Implementing Isolation for Service-Based Applications

Wei Chen[1], Alan Fekete[1], Paul Greenfield[2], and Julian Jang[3]

[1] School of Information Technologies, University of Sydney, NSW 2006 Australia
[2] CSIRO Mathematical and Information Sciences
Locked Bag 17, North Ryde NSW 1670 Australia
[3] CSIRO ICT Centre, Locked Bag 17, North Ryde NSW 1670 Australia
{weichen,fekete}@it.usyd.edu.au,
{paul.greenfield,julian.jang}@csiro.au

Abstract. Loosely-coupled distributed systems can be difficult to design and implement correctly, with time-of-check-to-time-of-use flaws arising from the lack of isolation being of particular concern. It is not feasible to use traditional distributed ACID transactions to solve such problems because the business activities being integrated are typically long-running and the interacting participants have incomplete mutual trust. 'Promises' were recently proposed as a solution to this problem. This paper discusses how promise-based isolation can be implemented when resources are described by predicates over properties, rather than being identified explicitly.

1 Introduction

Many distributed applications are based on long-running business processes that call each other back-and-forth in extended patterns of interaction. The designers of these systems have to correctly handle all of the different possible sequences of interaction, a task which can be hard for the normal ('happy') case, and much harder still when all possible error paths and all possible application states are considered.

One particular difficulty for application developers comes when services may fail if they are called while they are in an unsuitable internal state. For example, in an accommodation service, the operation used to book a hotel room will return an error if no appropriate room is available, and the calling application must include code to do something sensible on receiving such an 'operation unavailable in this state' error.

The traditional way of solving this problem is for such services to provide operations that clients can use to check that this internal state is appropriate before going on to call the state-sensitive operations. If this preliminary check succeeds the client can call subsequent operations that should not fail because the service is in an inappropriate state, and if it fails the client can take an early remedial/alternative processing path. For example, an accommodation service could have operations that report on room availability, allowing the client to check that a room was actually available before trying to book one. In an environment with concurrent threads or concurrent clients, the correctness of this 'check-then-use' programming style depends on the platform providing 'isolation' support to ensure that the relevant internal state does not change from being 'suitable' to 'unsuitable' between the check and the subsequent use.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 365–372, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Traditional 'ACID' transactional technologies implement isolation through locking, logging and two-phase commit [1] but these mechanisms are not appropriate for loosely-coupled, inter-organisational systems because they are based on unviable assumptions of trust and timeliness. As a result, the designers of these kinds of systems cannot assume that a state-sensitive operation will succeed just because they had earlier checked that it ought to succeed, and not considering this failure scenario, where a precondition value changes between time-of-check and time-of-use, can be a common source of programming errors.

We recently proposed an approach called 'Promises' [2, 3] which generalises coding patterns such as 'soft locks' and 'reservations' into a framework that brings the benefits of isolation to loosely-coupled systems. Promise-based applications can obtain 'promises' of later resource availability and then invoke state-dependent operations with confidence that they will succeed.

A promise is a time-limited commitment from a service that a requested resource will be available for later use. In this paper we go beyond our earlier work and explain how to implement promise-management for resources which are described implicitly and non-deterministically by their properties. An additional contribution of this paper is showing how the work of promise-management can be divided among several subsystems, separating for the handling of promises over one type of resource (e.g. hotel rooms) with its own syntax and semantics for promises, from the handling of promises for another type of resource (e.g. bank balances). We also describe a proof-of-concept implementation that includes these novel features and shows that acceptable performance can be obtained in promise-based systems.

2 Background and Related Work

As defined in our previous papers on Promises [2, 3], a 'promise' is a resource availability commitment given by a service to a client application. By accepting a promise request, a service guarantees that a client-specified set of conditions ('predicates') will be maintained over a set of resources for an agreed period of time.

Promises provide an isolation mechanism for loosely-coupled business processes. Client applications have to accumulate a set of promises that guarantee the availability of the resources required for the successful execution of their proposed actions. Then, the client application can send 'action' service requests to the corresponding application services to complete its business processes, knowing that these requests cannot fail as a result of conflicts over promised resources.

Promises are made over the availability of abstract resources and [2] defined three important ways that clients can view these resources. *Anonymous View*: Clients see a pool of identical and indistinguishable resource instances, such as the copies of a title in a bookstore. *Named View*: Clients see each resource instance as unique, with its own unique identifier that can be used in promise predicates. Specific airline seats (seat 24G on QF1 departing on 8/10/2009) are an example. *Property View*: Clients see abstract resources that are identified by their properties. For example, a hotel room may or may not have an ocean view, and may contain twin, double or king beds. These particular properties could be used in promise predicates that ask for any room with a view and a double bed.

The need for mechanisms that provide isolation across the different steps of an activity has attracted many researchers. Traditional ACID transactions can be used within a trust boundary, and can be implemented with a range of techniques including shared and exclusive multi-granularity locking [1]. For numeric values with increment and decrement operations, there are special high-performance mechanisms such as escrow locking [4]. A technique for locking predicates has been proposed in [5], but this requires determining the NP-hard solution to satisfiability of an arbitrary Boolean expression. We are not aware of any techniques in the database domain that correspond to our proposal for consistency checking with property-based resources. Rather than using locks in the DBMS, some applications implement isolation through application-level soft locks. Promises over property-based resources give the application more flexibility as soft locks can only be taken on named resources.

The needs of long-running applications, such as cooperative design tools, led to the study of extended transaction models. The ConTract model [6] specifies preconditions needed for operations to be successful. In ConTract there are defined language mechanisms to indicate these conditions, whereas Promises allows diversity in the way that properties for different types of resources are defined and supported by Promise Handlers. Another extended model is Sagas [7], where a business activity is treated as a sequence of separate steps with a compensator defined for each. Sagas provided all-or-nothing outcomes but do not give Isolation.

Several protocols have been published for managing activities in loosely-coupled distributed systems and B2Bi domains. Tentative Hold Protocol [8] does not itself ensure consistency of granted Holds and so does not prohibit several clients getting a Hold on a single resource at the same time. The reservation-coordination protocol [9] restricts the application to a two-phase structure, first obtaining reservations on a number of resources and then making use of those resources. In contrast, promise allows the application to have complex sequences of interactions, using some promises and then obtaining others. None of these protocols include consistency-checking algorithms for property-based resources which we describe for Promises.

Much other research has been focused on ensuring common all-or-nothing outcomes for a long-running activity spread over different sites, rather than isolation support. WS-BA is a standard for this; and the BTP protocol was proposed as a standard. Our ideas on achieving outcome consistency are described in [10].

3 Conceptual System Architecture

The conceptual architecture of our current (second-generation) prototype is shown in Fig. 1. The Promise Server is an intermediary along the path from the client to the service application and its role is to process promise-related elements found in messages sent from the client to the application server. These promise elements are passed to Promise Handlers, each of which deals with resources of a particular kind only, and is closely associated with a corresponding Resource Manager.

The Promise Server is implemented as a SOAP intermediary that intercepts messages sent from the client to application services. These messages may contain promise-related header elements and application service requests ('actions'). The Promise Server parses incoming messages, split them into promise parts and actions, and forwards the promise parts to the appropriate Promise Handlers and the actions to the Application Servers.

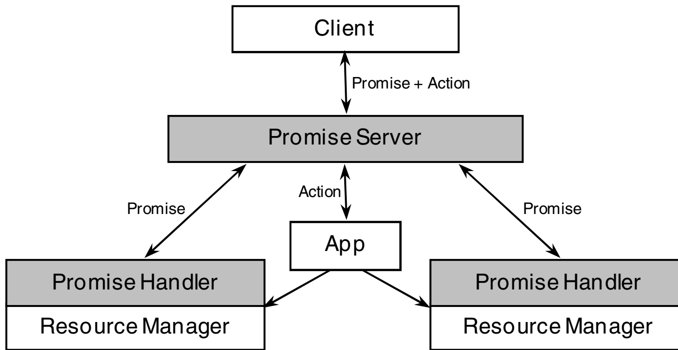

Fig. 1. Architecture of promise-based system

Promise operations have to support atomicity and consistency across internal promise state and resource allocations, even when errant or non-conforming application code allocate resources that could result in promises being violated. Our current prototype uses conventional ACID transactions as part of the implementation of this functionality. Transactions are started by the Promise Server and propagated to (multiple) Promise Handlers and Application Servers. Transactional isolation is used to ensure that concurrency problems cannot arise while promises and resources are being checked for consistency. Transactional atomicity is used to ensure that we can back out changes to promises or resource availability if promise requests cannot be granted or are violated. This use of ACID transactions is regarded as 'trusted' because client applications are excluded from the transaction scope.

Each class of resources supported by a Promise system has a corresponding Promise Handler that processes promise parts and maintains sets of promises over its resources. Promise Handlers are responsible for implementing promise operations, such as granting and releasing promises, and for consistency checking between promises and actual resource availability. Promise Handlers are resource-specific and so have close links to their related Resource Manager, including understanding the schema and semantics of the data used to define resource availability.

Promise Handlers maintain internally-consistent sets of promises that are also consistent with the resource availability state maintained by the Resource Manager. The code that is used to ensure this consistency has to have fast access to the availability state held within Resource Managers if the performance of promise-based systems is to be acceptable. Our previous papers suggested that Promise Handler functionality could be moved into Resource Managers, letting them manage both promised-based resource constraints as well as more conventional resource allocation state based on data values held in database tables. Our current prototype maintains the conceptual separation between Promise Handlers and Resource Managers, but links the two together closely through the use of a shared and unified data structure that describes all allocatable resources. This shared data structure is accessed and updated by both the Promise Handler and the Resource Manager.

4 Promises over Property-Based Resources

The actual work that has to be performed by a Promise Handler to make and keep promises depends on the nature of the resources involved. We have previously defined [2] three ways that resources can be viewed: anonymous view, named view and view via properties. The implementation issues for the first two views have been discussed in [3] and in this paper we focus on novel aspects in the implementation of promises over property-based resources.

4.1 Representing Resource Properties

Promises are made over resources that are modelled by one or more database tables held by Resource Managers. Columns of these resource tables provide the values for each property of a resource, including its current availability status. Promise predicates specify a set of properties and values that define needed resources in terms of these property columns. For example, an accommodation service could define information on hotel rooms, such as the number and size of beds, whether or not there is a sea-view, and which floor it is on. These properties can then be used in predicates to request that double-bed room with a sea view be kept available on a given date. The syntax used to specify these predicates is resource-specific as it has to reflect the semantics of the resource tables and their property columns.

Promise Handlers receive promise requests containing these predicates and turn them into searches for sets of satisfiable resources. If satisfiable resources are available, the new requests are checked to ensure that they are consistent with any already-granted promises before returning a 'promise granted' response to the caller.

4.2 Consistency Checking

As described in [3], checking the consistency of a set of promises is an essential step that takes place on almost every promise operation. The Promise Handler must ensure that the set of resources that are available are always sufficient to meet every promise that has been granted. This was easier for named and anonymous resources [3], but, where promises are based on predicates that implicitly identify a set of possible resources viewed via properties, it is not straightforward to see whether or not a group of competing promise requests based on different sets of properties can all be granted.

This problem can be resolved by using a bipartite graph matching [11] algorithm. A set of available resources form one partition of nodes, and promises form another. An edge links a resource node to a promise node whenever the promise can be satisfied by the resource (that is, the resource has the properties required by the predicate of the promise). The consistency check is the existence of a match in this bipartite graph which covers every promise, that is, each promise is satisfied by a resource, and each resource is only matched at most once.

The standard bipartite graph matching algorithm either finds a match or decides that there is no match. The algorithm is incremental, extending a partial match or else re-arranging the matches along an *augmenting path* of edges in the graph which are alternately edges which are used in the matching, and edges which are not used. We exploit this property for performance reasons by maintaining a match at all times, and incrementally updating it as new promises are made or resource availability changes.

4.3 Resource Suggestion and Resource Query Process

The applications discussed in [3] only dealt with named and anonymous resource views and were completely unaware of promises and promised resources. We have found that this simple approach is not possible with property-based views of resources because the correct choice of which resource to use in fulfilling an action depends on both the set of granted promises and availability data as recorded in database tables. Applications coded without knowledge of promises can only be aware of the database-resident availability state, and can mistakenly assign resources that are really needed to meet promises granted to other clients. This conflict will be picked up by the consistency check that follows the completion of the action. The problematic resource allocations (and other application actions) will then be undone to ensure the integrity of the already-granted promises, even if the application was executing under the protection of promises that guaranteed that suitable resources would be available. This is the very situation that promises were intended to prevent from happening.

To illustrate this problem, suppose a Promise Handler first receives a promise request for a room with sea-view. It finds that rooms 15 and 17 could be used to satisfy this request so the promise is granted. A promise request then arrives for a non-smoking room. This can be satisfied by room 15 and so this promise is granted as well. An action then arrives that is covered by the first promise, such as a request to book a room with sea-view. In the previous architecture of [3], the hotel booking service knows nothing about promises, so when its database queries on resource availability say that both rooms 15 and 17 are available and have sea-views, it can validly decide to use room 15 to meet the booking request. Unfortunately, this will result in the second promise being broken, and the completed action will be aborted by the post-execution consistency check, even though it was promised a suitable resource and such a resource is available. This would not have happened if the service had chosen room 17, and we need some way of conveying this hint to the application.

In our new architecture, application services find out about allocatable resources by sending a Query message to the Resource Manager responsible for the resource class. The Resource Manager then forwards this query on to the Promise Handlers responsible for the promises held over this resource class. These Promise Handlers then examine both their sets of promises and the available resources, and make a suggestion as to which resources can be safely used by the service. These suggestions are passed back to the Resource Manager and thence back to the application. In many cases, there will be multiple promise-consistent resources that could be used by the application and our design lets Promise Handlers suggest lists of allocatable resources, not merely one suitable resource. When the application service code receives the suggestion response, it chooses one resource from the suggested set, and uses it in completing the action.

5 Prototype Implementation and Performance

The second-generation prototype described in this section was built to explore issues arising from providing support for promises over property-based resource views. This new prototype is built on Windows Communication Foundation (WCF), a framework

provided by Microsoft for service-oriented applications. Microsoft SQL Server is used as the Resource Manager and the Microsoft Distributed Transaction Coordinator (MSDTC) is used as the coordinator for the two-phase commit protocol. The Promise Server is implemented as a routing service [12] that forwards client messages (or parts of them) to Promise Handlers and application services. This routing architecture is extensible, and additional resources and Promise Handlers can be added by simply adding new Promise Server configuration entries that specify the resource name and the endpoint of the corresponding Promise Handler.

The prototype maintains an in-memory data structure that represents the current bipartite graph of promises and all resources that can satisfy each promise. We also store a current matching within this graph. This data structure represents a unified view of allocatable resources and is (conceptually) shared by both a Promise Handler and its related Resource Manager. This data structure is contained within a Common Language Runtime (CLR) library hosted by the database engine and is built from the database tables holding resource availability state and granted promises whenever the system starts up. It is kept up-to-date by stored procedures and triggers that are invoked whenever resource availability or promise status is changed. Keeping a current matching allows incremental calculation of the consistency check whenever the graph changes through searching for an augmenting path.

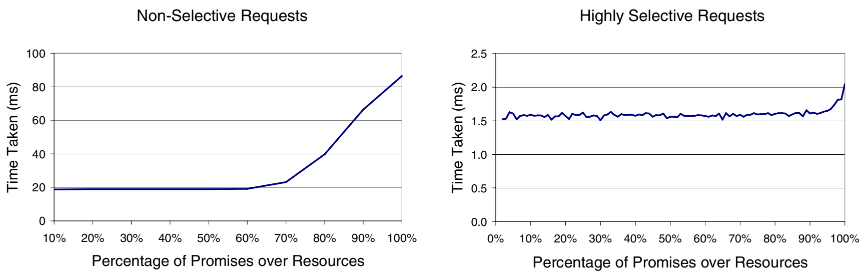

Fig. 2. Time taken to find path

Two application scenarios are used to investigate how the performance of promise-related operations changed as the number of satisfiable resources is scaled up. One scenario is *Non-selective Requests*, where each promise request can be satisfied by about 25% of all resources. The other is *Highly Selective Request*, where only 0.1% of resources can be used to satisfy any given promise request and these are distributed widely across all the available resources. The effects of these scaling characteristics can be seen in the time taken to process a new promise request (Fig. 2). The overhead caused by the consistency check is quite reasonable in both scenarios.

The time spent on allocating a resource and updating the allocatable resources graph is shown as 'Allocate + Update' in Fig. 3. It represents the whole cost of using a promised resource to fulfil an action. Compared with a simple SQL update, the overhead cost in this part of the Promises system is also quite acceptable.

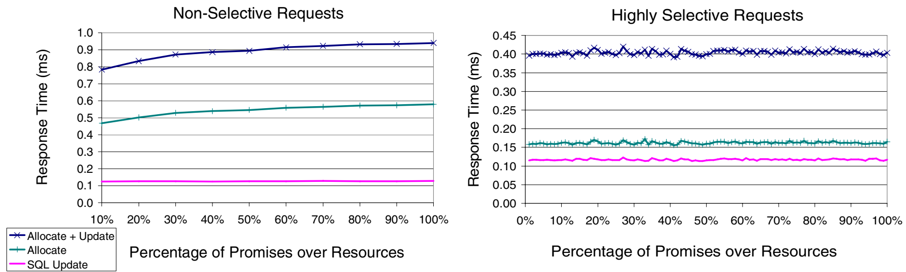

Fig. 3. Time taken to allocate & update

References

1. Gray, J., Reuter, A.: Transaction processing: concepts and techniques. Morgan Kaufmann, San Francisco (1993)
2. Greenfield, P., Fekete, A., Jang, J., Kuo, D., Nepal, S.: Isolation Support for Service-based Applications: A Position Paper. In: Proc. of CIDR, pp. 314–323 (2007)
3. Jang, J., Fekete, A., Greenfield, P.: Delivering Promises for Web Services Applications. In: Proc. of IEEE ICWS, pp. 599–606 (2007)
4. O'Neil, P.: The Escrow Transactional Methods. ACM TODS 11(4), 405–430 (1986)
5. Eswaran, K., Gray, J., Lorie, R., Traiger, I.: The Notions of Consistency and Predicate Locks in a Database System. Comm. ACM 19(11), 624–633 (1976)
6. Wachter, H., Reuter, A.: The ConTract Model. In: Elmagarmid, A. (ed.) Database Transaction Models for Advanced Applications, pp. 219–263 (1992)
7. Garcia-Molina, H., Salem, K.: Sagas. In: Proc. of ACM SIGMOD, pp. 249–259 (1987)
8. Srinivasan, K., Malu, P., Moakley, G.: Automatic Multibusiness Transactions. IEEE Internet Computing 7(3), 66–73 (2003)
9. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: A Reservation-Based Coordination Protocol for Web Services. In: Proc. of IEEE ICWS, pp. 49–56 (2005)
10. Greenfield, P., Kuo, D., Nepal, S., Fekete, A.: Consistency for Web Services Applications. In: Proc. of VLDB, pp. 1199–1203 (2005)
11. Kleinberg, J., Tardos, E.: Algorithm Design. Addison-Wesley, Reading (2006)
12. Bustamante, M.: Building a WCF Router. In Service Station, MSDN Magazine (April 2008), `http://msdn.microsoft.com/en-us/magazine`