CTP: An Optimistic Commit Protocol for Conversational Transactions

Jinsong Ouyang, Akhil Sahai, Vijay Machiraju
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-20
January 26th , 2001

Internet, conversations, management, transactions

E-services are composite in nature. The composition can be static or dynamic. As these e-services are being deployed by different enterprises, they are cross-domain and federated in nature. To conduct a business task, an e-service undertakes a conversation that spans across multiple e-services, which is often asynchronous and asymmetric. Within a conversation, the unit of business at each e-service is called a component transaction. The component transactions within a conversation form a conversational transaction. Each component transaction is autonomous and independent while it is relative to each other only in the context of a specific conversation. A conversational transaction is committed if each component is committed, or recommitted after compensation. If one or more component transactions abort, the conversational transaction, depending on the business logic, may need to be canceled—the committed component transactions need to be canceled—though not every transaction is cancellable after a certain point. In this paper, we argue that the traditional transaction semantics and mechanisms do not apply well in the e-service world, and propose an optimistic commit protocol to provide transaction support, i.e., all-or-nothing semantics, for conversational transactions.

* Internal Accession Date Only Approved for External Publication

CTP: An Optimistic Commit Protocol for Conversational Transactions

Jinsong Ouyang, Akhil Sahai, Vijay Machiraju
E-Services Software Research Department
HP Laboratories, 1501 Page Mill Road, Palo-Alto, CA 94034

Abstract: Web based services or E-services are being deployed by different enterprises. These E-services in turn use other E-services. This makes them collaborate across management domains. To conduct a business task, an e-service undertakes a conversation that often spans across multiple e-services, and is often asynchronous and asymmetric. Within a conversation, the unit of business at each e-service is called a component transaction. The component transactions within a conversation form a conversational transaction. Each component transaction is autonomous and independent while it is relative to each other only in the context of a specific conversation. A conversational transaction is committed if each component is committed, or re-committed after compensation. If one or more component transactions abort, the conversational transaction, depending on the business logic, may need to be canceled—the committed component transactions need to be canceled—though not every transaction is cancellable after a certain point. In this paper, we argue that the traditional transaction semantics and mechanisms do not apply well in the e-service world, and propose an optimistic commit protocol to provide transaction support, i.e., all-or-nothing semantics, for conversational transactions.

A. INTRODUCTION

An *e-service* is a service available via the Internet that completes tasks, solves problems, or conducts transactions. These e-services are accessible on the Internet at a particular Uniform Resource Locator. An e-service may depend on other e-services. These e-services are termed composite e-services. This composition could be static or dynamic in nature. There is an increasing trend towards dynamic composition where e-services dynamically choose their trading partners or service providers. In figure 1, the root e-service is composed of two sub e-services, one of which in turn is composed of another sub e-service.

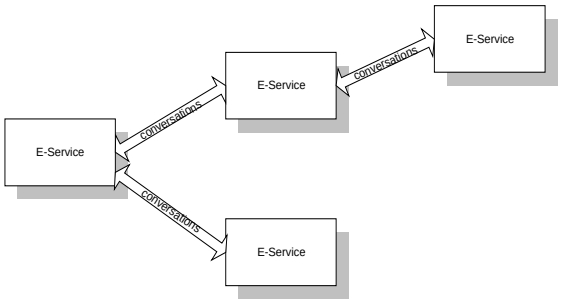

Figure 1. A composite e-service

These e-services are federated in nature as they interact across management domains and enterprise networks. Their implementations could be vastly different in nature. They could be based on CORBA [1], BizTalk [2], COM, E-speak [3] or on other platforms.

E-services interact with each other using asynchronous messages. The interactions between e-services for conducting a specific task create a *conversation* [4, 5]. Within a conversation, we call the unit of business at each e-service a *component transaction*. The component transactions within a conversation form a *conversational transaction*. As independent e-services are created and deployed by autonomous e-service providers, component transactions are independent and autonomous. On the other hand, the component transactions are relative to each other in the context of a particular conversation. For example, a travel agency may need services provided by a flight and a hotel e-service. The business logic of the transactions in the two e-services is independent of each other. The lifecyles of the transactions in the two e-services are totally unrelated, and could be largely different. However, for a particular travel request, the travel conversation will initiate a flight booking transaction and a hotel booking transaction. The two transactions are relative to each other in the context of this particular conversation—The booked flight and hotel are part of a particular travel arrangement.

Conversational transactions have different semantics compared to tranditional transactions. A conversational transaction is committed if each component transaction is committed. If one or more component transactions abort, the conversional transaction, unlike a traditional transaction, may or may not need to be canceled depending on the business logic. This is because the component transactions are independent of each other, and it is totally up to the e-service starting the conversation to decide if *all-or-nothing* semantics should be enforced. A conversational transaction, by itself, does not require all-or-nothing semantics.

This paper proposes an optimistic commit protocol, called conversational transaction protocol (CTP), to provide all-or-nothing property for conversational transactions. The remainder of this paper is organized as follows. Section B describes the all-or-nothing problem for conversational transactions, and presents the related work. Section C presents the system model on which

CTP is built. Section D formally describes the protocol and proves its correctness. Section E proposes a reference API on top of the protocol, and section F draws the conclusion.

B. THE PROBLEM AND RELATED WORK

For traditional distributed transactions, a key technology to guarantee the data consistency is the two-phase commit protocol (2-pc) [6]. It is used to ensure agreement by all parties regarding the outcome of the work (all-or-nothing). Current 2-pc protocols and transaction managers are only suitable for single-domain distributed applications [5]. To address the problem, The Transaction Internet Protocol (TIP) [5] was proposed. TIP is a 2-pc protocol which provides ubiquitous distributed transaction support in a hetergeneous and cross-domain environment. It is made possible by separating the transaction protocol from the application communications protocol (the two-pipe model).

2-pc protocols including TIP do not fit in well with the conversational transaction paradigm. If TIP is used to coordinate a conversational transaction, the executions of component transactions are bundled together in a 2-pc manner. This is not desirable, sometimes not acceptable due to the following reasons.

- The component transactions within a conversation are autonomous and independent, and they are relative to each other only in the context of a particular conversation.
- The lifecycles of different component transactions could be largely different. In the e-service world, individual service requests are usually asynchronous. That is, response to a request may not arrive immediately, and the response time may be unpredictable. There are many factors contributing to the response time such as business logic (long-lived or short-lived), delayed inputs from human operators, the network delays, and so forth. The reponses times of component transactions are independent of each other.
- If a conversational transaction consists of component transactions with largely different response times, TIP will commit the conversational transaction when everyone is ready. That is, the short-lived transactions (their resources) will be blocked for probably an unaceptable amount of time, and the resources cannot be released for processing new service requests. This is usually an undesirable, sometimes an unacceptable option from an autonomous e-service provider's point of view.

Another type of approach [4] would be to let the applications explicitly deal with the possible failure scenarios. The protocol would work as follow.

- Starts a conversation by starting a root transaction upon receiving a service request.
- Each component transaction, after finishing its business logic, commits or aborts the transaction based on its local result.
- Commits the conversational transaction if every component transaction commits.
- If one or more component transactions abort and the committed component transactions need to be canceled, the root transaction sends explicit cancel requests to the e-services to cancel the previous agreements.

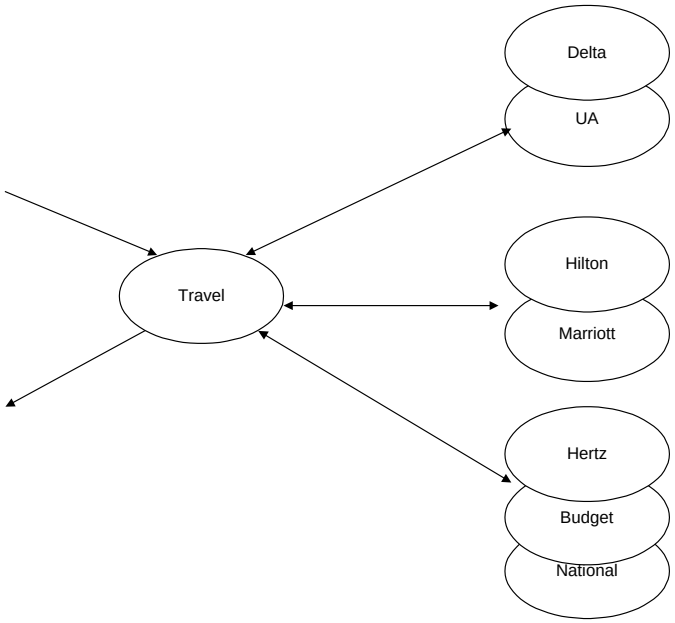

Figure 2. A dynamic travel e-service

The above protocol has two problems. First, it does not provide all-or-nothing semantics. If a conversational transaction needs to be canceled and some committed component transactions cannot be undone, it will produce an inconsistent result. Second, even without all-or-nothing requirement, it works fine only if the composite e-services are static. It would be extremely difficult, if not impossible, when the composition is dynamic. Consider a travel e-service that interacts with flight, hotel, and car rental e-services. As shown figure 2, there are a number of e-services available for flight, hotel, and car rental. For each travel request, the travel e-service dynamically chooses a set of e-services (e.g., UA, Hilton, and Hertz) based on some criteria such as availability, rates, etc. If the travel e-service is involved

in a higher level conversation, the travel transaction becomes a component transaction within that conversation. To enable the cancellation of a travel arrangement, the travel component transaction must remember the context of each travel arrangement (i.e., the request itself and the sub e-service providers for the request). Thus the cancellation request can be forwarded to the corresponding flight, hotel, and car e-services. This would impose tremendous burden on the development of these e-services.

To overcome the limitations of the above approaches, we adopt and extend the idea of *transaction compensation* [10]. The idea was to perform a "semantic undo" of a "erroneoulsy" or "prematurely" committed local transaction if the global transaction aborts due to failures. Based on compensation, a protocol [11] was proposed to enable each participating transaction to commit independently, and be undone when the global transaction aborts. However, the protocol is not sufficient to manage conversational transactions due to the following reasons.

First, a committed component transaction may need to be undone before it passes its deadline and becomes uncancellable. Then instead of just an "undo", a new component transaction may need to be initiated, perform the same business logic, and send its parent an updated service response. We term the new transaction the *transaction in update*. If transactions in update are not handled properly, the correctness of a committed conversational transaction cannot be guaranteed. Consider the example in figure 3, t_0, after receiving responses from t_1 and t_2, sends a global commit. Meanwhile, t_3 has passed its deadline, and has to perform an "undo" and an "update". As illustrated in the figure, a problem occurs in the following scenario. t_2 sends a commit response to t_0 before receiving the updated service response from t_3, t_0 then commits the conversational transaction. If t_2 fails to do the update, it aborts and sends an abort message to t_0 that has already committed. Even if t_0 can, once again, try to cancel the global transaction, it may not be able to do so if t_1, after globally committed, has passed its deadline and become uncancellable.

We define that a conversational transaction is committed when each component transaction has been committed, and each transaction in update (if any) has been caught.

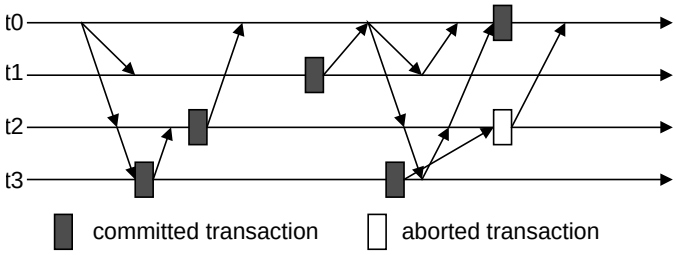

Figure 3. A transaction in update

That is, the result of each transaction in update has been reflected permanently in the conversational transaction once committed. The protocol [11] did not provide mechanisms to catch the transactions in update. Second, a conversational transaction can dynamically span across multiple e-services, and form a spanning tree. In such a spanning tree, each node only knows its immediate parent and children—no node has a global picture of what the tree looks like. The above protocol can only deal with a typical "coordinator-participants" scenario (i.e., a two-tier workflow). That is, the coordinating transaction knows the global picture and deal with each participating transaction directly.

To fill the gap in the existing approachs, we propose CTP to provide transaction support for conversational transactions. The key contribution in this protocol is to enable the catching of transactions in update for conversational transactions that are dynamically composed.

C. SYSTEM MODEL

A conversational transaction is used to identify the units of business spanning across multiple e-services while a conversation is used to capture the interactions between the e-services. The following characteristics usually exist in the conversation-based e-services environment, and our conversational transaction protocol is built in such type of environment.

E-service independence. Each e-service may be developed and deployed by an independent entity. Thus, the business logic of each e-service is completely separated from each other.

E-service Autonomy. Within a conversation, the state and status of a component transaction will neither affect nor be affected by those of its peer transactions.

Dynamic Composition. E-services can be composed dynamically.

DOM based Interactions. E-services essentially use the Document Object Model of interactions.

Unpredictable Response Times. Asynchronous processing, different service lifecycles, human and/or network delays make the response times of component transactions unpredictable. The response time of one component transaction is of little use to predict another one. It is undesirable, sometimes unacceptable to synchronize the executions of the peer e-services.

System and network failures. System and/or network failures may occur during a conversation. We assume that each e-service is able to recover from a failure within a definite amount of time.

Commit, cancellation, and compensation. Depending on the business logic, the e-service starting a conversational transaction can request an all-or-nothing property for the conversational transaction. However services may not always be cancellable. Some payments are not refundable, and booking cannot be canceled after a certain point of time. Thus some scheme is needed to prevent the situation where some committed component transactions become uncancellable while others are still outstanding.

D. CONVERSATIONAL TRANSACTION PROTOCOL

1st. System definitions

We define an e-service conversation as *C*, $C = \{s_0, s_1, \ldots, s_{n-1}\}$ where s_0 is the root e-service starting the conversation, and others are the participating children e-services. A participating e-service may be an immediate child of the root e-service, or may be an immediate child of another child e-service.

Corresponding to *C*, a conversational transaction is defined as *T*, $T = \{t_0, t_1, \ldots, t_{n-1}\}$ where t_0 is the root transaction starting the conversational transaction, and t_i is a component transaction at e-service s_i. A component transaction may be an immediate child of the root transaction, or may be an immediate child of another component transaction.

As e-services interact with each other by exchanging documents, d_{ij} is used to represent a document from t_i to t_j. We define two functions for each component transaction: $f(d_{ij})$ and $\varphi(d[])$. $f(d_{ij})$ is called by t_j to perform the business logic when receiving a request from t_i. $\varphi(d[])$ is called by t_j to cancel the commit status of the service due to request d_{ij}. $d[]$ including d_{ij} are the messages t_j received from other e-services. Based on $d[]$, $\varphi(d[])$ undoes the committed transaction, and performs the business logic one more time without making a commitment (t_j is in update).

A converstional transaction has the following properties:

Property 1 A conversational transaction is committed if each component transaction has been committed and each transaction in update, if any, has been caught.

Property 2 If a conversational transaction is canceled, each of its component transactions is canceled.

CTP is designed to guarantee the properties, and provide transaction support for conversational transactions.

2nd. The techniques

Before presenting the conversational transaction protocol, we describe the techniques used for the protocol: transaction correlation [12, 13] and message logging.

A conversational transaction draws a spanning tree where each node represents a component transaction. To forward a commit or cancellation request, each node is correlated to its parent and children by using a *correlator*. A correlator consists of the CTP handles of a node, its parent, and its immidiate children. The CTP handle is used to identify a component transaction. It contains the transaction's listening endpoint (i.e., its URL) and the transaction identifier. By using the correlators, a CTP message (e.g., a commit or cancellation request) from the root can be forwarded to all the nodes as each node knows where their immediate children are. For the same reason, a CTP message (e.g., a commit or cancellation confirmation) can be delivered all the way back to the root node. The following are the XML [7] definitions of the correlator.

```
<complexType name = "CTPHandleType">
  <element   name = "CTPURL" type = "string"/>
  <element   name = "TranID" type ="decimal"/>
</complexType>
```

```
<complexType name = "CorrelatorType" >
  <element name = "ParentHandle"
           type = "CTPHandleType"
       minOccurs = "0" maxOccurs = "1">
  <element name = "TranHandle"
           type = "CTPHandleType"/>
  <element name = "ChildHandle"
           type = "CTPHandleType"
       minOccurs = "0" maxOccurs = "unbounded">
</complexType>
```

For traditonal transaction processing, a 2-pc protocol is used to coordinate a distributed transaction such that a local transaction cannot commit unless every participating transaction is ready to do so. As described above, it is expensive, if not unacceptable, to do so for conversational transactions. During a conversation, a component transaction should commit or abort independently whenever possible. To enable the cancellation of a committed transaction, there are basically two types of approaches.

One approach is based on the checkpointing of databases and files [8, 9]. The advantage of this approach is that the cancellation of a committed transaction is totally transparent to the business logic. However the disadvantage of this approach is that the complexity is high. It involves modifications to the database/file I/O libraries, which makes it less practical.

The other approach is based on message-logging. Conversational/component transactions are initiated by service requests. When receiving service request d_{ij} from t_i, t_j is initiated, and d_{ij} is logged. Then t_j calls $f(d_{ij})$ to process the request, and commits the transaction if possible. Besides requests, responses received from other e-services also need to be logged. To cancel its commit status, t_j retrieves the logged messages $d[]$ and calls the callback function $\varphi(d[])$ to undo the previous service and perform the business logic one more time. If a failure occurs, t_j aborts; otherwise it is in update until it is informed to commit again.

Compared to the first approach, message-logging is easy to implement, and does not need the involvement from the system. The disadvantage of this approach is that it requires applications to provide a callback function to cancel the commit status of a previously committed service. CTP uses message-logging for the cancellation purpose.

3rd. The protocol

CTP is designed to provide all-or-nothing transactional property for conversational transactions. It has the following features.

- It is distributed. In the cross-domain e-services environment, each e-service is responsible for its own behaviors and commitments. There is no central control point which could otherwise have the global knowledge of a conversation and coordinate all the participating e-services directly. As a result, CTP does not provide a central control point coordinating each participating component. Instead, it relies on transaction correlation to make sure that commit or cancellation actions are taken consistently within a conversational transaction.
- It is partly asynchronous. Unlike 2-pc protocols, CTP allows the component transactions within a conversation to locally commit if they are cancellable. CTP commits the conversational transaction if each component is committed and each transaction in update is caught. Otherwise, CTP will try to automatically cancel the committed component transactions.

A component transaction, if not cancellable, will not commit until receiving a commit request. A locally committed component transaction needs to roll back to, and be blocked at the pre-commit state when it reaches the point beyond which it will not be cancellable. Thus, CTP, in its worse scenario where each component transaction is or becomes uncancellable, will work in the same manner as TIP. However in the normal situation, CTP should outperform TIP as all or most of the component transactions commit independently.

The protocol consists of two parts. The first part specifies the behavior of the root transaction manager (RTM) that starts a particular conversation. The second part specifies the behavior of a component transaction manager (CTM)—how a component transaction participates in a conversational transaction.

The root transaction manager is responsible for starting, committing, or canceling a conversational transaction upon receiving an event from the root e-service or its immediate component transaction managers. The RTM protocol describes how to respond to these events.

- The start event. The RTM starts a root transaction upon receiving a start event from the root e-service. It creates a CTP handle for the root transaction and returns it to the root e-service. The CTP handle must be tagged on each of the request documents sent to other e-services.
- The connection event. After a service request is delivered at a sub e-service and a component transaction is created, the CTM will send its parent a connection request. Then the root/parent transaction will build a connection with the component transaction using their CTPs, and add the child's CTP to the children list of its correlator structure. Note that, due to the asynchronous nature of e-services, the connection is really just a "hand-shake" so that the parent and the child knows the existence of each other.

- The response event. When a component transaction ends, the CTM will send a response to the root/parent manager. Based on the status of the component transaction, the root/parent transaction will update the corresponding child entry in its correlator structure.
- The timeout event. A timeout event will be generated internally if there are outstanding service requests for a certain amount of time. When the event occurs, the transaction manager finds out the outstanding component transactions from its correlator, and sends a "ping" message to the corresponding CTMs. If one or more CTMs do not reply or reply with an error, failures are assumed to have occurred, and an alarm is generated and sent to the local e-service.
- The update event. Before a committed component transaction becomes uncancellable, the CTM sends its parent/root an "update" request. After receiving the request, the root transaction, based on its current status, will send back a reply indicating how the component transaction should proceed: whether to allow the component transaction to perform $\varphi(d[])$. If $\varphi(d[])$ is allowed, the update count of the root transaction is incremented. The update count is used to catch the transactions in transit. It indicates the number of the updated service responses that the root transaction will receive. The root transaction will not commit until those number of updated responses have been received.
- The end event. The RTM ends the root transaction upon receiving an end event from the root e-service. If the root e-service wants to commit the conversational transaction, the RTM sends a "local commit" request to each of its immediate CTMs. If each response is "locally committed", then the RTM sends a "global commit" message and commits the root transaction. If the root e-service wants to cancel the conversational transaction due to an error, the RTM sends a "cancel" request to each of its immediate CTMs. When receiving a "canceled" response from each CTM, the RTM marks the root transaction as "canceled". The RTM will forget the transaction after returning control to the root e-service.

The RTM protocol is shown as follows.

```
RTM()
   When (an event occurs)
      if (start) then
         corr ← new correlator();
         corr.CTPHandle.URL ← RTM.URL;
         corr.CTPHandle.ID ← a new tran ID;
         return corr.CTPHandle;
      else if (connection) then
         connect(corr.CTPHandle,evt.CTPHandle);
         corr.children ← evt.CTPHandle;
      else if (response) then
         if (corr.children[i] = evt.CTPHandle)
         then
            corr.children[i] ← evt.tranStatus;
      else if (timeout) then
         ping(corr.children);
         if (error) then
            alarm(e-service);
      else if (update) then
         if (allowed) then
            corr.updates++;
         send(decision, evt.CTPHandle);
      else if (end) then
         if (commit) then
            send("local_commit",corr.children);
            wait until all responses or timeout
            if (local commit is OK) then
               ret ← commit(corr.CTPHandle);
               if (ret != OK) then
                  goto cancel;
               send("global_commit",
                     corr.children);
               wait until all responses
            else
               goto cancel;
         else if (cancel) then
cancel:     send("cancel", corr.children);
             wait until all responses
end
```

A component transaction manager is responsible for starting, committing, or canceling a component transaction upon receiving an event from the local e-service, its parent CTM/RTM, or its own immediate CTMs. The CTM protocol describes how to respond to these events.

- The start event. When receiving a request, an e-service first checks if there is a CTP handle tagged on the request. If so, it informs local CTM to start a component transaction. Based on the request and its own business logic, the e-service should also let the CTM know if and/or until when the transaction will be cancellable. Then the CTM creates a CTP handle for the transaction, and sends a connection request to the parent transaction manager which will build the connection.
- The connection event. Same as the RTM protocol.
- The response event. Same as the RTM protocol.
- The ping event. When receiving a ping message, the CTM checks the corresponding component transaction's correlator and sees if there are outstanding sub transactions. If so, it sends a "ping" message to the corresponding CTMs. If one or more CTMs do not reply or reply with an error, failures are assumed to have occurred, then the CTM sends

an error reply to its parent. Otherwise, a "in-progress" reply is sent back.

- The end event. The CTM ends the component transaction upon receiving an end event from the local e-service. The transaction aborts if there is an error in itself or some of its component transactions. Otherwise, the CTM optimistically commits the transaction ("self-committed") if it is cancellable, or puts it in a pre-commit state. Then the CTM sends a "ok" or "aborted" response to the parent transaction.
- The checkpoint event. A checkpoint of a transaction is a point beyond which it will not be cancellable. The checkpoint event can be triggered by time and/or some business logic. When it occurs, the CTM checks if there are any self-committed transactions that have reached their cancellation deadlines. For each of these transactions, the CTM sends an "update" request to its parent. If the reply indicates that the local transaction should proceed, the CTM retrieves the logged service requests/responses. Then it calls the transaction's callback function $\varphi(d[])$ to undo the committed transaction, perform the business logic one more time, and put it in the pre-commit state (the transaction is in update). If the reply indicates that the local transaction should abort, the CTM calls $\varphi(d[])$ just to undo the transaction, sends a "cancel" request to the children transactions, and then sends an "aborted" response to the parent transaction. If the reply indicates that a distributed commit has been initiated, nothing should be done at this point.
- The update event. When receiving an "update" request from one of its children, the CTM, if the local transaction's status is "locally committed", sends back a reply indicating a distributed commit has been initiated. Otherwise, it forwards the request to its parent. If the reply is positive, the CTM increments the update count of the local transaction, and changes its status to "pre-commit" if its current status "self-committed". Therefore, the local transaction can expect an updated service response from that child transaction in update, and it will not commit until all the updated responses have been received. Regardless of the content of the reply, the CTM will then forward the reply to the child.
- The local commit event. When receiving a "local commit" message, the CTM checks the status of the corresponding local transaction, and forwards the message to the transaction's immediate children. If each response from the children is "locally committed", the CTM commits the local transaction if it is in the pre-commit state. If the commit is successful, the CTM marks the local transaction as "locally committed", and sends a "locally committed" response to the parent transaction. Otherwise, the CTM aborts/cancels the local transaction as well as its children, and sends back a "aborted" response.
- The global commit event. When receiving a "global commit" message, the CTM marked the local transaction as "globally committed", and forwards the message to the transaction's immediate children. Once all the responses come back, the CTM sends a "globally committed" response to the parent transaction.
- The cancel event. When receiving a "cancel" message, the CTM aborts or cancels the local transaction if its status is not aborted (e.g., pre-commit, self-committed, locally committed). Then it forwards the "cancel" message to its immediate CTMs. Once all the responses come back, the CTM sends a "canceled" response to the parent transaction.

The pseudo code of the CTM protocol is shown as follows.

```
CTM()
   When (an event occurs)
      if (start) then
         corr ← new correlator();
         corr.CTPHandle.URL ← CTM.URL;
         corr.CTPHandle.ID ← a new tran ID;
         corr.parent ← evt.CTPHandle;
         corr.checkpoint ← deadline;
         connect(corr.CTPHandle, corr.parent);
         return corr.CTPHandle;
      else if (connection or response) then
         /* Same as RTM */
      else if (ping) then
         ping(corr.children);
         if (error) then
            send("error", corr.parent);
         else
            send("in-progress", corr.parent);
      else if (end) then
         if (commit) then
            if (checkpoint) then
               commit(corr.CTPHandle);
               corr.status ← "self-committed";
            else
               corr.status ← "pre-commit";
            send("ok",corr.parent);
         else
abort:      send("cancel",corr.children);
            wait until all responses
            abort(corr.CTPHandle);
            corr.status ← "canceled";
            send("aborted", corr.parent);
      else if (checkpoint) then
         for (each local transation)
           corr ← t.correlator;
           if(corr.checkpoint >= δ(deadline) &&
```

```
            corr.status ="self-committed") then
              send("update", corr.parent);
              if (reply = "allowed") then
                d[] ← retrieve(corr.CTPHandle);
                thread(φ, corr.CTPHandle, d[]);
              else if ("not allowed") then
                d[] ← retrieve(corr.CTPHandle);
                φ(d[]);
                goto abort;
      else if (update) then
         if (corr.status = "locally committed")
         then send("wait", evt.CTPHandle);
         else
           send("update", corr.parent);
           if (reply = "allowed") then
            corr.updates++;
            if (corr.status = "self-committed")
            then  corr.status ← "pre-commit";
           send(reply, evt.CTPHandle);
      else if (local commit) then
         if (no error) then
           send("local_commit",corr.children);
           wait until all responses
         if (corr.status = "pre-commit"
         &&  no error) then
           commit(corr.CTPHandle);
         if (no error) then
         corr.status ← "locally committed";
         send("locally-committed",corr.parent);
         if (error) then
            goto abort;
      else if (global commit) then
         corr.status ← "globally committed";
         send("global_commit",corr.children);
         wait until all responses
         send("globally-committed",
                    corr.parent);
      else if (cancel) then
         if (corr.status = "canceled") then
            send("aborted", corr.parent);
         if (corr.status = "pre-commit") then
            goto abort;
         else
            d[] ← retrieve(corr.CTPHandle);
            φ(d[]);
            goto abort;
end
```

Note that the conversational transaction protocol does not specify how a component transaction should be committed locally. It is our intention to separate this from the protocol. Therefore, any existing transaction managers can participate in CTP without any modifications.

To demonstrate how CTP works, figure 4 shows an example where a travel e-service is making a travel arrangement by interacting with a flight, a car, and a hotel e-service. For simplicity, figure 4 only displays the CTP messages between the e-services whereas the e-service-specific messages are not included.

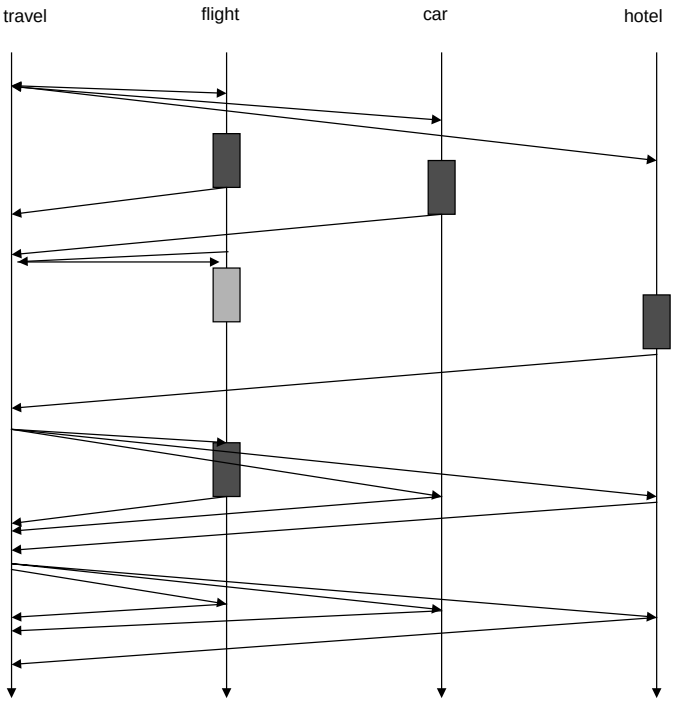

Figure 4. A CTP workflow

In the example, the flight and car e-services book and confirm a flight and a car, then commit the transactions accordingly. Due to some delay at the hotel e-service (e.g., no vacancy), it is about to reach a point when the booked flight cannot be canceled while a hotel room has not been booked. After being informed of the status of the flight transaction, the travel transaction decides to continue this arrangement and knows that it will not only hear from the hotel transaction, but also get a flight update from the flight transaction. When receiving the decision, the flight transaction cancels the confirmed flight. Based on the logged flight request, it will then book another flight and send an update to its parent. But this time, the flight transaction will not commit itself until it hears from the travel transaction. After receiving the flight update and the hotel confirmation, the travel transaction requests each sub transaction to commit—the flight transaction has not committed—the other two have already committed. Once receiving the responses from the sub transactions, the travel transaction sends a "global commit" message. The message informs each participating entity that the conversational transaction has committed, and the local transaction can be discarded.

4th. Proof of correctness

We prove in this section the correctness of the conversational transaction protocol. The protocol must ensure the safety and liveness properties.

Lemma 1 The conversational transaction protocol is deadlock free.

Proof:
First, we prove that the root transaction will commit or cancel a conversational transaction within a definite amount of time. Suppose that root transaction t_0 has initiatied conversational transaction *T*. If t_0 has not received responses from some component transactions within a timeout interval, a timeout event is generated, and a "ping" message is forwarded to each t_i *(0 < i < n)*. If errors are detected, t_0 will abort *T* to avoid a potential deadlock. Otherwise, t_0 will eventually receive the response from each t_i *(0 < i < n)* because the latencies due to asynchronous processing and network delivery are bounded. Following the business logic, t_0 will send two rounds of synchronous messages. If errors are detected in the first phase, t_0 cancels *T*; otherwise it commits *T*. t_0, after every certain interval, will keep sending the final commit or cancel message until it receives the response from each t_i *(0 < i < n)*. This will not lead to a deadlock as we assume each e-service is able to recover from a failure within a definite amount of time. Thus, each t_i *(0 < i < n)* will eventually receive the commit or cancel decision, so will t_0 receive all the responses.

Second, we prove that each t_i *(0 < i < n)* will not be blocked indefinitely. In CTP, t_i behaves independently except for two points where t_i can be blocked: waiting for a reply after sending its parent an "update" request; and waiting for a "global commit" or "cancel" message. For the same reason as above, this will not lead to a deadlock.

Lemma 2 The conversational transaction protocol is livelock free.

Proof:
A livelock would occur if t_i, before receiving the "local commit" message, is committed and undone repeatedly due to the checkpoint events. CTP prevents this by allowing t_i to be undone at most once before t_0 commits or cancels *T*. When a checkpoint event occurs, $\varphi(d[])$ is called to undo the transaction, perform the business logic one more time, and change t_i's status from "self-committed" to "pre-commit". That is, the following checkpoint events will have no impact on t_i. Hence no livelock. t_i will not proceed until receiving a "local commit" or "cancel" messsage. Again, this will not lead to a deadlock, as proved in Lemma 1.

E. THE CTP API

Based on the conversational transaction protocol, we propose an application programming interface for the development of conversation-based e-services. The API is defined in the CTransaction class.

```
public class CTransaction extends Object {
// Public Constructors
        public   CTransaction();
// Public Instance Methods
        public XMLDocument
              push(XMLDocument doc,
                    CTPHandle Handle);
        public CTPHandle
              pull(XMLDocument doc);
        public CTPHandle
              begin(Time Checkpoint);
        public CTPHandle
              begin(CTPHandle Parent,
                    Time Checkpoint);
        public int
              query(CTPHandle Handle);
        public short
              end(CTPHandle Handle,
                    int CompletionType);
}
```

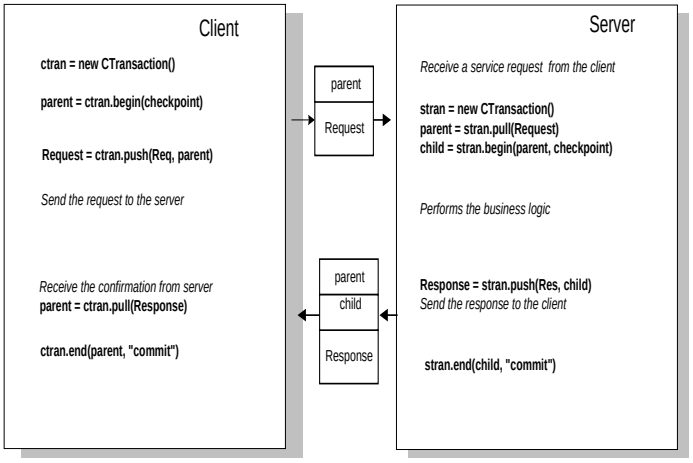

Figure 5. The CTP API

Method push() is used to tag the outgoing service request/response with the CTP handles of the local transaction and its parent (if a response). When receiving a request or response, method pull() is called to log the message, and retrieve the CTP handle of the parent or local transaction. There are two versions of begin(). One version is used to start a root transaction while the other is used to start a component transaction. Method query() is called to get the number of transactions in update. A transaction can commit only after it receives the response from each child transaction

in update. Method end() is called to commit or abort a root/component transaction, depending on the transaction completion type. Figure 5 is a simplest sample demonstrating how the CTP API is used.

F. CONCLUSION

E-services are federated, composite, and autonomous. Due to the nature of e-services, the traditional transaction processing mechanisms do not serve well in the e-service world. We proposed in this paper a new optimistic commit protocol called CTP to provide transaction support for e-service transactions (i.e., conversational transactions). That is to provide all-or-nothing semantics for conversational transactions. A reference API was also proposed to enable the development of the conversation-based e-services.

G. REFERENCES

[1] Object Management Group. *The common object request broker: Architecture and specification*. Revision 2.0, July 1995 http://www.omg.org

[2] D. Rogers. *BizTalk service framework*. Microsoft Corporation. http://www.biztalk.org

[3] Hewlett-Packard Company. *E-Speak Architecture Specification*. Version Beta2.2. December 1999. http://www.e-speak.net/library/pdfs/E-speakArch.pdf

[4] A. Dan and F. Parr. An Object implementation of network centric business service application (NCBSAs): conversational service transactions, service monitor, and an application style. *OOPSLA'97, Business Object Workshop III*.

[5] K. Evans, J. Klein, and J. Lyon. Transaction Internet Protocol – Requirements and supplemental information. 1998. http://www.landfield.com/rfcs/rfc2372.html

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, 1987.

[7] XML at World Wide Web (WWW) Consortium. http://www.w3.org/xml

[8] J. Ouyang and P. Maheshwari. Supporting cost-effective fault tolerance in distributed message-passing applications with file operations. *Journal of Supercomputing*, 14, 207-232, 1999.

[9] R. Strom, S. Yemini, and D. Bacon. A recoverable object store. In *Proceedings of International Conference on System Sciences*, Vol. 2, 215-221, 1988.

[10] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Databases*, Brisbane, pages 95-106, 1990.

[11] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data*, pages 88-97, 1991.

[12] A. Sahai, J. Ouyang, and V. Machiraju. End-to-end e-service transaction and conversation management through distributed correlation. Technical Report HPL-2000-145, Hewlett-Packard Labs, 2000.

[13] Vantage Point Web Transaction Observer http://www.openview.hp.com/products/webtransobserver/