Transactional Attitudes: Reliable Composition of Autonomous Web Services

Thomas Mikalsen, Stefan Tai, Isabelle Rouvellou

IBM T.J. Watson Research Center, Hawthorne, New York, USA
{tommi | stai | rouvellou@us.ibm.com}

Abstract

The Web services platform offers a distributed computing environment where autonomous applications interact using standard Internet technology. In this environment, diverse applications and systems become the components of intra- and inter-enterprise integration. Yet, transactional reliability, an often critical requirement on such integration, is presently missing from the Web services platform. In this paper, we address this shortcoming and propose the WSTx framework as an approach to Web service reliability. WSTx introduces transactional attitudes *to explicitly describe the otherwise implicit transactional semantics, capabilities, and requirements of individual applications. We show how explicit transactional attitude descriptions can be used by a middleware system to automate the reliable composition of applications into larger Web transactions, while maintaining autonomy of the individual applications.*

1. Introduction

The development of software systems frequently entails the need to integrate diverse applications within an enterprise and across enterprises. Different kinds of application integration technologies, such as object-oriented middleware, message-oriented middleware, and, more recently, the Web services platform, have been proposed for this purpose.

Transactions are a commonly employed approach to address system reliability and fault-tolerance. Object-oriented middleware and message-oriented middleware support transaction processing; examples of middleware transaction models are CORBA OTS/JTS distributed object transactions, EJB declarative transactions, and messaging transactions. The Web services platform, however, is lacking reliability features; no Web transaction model and corresponding Web transaction infrastructure support has been established so far.

In this paper, we address this shortcoming of the Web services platform and propose a solution for introducing transactional reliability to Web services.

Web transactions are a difficult matter, as unlike conventional middleware transactions, no common transaction semantic, transaction context representation, and coordination protocol can be assumed to exist for transaction participants (the individual applications) in a Web services environment. It is more likely that participants are autonomous (with respect to their implementation and execution environment), that different, seemingly incompatible transaction models and middleware technologies may be involved in the same Web transaction, and that context representation and service coordination and management must be achieved in a decoupled, decentralized manner.

In this paper, we describe first results in the development of the *WSTx (Web Services Transactions)* framework addressing these challenges. We introduce the concept of *“transactional attitudes”*, where Web service providers declare their individual transactional capabilities and semantics, and Web service clients declare their transactional requirements (of providers). We further propose a Web service middleware, based on intermediaries, which supports different Web service transactional capabilities, and provides global context management for execution monitoring, transaction completion, failure-detection, and recovery.

2. Web Services

The Web services platform comprises different kinds of technologies and standards that are organized into the five layers of network, transport, packaging, description, and discovery, as illustrated in Figure 1.

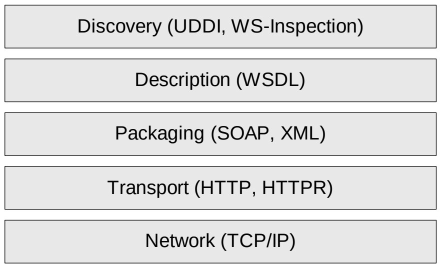

Figure 1. Web Services Technology Stack

The Web services standard of primary interest in this paper is WSDL [1]. WSDL is the *Web Services Description Language*; it is used to describe a Web service in terms of *ports* (addresses implementing the service), *port types* (the abstract definition of operations and ordered exchanges of messages), and *bindings* (the concrete definition of which packaging protocols, such as SOAP [2], are used).

WSDL is an interface standard that abstracts from any platform and programming language-specific details of how application code is actually invoked. As such, Web services are independent of any particular implementation technology used; they can be deployed anywhere that common Internet technology is supported.

Therefore, individual (implementations of) Web services may use different kinds of (middleware) transaction technology. A Web services transaction model that aims to support transactions across Web services hence must tolerate such autonomy; a Web transaction is a *global* transaction that spans across diverse transactional (and non-transactional) service implementations.

3. WSTx

In this section, we introduce the *Web services transaction (WSTx) framework* comprising *provider transactional attitudes* (Section 3.2) and *client transactional attitudes* (Section 3.3.). We then show how this framework is supported as *middleware* (Section 3.4). We begin by describing a travel booking scenario (Section 3.1), which is used throughout the paper to motivate and illustrate the WSTx approach.

3.1. Travel Booking Scenario

In the Travel Booking scenario illustrated in Figure 2, a client application wishes to arrange a trip using three independent travel services: a *flights* Web service, a *rooms* Web service, and a *taxis* Web services. These services are defined below.

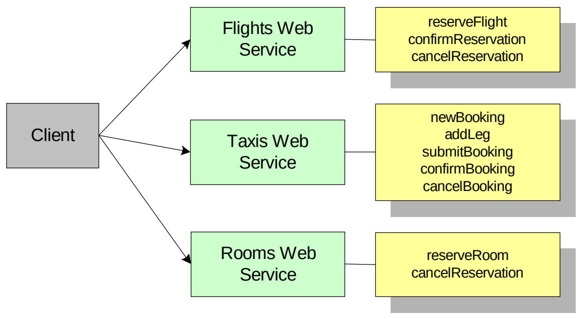

Figure 2. Travel Booking Services

The Flights Web Service. The *Flights* Web service provides the three operations `reserveFlight`, `confirmReservation`, and `cancelReservation`. The `reserveFlight` operation returns a reservation number to the client, which the client needs to either confirm a reservation, or to cancel a reservation. This service is defined by the following WSDL specification:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Flights reservation Web service -->
<definitions name="Flights"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
...
<!-- abstract messages -->
...
<message name="ReserveFlightOut">
  <part name="resvNo" type="xsd:string"/>
</message>
<message name="ConfirmReservationIn">
  <part name="resvNo" type="xsd:string"/>
</message>
<message name="CancelReservationIn">
  <part name="resvNo" type="xsd:string"/>
</message>
...
<!-- port type -->
<portType name="FlightsPortType">
<operation name="reserveFlight">
  <input message="ReserveFlightIn"/>
  <output message="ReserveFlightOut"/>
</operation>
<operation name="confirmReservation">
  <input message="ConfirmReservationIn"/>
  <output message="ConfirmReservationOut"/>
</operation>
<operation name="cancelReservation">
  <input message="CancelReservationIn"/>
  <output message="CancelReservationOut"/>
</operation>
</portType>
</definitions>
```

The Taxis Web Service. The *Taxis* Web service offers operations to group multiple taxi leg requests into a single (atomic) reservation. The `newBooking` operation is used to start a reservation; it returns a booking number which the client uses for all subsequent operations. The `addLeg` operation is used to add taxi legs, the `submitBooking` operation is used to submit the request, and the `confirmBooking` and the `cancelBooking` operations are used to complete or cancel a reservation. The *Taxis* Web service is described as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Taxis reservation Web service -->
<definitions name="Taxis"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
...
<!-- abstract messages -->
...
<message name="NewBookingOut">
  <part name="bookingNo" type="xsd:string"/>
</message>
<message name="ConfirmBookingOut">
  <part name="confNo" type="xsd:string"/>
</message>
...
<!-- port type -->
<portType name="TaxisPortType">
<operation name="newBooking">
  <input message="NewBookingIn"/>
  <output message="NewBookingOut"/>
</operation>
```

```
<operation name="addLeg"> ...
</operation>
<operation name="submitBooking"> ...
</operation>
<operation name="confirmBooking">
   <input message="ConfirmBookingIn"/>
   <output message="ConfirmBookingOut"/>
</operation>
<operation name="cancelBooking"> ...
</operation>
</portType>
</definitions>
```

The Rooms Web Service. The *Rooms* Web service provides the two operations `reserveRoom` and `cancelReservation`. The `reserveRoom` operation returns a confirmation number to the client, which is needed for the client to cancel (undo) a reservation later on. The following WSDL fragment shows the interface of the service:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Hotel room reservation Web service -->
<definitions name="Rooms"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
   ...>
<!-- types -->
<types ...>
<schema>
<complexType name=”ConfInfo”>
<sequence>
   <element name=”confNo” type=”xsd:string”/>
   ...
</sequence>
</schema>
</complexType>
</types>
<!-- abstract messages -->
...
<message name="ReserveRoomOut">
   <part name="conf" type="types:ConfInfo"/>
</message>
<message name="CancelReservationIn">
   <part name="confNo" type="xsd:string"/>
</message>
...
<!-- port type -->
<portType name="RoomsPortType">
<operation name="reserveRoom">
   <input message="ReserveRoomIn"/>
   <output message="ReserveRoomOut"/>
</operation>
<operation name="cancelReservation">
   <input message="CancelReservationIn"/>
   <outputmessage="CancelReservationOut"/>
</operation>
</portType>
</definitions>
```

Each Web service implements a different transactional behavior, which, using standard WSDL, is only *implicitly* described. For example, in the Flights Web service, the `cancelReservation` operation aborts a tentative reservation, whereas in the Rooms Web service, the `cancelReservation` operation compensates for a completed reservation.

If the client application wishes to establish a transactional dependency between the individual services, so that only well defined combinations of reservations/ bookings are possible, ad hoc code will be required to weave these services into a meaningful transaction.

3.2. Provider Transactional Attitudes

In order to avoid ad hoc programming of compositions of Web services that have different transactional semantics, we propose the concept of *Provider Transactional Attitudes*.

Provider transactional attitudes (PTAs) are a mechanism for Web service providers to explicitly describe their specific transactional behavior. By making transactional semantics explicit, PTAs can be used in automating the composition of individual transactional Web services into larger transactional patterns, while maintaining the autonomy of the individual services.

A PTA includes the name of an abstract transactional pattern, plus any additional port-specific information needed to make the pattern concrete. The abstract pattern implies a well-defined state machine describing the valid transactional states, state transitions, and transition-triggering events. The name and semantic of each state is implied by the name of the transactional pattern. State transitions are triggered by externally observable events, such as operation invocations on ports and time-outs. To complete a pattern, a provider must

- identify transactional port operations (i.e., the port operations that can trigger transactional-state transitions),
- describe how invocations on these operations are associated with corresponding state machines instances (e.g., operation correlation encoding), and
- describe how transactional-operation outcomes are encoded.

We propose using the standard WSDL extensibility mechanisms (so called *extensibility elements*) to annotate port bindings with transactional semantics (i.e., to define an XML vocabulary for transactions).

The WSTx includes the following extension elements (applied to `wsdl:binding`):

- a binding extension element for declaring the transactional attitude (i.e., transactional pattern) of a port,
- operation extension elements for declaring transactional-event triggering operations,
- input/output extension elements for describing operation correlation encoding, and
- output/fault extension elements for describing operation-outcome encoding.

For example, a WSTx port binding element is used to declare a port’s transactional attitude: annotating a port binding with

```
     <wstx:binding
          attitude=’pending-commit’/>
```

declares that the port has a pending-commit attitude towards transactions. The name pending-commit refers to a well-defined WSTx transactional pattern.

The WSTx vocabulary presently defines three provider transactional attitudes:

- "*pending-commit*" (PC),
- "*group-pending-commit*" (GPC), and
- "*commit-compensate*" (CC).

We have found these PTAs to describe transactional behavior that is common to many Web services; in particular, these PTAs can be used to describe the transactional behavior of the three Web services in our Travel Booking scenario. We expect the number attitudes supported by the WSTx to grow as additional transactional Web service patterns emerge.

Following, we describe each of these transactional attitudes in detail, and describe the associated WSTx extension elements used to represent them.

3.2.1. Pending-commit PTA

The *pending-commit* (PC) attitude describes a transactional port of a single Web service where the effect of a single *forward operation* invocation can be held in a pending state; the operation-effect remains pending until the subsequent occurrence of an event (e.g., the invocation of a commit or abort operation) triggers either acceptance or rejection (of the operation-effect).

Forward operations are annotated with a `<wstx:forwardOperation>` element. The effect of a forward operation is brought to a pending state if the operation is invoked successfully; otherwise, the effect is rejected.

The effect of a forward operation can be explicitly realized by invoking a *commit* operation. Commit operations are annotated using a `<wstx:commitOperation>` operation extension element.

The effect of a forward operation can be explicitly dismissed by invoking an *abort* operation. Abort operations are annotated using a `<wstx:abortOperation>` operation extension element.

In order to associate a forward operation with a commit (or abort) operation, a PC attitude must include a *correlation encoding*. WSTx provides the `<wstx:simpleCorrelation>` input/output extension element for describing simple correlation encodings, where a correlation identifier is embedded in a message part. This element allows the provider to specify the part, and optionally the sub-part (e.g., using an XPath [3] expression), of a message that holds the correlation identifier.

Returning to the Travel Booking scenario from above, the Flights Web service implicitly describes the transactional semantics of a pending-commit PTA: the process of reserving a flight includes a pending state, before the reservation is either committed (using the confirmReservation operation) or aborted (using the cancelReservation operation). This transactional behavior can be made explicit in a WSDL binding using WSTx extension elements as follows[1]:

```
<binding name="FlightsPortBinding"
        type="tns:FlightsPortType">
<wstx:binding attitude="pending-commit"/>
<operation name="reserveFlight">
  <wstx:forwardOperation/>
  <output>
  <wstx:simpleCorrelation partName="resvNo"/>
  </output>
</operation>
<operation name="confirmReservation">
  <wstx:commitOperation/>
  <input>
  <wstx:simpleCorrelation partName="resvNo"/>
  </input>
</operation>
<operation name="cancelReservation">
  <wstx:abortOperation/>
  <input>
  <wstx:simpleCorrelation partName="resvNo"/>
  </input>
</operation>
</binding>
```

3.2.2. Group-pending-commit PTA

The *group-pending-commit* (GPC) attitude describes a transactional port of a single Web service where the effects of a *group of forward operation* invocations can be held in a pending state; the group-effect remains pending until the subsequent occurrence of an event (e.g., the invocation of a commit or abort operation) triggers either acceptance or rejection (of the group-effect).

A new operation group is created either explicitly with a *begin* operation, or implicitly as part of a *forward* operation.

Explicit begin operations are annotated with a `<wstx:beginOperation>` element. Forward operations are annotated with a `<wstx:forwardOperation>` element.

The group-effect is brought to a pending state by invoking a single *prepare* operation. Prepare operations are annotated with a `<wstx:prepareOperation>` element. The group-effect is brought to a pending state if the prepare operation is invoked successfully; otherwise, the group-effect is rejected.

The group-effect of all forward operations can be explicitly accepted by invoking a single *commit* operation. A commit operation is annotated with a `<wstx:commitOperation>` element. The group-effect is accepted if the commit operation is invoked successfully; otherwise, the group-effect is rejected.

The group-effect of all forward operations can be explicitly rejected by invoking a single *abort* operation. Abort operations are annotated using the `<wstx:abortOperation>` element.

In order to associate individual operations (forward operations, prepare operations, etc.) with a group, a GPC attitude must include a correlation encoding. As with the pending-commit PTA (see "Pending-commit PTA"above), simple correlation encodings can be

1. The bindings illustrated here have been abbreviated to save space; a complete binding would likely include additional communication protocol elements, such as SOAP extension elements.

described using the `<wstx:simpleCorrelation>` element.

The "Taxis" Web service (from the scenario above) implicitly describes a group-pending-commit PTA: the Web service provides control operations to begin, prepare, commit or abort a group of actions, and it provides a forward operation that represents the group action. This transactional behavior can be described using WSTx as follows:

```
<binding name="BookingBinding"
   type="tns:BookingPortType">
<wstx:binding
   attitude="group-pending-commit"/>
<operation name="newBooking">
<wstx:beginOperation/>
<output>
   <wstx:simpleCorrelation
      partName="bookingNo"/>
</output>
</operation>
<operation name="addLeg">
<wstx:forwardOperation/>
<input>
   <wstx:simpleCorrelation
      partName="bookingNo"/>
</input>
</operation>
<operation name="submitBooking">
<wstx:prepareOperation/>
<input>
   <wstx:simpleCorrelation
      partName="bookingNo"/>
</input>
</operation>
<operation name="confirmBooking">
<wstx:commitOperation/>
<input>
   <wstx:simpleCorrelation
      partName="bookingNo"/>
</input>
</operation>
<operation name="cancelBooking">
<wstx:abortOperation/>
<input>
   <wstx:simpleCorrelation
      partName="bookingNo"/>
</input>
</operation>
</binding>
```

3.2.3. Commit-compensate PTA

The *commit-compensate* (CC) attitude describes a transactional port of a single Web service where the effect of a *single forward operation* invocation is immediately accepted, yet can later be semantically reversed by invoking an associated *compensation* operation on the port.

Forward operations are annotated with a `<wstx:forwardOperation>` element. The effect of a forward operation is immediately accepted if the operation is invoked successfully; otherwise, the effect is rejected.

The effect of a forward operation can be semantically reversed by invoking a *compensation* operation. Compensation operations are annotated using a `<wstx:compensationOperation>` element.

The transactional behavior of the Rooms Web service (from above) is consistent with the commit-compensate PTA: an incoming request that can be executed is immediately committed. A compensating operation (the `cancelReservation` operation) is defined for a client to undo a previously committed reservation. This behavior can be made explicit using the following WSTx port binding:

```
<binding name="ReservationBinding"
   type="tns:ReservationPortType">
<wstx:binding attitude="commit-compensate"/>
<operation name="reserveRoom">
<wstx:forwardOperation/>
<input>
<output>
   <wstx:simpleCorellation
      partName="conf" select="confNo"/>
</output>
</operation>
<operation name="cancelReservation">
<wstx:compensationOperation/>
<input>
   <wstx:simpleCorrelation partName="confNo"/>
</input>
</operation>
</binding>
```

In the examples above, a default operation-outcome encoding is assumed. Each operation, in all three Web services, has an associated single `fault` message (which has been elided to save space). If an operation generates a `fault`, then the operation-outcome is `failure`; otherwise, the operation-outcome is `success`.[2] The meaning of `success` and `failure` for a given operation is defined by the PTA; for example, failure of a `prepareOperation` in the group-pending-commit PTA results in the immediate rejection of the group-effect (as if an `abortOperation` was invoked).

Figure 3 illustrates in summary the provider transactional attitudes of the three autonomous Web services of the Travel Booking scenario. Using WSTx, the otherwise implicit transactional behavior of the Web services are now explicit as part of the WSDL interface.

3.3. Client Transactional Attitudes

Reconsider the client of the travel booking example. Let us assume that the client wants to reserve two flights, one hotel room, and multiple taxis, using the three independent Web services. The client wishes to establish a transactional dependency between the services, so that only the following defined outcomes of the combined use of the Web services are possible:

2. Other, more flexible, operation-outcome encodings are being considered.

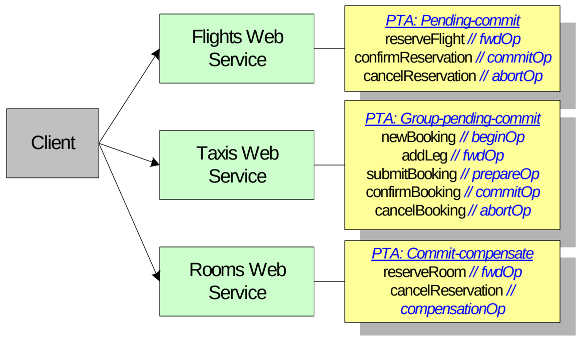

Figure 3. Travel Booking Services with PTAs

- the two flight reservations, the hotel reservation, and the taxis reservation are all committed
- the two flight reservations and the hotel reservation are committed, but the taxis reservation is aborted
- the two flight reservations, the hotel reservation, and the taxis reservation are all aborted.

Therefore, the flights reservations and the hotel reservation *in combination* are vital to the success of the client transaction, whereas the taxis reservation is not critical to the success of the transaction (but is nevertheless part of the transaction).

3.3.1. Implicit client attitudes

Without the WSTx framework, the client needs to manually program the use of the Web services to ensure that exactly one of the desired outcomes is reached. This is a complex exercise; a possible naive solution (neglecting any management of system failures) is illustrated below.

```
// pseudo-code for client without WSTx
cR = Rooms.reserveRoom // commit room
pF1 = Flights.reserveFlight //prepare flight1
pF2 = Flights.reserveFlight //prepare flight2
Taxis.newBooking {
   Taxis.addLeg *
   pT = Taxis.submit // prepare taxis
}
if(cR && pF1 && pF2) { // commit condition
   cF1 = Flights.confirmFlight // commit flight1
   cF2 = Flights.confirmFlight // commit flight2
   if(cF1 && cF2) {
      if(pT){
         cT = Taxis.confirmBooking // commit
      }
   } else {
      Rooms.cancelResv; // compensate room
      if(pT){
         Taxis.cancelBooking // abort
      }
   }
} else { // rollback
   if(pF1) {
      Flights.cancelResv // abort flight1
   }
   if(pF2) {
      Flights.cancelResv // abort flight2
   }
   if(pT) {
      Taxis.cancelBooking // abort taxis
   }
   if(cR) {
      Rooms.cancelResv // compensate room
   }
}
```

Here, the client *implicitly* describes the transactional expectations: which of the providers' commit operations are critical (form an atomic group), and which providers' commit operations are non-critical, but associated to the atomic group.

3.3.2. Explicit client attitudes

The WSTx framework proposes to describe such client expectations *explicitly* using *Client Transactional Attitudes (CTAs).*

A client's transactional attitude is established by its use of a particular WSDL port type to manage (create, complete, etc.) Web transactions, where the port type represents some pre-defined transactional pattern. Within the scope of a Web transaction, the client executes one or more named *actions*, where each action represents a provider transaction (associated with some PTA) executing within the context of the larger Web transaction.

The client initiates an action by binding to an *action port*, which serves as a proxy to a participating provider's transactional port. Each action port represents a unique provider transaction executing within the context of the client's Web transaction. When using an action port, the client may invoke *only the forward operations* of the provider; that is to say, the client cannot invoke *completion* operations (commit, abort, and compensation operations). (Actions and action ports are described in more detail below in Section 3.4.)

Flexible Atom CTA. The *flexible atom (FA)* attitude describes a client transaction where a set of client actions (i.e., provider transactions) are grouped into an atomic group that can have one out of a set of defined group outcomes; that is to say, some actions are declared to be *critical* to the success of the transaction, whereas other actions are part of the transaction though not pivotal to its success. The client specifies the acceptable outcomes as an *outcome condition*, described in terms of the success or failure of the individual actions, and when ready (i.e., after executing the forward operations of these actions), requests the completion of the flexible atom according to that condition. The WSTx middleware (described below in Section 3.4) then attempts to satisfy the condition by invoking the appropriate completion operations on the providers represented by the associated actions.

The code below illustrates the client for the Travel Booking example using the WSTx:

```
// Create a new Flexible Atom
wstxId = flexAtom.beginAtom();
```

```
// Action-port bindings
FlightsPortType flight1 =
   wstxPort.bindAction(wstxId, "flight-1",
      "FlightsPort","url:FlightsService.wsdl")
FlightsPortType flight2=
   wstxPort.bindAction(wstxId, "flight-2",
      "FlightsPort","url:FlightsService.wsdl")
RoomsPortType room1 =
   wstxPort.bindAction(wstxId, "room-1",
      "RoomsPort","url:RoomsService.wsdl")
TaxisPortType taxis =
   wstxPort.bindAction(wstxId, "taxis",
      "TaxisPort","url:TaxisService.wsdl")

// invoke forward operations on participants
room1.reserveRoom(/*...*/);
flight1.reserveFlight(/*...*/);
flight2.reserveFlight(/*...*/);
String t = taxis.newBooking(/*...*/);
   taxis.addLeg(t,/*...*/);
   taxis.addLeg(t,/*...*/);
taxis.submit(t);

// Establish condition (valid outcomes)
String condition = "(flight-1 && flight-2 &&
room-1 && (taxis || !taxis)) || !(flight-1 ||
flight-2 || room-1 || taxis)";

// End the atom
String outcome = flexAtom.endAtom(wstxId,
                          condition);
```

Here, the client initiates four actions named `flight-1`, `flight-2`, `room-1`, and `taxis`. Each action is bound to an action port using the WSTx middleware `bindAction` operation. The action ports serve as proxies to the providers transactional ports, allowing invocations to be associated with the correct client Web transaction. The client invokes forward operations using the action ports, and then establishes an outcome condition representing the desired group outcomes and requests that the flexible atom Web transaction be completed according to this condition.

3.4. WSTx Middleware

As suggested earlier, explicit transactional semantic descriptions, such as the PTAs and CTAs described above, can be used to automate the reliable execution of Web transactions. To support such automation, the WSTx framework includes *middleware* which acts as an intermediary between a client and a set of Web service providers. This middleware harmonizes client transactional requirements (CTAs) with the diverse transactional capabilities of individual providers (PTAs), and reliably executes and monitors Web transactions accordingly.

In the following, we first enumerate some general requirements on a WSTx middleware implementation. We then offer a conceptual design for one possible implementation – the *Smart Attitude Monitor* – which supports the *flexible atom* CTA and the *pending-commit*, *group-pending-commit*, and *commit-compensate* PTAs described above.

3.4.1. General requirements

In general, the WSTx middleware must provide the following functions:

- understand and support some set of PTAs (and the proposed WSTx binding extensions used to describe them)
- understand and support some set of CTAs (i.e., provide port and port bindings for the supported CTA port types)
- intercept all transactional interaction between a client and the participating Web service providers
- establish and manage Web transaction contexts (i.e., global contexts) on behalf of the client
- associate transactional interactions (messages) with the corresponding global contexts and participating local provider transaction contexts
- provide records (logs) of the actions performed for each (global) transaction

To ensure reliable execution, the WSTx middleware must comprise

- a persistent log of all ongoing transactions and their state,
- a persistent log of all relevant transactional-state transformations for each global transaction, and
- a time service that serves for detecting timeouts in transactional interactions with participants.

The WSTx middleware includes interfaces to manage Web transactions according to specific CTAs. Once a client ends a WSTx transaction, it is the middleware's responsibility to reliably perform the completion of the transaction (e.g., invoking commit, abort, and compensation operations on the providers). Thus, the WSTx middleware must further comprise components that monitor participants to guarantee that completion operations are successfully performed.

3.4.2. SAM - the "Smart" Attitude Monitor

We now describe the conceptual design of a specific WSTx middleware implementation – the *Smart Attitude Monitor*, a.k.a. *SAM*.

SAM is itself a Web service, and serves as an intermediary between a transactional client and one or more transactional providers. Figure 4 illustrates SAM, within the context of the Travel Booking scenario, supporting the *flexible atom* CTA and the *pending-commit*, *group-pending-commit*, and *commit-compensate* PTAs described above.

SAM comprises three types of ports and a recovery log:

WSTx Port. This port provides general Web transaction and configuration operations, and is independent of any particular CTA. An example is the `bindAction` operation (described above).

CTA Ports. These ports are used to manage Web transactions associated with specific CTAs. In the figure, a CTA port supporting the *flexible atom* CTA is

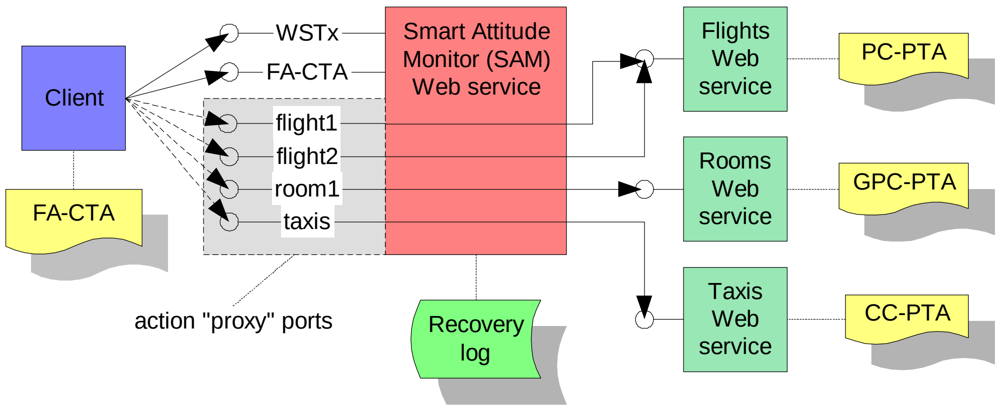

Figure 4. Travel Booking Scenario using SAM

shown. Sample operations are the `beginAtom` operation and `endAtom` operation.

Action Ports. These are *proxy* ports through which a client interacts with providers. An action port is created dynamically (by SAM) whenever a client initiates an action (representing a provider transaction) within the context of a Web transaction.

Recovery Log. SAM's recovery log is used to durably record all critical interactions and transactional state transitions, in order to recover in the event of a failure.

With reference again to Figure 4, we will now elaborate on the client programming model introduced above, and further describe how SAM supports this model to execute Web transactions.

In the example, the client uses SAM's FA-CTA port to create a new *flexible atom* Web transaction:

```
wstxId = flexAtom.beginAtom();
```

The client then initiates actions using SAM's `WSTx` port `bindAction` operation. For example, the client creates an action called "flight-1" associated with the `FlightsPort` port provided by the Web service described in *`url:`*`FlightsService.wsdl`[3]:

```
FlightsPortType flight1 =
   wstxPort.bindAction(wstxId, "flight-1",
      "FlightsPort","url:FlightsService.wsdl")
```

In response, SAM creates a new action proxy port for the action named "flight-1", and associates it with the given Web transaction context (`wstxId`). Subsequent client invocations on this port, which are expected to be forward operations within the same provider transaction context, are routed to the `FlightsPort` port of the `FlightsService` provider. For example, the invocation `flight1.reserveFlight` is routed to the `FlightsService` provider, which creates a new provider transaction context identified by the returned reservation number correlator.

For each client action, SAM observes the transactional events (e.g., client invocations) that can affect the state of the associated provider transaction. In particular, SAM intercepts all messages exchanged through its action ports, and uses the provider's PTA to determine the meaning of these messages within the context of the provider's transaction. In doing so, SAM reflects the state of all provider transactions associated with the client's actions (i.e., the client's Web transaction). SAM persistently records the events associated with critical state transitions, and uses these records to recover in the event of a failure.

When completing a Web transaction, SAM drives the associated provider transactions to completion according to the client's CTA. For example, when completing a *flexible atom* Web transaction, SAM evaluates the actions named in the outcome condition, and attempts to satisfy this condition by invoking the appropriate completion operations (e.g., commit, abort, compensate) on the providers represented by those actions.

3. The specified WSDL file is expected to include the PTA of the provider.

4. Summary and Discussion

In this paper, we introduced the *WSTx framework* for building reliable transactional compositions from Web services with diverse transactional behavior. We showed how *transactional attitudes* are used to capture and communicate otherwise implicit transactional semantics and requirements, without compromising the autonomy of the individual transaction participants. *Provider transactional attitudes* (PTAs) use WSDL extension elements to annotate Web service provider interfaces for Web transactions, according to well-defined transactional patterns; *client transactional attitudes* (CTAs) are described in terms of well-defined WSDL port types and outcome acceptance criteria. We further outlined the requirements on a middleware system which uses these explicit transactional attitude descriptions to reliably execute web transactions, and offered a conceptual design for a specific middleware implementation that meets these requirements.

The WSTx framework uniquely enables a client to program a Web transaction without requiring transaction participants to agree on a common transaction semantic, transaction context representation, and coordination protocol.

Further, the concept of transactional attitudes follows and promotes a clean separation of concerns for software engineering. Transactional properties can be isolated from other aspects of a service description, allowing, for example, capability-based service queries.

Regarding the system implementation, the design of the WSTx framework uses standard Web services technology only; WSDL and existing WSDL extension mechanisms are sufficient to support the idea of transactional attitudes. The design is also open to accommodate some technologies (such as those described below) emerging in the highly dynamic field of Web services.

The implementation design for SAM can easily be realized using existing Web services toolkits and standard Web Application servers (such as IBM's Web Services Toolkit [4]).

There are, however, open issues that need to be further explored. For example, it is possible that completion operations (such as commit, abort, and compensate operations) have signatures for which application specific actual arguments will be needed, or that there are alternative operations for the same completion function (e.g., multiple commit operations). In such cases, the client must somehow communicate this additional information in order for the WSTx middleware to complete a Web transaction.

There are also issues related to the implementation of a mature WSTx middleware. In order to scale, the WSTx middleware may not be a single Web service, but a distributed system of networked entities, or a set of Web service intermediaries. Appropriate message routing support is required for messages traveling through multiple WSTx components and WSTx intermediaries to a specific provider, though no agreed standard for Web service routing exists to date.

The WSTx approach also creates additional opportunities. For example, WSTx clients do not necessarily need to understand the PTAs of the participating providers. Rather, a client is required only to understand its chosen CTA and the semantics of the forward operations of the providers it uses. The CTA itself may impose restrictions on the PTAs that can be included in a Web transaction, and this may result in the client's erroneous attempt to bind an action to an unsupported provider. However, the client's CTA can be used to select appropriate providers (e.g., to dynamically locate providers with compatible transactional attitudes).

5. Related Work

The Web services framework to-date does not provide a model for the reliable (fault-tolerant, transactional) execution of a group of Web services. The need to address this problem, however, has been identified in the context of a number of diverse activities. These activities relate to Web services transactions and the WSTx as follows.

Process modeling support. Extensions to WSDL have been proposed to model application process (work-) flows that involve multiple Web services invocations. IBM's Web Services Flow Language (WSFL) [5] and Microsoft's XLANG [6] fall in this category. Though neither WSFL nor XLANG support distributed transactions, they define constructs that support the composition (coordination) of Web services based on rules. These include constructs for defining invocation orders and dependencies, exception handling, and service correlation and (basic) compensation.

Thus, languages like WSFL and XLANG promise to be very suitable for modeling the client application flow in a WSTx CTA.

Web transaction protocols. A number of specific transaction protocols for the Web are currently being discussed. These include the OASIS Business Transaction Protocol (BTP) [7], and the W3C Tentative Hold Protocol (THP) [8]. These protocols define a model for the automatic coordination of loosely-coupled Web services based on a defined set of transaction messages. They also suggest new notions of transactions that relax some of the ACID properties of conventional transactions. BTP, for example, introduces the notion of a cohesion, which allows the individual participants of a transaction to have different outcomes.

Transactions like BTP cohesions are supported by the WSTx. In fact, the travel booking example discussed in this paper describes a cohesion-like client model. The WSTx is a framework to support diverse transaction protocols; different protocols are regarded as particular transactional attitudes.

Enterprise Java and XML transactions. The state-of-the-art in transaction processing for enterprise systems is arguably reflected in the emergence of transaction coordination frameworks such as the J2EE Activity Service specification [9] and related XML Transactioning API for Java [10]. These frameworks are proposed as implementation-level frameworks for building the transaction monitors that modern Java or Web-based applications use. They define a set of interfaces and XML messages to achieve coordination among application participants and transaction middleware components, independent of the particular transaction protocol that is to be supported.

We believe that these frameworks can help in the design and implementation of a concrete WSTx middleware implementation (SAM).

Dependency-Spheres. In our previous work on Dependency-Spheres (D-Spheres) [11], we developed a transaction model and middleware for applications that communicate with each other using synchronous and/or asynchronous messaging. A D-Sphere is a global transaction context that can include various local transaction contexts of individual participants. WSTx extends some D-Spheres concepts for the Web.

6. References

[1] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, W3C Note 15 March 2001. http://www.w3.org/TR/wsdl.html

[2] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, D. Winer. *Simple Object Access Protocol (SOAP) 1.1*, W3C Note 08 May 2000. http://www.w3.org/TR/SOAP

[3] *XML Path Language (XPath) Version 1.0*, W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath

[4] *IBM Web Services Toolkit (WSTK).* http://www.alphaworks.ibm.com/tech/webservicestoolkit

[5] F. Leymann. *Web Services Flow Language (WSFL 1.0)*, IBM Software Group, May 2001. http://www-4.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf

[6] S. Thatte. *XLANG: Web Services for Business Process Design*. Microsoft Corporation, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[7] OASIS Business Transaction Protocol (BTP). http://www.oasis-open.org/committees/business-transactions/

[8] J. Roberts, K. Srinivasan. *Tentative Hold Protocol Part 1: White Paper*, W3C Note 28 November 2001. http://www.w3.org/TR/tenthold-1/

[9] JSR 95 J2EE Activity Service for Extended Transactions. http://jcp.org/jsr/detail/95.jsp

[10] JSR 156 XML Transactioning API for Java (JAXTX). http://www.jcp.org/jsr/detail/156.jsp

[11] S. Tai, T. Mikalsen, I. Rouvellou, S. Sutton. Dependency Spheres: A Global Transaction Context for Distributed Objects and Messages. In *Proceedings 5th International Enterprise Distributed Object Computing Conference* (EDOC 2001, Seattle, Washington, USA), IEEE Computer Society, September 2001.