Robust Web Services via Interaction Contracts

David Lomet

Microsoft Research
lomet@microsoft.com

Abstract. Web services represent the latest effort of the information technology industry to provide a framework for cross enterprise automation. One principal characteristic of this framework is information hiding, where only the message protocol is visible to other services or client software. This environment makes it difficult to provide robust behavior for applications. Traditional transaction processing uses distributed transactions. But these involve inter-site dependencies that enterprises are likely to resist so as to preserve site autonomy. We propose a web services interaction contract (WSIC), a unilateral pledge by a web service. A WSIC avoids dependencies while enabling, but not requiring, an application to be written so that it can provide exactly once execution semantics, even in the presence of failures. And exactly once semantics is essential whenever commercial interactions involve money.

1 Introduction

1.1 Using the Web for Commerce

The importance of the web for commerce only continues to grow. But that growth has been a bit "cranky". Growth is impeded by the barriers of legacy systems, differing protocols, the lack of robustness, and a lack of an integrative framework. But things are changing, as industry is well aware of these difficulties. The current "horse" being ridden is that of "web services", which offers the start of a compelling story about how to break down these barriers and produce the flourishing of commerce on the web.

So, abstractly, what are web services? And why are web services the way to go? The web services effort places special emphasis on "information hiding", an idea that began with David Parnas over 30 years ago [10]. The idea is to minimize what applications need to know in order to interact meaningfully with the service. Web services exploit message protocols, using a common syntactic framework (XML with SOAP) but are otherwise opaque in terms of how they are implemented behind their service front. This makes it possible for diverse systems to interact, by hiding the "ugly details" behind the service interface.

There is a lot to the web services story (UDDI [9], WSDL [11], etc.) but the focus here will be on the robustness of web services and the applications that use them. Currently, web applications either are not very robust, or need hand crafting to be so. But for pervasive use of web services, we need to ensure both robustness and a standard, non-intrusive way of making it possible.

1.2 Making Web Services Robust

There is a long history of efforts to make distributed applications robust [6]. The goal of most of these efforts has been to reduce the work lost in a long running application due to a system crash. Commercial systems need to ensure that no committed work is lost. Most commercial efforts evolved in the transaction processing community, with TP monitors and distributed transactions [1, 7]. And such transaction processing efforts have been quite successful in the context of relatively closed systems. There are efforts to extend this style of support to the web services world (e.g. WS- Coordination [12]). However, TP has had much less impact on highly distributed, largely autonomous, and heterogeneous commercial systems.

Classic transaction processing relies heavily on distributed transactions. But even in the classic TP world, distributed transactions are not usually very "distributed". They are used mostly for partitioned databases and for stateless applications interacting with queue managers. It is rare for this technology to be used across enterprise boundaries, and is not widely used even across organizational boundaries within a single enterprise. One can hypothesize a number of reasons for this. Protocols are too tied to a particular implementation base. Or these protocols require too much compromise of autonomy. For example, who coordinates a transaction and who will block if there is a failure at the wrong time? But without distributed transactions, there are no TP-based solutions. The situation then reverts either to non-robust applications and systems or to roll-your-own special case solutions.

1.3 Interaction Contracts to the Rescue?

I want to make a case for "interaction contracts" [5] as the paradigm for robust web services. Interaction contracts enable applications to be persistent across system crashes, without distributed transactions. They enable applications to interact with "transactional services" with the application neither participating in the transaction nor perhaps even being aware that there is a transaction involved. Thus, atomicity can be encapsulated within an "opaque" web service, and supported by transactions whose distribution can be limited to the traditional TP domain, and whose participants could be databases, queue managers, etc.

It is important to emphasize here that atomicity within web services is still highly desirable, perhaps essential. One wants an e-commerce sale to be all or nothing: no money—no product; money—product. The in-between states are unacceptable. But whether this atomicity is provided by a single local transaction, a distributed transaction, a workflow system, or some other technology, should be immaterial at the level of the web service.

If the applications using web services are not participating in transactions, then how are they made robust? The answer here is that interaction contracts capture precisely what needs to be done for applications to be persistent, surviving system failures. These applications live outside of any transaction, and because they are persistent, they can robustly provide program logic for dealing with failures, which might have resulted from one or more transaction aborts or system crashes. This is important, as traditional TP with stateless applications had no convenient place to put program logic dealing with transaction failures.

There is more to robustness than persistence (the ability for an application to survive system crashes). Traditional TP can provide not just applications that survive system failures and have certain atomicity properties. It also provides solutions for other robustness attributes: availability, scalability, and load balancing. Fortunately, these attributes can also be provided with interaction contracts and persistent applications. We have discussed these attributes in papers describing our prototype Phoenix/App system [2, 3] that provides transparent support for persistent .NET applications [8]. So I will not discuss these here.

2 Interaction Contracts

2.1 Overview

Most of what is described in this section is based on recovery guarantees [4, 5] that provide exactly once execution. Interaction contracts are obligations for each party involved in sending or receiving a message. The intent of the obligations is to ensure that both sides of the interaction can agree, even in the presence of a system crash of one or both of the parties, as to what the interaction consisted of and whether it occurred. Each party to the contract guarantees that enough information is stably recorded so that it can continue execution after a system crash, with agreement as to whether the interaction occurred and the message that was exchanged.

Each element of a distributed system is characterized with a component type.

1. **Persistent Component (Pcom):** a component whose state is guaranteed to survive system crashes, the execution result being as if the crash did not occur;
2. **External Component (Xcom):** a component that is outside of the control of our infrastructure and for which we cannot make a guarantee;
3. **Transactional Component (Tcom):** a component that executes transactions, and that guarantees that the effects of committed transactions will persist and the effects of uncommitted transactions will be erased.

Between persistent components and each component type, there is a particular flavor of interaction contract that needs to be honored for persistence and exactly once execution to be assured. *But first—an admission.* No system can guarantee failure masking under all circumstances. For example, in a system interaction with a human user, if the system crashes before all input from the user is stored stably (usually by logging), then the user will need to re-enter the input. Despite this, the "interaction" can be "exactly once". And we can minimize the window of vulnerability, and that is what an external interaction contract does. For the other interactions, we "shut" the window.

As there are three component types, we also have three flavors of interaction contracts.

1. **Committed Interaction Contract (CIC):** between persistent components;
2. **External Interaction Contract (XIC)**: between external component and persistent component;
3. **Transaction Interaction Contract (TIC)**: between persistent component and transactional component.

2.2 Committed Interaction Contract

I'll describe a committed interaction contract here and refer you to [5] for external interaction contracts. Transaction interaction contracts, which underlie our web services contract, are described in the next section.

A **committed interaction contract** between two Pcoms consists of the following obligations:

Sender Obligations:

- **S1: Persistent State**: The sender promises that its state at the time of the message or later is persistent.
- **S2: Unique Persistent Message**
  - **S2a:** Sender promises that each message is unique and that it will send message repeatedly until receiver releases it from this obligation (R2a)
  - **S2b:** Sender promises to resend the message upon explicit receiver request until receiver releases it from this obligation (R2b).

Sender obligations ensure that an interaction is **recoverable,** i.e. it is guaranteed to occur, though not with the receiver guaranteed to be in exactly the same state.

Receiver Obligations:

- **R1: Duplicate Message Elimination:** Receiver promises to eliminate duplicate messages (which sender may send to satisfy S2a).
- **R2: Persistent State**
  - **R2a:** Receiver promises that before releasing sender obligation S2a, its state at the time of message receive or later is persistent. The interaction is now **stable**, i.e. the receiver knows that the message was sent.
  - **R2b:** Receiver promises that before releasing the sender from obligation S2b, its state at the time of the message receive or later is persistent without the need to request the message from the sender. The interaction is now **installed,** i.e., the receiver knows that the message was sent, and its contents.

CIC's ensure that the states of Pcoms not only persist but that all components "agree" on which messages have been sent and what the progress is in a multi-component distributed system. S2 and R1 are essentially the reliable messaging protocol for assuring exactly once message delivery, coupled with message persistence, while S1 and R2 provide persistent state for sender and receiver. Reliable messaging is not sufficient by itself to provide robust applications. Also, the CIC describes the requirements more abstractly than a protocol in order to maximize optimization opportunities. For example, it is not necessary to log an output message to persist it if the message can be recreated by replaying the application from an earlier state.

In our Phoenix/App prototype [2, 3], our infrastructure intercepts messages between Pcoms. It adds information to the messages to ensure uniqueness, resends messages, and maintains tables to eliminate duplicates; it logs messages as needed to ensure message durability. Message durability permits Phoenix to replay components from the log, so as to guarantee state persistence (see, e.g. [3]). Because the interception is transparent, the application executing as a Pcom needs no special provisions in order to be persistent, though some limitations exist on what Pcoms can do.

2.3 Transaction Interaction Contract

A transaction interaction contract[1] is a model for how we deal with web services. Thus we describe it in detail. The TIC explains how a Pcom reliably interacts with, e.g., transactional database systems- and what a transactional resource manager (e.g. DBMS) needs to do, not all of which is normally provided, so as to ensure exactly once execution.

When a Pcom interacts with a Tcom, the interactions are within the context of a transaction. These interactions are of two forms:

- **Explicit** transactions with a "begin transaction" message followed by a number of interactions within the transaction. A final message (for a successful transaction) is the "commit request", where the Pcom asks the Tcom to commit the work that it has done on behalf of the Pcom.
- **Implicit** transactions, where a single message begins the transaction and simultaneously requests the commit of the work. This message functions as the "commit request" message.

For explicit transactions, the messages preceding the commit request message do not require any guarantees. Should either Pcom or Tcom crash, the other party knows how to "forget" the associated work. The components do not get confused between a transaction that started before a crash, and another that may start after the crash. The later is a new transaction. Further, at any point before the "commit request" message, either party can abandon the work and declare the transaction aborted. And it can do so unilaterally and without notice. The only Tcom obligation is the ability to truthfully reply to subsequent Pcom messages with an "I don't know what you are talking about" message (though more informative messages are not precluded).

At the end of a transaction that we want to commit, we re-enter the world where contracts are required. Interactions between Pcom and Tcom follow the request/reply paradigm. For this reason, we can express a TIC in terms that include both the request message and the reply message. We now describe the obligations of Pcom and Tcom for a transaction interaction contract initiated by the Pcom's "commit request" message.

Persistent Component (Pcom) Obligations:

PS1: Persistent Reply-Expected State. The Pcom's state as of the ***time at which the reply to the commit request is expected***, or later, must persist without having to contact the Tcom to repeat its earlier sent messages.

- The persistent state guarantee thus includes the installation of all earlier Tcom replies within the same transaction, e.g., SQL results, return codes.
- Persistence by the Pcom of its reply-expected state means that the Tcom, rather than repeatedly sending its reply (under TS1), need send it only once. The Pcom explicitly requests the reply message, should it not receive it, by resending its commit request message.

[1] The form of transaction interaction contract presented here is the more complete specification given in the TOIT paper [4].

PS2: Unique Persistent Commit Request Message: The Pcom's commit request message must persist and be resent, driven by timeouts, until the Pcom receives the Tcom's reply message.

PR1: Duplicate Message Elimination: The Pcom promises to eliminate duplicate reply messages to its commit request message (which the Tcom may send as a result of Tcom receiving multiple duplicate commit request messages because of PS2).

PR2: Persistent Reply Installed State: The Pcom promises that, before releasing Tcom from its obligation under TS1, its state at the time of the Tcom commit reply message receive or later is persistent without the need to request the reply message again from the Tcom.

Transactional Component (Tcom) Obligations:

TR1: Duplicate Elimination: Tcom promises to eliminate duplicate commit request messages (which Pcom may send to satisfy PS2). It treats duplicate copies of the message as requests to resend the reply message.

TR2: Atomic, Isolated, and Persistent State Transition: The Tcom promises that before releasing Pcom from its obligations under PS2 by sending a reply message, that it has proceeded to one of two possible states, either committing or aborting the transaction (or not executing it at all, equivalent to aborting), and that the resulting state is persistent.

TS1: Unique Persistent (Faithful) Reply Message: Once the transaction terminates, the Tcom replies acknowledging the commit request, and guarantees persistence of this reply until released from this guarantee by the Pcom. The Tcom promises to resend the message upon explicit Pcom request, as indicated in TR1 above. The Tcom reply message identifies the transaction named in the commit request message and faithfully reports whether it has committed or aborted.

A TIC has the guarantees associated with reliable message delivery both for the commit request message (PS2 and TR1) and the reply message (PR2 and TS1). As with the CIC, these guarantees also include message persistent. In addition, in the case of a commit, both Pcom (PS1) and Tcom (TR2) guarantee state persistence as well. As with CIC, we stated TIC requirements abstractly.

2.4 Comparison with Transaction Processing

Unlike persistent components, traditional transaction processing applications are stateless. That is, there is no meaningful state outside of a transaction except what is stored explicitly in a queue or database. And, in particular, there is no active execution state. Each application "step" is a transaction; the step usually is associated with the processing of a single message.

A typical step involves reading an initial state from a queue, doing some processing, updating a database, digesting a message, and writing a queue (perhaps a different queue), and committing a distributed transaction involving queues, databases, and message participants. This requires 2PC unless read/write queues are supported by the same database that is also involved. When dealing with distributed TP, queues are frequently different resource managers, and so is the database. So typically, there are at least two log forces per participant in this distributed transaction.

Because all processing is done within a transaction, handling transaction failures requires special case mechanisms. Typically, a TP monitor will retry a transaction some fixed number of times, hoping that it will succeed. If it fails repeatedly, then an error is posted to an administrative (error) queue to be examined manually. The problem here is that no application logic executes outside of a transaction. Program logic fails when a transaction fails. So how does an application understand what happened to its request if a reply is enqueued on an error queue?

There is, not unexpectedly, a relationship between interaction contracts and distributed transactions. Both typically require that logs be forced from time to time so as to make information stable. But the interaction contract is, in fact, the more primitive notion. And, if one's goal is application program persistence, only the more primitive notion is required. Only if rollback is needed is the full mechanism of a transaction required. Further, in web services, as for workflow in general, "rollback" (compensation) is frequently separate from the transaction doing the forward work. For distributed, web services based computing, we do not believe that the tight coupling and high overhead needed for transactions will make them the preferred approach.

There is an even deeper connection between interaction contracts and 2PC. The message protocols in two phase commit are, in fact, instances of transaction interaction contracts. However, by unlocking the TIC from the commit coordination protocol, we make it possible to have a better, more flexible, efficient, and opaque end-to-end protocol for making applications robust. It is these properties that make interaction contracts well suited for web services applications. We'll see this in the next section.

3 Opaque Web Services Using Interaction Contracts

3.1 Web Services Characteristics

The setting for web services is quite different from traditional transaction processing. The TP world was usually completely within an enterprise, or where that wasn't the case, between limited clients and a service or services within a single enterprise. But web services are intended specifically for the multi-enterprise or at least multi-organizational situation. Site autonomy is paramount. This is why web services are "arms length" and opaque, based on messages, not RPC. This is why the messages are self-describing and based on XML.

Because web services are opaque, a transaction interaction contract is not quite appropriate. An application is not entitled to know whether a web service is performing a "real" transaction. It is entitled to know only something about the end state. Further, a web service provider will be very reluctant to enter into a commit protocol with applications that are outside of its control. But web services need to be concerned about robust applications. In particular, a web service should be concerned with enabling exactly-once execution for applications. This is the intent of the web services interaction contract.

3.2 Interacting with a Web Service

There can be many application interactions with a web service that require no special guarantees. For example, an application asks about the price and availability of some product. Going even further, a customer may be shopping, and have placed a number of products in his shopping cart. There is, at this point, no "guarantee" either that the customer will purchase these products, or that the web service will have them in stock, or at the same price, when a decision is eventually made. While remembering a shopping cart by the web service is a desirable characteristic, it is sufficient that this remembering be a "best effort", not a guarantee. Such a "best effort" can be expected to succeed well over 99% of the time. Because of this, it is not necessary to take strenuous measures, only ordinary measures, to do the remembering. But a guarantee must always succeed, and this can require strenuous efforts. We want, however, to reserve these efforts for the cases that actually require them.

By analogy with the transaction interaction contract, guarantees are needed only when work is to be "committed". As Tcom's can forget transactions before commit, so web services can forget units of work. An application can likewise forget this unit of work. No one has agreed to anything yet. There is no direct way an application can tell whether a work unit is a transaction, or not. (Of course, it might start other sessions with other work units and see whether various actions are prevented, and perhaps infer what is going on.) Whether there is a transaction going on during this time is not part of any contract.

A web service may require that an application remember something, e.g. the id for the unit of work, as it might for a transaction, or as it might for a shopping session with a particular shopping cart. But this is not a subject of the guarantee. The guarantee applies exactly to the message in which work is going to be "committed". Everything up to this point has been "hypothetical". Further, if the web service finds the "commit request" message not to its liking, it can "abort". Indeed, it can have amnesia in any circumstance, and when that happens, there is no guarantee that the web service will remember any of the prior activity.

3.3 A Web Services Interaction Contract

It is when we get to the "final" message, e.g. when a user is to purchase an airline ticket, that we need a guarantee. In a web services interaction contract (**WSIC**), only the web service makes guarantees. If the application also takes actions of the sort required of a Pcom in the transaction interaction contract, then it can ensure its persistence. Without the WSIC, it would not be possible to implement Pcoms interacting with web services. But the web service contract is independent of such an arrangement. Thus a WSIC is a *unilateral* pledge to the outside world of its applications. Nothing is required of the applications.

The WSIC requirements for the web service resemble the requirements imposed on transactional components by a TIC.

Web Service (WS) Obligations:

WS1: Duplicate Elimination: WS promises to eliminate duplicate commit request messages. It treats duplicate copies of the message as requests to resend the reply message.

WS2: Persistent State Transition: The WS promises that it has proceeded to one of two possible states, either "committing" or "aborting" and that the resulting state is persistent. [These states are in quotation marks because there may not be a connection with any particular transaction.]

WS3: Unique Persistent (Faithful) Reply Message: WS awaits prompting from the application to resend, accomplished by the application repeating its commit request message. Once the requested action terminates, the WS replies acknowledging the commit request, and guarantees persistence of this reply until released from this guarantee. The WS promises to resend the message upon explicit request. Message uniqueness permits an application to detect duplicates.

An important aspect of the WSIC is that it says nothing about how the web service meets its WSIC obligations. This is unlike the TIC, where the transaction component is required to have an atomic and isolated action (transaction). Thus, a web service might meet its requirements using perhaps a persistent application, or a workflow. It might exploit transactional queues or databases or file systems. This is the other side of the value of the opaque interaction contract. It does not prescribe how a web service meets its obligations, it only describes the obligations. This obligation ensures that the request to the web service is "idempotent". Note that the WS action taken, like a logged database operation, does not need to be idempotent, and in general is not. It is the web service that provides idempotence, i.e. exactly once execution, by ensuring that the action is only executed once. We discuss next how an application can use a WSIC to ensure robust behavior.

3.4 Robust Applications

What the WSIC guarantees, as with the TIC, is that **if** an application is written as a persistent component, following the rules for the Pcom in the TIC, then the application can be made to survive system crashes. We showed how Pcoms can be made persistent previously [2, 3, and 4].

The key role that the WSIC plays is to ensure that an interaction can be re-requested should a failure occur during a web service execution. In this case, the WSIC ensures that the web service activity is executed exactly once, despite potentially receiving multiple duplicate requests (WS1), that the web service "committed" state, once reached is persistent (WS2), and that the reply message will not be lost because it is persistent, and it is unique to ensure that it cannot be mistaken for any other message (WS3), so that duplicates can be eliminated by the application.

We have argued before that programming using persistent applications is easier and more natural than programming using stateless applications. A stateful persistent application need not be arranged into a "string of beads" style, where each "bead" is a transaction that moves the state from one persistent queue to another. Rather, the application is simply written as the application logic demands, with persistence provided by logging and by the ability to replay crash interrupted executions [3]. And it is this program logic that can deal with errors, either exercising other execution paths or at least reporting errors to end users, or both.

4 An Example

4.1 The Application

In this section, we explore ways to implement a web service that satisfy the WSIC. This will illustrate how the flexibility permitted by an opaque web service can be exploited for both implementation ease and to enable persistent applications.

Our application is a generic order entry system. We do not describe it in detail. But it responds to requests about the stock of items it sells, it may permit a user to accumulate potential purchases in a "shopping cart". And, finally, when the client (application or end user) decides to make a purchase, the "commit request" for this purchase is supported by a web service interaction contract that will guarantee exactly once execution.

4.2 Using Transactional Message Queues

A "conventional" transaction processing method of implementing our web service might be to use message queues [1, 7]. When an application makes a request, the web service enqueues the request on its message queue. This executes as a transaction to ensure the capture of the request. The application must provide a request id used to uniquely identify the work item on the queue. The message queue permits only a single instance of a request or reply with a given request id.

The request is executed by being dequeued from the message queue, and the order is entered into the order database, perhaps inventory is checked, etc. When this is complete, a reply message is enqueued to the message queue. This reply message may indicate the order status, what the ship date might be, the shipping cost and taxes, etc. This is the WS2 obligation and part of the WS3 obligation, since both the state and the message are guaranteed to persist.

The application resubmits its request should a reply not arrive in the expected time. The web service checks the message queue for the presence of the request. If not found, the request is enqueued. If it is found, the web service waits for the request to be processed and the reply entry available. If the reply is already present, it is returned directly to the application. The queue has made the reply durable. The web service ensures that request and reply are unique so that duplicates are eliminated, satisfying WS1.

Note here that if the application executes within a transaction, as is the case for traditional transaction processing, and its transaction fails, there is no convenient place to handle the failure. But here, even when the web service uses conventional queued transaction processing, the application can be a persistent one. When that is the case, application logic can handle web service failures. The application is indifferent to how the web service provides the WSIC guarantees, only that they are provided, enabling application persistent.

4.3 Another Approach

We can change the implementation of the web service to provide the WSIC guarantees in a simpler and more efficient manner if the web service has the freedom to modify database design. We guarantee duplicate elimination, a persistent state transition and a persistent output message all by adding a request id field to each order in our order database. We enter an order in the order table using a SQL insert statement. We define this table with a uniqueness constraint, permitting only one order with the given request id to be entered.

Should our activity in the web service be interrupted by a system crash, then there are a number of possible cases.

1. We have no trace of the request because the transaction updating the order table was not committed prior to the crash. In this case, it is as if we have not seen the request. A "persistent" application will resend the request.
2. We have committed the transaction that updates the order table. Subsequent duplicate requests are detected when we again try to insert an order with the given request id into the order table. The duplicate is detected, the transaction is bypassed, and the original reply message is generated again based on the order information in the table.

The implementation strategy we sketch here avoids the need to have the persistent message part of the WSIC released explicitly. The idea is that the request id remains with the order for the entire time that the order is relevant. With the traditional transaction processing approach, the release for the persistence guarantee is done with the commit of a dequeuing operation for the reply on the message queue.

The bottom line here is that the WSIC provides abstract requirements. The web service can decide how to realize them. Message queues are one way of doing this. But, as can be seen, there can be other, perhaps more effective, approaches.

4.4 The Web Service Client

Because the WSIC is an abstract, opaque characterization of requirements for a web service, an application program using the web service can essentially do whatever it wants, since the WSIC is a unilateral guarantee by the web service. The application has no obligations under the WSIC.

If the application doesn't do anything special, the program state will not be persistent across system crashes. But the WSIC guarantees are useful in any case. With respect to our order entry system example, for instance, it is surely useful for an end user to be able to ask about an order's status. And having a persistent state reflecting the order is usually considered a minimal requirement for business data processing needs.

If the application wants to realize exactly once semantics for its request, then the WSIC enables this robustness property to be realized. Not surprisingly, the application needs to implement the Pcom side of the transaction interaction contract (TIC). That is, the application becomes persistent when it assumes the obligations of the persistent component in a TIC.

5 Discussion

A number of additional subjects are worth mentioning briefly.

5.1 Undo Actions for Business Processes

To construct long duration workflows, it is usually a requirement that some form of compensation action be possible for each forward action of the workflow. To support this kind of scenario, we can associate with each "action" of a web service a "cancel" (or undo) action. This says nothing about atomicity. The "cancel" activity can be as opaque as the original action. It says nothing about the details of the "inverse" action. But it puts the responsibility for the cancel (or undo) action on the web service, which, after all is the only autonomous entity really capable of doing it.

One way of dealing with this is to submit the same id used for the commit request with a cancel request (which itself is a form of "commit request" obeying the TIC obligations). There may be a charge assessed for this under some circumstances, and that should be part of the WSDL description of the web service. The response message to this cancel request should be something like "Request cancelled". This is independent of whether the "forward" request was ever received or executed since once the request is cancelled, there should be no requirement that the web service remember the original request.

By supporting a "cancel request", a web server enables an application to program a compensation action should the application need to "change plans". Note that this says nothing about how the application figures out what needs undoing, etc. Again, web services are opaque. But by providing a "cancel request", they enable an application to be written that undoes earlier work as appropriate.

There need be no requirement that a web service provide a "cancel request". But there is no requirement also, that an application program use any specific web service. But many e-commerce sites support canceling orders, and many web services should be willing to support "cancel request", especially if it were possible to charge for it.

A "cancel" request may only be a best efforts cancellation, e.g. the canceling of an order to buy or sell shares of stock. If the cancellation fails, then there is an obligation to faithfully report that failure. In this case, the web service is obligated to maintain the original action state so as to be able to generate a persistent "cancellation failed" message.

5.2 Releasing Contracts

Persistent states and persistent message obligations of the parties to an interaction contract may require an eventual release. Application programs that exploit the obligations eventually terminate. An order is eventually filled and at that point becomes of historical interest but not of current interest. We have not discussed up to this point how the contracts might be released. For web services, recall, the obligations are unilateral and apply only to the web service. Any application effort to exploit the WSIC, e.g. to provide for its persistence, is purely at the its own discretion. Below we discuss some alternatives for the web service.

No Release Required: This is both very useful and very simple for applications. It means that no matter how long the application runs, the web service will retain the information needed to effectively replay the interaction. A variant of this is that, for example, once an order is shipped, the shipped object ends the WSIC obligations. So it is frequently possible to remove information about old orders (old interactions) from the online system without compromising the WSIC guarantees, and without requiring anything from the application.

Release Encouraged: In this scenario, when an application releases the web service from its WSIC obligations, the web service can remove information associated with the interactions from the online system. If only a small number of apps do not cooperate, that will not be a major issue. Storage is cheap and plentiful. A further step here is to "strongly" encourage release, e.g. by giving cooperators a small discount. Another possibility is to eventually deny future service until some old WSIC's are released. This distinguishes well-behaved applications from rogues, and eventually limits what the rogues can do.

Release Required: When web service providers feel that it is too much of a burden on them to maintain interaction information in their online system, an application can be required to release the contract. A frequent strategy is to stipulate that the contract is released at the time of the next contact or the next commit request. If there is no such additional contact that flows from the application logic, then release can be done at the time the application terminates (or its session terminates). Of course, stipulations can be ignored. So a further "clause" in the web service persistence guarantee might "publish" a time limit on the guarantee. For example, one might safely conclude that most applications requesting a web service would be complete within four hours of the commit request having been received, or within one day, etc. This permits "garbage collecting" the information that is older than the published guarantee.

5.3 Optimizations

The existence of web services supporting WSIC's can frequently enable persistence for applications more efficiently than can traditional transaction processing. Typically, with traditional transaction processing, each message exchange is within a separate transaction. Input state on a queue and one message are consumed, and the output state is placed on another queue, then the transaction is committed. Thus there are two log forces per resource manager (message source, input queue, and output queue) and at least one, perhaps two, for the transaction coordinator.

Using interaction contracts, we have an opportunity to avoid multiple forced log writes. The application may have several interactions, each with a different web service. Assuming that contract release is either not required, or occurs on the next interaction, it is not necessary for the application to log each of these interactions immediately to make them stable, as is done by committing transactions in classic TP. These interactions are stable via replay, using the sender's stable message. This results in many fewer forced log writes. Each web service may require a log force, and eventually the application will need to force the log, but this can be amortized over multiple web services. So, "asymptotically, we might have as few as one log force per method call instead of several.

5.4 Summary

We have shown how robust web based applications can be enabled by web services that meet the relatively modest requirements for a unilateral web services interaction contract. The WSIC is an opaque requirement in that it does not specify (or even reveal) how it is that the web service satisfies the WSIC. Further, it usually permits robust, i.e. persistent applications, to be realized at lower cost than traditional transaction processing. Finally, the application can be a stateful one. This has two advantages: (i) it is a more natural programming style than the "string of beads" style required by traditional TP; (ii) it enables program logic in the persistent application to deal with transaction failures, something that is not easily accommodated in traditional TP.

References

1. P. Bernstein, and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.
2. R. Barga, S. Chen, D. Lomet. Improving Logging and Recovery Performance in Phoenix/App, *ICDE* (March 2004) 486–497.
3. R. Barga, D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran. Persistent Applications via Automatic Recovery. *IDEAS* (July, 2003) 258–267.
4. R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery Guarantees for Internet Applications *ACM TOIT* 4(3) (August, 2004) 289–328.
5. R. Barga, D. Lomet, and G. Weikum. Recovery Guarantees for General Multi-Tier Applications. *ICDE* (March 2002) 543–554.
6. E.N. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comp. Surv.* 34(3), 2002.
7. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
8. Microsoft. Microsoft .Net Framework Developer Center. http://msdn.microsoft.com /netframework/
9. Oasis. Universal Description, Discovery and Integration of Web Services http://www.uddi.org/specification.html
10. D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *CACM* 15(12): (December 1972) 1053–1058.
11. WC3. Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl .
12. WC3. Web Services Coordination. 106.ibm.com/developerworks/library/ws-coor/.