Enabling Persistent Web Services via Commitments

Feng Wan

Department of Computer Science

North Carolina State University

Raleigh, NC 27695-7535, USA

fwan@eos.ncsu.edu

919-858-8898x105

Munindar P. Singh*

Department of Computer Science

North Carolina State University

Raleigh, NC 27695-7535, USA

singh@ncsu.edu

Note: Feng Wan is the contact author.

Abstract

Web services are gaining popularity for supporting reusable business processes across distributed and heterogeneous environments. Current Web services are understood as taking inputs, executing their internal logic, and delivering outputs. When outputs are delivered, the interaction related to the given service ends. However, in many cases, the delivery of a service does not mean that the *business transaction* has ended, because there may be a change or cancellation of the original request (from the service requester) or an update of results (from the service provider). Current approaches deal with such scenarios by creating additional operations, thereby arbitrarily splitting the business logic and thus complicating service modeling and execution.

This paper introduces the persistence of services via commitments. Commitments represent agreements between service requesters and providers. The commitments must be fulfilled to ensure successful business transactions. A commitment may last longer than individual episodes of service request and delivery. Operations on commitments, such as create, update, cancel, and discharge, reflect the persistence of the corresponding services and assist in constructing service models and protocols. We show that commitment-enhanced Web service descriptions are simpler and yet more general than current approaches, so that reliable and flexible service compositions can be produced.

Keywords: Web services; commitments; business processes; business transactions

1 Introduction

Web services are gaining popularity, partly because they exploit XML (the eXtensible Markup Language [1]) as the universal encoding language, and ubiquitous protocols such as HTTP as the communication platform. Web services create a new standard of function invocations across heterogenous environments. Unlike previous technologies, such as TCP/IP, RPC, RMI, CORBA, and DCOM, Web services are described, discovered, and invoked through a triad of standards, namely, WSDL (Web Services Description Language [5]), UDDI (Universal Description, Discovery, and Integration [11]), and SOAP (Simple Object Access Protocol [13]), respectively.

Although described in XML, each individual service merely represents a set of low-level functions that can be invoked remotely. To form real B2B (business-to-business) or B2C (business-to-consumer) interactions, a set of services need to work together and be executed in a specified order. Such execution is termed *service composition* or *choreography*. Several standards have been proposed and are being introduced into practice, especially BPML (Business Process Modeling Language [3]), BPEL4WS (Business Process Execution Language for Web Services [2]) and WSCI (Web Service Choreography Interface [17]). However, these approaches are conceptually no more than simple extensions of traditional workflow technologies wherein Web services are used to represent tasks instead of ad hoc and legacy processes.

Web services are sometimes used to form conversational messaging systems among parties that interact with each other. This is also a limited extension of traditional messaging systems wherein we simply replace the sending and receiving of messages with service invocations. This change of the transport layer has no effect on high-level abstractions.

Since Web applications face greater heterogeneity than other applications, merely exposing a reusable software component as a Web service is rarely adequate. To solve a specific business problem, additional services must be defined and additional process flows must be enacted. The main cause of this problem is that the Web service specifications mostly concentrate on lower levels and do not offer high-level abstractions to accommodate variations in service invocations and business process models.

Because multiparty collaborations in business and elsewhere involve autonomous and persistent parties with long-lived interactions, a model based on service invocation or "run-once" workflows is not adequate for such applications. Communication among the interacting parties requires continually updating or canceling any previous service requests and creating new service requests. The updating of a request could involve changes of the input parameters as well as the removal or addition of parameters. Likewise, a response could be updated because the underlying data available to the service provider could have changed. In such an environment, if the service requesters and providers can understand each other, then the system would be reliable and adaptable, thereby enabling flexible treatment of varied business scenarios.

To accomplish the above objective, we introduce the concept of *commitments*, which are now widely recognized as a key representation for the interactions in a multiagent system [10]. This is because commitments enable us to model and analyze the behavior of autonomous agents which represent the service providers and requesters, especially in settings such as supply chain integration and Web service composition for business process integration.

Commitments are a key element of the semantics of agent communications [6]. Of particular relevance here is recent work on operationalizing commitments. Fornara and Colombetti model the life cycle of a commitment and develop an operational semantics for commitments [8]. This work gives a foundation for deriving commitments from low-level messaging protocols. Economou *et al.* show how deontic states and commitments can be inferred from agents' finite state machines and how successful agent communications rely upon the protocols that each agent executes [7]. Singh and colleagues show how protocols among agents can be encoded as commitment machines and automatically executed [4, 19].

Based on the foregoing discussion, the main proposal of this paper is to incorporate commitments into service specifications. Doing so enables service requests and responses to be updatable and cancellable, effectively making agent interactions longer than single episodes of service delivery. Below, we use a variation of the classical travel agent system as our running example. We first derive the necessary domain-level Web services and next identify the commitments underlying each service request or response message. Further, we show that the essential states of the service protocol correspond to the creation, updating, cancellation, or fulfillment of each commitment. We conclude that the commitment enabled Web service interactions will produce more flexible and reliable business process models.

This paper is organized as follows. Section 2 introduces some background on commitments. Section 3 describes Web services and their composition and choreography. Section 4 illustrates how to incorporate commitments into Web service descriptions. Section 5 shows our running example and describes the advantages of using the proposed commitment-based approach. Finally, Section 6 compares our work to previous approaches and discusses some future directions.

2 Commitment Concepts

We now present some background on commitments. Intuitively, a commitment may be understood as an obligation from a debtor to a creditor about a particular condition. For debtor x, creditor y, and condition p, the relevant commitment is notated $C(x, y, p)$. Two main variants can be identified for practical purposes of business modeling:

- *Unconditional commitment.* A commitment whose condition is a simple proposition. For example, the commitment

  $$C(\text{TravelAgent, Customer, ConfirmTicket})$$

  states that the travel agent makes a commitment to the customer that the given ticket is confirmed. After making the commitment, the travel agent ought to hold the ticket until the customer pays for it or cancels the booking.

- *Conditional commitment.* A commitment of the form $C(x, y, e \rightarrow p)$, where e is a condition (possibly interpreted as an event) and p is a condition to be brought about (possibly interpreted as an action). Intuitively, p is activated when e becomes true. For example, the commitment

  $$C(\text{Customer, TravelAgent, ConfirmTicket} \rightarrow \text{BuyTicket})$$

states that the customer makes a commitment to the travel agent if the latter confirms a ticket then the former will buy it.

Some operations on commitments are usually implied in each agent message. These operations provide a basis for determining the compliance of each agent to the protocols and agreements [12]. By exchanging commitments, agents can validate each other's behavior and rationally proceed with business transactions.

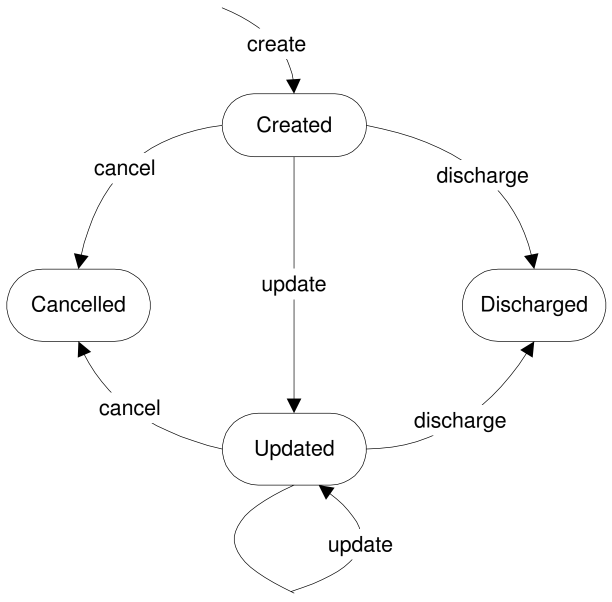

Figure 1: Commitment life cycle

The life cycle of each commitment consists of four states, as described next. Once a commitment is created, the debtor is committed to the creditor that he (the debtor) will eventually make the condition true or perform an action. If this condition finally become true, usually because of the debtor's effort, then the commitment is discharged. Before the commitment is discharged, agents can possibly update the commitment on either its conditions or actions. The update operation gives flexibility in manipulating agents' context to react to any potential requirement changes or exceptions. Finally, agents can also cancel their commitments due to exceptions or simply by their own wills. To cancel a commitment, agents usually encounter penalties that compensate for whatever inconsistencies that they may have introduced. Figure 1 shows the state diagram of the commitment life cycle.

3 Web Services

Web services are modeled as sets of methods wrapped by the SOAP messaging protocol. The common elements associated with a Web service message are:

- Service name
- Service description URI
- Message ID for request and response messages
- Agreement ID for conversational messages
- Inputs for service requests or outputs for service response

3.1 Web Services Interaction Atoms

Based on this message template, different interaction atoms can be formed. Service requesters either synchronously send requests and receive responses or asynchronously send requests and let the service provider send a response (possibly accomplished via a callback to a specified service on the service requester side). Some variations could be that requesters send unsolicited requests without response or that providers send multiple responses for a single request. The following summarizes the major interaction scenarios specified by the W3C [16].

- *Asynchronous (unsolicited one-way) messages.* A SOAP sender sends a message to one or more receivers and does not expect a response. An example of this scenario is sending a piece of news.
- *Synchronized Request and Response.* A SOAP sender sends a request message to a receiver and waits for a response message from the receiver in the same session. An example of this scenario is placing an order online and waiting for an order confirmation.

- *Asynchronous Request and Responses.* A SOAP sender sends a request message to a receiver, but does not block waiting for a response message. After the receiver processes the message, it sends a message back to the original sender. An example is creating an account and receiving email notifications of the user-name and password. A variation of this scenario would be to allow multiple responses to the same request, which allows partial result delivery.

- *Event Notification.* A SOAP sender sends a subscription request to a SOAP receiver and the latter keeps notifying the subscriber of any matching events. An example is subscribing to an online stock trader and receiving stock price updates every 15 minutes. The difference between this scenario and the unsolicited messages is that here the SOAP sender receives notifications after sending its initial request (hence the notifications are not exactly unsolicited).

- *Conversational Messages.* Two parties form a conversation to exchange goods, money, or information. Each party sends a SOAP message to the other, by following a previously agreed-upon protocol. An example of this scenario is that a buyer negotiates with a seller about prices and delivery dates of a particular set of goods, the seller delivers the goods, the buyer pays for the goods, and finally the seller sends a receipt to the buyer.

3.2 Service Composition and Choreography

In a supply chain or a B2B system, multiple parties coordinate with each other to accomplish a global task. This task is composed of the services provided by each party with the services being executed in a specified order. The process of discovering the services and producing a plan of execution is called service composition or choreography.

Traditional workflow technologies combined with the SOAP, UDDI, and WSDL standards have evolved into several service composition and business process model standards. In particular, these are BPML (Business Process Modeling Language [3], BPEL4WS (Business Process Execution Language for Web Services) [2], and WSCI (Web Service Choreography Interface) [17]. These standards specify how services interact and coordinate with each other to achieve a business process flow.

While ensuring the normal execution sequence, the standards also specify the exception handling strategies. For example, BPEL4WS defines a compensation operation for each service so that if the execution of the service has to roll back, its corresponding compensation operation needs to be performed. This technique has the following advantage over traditional transactions, since Web services are inherently distributed and business processes correspond to long-lived transactions. The distribution and autonomy of business participants, and the long-lived nature of business processes render it impossible in practice to support the well-known ACID properties (atomicity, consistency, isolation, and durability) of traditional transactions [9] across multiple parties. However, the compensation mechanism of BPEL4WS is still not the most desirable approach as we explain in the next section.

4 Augment Service Description with Commitments

In the scenarios that the above standards deal with, we can see that each of the messages occurs only once according to the service being requested. Once services are delivered, there is no mechanism to change the results or even to update the original requests. Usually, these kinds of changes are specified in different methods or services, say, something like `UpdateTripSchedule()` or `ChangeCarRental()`. However, due to the complexity of domain level logic, we cannot simply add such methods or services without exploding the size of the service models.

Similar to the above, the compensation mechanism also introduces more service operations. For example, the compensation operation for `RentCar()` would be the operation `CancelRentCar()`. In such a manner, there would be as many compensations as there are service methods. Adding the compensations obviously add complexity to service descriptions. Further, since normal service operations and their compensations become separate operations, it is a burden to maintain their associations whenever there is a change of interface or of the underlying implementations.

The major cause of the above problem is the lack of a high-level abstraction that can relate the update and cancellation messages to the given service so that we do not need to expand service operations. Incorporating commitments into service models enables us to relate the updates and cancellations naturally to the appropriate services.

Agents request and provide services by making commitments. The creation of the commitments corresponds to sending service requests and delivering services. This is mapped to the traditional service model. However, unlike traditional services that end their transaction when a result is delivered, in our approach, it is just the beginning of the life cycle of commitments. Agents can update or cancel these commitments as often as required by the given business logic. These updates and cancellations automatically modify the underlying service requests and responses.

The involvement of commitments not only helps reduce the complexity of service descriptions, but also adds constraints and conditions that agents promise to keep true, so that transaction histories can be recorded and agent behavior can be validated. If there is any violation of service agreements, it can be detected by checking the appropriate commitment conditions [12].

To incorporate commitments, we need to modify service definitions to support commitment operations, by doing which the services become updatable and cancellable without further methods being added. Figure 2 shows the new service model expressed in the Unified Modeling Language (UML).

Service requesters and providers communicate with each other by sending *Messages*. Each *Message* includes the *Service* being requested or provided, the *InputItem* or *OutputItem* to be passed or delivered, the *Commitment* involved, and the *CommitmentOp* that manipulates the *Commitment*. Each *Commitment* has two associated *Roles*, debtor and creditor. To make a valid *Commitment*, agents must agree to perform the commitment's roles [14]. We capture *Role* as an independent entity to explicitly document the parties involved in a service episode with its associated commitments. Doing so enables us to let the agents change roles. For example, an agent may delegate some of its commitments to another agent who was not initially involved in the service episode.

A *Commitment* can be of two types, *RequesterCommitment* and *ProviderCommitment*. A *RequesterCommitment* usually conveys a requester agent's willingness to perform some actions for his provider if the

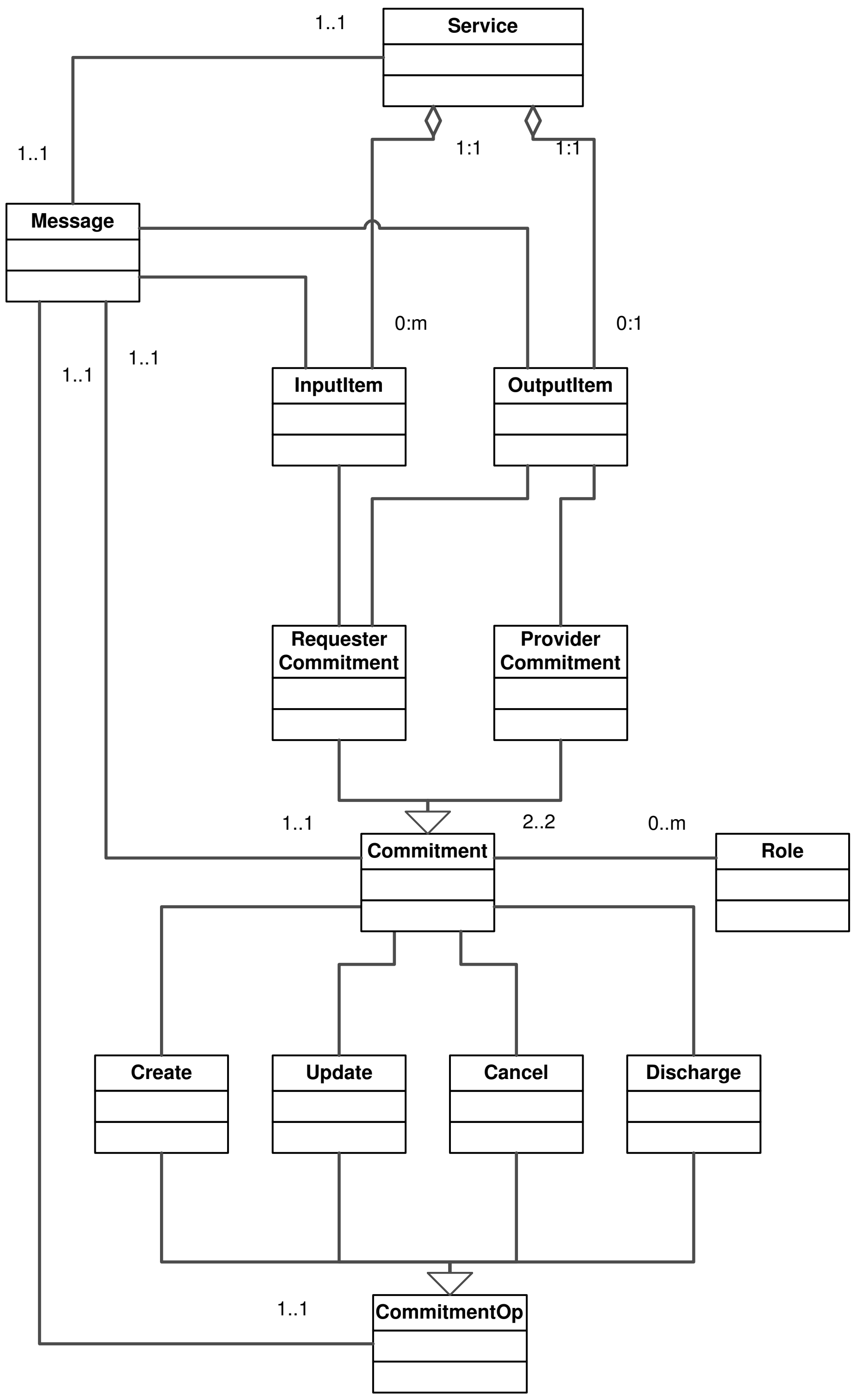

Figure 2: Service model expressed in UML

latter will satisfy the former's service request. It is associated with both the *InputItem* and *OutputItem* in that it passes the *InputItem* to the service provider and also uses the properties of the *OutputItem* to specify commitment conditions. For example, a traveler may commit to paying an airline if the latter finds him a flight. A *ProviderCommitment* is created by a service provider who commits to the satisfaction to his requester. For example, an airline would commit to rebooking the traveler if the flight schedule changes (and letting him know of the change). It is associated only with the *OutputItem* since it only applies to what the provider produces for the requester. For this reason, the two kinds of commitments do not have the same structure. The following shows the message templates of different commitment operations performed on the two types of commitments.

4.1 Creation of RequesterCommitment

The creation of a commitment for requesting a service. It should contain all initial input data required by the service. The commitment involved specifies the liabilities on the requester upon delivery of the service or the cancellation of the service request.

```
⟨env:Header⟩
   ⟨m:ServiceName⟩
   ...
   ⟨/m:ServiceName⟩
⟨/env:Header⟩
⟨env:Body⟩
   ⟨CommitmentOp type="Creation"⟩
   ⟨Input_1⟩...⟨/Input_1⟩
   ...
   ⟨Input_n⟩...⟨/Input_n⟩
   ⟨RequesterCommitment⟩
```

```
        ⟨Id⟩...⟨/Id⟩
        ⟨Condition⟩...⟨/Condition⟩
        ⟨Action⟩...⟨/Action⟩
    ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

4.2 Update of RequesterCommitment

The update of a commitment for the service requested. It passes part of the initial input data that need to be updated.

```
⟨env:Header⟩
    ⟨m:ServiceName⟩
    ...
    ⟨/m:ServiceName⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Update"⟩
    ⟨UpdateInput_1⟩...⟨/UpdateInput_1⟩
    ...
    ⟨UpdateInput_m⟩...⟨/UpdateInput_m⟩
    ⟨RequesterCommitment⟩
        ⟨Id⟩...⟨/Id⟩
        ⟨Condition⟩...⟨/Condition⟩
        ⟨Action⟩...⟨/Action⟩
    ⟨/RequesterCommitment⟩
```

```
⟨/env:Body⟩
```

4.3 Cancellation of RequesterCommitment

The cancellation of a commitment for the service requested. It voids the original service request and also takes any liabilities if required.

```
⟨env:Header⟩
   ⟨m:ServiceName⟩
   ...
   ⟨/m:ServiceName⟩
⟨/env:Header⟩
⟨env:Body⟩
   ⟨CommitmentOp type="Cancel"⟩
   ⟨RequesterCommitment⟩
      ⟨Id⟩ ... ⟨/Id⟩
      ⟨Condition⟩...⟨/Condition⟩
      ⟨Action⟩...⟨/Action⟩
   ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

4.4 Discharge of RequesterCommitment

The discharge of a commitment for a requested service. It releases any commitment associated with the original service request and ends the history for this transaction.

```
⟨env:Header⟩
```

```
  ⟨m:ServiceName⟩
  ...
  ⟨/m:ServiceName⟩
⟨/env:Header⟩
⟨env:Body⟩
  ⟨CommitmentOp type="Discharge"⟩
  ⟨RequesterCommitment⟩
    ⟨Id⟩...⟨/Id⟩
    ⟨Condition⟩...⟨/Condition⟩
    ⟨Action⟩...⟨/Action⟩
  ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

4.5 Creation of ProviderCommitment

The creation of a commitment for service delivery. It should contain all initial output data required by the service response. The commitment involved specifies the conditions that need to be satisfied upon service delivery.

```
⟨env:Header⟩
  ⟨m:ServiceResponseName⟩
  ...
  ⟨/m:ServiceResponseName⟩
⟨/env:Header⟩
⟨env:Body⟩
  ⟨CommitmentOp type="Creation"⟩
```

$\langle Output_1 \rangle \ldots \langle /Output_1 \rangle$

$\ldots$

$\langle Output_n \rangle \ldots \langle /Output_n \rangle$

⟨ProviderCommitment⟩

$\langle Id \rangle \ldots$ ⟨/Id⟩

⟨Condition⟩ ... ⟨/Condition⟩

⟨Action⟩ ... ⟨/Action⟩

⟨/ProviderCommitment⟩

⟨/env:Body⟩

4.6 Update of ProviderCommitment

The update of a commitment for a delivered service. It contains part of the output data that needs to be updated because some changes occur within the service provider or in response to the chain of updates triggered from other services.

⟨env:Header⟩

⟨m:ServiceResponseName⟩

...

⟨/m:ServiceResponseName⟩

⟨/env:Header⟩

⟨env:Body⟩

⟨CommitmentOp type="Update"⟩

$\langle UpdateOutput_1 \rangle \ldots \langle /UpdateOutput_1 \rangle$

$\ldots$

$\langle UpdateOutput_m \rangle \ldots \langle /UpdateOutput_m \rangle$

```
    ⟨ProviderCommitment⟩
        ⟨Id⟩...⟨/Id⟩
        ⟨Condition⟩...⟨/Condition⟩
        ⟨Action⟩...⟨/Action⟩
    ⟨/ProviderCommitment⟩
⟨/env:Body⟩
```

4.7 Cancellation of ProviderCommitment

The cancellation of a commitment for a delivered service. This happens when the service results are no longer available. The service has to be canceled.

```
⟨env:Header⟩
    ⟨m:ServiceResponseName⟩
    ...
    ⟨/m:ServiceResponseName⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Cancel"⟩
    ⟨ProviderCommitment⟩
        ⟨Id⟩...⟨/Id⟩
        ⟨Condition⟩...⟨/Condition⟩
        ⟨Action⟩...⟨/Action⟩
    ⟨/ProviderCommitment⟩
⟨/env:Body⟩
```

4.8 Discharge of ProviderCommitment

The discharge of a commitment for a delivered service. It releases any commitment associated with the service delivery and ends the history for this transaction.

```
⟨env:Header⟩
   ⟨m:ServiceResponseName⟩
   ...
   ⟨/m:ServiceResponseName⟩
⟨/env:Header⟩
⟨env:Body⟩
   ⟨CommitmentOp type="Discharge"⟩
   ⟨ProviderCommitment⟩
      ⟨Id⟩...⟨/Id⟩
      ⟨Condition⟩...⟨/Condition⟩
      ⟨Action⟩...⟨/Action⟩
   ⟨/ProviderCommitment⟩
⟨/env:Body⟩
```

Once commitments are incorporated into service specifications, we are able to examine an agent interaction specification or activity flow diagram to identify the Web services involved and their commitments, and to construct the corresponding messages in the communications. One of the contributions of this approach is to simplify the Web service semantics using commitments, so that service composition can follow a set of commitment patterns [18]. Commitment patterns accommodate a generic way to deal with exceptions, or any input and output changes.

5 Running Example

Here we introduce an extended example, which is based on a travel planning scenario. A customer (or passenger P) calls his travel agent (T) to book a trip. P makes a commitment that if T books the trip for him, he will pay for the trip as well as any processing costs (usually P only pays for the flight ticket to T and pays for hotel and rental car when he shows up for the trip). Upon receiving the order, T sends requests to airline (A), hotel (H), and car rental (R) agents to reserve air tickets, hotels, and cars, respectively. T also makes commitments that if A accepts his requests and if P purchases the trip, then he will pay them. If A finds an available flight, he will make a commitment to reserve it. So will H and R, if they have available spots. Eventually, if all goes well, T will make a commitment to P to confirm the trip. However, P may cancel all or part of the trip. For instance, P may cancel the car rental if he will get a ride with a colleague. In this case, P updates his request; that is, he updates his original commitment. This update may cause T to update and cancel some of his (T's) commitments.

Figure 3 shows the sequence diagram of this example. From the diagram, we can identify the following Web services.

- *Travel Agent Service*

```
Service Request Name:  BookTrip
Input:
   Passenger, Departure, Arrival, Hotel, Car
Service Response Name:  BookTripResponse
Output:
   FlightTicket, Lodging, Car Rental
```

- *Airline Ticketing Service*

```
Service Request Name:  BuyTicket
```

```
Input:
   Passenger, Departure, Arrival
Service Response Name:  BuyTicketResponse
Output:
   FlightTicket
```

- *Hotel Reservation Service*

```
Service Request Name:  ReserveHotel
Input:
   Passenger, Departure, Arrival, Hotel
Service Response Name:  ReserveHotelResponse
Output:
   Lodging
```

- *Car Rental Service*

```
Service Request Name:  RentCar
Input:
   Passenger, Departure, Arrival, Car
Service Response Name:  RentCarResponse
Output:
   CarRental
```

- *Customer Inquiry Service*

```
Service Request Name:  QueryCustomer

Input:

   Alternative Hotel, ...

Service Response Name:  QueryCustomerResponse

Output:

   Yes/No
```

We can also derive the commitments underlying each request or response messages. An algorithm for generating these commitments can be found in [15]. The basic idea behind this algorithm is to convert the sequence diagram into a conversation table by identifying the speech act expressed by each message and marking the relationships between pairs of messages. The communicative acts considered are Request, Refuse, Commit, Question, Inform, and Act. The relations considered are Respond, Reply, Resolve, Complete, and Update. From the conversation table, we then derive the requester and provider commitments by capturing the relational paths. For example, any Request message expresses a Requester commitment and its discharge conditions and actions can be determined in the message that Completes the original Request message as well as the messages to which it Responds.

- C_1 = C(P, T, C(T, P, SendItinerary)$\rightarrow$Pay(P, T))
- C_2 = C(T, A, C(A, T, Confirm) $\wedge$ Pay(P, T) $\rightarrow$Pay(T, A))
- C_3 = C(T, H_1,TBD(To Be Decided))
- C_4 = C(T, R, TBD)
- C_5 = C(A, T, Confirm)
- C_6 = C(H_1, T, Confirm)
- C_7 = C(R, T, Confirm)

- C_8 = C(T, P, SendItinerary)
- C_9 = C(T, H_2, TBD)
- C_{10} = C(H_2, T, Confirm)

Notice that commitment C_3, C_4 and C_9 have undecided conditions which are marked as TBD. The reason why they are undecided is because there is not sufficient detail in the messages included in the diagram to generate the conditions. For example, for commitment C_4, since there is no message from agent T that Completes message 4 in Figure 3, which creates commitment C_4, we do not know what agent T will do after the customer pays for the trip. Usually these incomplete commitments are finalized by incorporating domain-level knowledge. For example, agent T may need to make sure the customer gives a 24 hour notice if he cannot show up at the car rental agency; otherwise, he will be liable for a no-show charge.

Now we give some message examples which show how the above services and commitments can be specified using the templates described in the previous section.

5.1 Book Trip

A customer requests a BookTrip service from a travel agent and also commits to pay for the trip if the travel agent sends him a satisfactory itinerary. This is the creation of a RequesterCommitment. See message 1 in Figure 3.

```
⟨env:Header⟩
    ⟨m:BookTrip⟩
    ...
    ⟨/m:BookTrip⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Creation"⟩
    ⟨Passenger⟩ John Doe ⟨/Passenger⟩
    ⟨Departure⟩ RDU, 01/24/2003 ⟨/Departure⟩
    ⟨Arrival⟩ LAX ⟨/Arrival⟩
    ⟨Hotel⟩ Embassy Suite ⟨/Hotel⟩
    ⟨Car⟩ Hertz ⟨/Car⟩
    ⟨RequesterCommitment⟩
        ⟨Id⟩ C1 ⟨/Id⟩
        ⟨Condition⟩ C(T, P, SendItinerary) ⟨/Condition⟩
        ⟨Action⟩ Pay(P, T) ⟨/Action⟩
    ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

5.2 Buy Ticket

Upon receiving the BookTrip request from the customer, the travel agent sends a request to the airline agent to buy a ticket. The travel agent also makes a commitment that if the airline confirms a ticket and the customer pays for it, then he will pay the airline agent for the order. See message 2 in Figure 3.

```
⟨env:Header⟩
    ⟨m:BuyTicket⟩
    ...
    ⟨/m:BuyTicket⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Creation"⟩
    ⟨Passenger⟩ John Doe ⟨/Passenger⟩
    ⟨Departure⟩ RDU, 01/24/2003 ⟨/Departure⟩
    ⟨Arrival⟩ LAX ⟨/Arrival⟩
    ⟨RequesterCommitment⟩
        ⟨Id⟩ C_2 ⟨/Id⟩
        ⟨Condition⟩ C(A, T, Confirm) ∧ Pay(P, T) ⟨/Condition⟩
        ⟨Action⟩ Pay(T, A) ⟨/Action⟩
    ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

5.3 Send Itinerary

The travel agent sends the itinerary to the customer and makes a commitment that he will reserve the itinerary. The creation of a commitment for service delivery should contain all initial output data required by the service response. The commitment involved specifies the conditions that need to be satisfied upon service delivery. See message 8 in Figure 3.

```
⟨env:Header⟩
    ⟨m:BookTripResponse⟩
    ...
    ⟨/m:BookTripResponse⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Creation"⟩
    ⟨FlightTicket⟩ AA1296, 0124/2003 ⟨/FlightTicket⟩
    ⟨Lodging⟩ Embassy Suite, Non Smoking ⟨/Lodging⟩
    ⟨CarRental⟩ Hertz, Full Size, Never Lost ⟨/CarRental⟩
    ⟨ProviderCommitment⟩
        ⟨Id⟩ C8 ⟨/Id⟩
        ⟨Condition⟩N/A⟨/Condition⟩
        ⟨Action⟩SendItinerary⟨/Action⟩
    ⟨/ProviderCommitment⟩
⟨/env:Body⟩
```

5.4 Update Trip

The customer wishes to update his itinerary by canceling or changing some reservations. This maps to an update of the RequesterCommitment pattern. The following example shows a request to cancel car rental. See message 9 in Figure 3.

```
⟨env:Header⟩
    ⟨m:BookTrip⟩
    ...
    ⟨/m:BookTrip⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Update"⟩
    ⟨Car⟩ Cancel Rental ⟨/Car⟩
    ⟨RequesterCommitment⟩
        ⟨Id⟩ C1 ⟨/Id⟩
        ⟨Condition⟩ C(T, P, SendItinerary) ⟨/Condition⟩
        ⟨Action⟩ Pay(P, T) ⟨/Action⟩
    ⟨/RequesterCommitment⟩
```

```
⟨/env:Body⟩
```

5.5 Update Itinerary

The travel agent updates the itinerary after consulting with the customer for reserving with an alternative hotel agent (H_2) since the first hotel agent (H_1) cancels the hotel reservation. This case maps to the update of ProviderCommitment pattern. See message 11 in Figure 3.

```
⟨env:Header⟩
    ⟨m:BookTripResponse⟩
    ...
    ⟨/m:BookTripResponse⟩
⟨/env:Header⟩
⟨env:Body⟩
    ⟨CommitmentOp type="Update"⟩
    ⟨Lodging⟩ Changed to Marriott, Non Smoking, King Size
    ⟨/Lodging⟩
    ⟨ProviderCommitment⟩
        ⟨Id⟩ C8 ⟨/Id⟩
        ⟨Condition⟩N/A⟨/Condition⟩
        ⟨Action⟩SendItinerary⟨/Action⟩
    ⟨/ProviderCommitment⟩
⟨/env:Body⟩
```

5.6 Pay for the Trip

Eventually the customer is satisfied with the itinerary and pays for the trip. This message maps to the discharge of the commitment in which the action of the commitment is performed. See message 18 in Figure 3.

```
⟨env:Header⟩
   ⟨m:BookTrip⟩
   ...
   ⟨/m:BookTrip⟩
⟨/env:Header⟩
⟨env:Body⟩
   ⟨CommitmentOp type="Discharge"⟩
   ⟨RequesterCommitment⟩
      ⟨Id⟩ C1 ⟨/Id⟩
      ⟨Condition⟩ C(T, P, SendItinerary) ⟨/Condition⟩
      ⟨Action⟩ Pay(P, T) ⟨/Action⟩
   ⟨/RequesterCommitment⟩
⟨/env:Body⟩
```

6 Discussion

The key strength of Web services lies in enabling method invocation and service requests, using established standards such as HTTP and XML, so that heterogenous systems can communicate with each other and participate in distributed business processes. One of the major research challenges for the expansion of Web services is the ability to enable additional expressive power in models while maintaining simple service descriptions and hiding the implementation complexity into service providers. This is because a complicated service description adds complexity to service composition and workflow management, and would yield a rigid system vulnerable to exceptions and heterogeneity.

Current service descriptions treat each service as a group of operations performed on one business transaction. These operations usually manipulate the data involved in the transaction, such as create, update, or cancel. Based on different types of data or different data components, there could be many operations. As described in our travel example, the customer may change flight, hotel or car to his original booking request. To accommodate these changes, the current service description would define operations such as ChangeFlight(), ChangeHotel(), or ChangeCar(), respectively. Adding excessive operations not only splits the business logic (thus making it difficult to maintain), but also complicates service interfaces, which further complicates service composition. Our proposal is that each service should only maintain the description of what initial inputs are required and what initial outputs should be produced. If there is an update or a cancellation of some of the inputs or outputs, we let the commitment operations to convey these changes. Usually the variations of the input and output data indicate what underlying process should be invoked. The service composition would then use different commitment operations to accomplish cancellation or update without being restricted by using rigidly defined service operations.

Current technologies of service composition and choreography offer little more than putting service providers and requesters into a traditional workflow management system. The main difference is that the workflows run via the Internet instead of a proprietary system. In previous research [14, 18], we proposed incorporating a set of commitment patterns into an agent-enabled workflow system to enhance system interoperability and modularity. Here we introduce commitments into what may be thought of as a workflow system enabled by Web services. Commitments enable us not only to record the service delivery history, but also to specify what properties or conditions must be maintained. If any of these conditions is violated, we can always trace down the cause and restore the system to a consistent state again. Therefore, embedding commitments into service descriptions is the key to enabling reliable and persistent Web services.

Commitments are a key element of agent technology. Commitments help specify and determine whether the participating agents are complying with their agreements, and thus help attain a stable and trusted system. When implementing service requesters and providers as agents, we introduce autonomy and flexibility naturally into a supply chain system. Thus, when different business parties communicate via low-level messages, they effectively interact at a high level in terms of their ongoing collaboration and their contracts.

This paper relates two important concepts, namely, commitments and Web services. We showed the limitation of current Web service technologies and proposed a way to incorporate commitment into service descriptions so that agent-enabled service requesters and providers can interact at a high level. Our approach enables the persistence of Web services and makes long transactional systems more reliable and flexible. In future work, we will develop a new service discovery and composition model that exploits commitment-based service descriptions. We will also derive agent behavior models to accommodate this new architecture.

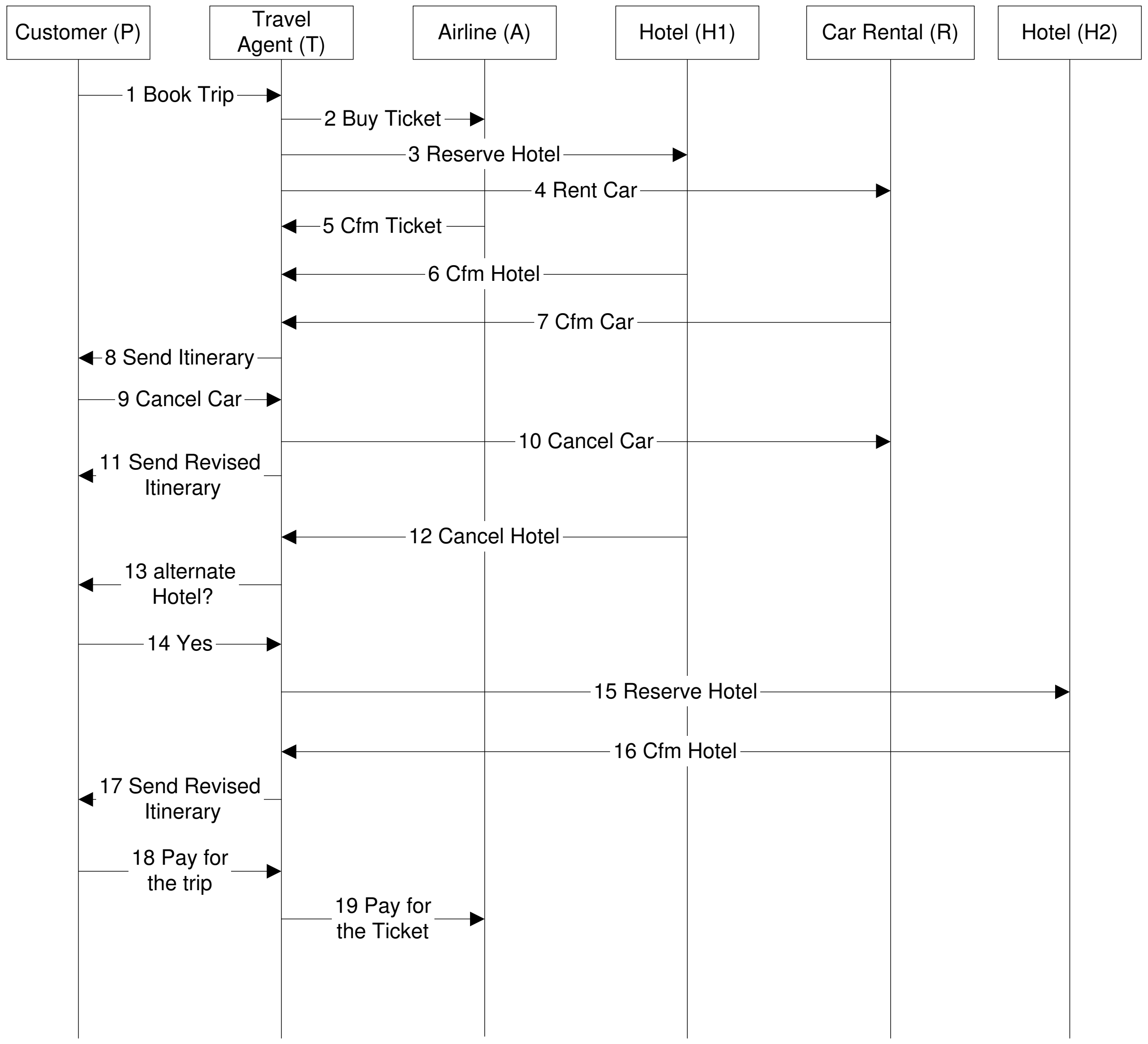

Figure 3: Sequence diagram of the travel agent example

Endnotes

* This work was supported by the National Science Foundation under grants IIS-9624425 and DST-0139037.

We are indebted to Jie Xing for several useful discussions.

References

[1] Box, Don, Aaron Skonnard, and John Lam. *Essential XML: Beyond Markup.* DevelopMentor Series. Addison Wesley, Boston, 2000.

[2] Business process execution language for Web services, version 1.0, July 2002. www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[3] Business process modelling language, November 2002. www.bpmi.org.

[4] Chopra, Amit K. and Munindar P. Singh. Nonmonotonic Commitment Machines. In *Proceedings of the Autonomous Agents Workshop on Agent Languages and Communication Policies*, July 2003.

[5] Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1, 2001. www.w3.org/TR/wsdl.

[6] Colombetti, Marco. A commitment-based approach to agent speech acts and conversations. In *Proceedings of the Autonomous Agents Workshop on Agent Languages and Communication Policies*, pages 21–29, May 2000.

[7] Economou, Gregg, Maksim Tsvetovat, Katia Sycara, and Massimo Paolucci. Implicit commitments through protocol-level semantics. In *Proceedings of the 2nd Workshop on Norms and Institutions in MAS*, 2001.

[8] Fornara, Nicoletta and Marco Colombetti. Operational specification of a commitment-based agent communication language. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 535–542. ACM Press, July 2002.

[9] Gray, Jim and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, 1993.

[10] Jain, Anuj K., Manuel Aparicio IV, and Munindar P. Singh. Agents for process coherence in virtual enterprises. *Communications of the ACM*, 42(3):62–69, March 1999.

[11] UDDI. UDDI technical white paper, 2000. www.uddi.org/pubs/Iru-UDDI-Technical-White-Paper.pdf.

[12] Venkatraman, Mahadevan and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, September 1999.

[13] W3C. SOAP version 1.2, 2001. http://www.w3.org/ TR/2001/WD-soap12-20010709/.

[14] Wan, Feng, Sudhir K. Rustogi, Jie Xing, and Munindar P. Singh. Multiagent workflow management. In *Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business*, 1999.

[15] Wan, Feng and Munindar P. Singh. Commitments and causality for multiagent design. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, July 2003. To appear.

[16] Web services architecture usage scenarios, July 2002. www.w3.org/TR/2002/WD-ws-arch-scenarios-20020730.

[17] Web service choreography interface 1.0, July 2002. wwws.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf.

[18] Xing, Jie, Feng Wan, Sudhir K. Rustogi, and Munindar P. Singh. A commitment-based approach for business process interoperation. *IEICE Transactions on Information and Systems*, E84-D(10):1324–1332, October 2001. Special issue on *Autonomous Decentralized Systems and Systems Assurance*.

[19] Yolum, Pınar and Munindar P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL-01)*, pages 235–247. Springer-Verlag, 2002.