ISOLATION SUPPORT FOR SERVICE-BASED APPLICATIONS
A POSITION PAPER
TECHNICAL REPORT 591

PAUL GREENFIELD¹, ALAN FEKETE¹,
JULIAN JANG², DEAN KUO³
AND SURYA NEPAL²

¹ SCHOOL OF INFORMATION TECHNOLOGIES, THE UNIVERSITY OF SYDNEY
² CSIRO ICT CENTRE
³ SCHOOL OF COMPUTER SCIENCE, UNIVERSITY OF MANCHESTER

AUGUST, 2006
This paper proposes an approach to providing the benefits of isolation in those service-oriented applications where it is not feasible to use the traditional locking techniques used to support ACID transactions. Our approach, called ‘Promises’, provides an uniform mechanism that clients can use to ensure that they can rely on the values of information resources which covers a wide range of implementation techniques on the service side, all allowing the client to check a condition and then later rely on that condition still holding.

1. INTRODUCTION

Web Services and service-oriented architectures are widely accepted as being the technologies that will be used to build the next generation of Internet-scale distributed applications. These applications are constructed by gluing together opaque and autonomous services, possibly supplied by business partners and third party service providers, to form loosely-coupled virtual applications. The services model is extremely simple but, unfortunately, this simplicity does not mean that service-based applications will prove to be easy to develop in practice or sufficiently reliable and robust.

Building robust large-scale stateful distributed systems is a long-standing and inherently hard problem. Some of the difficulties come as consequences of having to deal with the effects of concurrency and partial failures, and are made worse by the opaque and autonomous nature of services. Traditional distributed ACID transaction technologies provide an elegant and powerful solution to these problems, but depend on assumptions of trust and timeliness that no longer apply in the new loosely-coupled services-based world.

Our earlier work [3] on improving the robustness of service-based distributed applications focussed on the consistency problem: how to ensure that the set of autonomous services making up one of these applications always finish in consistent states despite failures, races and other such difficulties. Rather than attempting to provide the equivalent of traditional distributed transactions for the loosely-coupled
Web Services world, our approach instead was to develop tools, programming models and protocols for the detection and avoidance of consistency faults, at both design time and at run-time. The key to this work was establishing a relationship between internal service states, messages and application-level protocols. This insight let us transform the problem of ensuring consistent outcomes into a protocol problem that could be addressed using proven techniques from the world of protocol verification. We developed tools that could test whether the contracts defining the behaviour of two services were compatible and that their interactions would never lead to an inconsistent outcome. The same message-based definitions of correctness and consistency were also used as the basis for a protocol for dynamically checking for consistency failures at the termination of service-based applications, without requiring an overall coordinator or a global view of the entire application.

Or previous work addressed only the ‘atomicity’ part of the larger problem of simplifying the construction of robust and reliable service-based distributed applications. We could prove that the use of correctly designed contracts and the resulting application protocols could avoid inconsistent outcomes, but we still required the programmer to provide code to handle each message under every possible state. For example, the methodology of [3] requires a merchant service to have code for the situation where payment arrives for an accepted order but there is insufficient stock on hand. In the simpler world of ACID transactions, programmers could simply check stock levels when the order was accepted, and then rely on sufficient stock being available throughout the rest of the order process, regardless of any concurrent orders or other activities. The challenge we faced was providing degree of isolation in a services-based world where autonomy and lack of trust meant that traditional lock-based isolation mechanisms could not be used. Our approach to this problem was to first identify a range of real-world examples where the lack of isolation was actually a problem, and then to understand and generalise the solutions to these problems adopted in traditional business processes. The result of this work is a general pattern and protocol called ‘Promises’.

2. PROMISES

A Promise is an agreement between a client application (a ‘promise client’) and a resource manager (a ‘promise maker’). By accepting a promise request, a resource manager guarantees that some set of conditions (‘predicates’) will be maintained over a set of resources for a specified period.

Client applications can now determine what conditions they need to have hold in order to complete successfully, express these as a precise set of predicates and send them to the relevant resource manager as a promise request. Once a promise request is granted, the client application is isolated from the effects of concurrent activities with respect to the resources protected by its promises. For example, the merchant process we mentioned above can now ask the stock level manager for a promise that the goods required to meet an order still be available when needed later in the order handling process.

Traditional lock-based isolation can be seen as a very strong and monolithic form of promise; in this case the resource manager is promising that no other concurrent process can alter or even examine the state of a protected resource for the duration of a transaction. Our proposed promise-based isolation mechanism is weaker, but just as effective, as predicates can specify the exact requirements of the client application and
other concurrent clients can be granted promises covering the same resources as long as
they don’t conflict with already granted promises.

Promises are both a pattern and a protocol that supports this pattern. The pattern is
simply that client application determines the constraint they need to have hold over a set
of resources and express these as predicates that are sent within promise requests to a
resource manager. The resource manager will determine if it can grant the promise and
reply with either an acceptance or rejection response. Once a promise has been granted,
the client application can continue and make changes to the resources protected by its
promises, with the guarantee that they will be allowed if they are within the constraints
implied by its promises. Client applications then release their promises by sending
promise release messages to their resource managers. Promise release requests can be
combined with application request messages. In this case the promise release and the
application request could form an atomic unit, and the promise would then only be
released if the associated action succeeded.

The Promise model places no limitations on the nature or form of predicates, or on the
way that resource managers can implement predicates to guarantee that they hold
despite concurrent updates to the same resources. This flexibility means that resource
managers are free to implement whatever form of constraint checking or isolation is
best for the type of resource being protected.

Some forms of Promises can be implemented using the common business practice
sometimes called ‘soft locks’. This approach uses a field in the database record to show
whether an item has been allocated or reserved for a client. The record is not locked
against access once the allocation has been made; instead application code reads this
field, and ignores any record that has been allocated elsewhere. Our Promise pattern
accommodates this way of implementing isolation, but it is more general, separating the
model and its supporting protocol from any specific implementation. The flexibility that
results lets us support techniques where the actual allocation of a particular resource to
a client is delayed long after the promise is made. Section 5 discusses a range of
implementation alternatives

The motivation behind the development of this Promise approach to isolation was to
provide application programmers with something akin to the simplicity that comes from
the traditional ACID transaction model. By implementing weaker but effective
constraints over shared resources, we want to let programmers establish those pre-
conditions needed to ensure their application can complete successfully, with the
guarantee that concurrent activities cannot violate these promises. Promise violation is
still possible (an industrial accident might damage previously-promised stock, or a third
party may default on a promise they have made) but now it can be treated as a serious
exception, one rare enough to be classified as disaster that can be left to human
intervention to resolve. This is very far from the situation without isolation where the
effects of concurrency are common enough that they need to be included in normal
processing paths.

Our proposed Promise protocol fits very naturally into the SOAP and Web Services
model. All of our promise protocol messages can be transferred as elements in SOAP
message headers. The fit between the Promise protocol and SOAP is discussed more
fully in Section 6.

We are not the first to propose transaction-like models based on conditions that must be
preserved; Section 8 points to previous work. Our key innovations lie in a deep analysis
of the variety of resources and conditions, in considering how to atomically combine several related aspects of managing a single promise, and in integrating these ideas into the services-oriented message exchange framework.

3. RESOURCES AND PREDICATES

This section discusses the nature of resources and the way that this defines the types of predicates that can be used in promises over them. We describe three ways of viewing resources in detail and explore the relationship between these views and predicates. We also give examples that show how applications can use these different ways of viewing resources to obtain just the degree of isolation they need.

Promises are a general approach to the problem of providing isolation support for distributed applications. Client applications can specify predicates that represent conditions over resources and resource managers that accept these promises guarantee that these conditions will hold true until released by the client. Predicates are simply Boolean expressions over resources. Our model imposes no restrictions on the form these expressions can take, and their ideal form will normally depend on the nature of the resources involved and the way we want to view them at the time.

Predicates are expressions over resources but their form and structure depend on the way we regard a resource. Different applications may want to treat the same physical resource, such as a particular airline seat or an individual pink widget, in different ways, and will want to use different types of predicates to get the required level of isolation from other applications that might be using the same or related resources.

In this section we discuss three different ways of viewing resources: anonymous access, named access, and access via properties. These abstractions were derived from a study of different isolation mechanisms commonly used in existing business practices. To understand the power of our approach, it is important to consider the different ways of viewing resources and the way that their characteristics influence the sort of predicates that clients want to rely on to achieve the isolation they require to operate correctly.

3.1 Anonymous Access

Anonymous resources refer to the ones which are not distinguished from one to another. Resources in the same pool have the exactly same qualities and value; therefore it is not important to the client which item from a pool it uses. Anonymous resources are usually reflected in data that records ‘quantity on hand’ for each resource pool. For example, for most goods, a retailer carries multiple instances which are not distinguished. Barnes and Noble have many copies of each book title in stock, and a client who wishes to buy the book doesn’t care which copy they are given. Thus the book title represents a resource pool, consisting of many copies.

Financial applications, such as banking, use anonymous resources all the time. For example, if a promise is made that the customer will be able to withdraw $500, the bank is not obliged to have five specific $100 bills set aside.

3.2 Named Access

Named resources have identifiers: numbers or similar distinguishing names that allow clients to refer to individual items uniquely. The clients can check the availability of a resource based on its identifier, and they can later make use of that resource. For example, each item sold on eBay is identifiable by item number, and a worker’s laptop computer has a serial number.
Sometimes, a group of similar named resources can be treated as if they were anonymous in some situations. For example, each seat on a flight has a unique name (e.g. seat 24G), but many clients treat all economy seats as equivalent, and want to rely not on the availability of seat 24G, but just on availability of some economy class seat.

3.3 Access via Properties

In many cases, a service is managing a collection of distinguishable resources each of which has a variety of properties, and different clients might care about different properties. For example, requests for hotel rooms may refer to the size and type of beds, whether or not smoking has occurred in the room, whether or not there is a view, which floor it is on, etc. Thus room 524 may be suitable to fill a request for a room with a view, or a request for a 5th floor room, but it can’t be allocated to both requests at once. This type of resource is common in the travel industry; as well as hotel rooms, flights and entertainment are usually requested based on a wide range of properties.

Users may regard some properties as essential and others as desirable but not required. The interplay between essential and desirable properties when obtaining a promise may be complicated. The promise requestor and the promise maker can negotiate to find a promise that is both satisfiable and maximally desirable. For example, the client may initially request a nonsmoking room with a view and twin beds, and eventually accept a promise for a room with twin beds.

One interesting aspect of predicates over this type of resource view is the possibility that certain properties can be treated as ordered in acceptability, with it being understood that a promise can be satisfied by a resource that doesn’t actually meet the precise stated property, but a better one. For example, a customer who holds a promise for an economy class airline seat will not complain if, when they fly, they are upgraded to business class.

4. ATOMICITY AND PROMISES

In this section we identify three important requirements for the atomicity of promise-pattern activities. While the autonomy of service-providers means that there is no way to demand atomicity across long duration business processes, it is feasible to require that specific atomicity guarantees apply during the handling of a single Promise message. These requirements are:

Request guarantees on several predicates at once. While it may be common to seek a single guarantee such as ‘ensure that at least 5 widgets are available when I decide to buy them’, sometimes a client will want to ensure that several different properties (perhaps involving several resources) will all be true at later points. The classic example is from travel planning, where a client may want a promise that a flight and a rental car and a hotel room will all be available. Unless the service can promise all these together, the client has no use for any of the promises separately.

Perform an action which depends on, but destroys, a previously promised condition, together with releasing the promise. Making a promise forces the service to reject some later operations which would violate the promise; for example if an art gallery service has promised a client that a particular painting will be available, then other requests to buy that painting must be rejected. However, when the client who obtained the promise wants to buy, the purchase must be allowed; and when the purchase occurs the gallery
service is released from the promise (the client can’t rely on the painting still being there after they themselves bought it!)

Modify the predicate whose preservation is promised, by obtaining a new promise and releasing a previous one atomically. An important use-case is for the client to request changes to the promise they have received. The change can be to upgrade the promise, or to weaken it. For example, if a client has obtained a promise that an account will have at least $100, they may find that their anticipated later withdrawal has changed to $200 (a stronger promise is needed) or to $50 (a weaker promise). In either case, it would be too restrictive to force the service to provide the new guarantee as well as the previous one, nor would the client want to release the previous one until the new one was obtained. Thus obtaining a new promise should be atomic with releasing the old one, and the previous one should be retained if the service can’t guarantee the modified request.

5. IMPLEMENTATION TECHNIQUES

The Promise Pattern we propose is a style of interaction, in which a client can request another service to guarantee that a predicate will remain true (for a limited time into the future). The value of our proposal depends on the existence of mechanisms by which the provider can keep its promises. In this section we describe several well-known implementation techniques. Not all are applicable for every sort of promise. However for each resource type there is at least one (and sometimes more) of the following techniques that the resource manager can choose between, without the client needing to know which is being used. Thus the Promise pattern abstracts away the choices made autonomously by the promise maker.

• Resource Pool: In managing anonymous interchangeable physical resources, it is common to keep the available examples of each variety of objects in a pool, and move them to a separate ‘allocated’ pool once they are needed to meet a promise. For example, when we promise that we can supply 10 widgets, we remove 10 widgets from the pool of available widgets and place them in the allocated pool. The digital equivalent is managed by keeping a count of available and allocated items in the record corresponding to each type of resource. This technique is similar to escrow locking [6].

• Allocated Tags: When resources are named, and a promise refers to a particular resource, we can keep a status field in the data about the resource. The value of this field is the client to whom the item has been promised; a null value indicates that the item is still available.

• Satisfiability Check: The service keeps a record of all the promises it is currently committed to, and whenever an operation is received, the service checks the state which would result against every promise that might be impacted (this might be done by speculatively executing the request, and rolling it back if necessary; alternatively the service can use knowledge of the request’s semantics to calculate its effects). If any promise would become invalid, then the request is rejected. This mechanism resembles the theory of predicate locks [1].

• Tentative allocation: This is a hybrid mechanism, where requests for resources satisfying properties are met by marking matching resource items as ‘allocated’, and also remembering the specific predicate requested; if a later request is not satisfiable from unallocated items, the system considers rearranging the allocations to meet all
previous promises as well as the new request. For example, a request for a hotel room with a view may lead to allocating room 512 which has a view; when a later request is made to promise a 5th floor room, the system may reallocate 512 to the new request as long as a different room with a view can be provided for the earlier request.

- Delegate: Promises are made relying on the promises of third parties. For example, a purchase order can be accepted by the merchant if it has received a promise from the distributor that a backorder will be fulfilled on time. In this scenario, the promise ‘sufficient stock’ is delegated from the merchant to the merchant’s supplier.

6. CONCRETE SYNTAX

This section presents the structure and content of the protocol elements that could be used in a SOAP-based implementation of the Promise Pattern. Clients and resource managers exchange promise-related information using <promise> and <environment> message header elements. <Promise> elements are used in the creation and release of promises. <Environment> elements are used to establish a promise context for the SOAP requests carried in the associated message body.

The <promise> element can have zero or more <promise-request> elements; each representing one request for the recipient to make a promise that will guarantee the included predicates for a certain period of time. A <promise> element can also include zero or more <promise-response> elements which are used to return outcomes from previous requests that flowed in the reverse direction. Each participating service can act as both client and promise-maker, so a single <promise> element can include both <promise-request> and <promise-response> elements.

A <promise-request> defines:

- A request identifier that can uniquely identify each promise-request. This request identifier is used to correlate promise-requests and promise-responses.
- A set of predicates that specify the conditions on which the client will rely in a later interaction and that the promise-maker must maintain.
- A set of resources that specify the subjects of the promise.
- A promise duration that indicates how long the promise must be kept.
- An optional set of promise identifiers that refer to existing promises that can be released if this new promise request is granted.

Each promise-request must be treated atomically. All of the predicates over the specified resources must be promised or the entire promise must be rejected. A promise request may hand back previous promises in exchange for a new promise, and if this new promise cannot be granted, the existing promises must continue to hold.

Promise makers send promise responses back to promise requestors to inform them whether their promise requests have been accepted or rejected. The elements of a <promise response> are:

- A promise identifier that the promise maker uses to uniquely identify this promise.
- A promise result that says whether a promise request is accepted or rejected. Promise responses could also return other results, such as ‘pending’ or
accepted with the condition XX’ but these possibilities have still to be investigated.

- A promise correlation which is the request identifier of the earlier promise request

Successful promise requests establish promise environments. Application requests can specify that they must be executed relying on a specific promise environment, with the set of resource guarantees defined by its promises, by including an `<environment>` element in its header. An `<environment>` must define:

- A set of promise identifiers that define which promises will be important to the performance the execution of the request.
- A corresponding set of promise release options that indicate whether the associated promise should be released after the request has completed.

We note that each message may contain any subset of the different elements relating to promises, and these may be related to the message body or unrelated. For example, we allow an application message from A to B to contain a related request for B to make a promise, and it can also carry a piggybacked response reporting on the outcome of a request that B had sent to A.

7. PROMISES AND ISOLATION

The key contribution of the Promise Pattern is that it allows a client to request an operation from a service, with confidence that the operation will succeed (except for very rare catastrophic situations that might need human intervention). Programmers are relieved of the need to consider frequent but unwelcome situations where concurrent activity has changed the truth of needed conditions after they were checked.

We will illustrate how applications can use promises to achieve the precise degree of isolation they require through two examples based on the merchant example used earlier. Both of these examples make use of the Promise Pattern but differ in the resources involved, the way they view them and the predicates they use.

The first example shows how the ordering process can check for the availability of goods using a promise and then be guaranteed that these goods will continue to be available for purchase, regardless of any concurrent activities, until the order is completed or abandoned. In this example, the customer is trying to order 5 pink widgets. As our customer doesn’t care exactly which 5 of the many identical pink widgets in stock they will receive as a result of this order, we will use the anonymous access model defined in Section 3.1 for this example.

<table>
<thead>
<tr>
<th>Order process</th>
<th>Resource manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine we need 5 pink widgets to be in stock</td>
<td>Check stock levels of pink widgets and…</td>
</tr>
<tr>
<td>Send promise request that (quantity of ‘pink widgets’ &gt;= 5)</td>
<td>Accept promise if &gt;=5 currently available</td>
</tr>
<tr>
<td></td>
<td>Record promise as predicate over stock levels, guaranteeing that at least 5 units will always be available. This predicate will checked before any further promises or purchases requests are accepted.</td>
</tr>
<tr>
<td></td>
<td>Send 'accept' &lt;promise response&gt;</td>
</tr>
<tr>
<td></td>
<td>Reject promise request if &lt;5 units available</td>
</tr>
<tr>
<td></td>
<td>Send 'reject' &lt;promise response&gt;</td>
</tr>
</tbody>
</table>
If promise rejected  
   Terminate order process saying goods unavailable  
If promise accepted…  
   continue processing order (payment, shippers)

Send ‘purchase stock’ request to resource manager  
and release promise to keep stock level $\geq 5$

**Release 5 pink widgets for delivery**  
**Reduce stock-on-hand for pink widgets by 5**  
**Remove this promise from the set of predicates over the pink widget stock level**

The second example is more complex and illustrates the flexibility of promise predicates. In this example, our merchant offers ‘next day’ shipping to its customers for a fixed additional cost on all orders. The order process asks the shipping component for a promise of next day delivery, with the predicate making no assumptions about how this promise will be implemented or needing any information about the structure of the shipping component and its internal states. The shipping component could implement the promise by obtaining soft-locks on warehouse and shipping capacity, but other implementations are possible. For example, the merchant may have a number of shipping alternatives available, each with different capacity and costs, and the actual choice of which shipper to use can be deferred to reduce costs and optimise utilisation. This flexibility is not visible to the order process or the customer, all that they need to know is that the shipping component has promised next-day delivery and guarantees that this will occur.

8. RELATED WORK

One of our inspirations was the early ConTract work of Wachter and Reuter [7]. This introduced the importance of expressing preconditions (‘entry invariants’) needed to allow actions within a workflow to execute successfully. This paper identified several different styles of ensuring that these preconditions still hold at the time when applications rely on them later in an execution. Among the styles proposed were using semantic locks to preserve conditions and notifying the client when a checked condition changes. Our work extends the semantic lock ideas of ConTract to the services world with its interacting autonomous participants. Our consideration of atomically combining steps is also new. We provide a richer analysis of the variety of resource and predicate types, and of the ways to ensure that predicates remain true over an extended period. We also support a variety of possible implementation mechanisms, each tailored to the needs of specific ways of viewing and accessing resources.

In previous work [5] one of us developed a transaction model for spatial data which was based on explicit constraints that could be set and unset to limit concurrent modification of properties of the data. Our current paper extends this to a world of autonomous services; as well we now offer the analysis of predicate types, and a better mechanism to structure the operations by providing atomicity between aspects of a single step of the promise exchange.

Recently Dieter Gawlick and other members of the Grid Computing community have suggested an ‘Option’ protocol [2] for reserving access to resources. This has similarities to our Promises but our work deals with a wider class of conditions including those on anonymous resources and resources with properties, and supports a wider choice of implementation mechanisms. Also, our use of atomicity allows us to
unify concepts such as securing, modifying, confirming, and dropping which are represented as separate message types in [2].

The idea of an organisation making a promise about future performance or behaviour is quite common in bricks-and-mortar businesses, and most of the implementation mechanisms we considered have long precedents in business practice. For digital data, many implementation techniques have been proposed which offer the effect of promise keeping. Conventional database locking provides the semantic effect of ensuring that data is not altered between the time a condition is checked and the time it is needed, despite any concurrent activities but the locking mechanism assumes a particular environment, where activities run very quickly and remain within a trusted boundary. These assumptions are inflexible and not suited for data under high contention or for today’s service-based applications. Alternative mechanisms have been developed within database engines for allowing higher concurrency based on knowledge of the semantics of the data. For example, escrow locking [6] deals with numeric data under operations that add or subtract, by recording high and low limits for the possible values, while granular locks and predicate locking have been proposed as a means of preventing phantoms[1]. Our Promises pattern unifies and abstracts over these mechanisms, providing a common way for clients to work without knowledge of the implementation technique used inside a service that can maintain some property between the time it is checked and a later time when the client relies on the property.

9. CONCLUSION

In this paper we propose a unified approach to describing the interactions between a client and a service where the client can make sure that some condition over resources will hold at a later time, despite concurrent activities that occur between the check and the use of the condition. We have analysed the variety of resource types and conditions on those types, identifying an important distinction between resources which are accessed anonymously (where the key property is just whether a given amount or volume is available), resources which are accessed by name, and a wider class where access is based on values for some subset of a collection of properties. We have identified important cases where several promise-related activities need to be combined into an atomic unit in order to support valuable use-cases such as upgrading or weakening a previously obtained promise.

In future work, we will implement support for Promise interactions in several service-provision frameworks, including our own GAT engine [4] and also some commercial approaches. This will involve developing detailed implementations for checking predicates against resources, as discussed in Section 5; as well we will provide simple heuristics to choose an appropriate implementation technique for each class of resources. We also will integrate the processing of promise with other frameworks for service-oriented messaging, including the transaction support found in standards like WS-BusinessActivity.

10. REFERENCES


