related standards [1, 10], BTP [5] and WS-CAF[5] have been proposed to address the transactional problem in service-oriented distributed systems. WS-Coordination defines two types of transaction protocols: WS-AtomicTransaction [1] and WS-BusinessActivity [3]. WS-AtomicTransaction maps the traditional ACID semantics to the coordination framework and can only be used in a trusted domain. WS-BusinessActivity defines a set of patterns for distributed activity termination and is not meant to solve the problem of consistency and isolation in service-oriented system. BTP and WS-CAF also provide a set of patterns and protocols, but do not deal directly with the problem of consistency and isolation, leaving these issues to be resolved by programmers and developers.

4. Contracts, Protocols and Consistency

A service-based application is built by combining a number of autonomous, possibly stateful, services. Whenever one of these distributed applications finishes, all of these participating services must agree on the outcome and they must finish in one of an agreed set of consistent distributed final states.

This agreed set of allowable states could be defined using global integrity constraints over the internal state of the participating services. An application would then be in a consistent state at termination if all the participating services are finished and these consistency constraints are satisfied. Using internal state in this way, however, violates the 'services are opaque' principle of service-oriented architectures and instead, in our work, we restrict our consistency constraint expressions and only allow them to refer to the messaging history of the application.

Restricting our consistency constraints expressions in this way is based on the assertion that for a large and significant set of service-based applications, all internal service state that needs to be referenced in their consistency constraint expressions will be unambiguously reflected in specified protocol messages. For example, in our eProcurement example, we want to be able to ensure that both the customer and the merchant agree on whether the ordered goods have been paid for. Our consistency expression in this case could refer to internal payment process state within the merchant service, but this state is unambiguously reflected in the *receipt* message that the merchant always sends to the customer to indicate the satisfactory completion of its internal payment process. As long as the relationship between this internal state and the *receipt* message is unambiguous, then we can safely use the occurrence of this message in an application protocol sequence as a proxy for the corresponding internal state in our constraint expressions.

This assertion about state being reflected in messages links service state and consistency constraints to messages and application protocols, and lets us define the correctness of an application protocol and the consistency of an application purely in terms of message sequences and consistency constraints that refer only to messages.

Our approach to ensuring the consistency of service-based distributed applications starts with the concepts of contracts and application protocols. A contract is an abstract definition of the externally observable messaging behaviour of a single service. Contracts are an important concept in their own right and specify the messages that a service can send and receive, and the causal relationships between these messages. An application protocol is the set of all possible sequences of messages that can be exchanged between the services participating in a distributed application. As the messaging behaviour of a service is defined by its contract, it follows that the application protocol of a service-based application is the set of all possible message sequences that are allowed by the contracts of the participating services.

As described in the next section, we have used these definitions of contracts and application protocol correctness to statically prove that our service-based example applications always finish with their participants in globally consistent states. This work could be used as the basis for tools that will let designers show that their service-based systems are free of certain types of common errors, including not always finishing in consistent global states.

As part of this work, we needed a simple and expressive way to specify the contracts of the services participating in an application so that we could derive the associated application protocol. Our method of defining a contract uses conditions to state when a service can send or receive specified messages. This approach is similar to the ECA (Event-Condition-Actions) programming languages used in active databases and agent based systems [20]. Our contracts simply specify the messages that can be received or sent by a service (its *in* and *out* messages) and associate a condition with each message that defines precisely when it is allowed to be sent or received. These conditions are Boolean expressions over the messaging history of the service. Some of these messages are labelled as final messages to signify that a service's participation in a distributed application has completed once they have been sent or received. This approach to defining contracts is independent of our work on consistency and application protocol correctness, and can be used as a general way to define Web Services contracts. It has been used in the SOAP Services Description Language's (SSDL) [8] rules protocol framework [13] to define service contracts.

There are already a number of standards that address the problem of specifying the messaging behaviour of a service, including WSDL [8], BPEL4WS [4] and WSCI [7], and there are also proposals based on process algebras

such as CCS [18], CSP [11] and π-calculus [19]. We considered these alternative ways of defining contracts as part of this work and found that they either did not have sufficient expressiveness or they become very tedious to use for all but the simplest such message sequences, particularly when asynchronous messages were allowed.

One example that illustrates the complexity that can arise even in seemingly simple contracts is handling customer-initiated cancellation. Our eProcurement application lets customers send a cancellation request at any time once they have sent a quote request. This means that the merchant's contract must written to specify that a cancellation request can be received after each possible incoming and outgoing message once a quote request has been received. The permitted responses then depend on whether the cancellation request was received before or after the purchase order was received. It is possible to specify this type of messaging behaviour using BPEL4WS, CSP, CCP or π-calculus but the service descriptions quickly become verbose and tediously repetitive.

5. Static Consistency Checking

We have used these definitions of contracts and consistency to show that it is possible to develop design-time tools that can successfully check that service-based applications always terminate with their participating services in consistent states. This proof-of-concept starts from the formal definition of the contracts of the participating services and then uses a model checker (SPIN [17]) to test whether these contracts are compatible and that the resulting application protocol meets our specified correctness criteria, including consistency. The contracts are modelled using SPIN's Process Meta-Language (PROMELA) and the correctness properties are specified in Linear Temporal Logic (LTL).

The model checker tests for three formal correctness properties for application protocols that correspond to the common problems discussed in Section 2. These properties are:

1. That all messages sent or received by a service comply with the conditions defined in its contract. No messages are allowed to be left unprocessed after a distributed application has finished.
2. That the application protocol eventually terminates. We assume that the application protocol contains no unbounded loops and that each service will eventually send or receive a final message. This property ensures that the application protocol will not deadlock and that all the participating services will eventually terminate.
3. That services finish in consistent states and so agree on the final outcome of the distributed application We define consistency through the use of global consistency constraints that are expressed in terms of the messaging state of the application (what messages have or not been sent and received by the participating services). As an example of one of these consistency constraints, after a successful purchase both the merchant and customer should agree that goods have been delivered and payment has been received. In terms of messages, the merchant will have sent a *receipt* and received a *goods received* acknowledgement; and the customer will have received a *receipt* and sent a *goods received* acknowledgement. There can be other consistency constraints that must hold as well as this one, and there can be any number of alternative constraints that define other acceptable outcomes.

This last property is the one that lets us use tools originally intended for checking protocol correctness to verify that all message sequences belonging to a given application protocol finish with their participating services in globally consistent states.

At this time, our model checking work uses a set of global consistency constraints when checking that an application protocol meets this consistency-based correctness property. We are already considering defining local consistency constraints as part of a service's contract and then deriving the required global consistency constraints from these local constraints. Further investigation of this possibility is being planned as future work.

6. Designing for Consistency

We have successfully specified and verified a number of asynchronous two-party distributed applications using the SPIN-based approach just described. We initially found that defining error-free contracts and application protocols was harder than expected, and we repeatedly had problems with race conditions and unprocessed messages at termination. We found that two important guidelines helped us produce error-free designs.

The first, and most critical, design guideline is that any internal service state that is needed to determine whether a distributed application has reached a consistent outcome must always and unambiguously be reflected in protocol messages. We believe that this requirement just reflects traditional business processes and protocol design. For example, when the merchant service reaches the point where it regards the goods as having been successfully paid for, it must send a suitable message to indicate that it has reached this state – a *receipt* message in our example. The merchant service is not allowed to reach the 'paid' state without sending out such a message, and once this state has been reached the merchant service cannot change it without sending out another message to signal the 'paid' state has changed.

Our second guideline is to use the WS-BusinessActivity standard [3] to terminate each sub-protocol and protocol. Every application protocol and

sub-protocol must eventually terminate and WS-BusinessActivity provides a very general framework for coordinating the termination of stateful pair-wise interactions. We found that adopting WS-BusinessActivity allowed us to avoid certain classes of problems in protocol design such as deadlocks and not reaching agreed outcomes.

We found it important to think of WS-BusinessActivity as a set of related protocol elements, or a toolkit for terminating pair-wise stateful interactions, rather than as a single monolithic protocol. Not all possible paths defined in the WS-BusinessActivity specification need to be used in any one application protocol [10] and the system designer is free to use only those paths that are appropriate for their purposes. We initially found that it was very difficult to define correct application protocols, even for what seemed like relatively simple examples, as race situations were not being properly handled. However, once we adopted WS-BusinessActivity, and gained some experience, the task of designing correct contracts and application protocols became much easier.

7. Further Work

The work discussed so far does not guarantee that a service-based distributed application can never finish with its participating services in inconsistent states, but it can let a developer check that their application does not contain consistency-related errors resulting from incompatibilities between service contracts.

We have also developed a protocol for dynamic consistency checking that can be run at the termination of a service-based application. This protocol is based on the way that the reflection and transfer of critical state within messages links the local consistency expressions for each of the participating services and should let us verify global consistency at termination without needing global consistency expressions and an overall coordinator to evaluate them. The correctness of this protocol is based on formal work we have underway to show that we can derive global consistency constraints from local consistency constraints that are defined as part of an extended contract. This extension also completes our work on contracts and removes the need for global consistency constraints during verification.

The other major problem facing developers of loosely-coupled service-based applications is the lack of isolation and the subsequent risk that concurrent applications will interfere with each other. Our initial research indicates that this problem can also be resolved to some extent by taking a similar approach based on protocols and patterns such as reservations.

The work presented in this paper also needs to be generalised for distributed applications consisting of an arbitrary number of services. The discussions in this paper deal with only two services and WS-BusinessActivity is also just a pair-wise termination protocol. We are currently investigating whether composition and delegation mean that practical multi-party service-based applications can always be regarded as two-party for the purposes proving their correctness. The correctness properties and the methods for specifying a contract remain the same for multi-service applications, however specifying correct application protocols may be difficult if we need to coordinate termination between many services, not just two.

8. References

1. WS-Coordination. http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf
2. WS-AtomicTransaction. http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-AtomicTransaction.pdf
3. WS-BusinessActivity. http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-BusinessActivity.pdf
4. BPEL4WS. http://www-128.ibm.com/developerworks/library/ws-bpel/
5. BTP. http://www.oasis-open.org/business-transaction/
6. WS-CAF. http://developers.sun.com/techtopics/webservices/wscaf/
7. Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci/
8. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl
9. SOAP Service Description Languages (SSDL). www.ssdl.org
10. F. Cabrera, G. Copeland, J. Johnson and D. Langworthy. Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity. http://msdn.microsoft.com/library/default.asp?url=/library/en- us/dnwebsrv/html/wsacoord.asp
11. Communicating Sequential Processes, C.A.R. Hoare, Prentice-Hall 1985.
12. A. Elmagarmid, editor. Database Transaction Models for advanced Applications. Morgan Kaufmann, 1992.
13. Kuo, D., Greenfield, P., Parastatidis, S., Webber, J. Rules SSDL Protocol Framework V1.0. http://ssdl.org/docs/v1.0/html/Rules
14. H. Garcia-Molina and K. Salem. Sagas. In Proceedings of the ACM SIGMOD, pages 249-259. ACM Press, 1987.
15. P. Greenfield, A. Fekete, J. Jang and D. Kuo. Compensation is Not Enough. In proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC) 2003.
16. Jim Gray, Andreas Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann 1992.
17. G. Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley.
18. R. Milner. Communication and Concurrency, Prentice-Hall International, Englewood Cliffs, 1989.
19. R. Milner. Communicating and mobile systems: the π-calculus. Cambridge University Press 1999.
20. J. Widom and S. Ceri. Active Database Systems: Trigger and Rules for Advanced Database Processing. Morgan Kaufmann, 1995.