An Approach to Optimistic Commit and Transparent Compensation for E-Service Transactions

Jinsong Ouyang, Akhil Sahai, Vijay Machiraju
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-34
February 23rd , 2001*

E-mail: {jinsongo, asahai, vijaym} @ hpl.hp.com

distributed transaction processing, optimistic commit, transparent transaction compensation, e-services, Internet, e-commerce

Web based services, also termed E-services are either stand-alone or use other web services for performing their tasks. Such E-services that depend on other E-services are termed composite e-services. The composition can be static or dynamic, i.e. either these e-services interact amongst themselves in a pre-negotiated manner or dynamically discover each other and negotiate on the fly. E-service to e-services interaction takes place across enterprises and management domains. 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 related to each other only in the context of a specific conversation. A conversational transaction is committed if each component is committed. If one or more component transactions abort, its parent transaction, depending on the business logic, may abort or start one or more new component transactions. In this paper, we argue that the traditional transaction semantics and mechanisms do not apply well in the e-service world, and propose a new approach to enabling transaction support on the Internet.

* Internal Accession Date Only
Approved for External Publication

1. Introduction

An *e-service* is a web-based service that undertakes transactions, solves problem and does useful task. E-services are accessible at well know Uniform Resource Locator addresses. Some of these E-services are stand-alone and perform their tasks on their own. However, some of these E-services depend on other e-services to perform their tasks. Such composite e-services either know their partner e-services in advance through pre-negotiated understanding, (e.g. web based portal sites like expedia, travelocity) or dynamically discover other e-services and enter into contracts with them. There is an increasing trend towards dynamic composition where e-services dynamically choose their trading partners or service providers [2][3]. In the figure 1, the composite root e-service is composed of two sub e-services, one of which in turn is composed of another sub e-service.

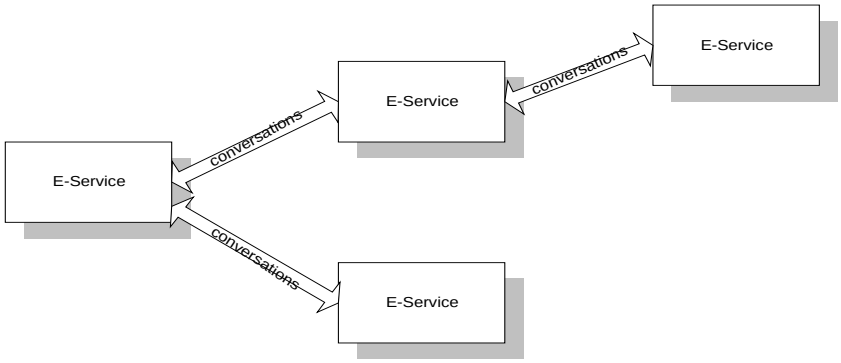

Figure 1. A composite e-service

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 typically 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 lifecycles 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 than traditional transactions. A conversational transaction is committed if each component transaction is committed. If one

component transaction aborts, its parent transaction usually tries a new sub transaction, instead of simply aborting itself. If not every component transaction can commit eventually, the conversational transaction, unlike a traditional distributed 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.

In this paper we propose a new approach called XIP (E-Service Transaction Internet Protocol) for transaction support in e-services. The approach consists of two parts. The first part is an optimistic commit protocol to systematically enable the Internet transaction semantics (e.g., commit, abort, retry, cancel). The second part is a transaction reversal algorithm for the cancellation/compensation of committed component transactions. The reversal of transactions is done by intercepting the invocations (e.g., ODBC [6], OLE DB [6], JDBC [7]) to the backend database servers. Thus, unlike the existing compensation-based approaches [10], the cancellation is transparently done without any involvement from the applications.

2. 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) [8]. 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 that provides ubiquitous distributed transaction support in a heterogeneous 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, the 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 response 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 unacceptable 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.

- Start 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.

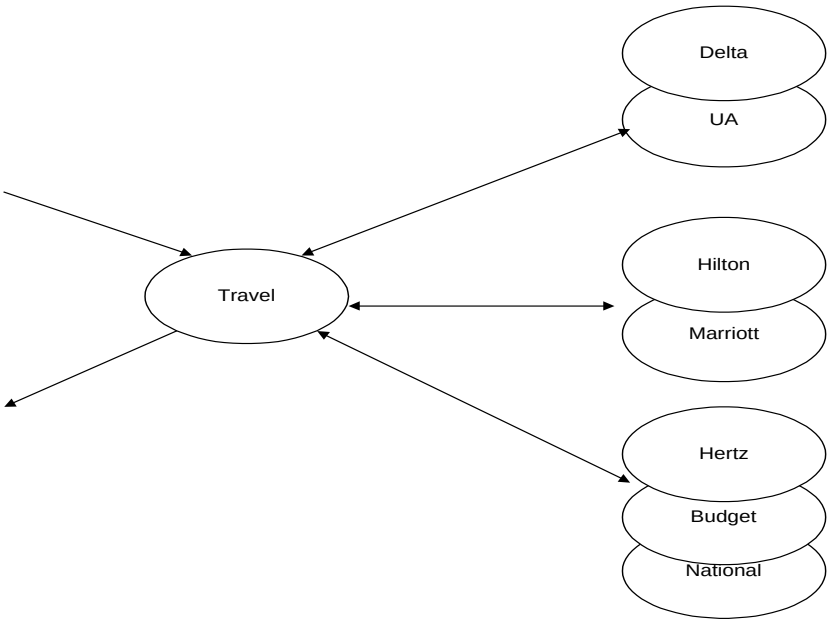

Figure 2. A dynamic travel e-service

The above protocol has two problems. First, it cannot guarantee all-or-nothing semantics when required. 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 e-service must remember the context of each travel arrangement (e.g., 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 an e-service.

There have been some optimistic commit protocols [10] based on *transaction compensation* [9]. The idea was to perform a "semantic undo" of an "erroneously" or "prematurely" committed local transaction if the global transaction aborts due to failures. The concept has also been used in some mobile transaction models [11, 12, 13]. However, the protocols and models did not cover some unique aspects of distributed transactions across the Internet. In the e-service world, a committed transaction, if cancellable, usually has a deadline. That is, it is cancellable until the deadline is reached. Thus even if no failures have occurred, a "semantic undo" may still be needed before the deadline is passed. When a transaction is compensated, one of the following scenarios can occur, depending on different protocols. First, its parent transaction that may or may not have been committed may choose to re-negotiate a new sub transaction with a different e-service, instead of simply aborting itself. It is likely that, with the new sub transaction, a different result will be produced. Also the new result must be propagated to its parent transaction, which will, in turn, change the result of its parent, and eventually that of the root transaction. Second, the set of transactions relative to the compensated transaction need to be compensated. The set consist of all of its children as well as its predecessors that have been committed. Then its nearest uncommitted predecessor (e.g., the root transaction) may choose to initiate a new path to perform the corresponding sub task. While the second scenario potentially involves more rollbacks, the first scenario is more difficult to deal with—a committed transaction need be updated upon receiving an updated response—and this imposes a strong coding requirement on the development of e-services.

Things can get complicated if the above situations occur after the root transaction initiates a global commit. It is essential that the states of the new sub transactions must be reflected permanently in the global transaction once committed.

Another problem for the existing compensation-based approaches is that the applications have to explicitly provide compensation functions. This imposes non-trivial burden on the development of e-services.

The main contribution of our approach is to, by assuming a certain computing and communication pattern for e-services, provide an optimistic commit protocol with transparent transaction compensation.

3. 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 XIP protocols are 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. E-services communicate with each other only via asynchronous documents.

Stateless transactions. Component transactions are stateless. Once a transaction is committed, its state is recorded in the backend databases, and/or sent to other e-services via a document.

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, retry, cancellation, and compensation. Depending on the business logic, the e-service starting a transaction can request an all-or-nothing property with respect to its sub transactions. The transaction can re-negotiate a new sub transaction if one is aborted or compensated. If "all" semantics cannot be achieved, the e-service can choose to cancel/compensate the committed sub transactions. 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.

4. Optimistic commit with transparent compensation

4.1 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 that 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 that of another component transaction.

E-services interact with each other by exchanging asynchronous 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 $\varphi(t_i)$. $f(d_{ij})$ is called by t_j to perform the business logic when receiving request d_{ij} from t_i. $\varphi(t_i)$ is the compensation function used to cancel the committed operations for t_i. Two attributes are defined for each transaction *t*: *t.deadline* and *t.status*. *t.deadline* defines whether/when *t*, once committed, is/becomes uncancellable. *t.status* defines *t*'s lifecycle that has five states: *pre-commit*, *prepared*, *local-committed*, *global-committed*, *aborted*, *canceled*.

4.2 The techniques

Before presenting the XIP protocol and algorithm, we describe the techniques used for the protocols: transaction correlation [14, 15] and the transaction reversal.

A conversational transaction forms 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 XIP handles of a node, its parent, and its immediate children. The XIP 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, an XIP 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, an XIP message (e.g., a commit confirmation) can be delivered all the way back to the root node. The following are the XML [16] definitions of the correlator.

```
<complexType name = "XIPHandleType">
  <element   name = "XIPURL" type = "string"/>
  <element   name = "TranID" type ="decimal"/>
</complexType>
```

```
<complexType name = "CorrelatorType" >
  <element name = "ParentHandle" type = "XIPHandleType"
                                                minOccurs = "0" maxOccurs = "1">
  <element name = "TranHandle"  type = "XIPHandleType"/>
  <element name = "ChildrenHandles" type = "XIPHandleType"
                                          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.

The first type of approach is that applications provide a compensation function for each business function. The function is called back by the transaction manager when a committed transaction need be canceled.

Because of the problem in the first approach, we propose a new approach called transparent transaction compensation. The approach is based on the concept of *transaction reversal*, the language-level compensation instead of the business-logic level compensation. The idea is to generate a reversing function for each update statement against the database (e.g., INSERT, UPDATE, DELETE), and log the associated data. At the time of cancellation, the transaction's effects can be removed by calling the corresponding reversing functions with the logged data. To make it work, the transaction-reversing algorithm assumes the following without losing the generality of e-services: the e-service transactions are stateless, and they use a standard programming interface (e.g., JDBC, ODBC, OLE DB) to access backend databases.

4.3 The optimistic commit protocol

The optimistic commit protocol in XIP is used to orchestrate the committing of a conversational transaction. It has the following features.

- Decentralized. In the cross-domain e-services environment, each e-service is responsible for its own behaviors and commitments. There is no central control point that could otherwise have the global knowledge of a conversation and coordinate all the participating e-services directly. As a result, the protocol 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.
- Asynchronous. Unlike 2-pc protocols, it allows the component transactions within a conversation to locally commit if they are cancellable. A locally committed transaction will be canceled if a global commit cannot be issued.
- Minimal coding requirement. The protocol does not require the applications to deal with the updating of previously committed transactions. And transparent transaction compensation is achieved by intercepting/augmenting the data access API with the transaction-reversing algorithm. Thus, no additional programming effort is induced, compared to the traditional transaction programming.

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 an XIP handle for the root transaction and returns it to the root e-service. The XIP 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 started, the CTM will send its parent a connection request. Then the root/parent transaction will build a connection with the component transaction using their XIP handles, and add the child's handle to the children list in 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 manager will update the corresponding child entry in the 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 compensation event. Before a committed component transaction becomes uncancellable, the CTM sends its parent a "compensation" request. After receiving the request, the RTM, if the root still has incomplete children that were not started due to "compensation" requests, sends back a confirmation so that the CTM can undo the component transaction by calling $\varphi(t)$. If each pending child was started due to a "compensation" request, the RTM cancels the conversational transaction to avoid a potential livelock. If there is no outstanding child transaction, the RTM sends a "denied" response in which case the root is about to, or has initiated a global commit. If the compensation is allowed, the RTM removes the child entry in the correlator structure, and informs the root e-service of the compensation. It is up to the root e-service to decide whether to start a new component transaction.
- 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 commits the root transaction, and sends a "global commit" request to each of its immediate children whose status is not "aborted". After receiving each response, the RTM sets the root transaction to "global-committed". 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.XIPHandle.URL ← RTM.URL;
         corr.XIPHandle.ID ← a new tran ID;
         return corr.XIPHandle;
      else if (connection) then
         connect(corr.XIPHandle,evt.XIPHandle);
         corr.children ← evt.XIPHandle;
      else if (response) then
         if (corr.children[i] = evt.XIPHandle)
         then
            corr.children[i].status ← evt.tranStatus;
      else if (timeout) then
         ping(corr.children);
         if (error) then
            alarm(error);
      else if (compensation) then
         if (no pending children) then
            decision = "denied";
         else if (each pending child is a replacement) then
            goto cancel;
         else
            decision = "confirmed";
            del(evt.XIPHandle, corr.children);
            alarm(evt);
         send(decision, evt.XIPHandle);
      else if (end) then
```

```
         if (commit) then
             ret ← commit(corr.XIPHandle);
             if (ret != OK) then
                goto cancel;
             send("global_commit", corr.children);
             wait for all responses
      else if (cancel) then
cancel:  send("cancel", corr.children);
         wait for 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 an XIP 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: its deadline. Then the CTM creates an XIP 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, an "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 CTM optimistically commits the transaction ("local-committed") if it is cancellable, otherwise prepares the transaction ("prepared"). If there is an error in itself or some of its children transactions, the CTM aborts the local transaction and sends a "cancel" request to its children transactions. Then the CTM sends an "ok" or "aborted" response to the parent transaction.
- The deadline event. A deadline of a transaction is a point beyond which it will not be cancellable. The deadline event can be triggered by time and/or some business logic. When it occurs, the CTM checks if there are any local-committed transactions that have reached their cancellation deadlines. For each of these transactions, the CTM sends a "compensation" request to its parent. If the reply indicates that the local transaction should be compensated, the CTM calls the transaction's $\varphi(t)$ to undo the transaction, and sends a "cancel" request to its children transactions.
- The compensation event. When receiving a "compensation" request from one of its children, the CTM checks the status of the local transaction. If the status is "local-committed" or "prepared", it forwards the request to its parent. When receiving the reply, the CTM forwards it to the child. If the reply is "confirmed", the CTM compensates or simply aborts the transaction, and sends a "cancel" request to the rest of its children transactions. If the local transaction's status is "pre-commit" and it still has incomplete children that were not started due to "compensation" requests, the CTM sends back a confirmation, removes the child entry in its correlator structure, and informs the

local e-service of the compensation. It is up to the local e-service to decide whether to start a new sub transaction. If each pending child was started due to a "compensation" request, the CTM aborts the local transaction and send a "cancels" request to its children. No action should be taken if there is no outstanding child transaction.

- The global commit event. When receiving a "global commit" message, the CTM commits the local transaction if its status is "prepared", and forwards the message to its immediate children. Once all the responses come back, the CTM sets the local transaction's status to "global-committed", and sends a "globally committed" response to the parent transaction.
- The cancel event. When receiving a "cancel" message, the CTM aborts or compensates the local transaction if its status is not aborted (e.g., pre-commit, prepared, local-committed). Then it forwards the "cancel" message to its immediate children. 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.XIPHandle.URL ← CTM.URL;
         corr.XIPHandle.ID ← a new tran ID;
         corr.parent ← evt.XIPHandle;
         corr.deadline ← deadline;
         connect(corr.XIPHandle, corr.parent);
         return corr.XIPHandle;
      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 (no error) then
            if (cancellable) then
              commit(corr.XIPHandle);
              corr.status ← "local-committed";
            else
              prepare(corr.XIPHandle);
              corr.status ← "pre-commit";
            send("ok", corr.parent);
         else
abort:      send("cancel",corr.children);
            wait for all responses
            if(corr.status = "local-committed")
            then
               φ(corr.XIPHandle);
            else
               abort(corr.XIPHandle);
            corr.status ← "canceled";
            send("canceled", corr.parent);
      else if (deadline) then
         for (each local transation)
            corr ← t.correlator;
            if(corr.deadline <= δ(curtime) && corr.status ="local-committed")
```

```
            then
               send("compensation", corr.parent);
               if (reply = "confirmed") then
                  φ(corr.XIPHandle);
                  send("cancel", corr.children);
      else if (compensation) then
         if (corr.status = "local-committed" ||  corr.status = "prepared")
         then
            send("compensation", corr.parent);
            receive(reply, corr.parent);
            send(reply, evt.XIPHandle);
            if (reply = "confirmed") then
               if(corr.status = "local-committed")
               then
                  φ(corr.XIPHandle);
               else
                  abort(corr.XIPHandle);
               send("cancel", other children);
         else if (corr.status = "pre-commit")
         then
             if (there are pending children)
             then
                if (each pending child is a replacement) then
                   goto abort;
                else
                   send("confirmed",evt.XIPHandle);
                   del(evt.XIPHandle, corr.children);
                   alarm(evt);
      else if (global_commit) then
         if (corr.status = "prepared") then
            commit(corr.XIPHandle);
         send("global_commit",corr.children);
         wait for all responses
         corr.status ← "global-committed";
         send("global-committed", corr.parent);
      else if (cancel) then
         if (corr.status = "canceled") then
            send("canceled", corr.parent);
         else
            goto abort;
end
```

Note that the protocol does not specify how a component transaction should be committed locally. It only assumes that, for uncancellable transactions, the local transaction managers should support 2 phase commit [17], and allow a transaction to be prepared and committed by potentially different processes.

Figure 3 shows how the protocol orchestrates the committing of a conversational transaction. In the example, s_0 is the root e-service, and interacts with s_1 and s_2. E-service s_2 has its own sub e-services $s_3$and s_4/s_5. For simplicity, figure 3 only displays the XIP messages between the e-services whereas the e-service-specific messages are not shown.

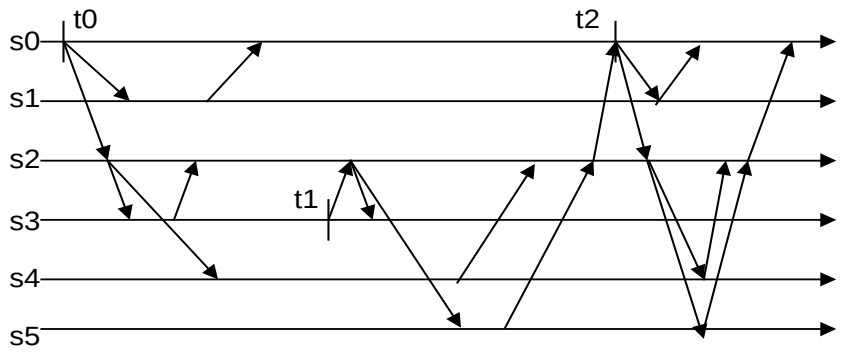

Figure 3. The committing of a conversational transaction

In the example, s_0 starts the root transaction at point t_0, then communicates with s_1 and s_2 which start their transactions accordingly. s_2 negotiates its own sub transactions by interacting with s_3 and s_4. Following its local commit, the transaction at s_3 reaches its deadline at point t_1 while it has not received the global commit request from its parent. Thus it sends a compensation request to its parent. When receiving the request, the transaction at s_2 has not heard from s_4, so it sends back a confirmation to cancel the committed transaction at s_3, and re-negotiates a new sub transaction with s_5. At point t_2 when informed of the completion of its sub transactions, s_0 commits the root transaction and sends a global commit to its children.

4.4 Proof of correctness

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

Lemma1 The optimistic commit 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 initiated 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 one round of synchronous messages to globally commit or cancel the conversational transaction. t_0, after every certain interval, will keep sending the global 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. With the protocol, t_i behaves independently except for two points where t_i can be blocked: waiting for a reply after sending its parent an "compensation" request; and waiting for the responses after forwarding a "global commit" or "cancel" message to its children. In the first scenario, t_i will keep sending

the request until it receives a "confirmed" or "denied" response. 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, the parents of t_i will eventually receive and process the request, so will t_i receive the response. In the second scenario, for the same reason, t_i will eventually receive all the responses.

Lemma 2 The optimistic commit protocol is livelock free.

Proof:
Transaction compensation could lead to a livelock in which an infinite number of compensations are initiated, preventing the conversational transaction from making progress. Consider an extreme scenario where t_i *(* $0 \leq i < n$*)* has two sub transactions: the first child that was previously committed reaches its deadline, and needs to be compensated while the second child is still pending. Suppose t_i starts a new sub transaction replacing the first child, then receives a response from the second child. But before the first child (the new sub transaction) finishes, the second child needs to be compensated. If t_i starts another sub transaction replacing the second child, the first child, following its local commit, could again reach its deadline before the second child (the new sub transaction) finishes: a livelock occur.

Our protocol avoids livelocks by checking the status of t_i's sub transactions before t_i can negotiates a new sub transaction. According to the protocol, t_i can start a new child only if it has pending children that were not started due to compensation requests. That is, t_i must be patient enough to hear from each of its original children. Then if there are still pending children each of which are a replacement of a compensated child, the protocol will cancel the conversational transaction when receiving any new compensation request. As a result, the number of compensations for a conversational transaction is bounded: no livelock.

4.5 The transaction-reversing algorithm

So far we have not addressed how the transaction compensation can be done transparently, except for assuming a compensation function $\varphi(t)$. In the following we propose a language-based approach enabling transparent transaction compensation. The approach assumes that databases are accessed via some APIs such as JDBC, ODBC, OLE DB, etc. The idea is to intercept each API call and augment it with the transaction-reversing algorithm. For each update statement (e.g., INSERT, DELETE, UPDATE), the reversing algorithm records the corresponding column information, and constructs the reversing SQL strings. At the time of compensation, the transaction manager prepares and executes the reversing statements with the reversing SQL strings and saved column information associated with the transaction. As a result, the state of the transaction is removed from the database.

Before presenting the algorithm, we formalize the interactions between applications and databases. $f_c(t, d)$ represents the operation creating a database table, where t is the name of the table to be created, and d is the definitions of the columns in the table.

$f_i(t, c)$, $f_d(t, q)$, $f_u(t, c, q)$, and $f_s(t, c, q)$ represents an insert, delete, update, and select operation respectively. The operation can be in the form of a statement, a prepared statement, or a stored procedure. Among the parameters, t is the set of the tables to be accessed. c is the set of

columns to be inserted, updated, or selected. q is the set of columns used to create the query (i.e., the WHERE clause).

The reversing algorithm works in the following format[1].

1. At the time of transaction start, associates the transaction with the current database connection: the database URL.
2. Saves t and d when performing $f_c(t, d)$: creating a table.
3. When preparing a statement or stored procedure, intercept and parse the SQL string and passed parameter. Then if it does not exist, a reversing function (i.e., a reversing SQL string) is constructed based on the type of the operation.
4. For $f_i(t, c)$, based on t and c, construct the reversing function $f_d(t, q)$ where q is a function of c: $q = \varepsilon(c)$. Then generate and save a compensation record. The record contains a pointer to the reversing function and the parameters to the function (t and q).
5. For $f_d(t, q)$, performs a select operation based on t and q. That is, $C = f_s(t, *, q)$, where C is the result set. Then construct the reversing function $f_i(t, d)$. For each $c \in C$, a compensation record is generated and saved. The record contains a pointer to the reversing function and the parameters to the function (t and c).
6. For $f_u(t, c, q)$, perform the following operations: $C' = f_s(t, c, q)$; $\overline{C} = \delta(c, C')$. $\forall c' \in C'$, c' is the set of columns to be updated. Function δ performs such that, $\forall \overline{c} \in \overline{C}$, $\overline{c}[i] = c[i]$ if $c[i]$ contains a delta value; otherwise, $\overline{c}[i] = c'[i]$. Then based on t, c, and q, construct the reversing function $\overline{f_u}(t, c, q)$. For each $\overline{c} \in \overline{C}$, a compensation record is generated and saved. The record contains a pointer to the reversing function and the parameters to the function (t, $\overline{c}$, and q).

Now we show an example where the algorithm prepares the compensation records for a transaction. Consider a transaction executing a piece of JDBC code that update the account balances at an indicated bank branch. Note that all of the following code are incomplete, and only present the logic of the algorithm.

```
Connection con = DriverManager.getConnection(url, id, pw);
PreparedStatement pstmt = con.prepareStatement(
         "UPDATE BRANCH SET BALANCE = BALANCE + ? DATE = ? WHERE BRANCH_ID = ?");
pstmt.setInt(1, delta);
pstmt.setString(2, curDate);
pstmt.setShort(3, branchID);
pstmt.executeUpdate();
```

When "prepareStatement()", "setInt()", "setString()", and "setShort()" are intercepted, the algorithm records the SQL string and the types and values of the updated (BALANCE and DATE) and indexing (BRANCH_ID) columns. It also constructs a select string and a reversing update string if they do not exist. The strings in this case are

```
"SELECT BALANCE DATE FROM BRANCH WHERE BRANCH_ID = ?"
"UPDATE BRANCH SET BALANCE = BALANCE + ? DATE = ? WHERE BRANCH_ID = ?"
```

[1] Full implementation details of the algorithm are out of scope of this paper

When “executeUpdate()” is intercepted, the algorithm does the following before executing the original statement.

```
PreparedStatement pstmt1 = con.prepareStatement(
                          “SELECT BALANCE DATE FROM BRANCH WHERE BRANCH_ID = ?”);
pstmt1.setShort(1, branchID);
ResultSet rs = pstmt1.executeQuery();
while (rs.next()) {
   compensationRecords[i].sqlString
   = “UPDATE BRANCH SET BALANCE = BALANCE + ? DATE = ? WHERE BRANCH_ID = ?”;
   compensationRecords[i].col[0].type = “int”;
   compensationRecords[i].col[0].value = -1*delta;
   compensationRecords[i].col[1].type = “String”
   compensationRecords[i].col[1].value = rs.getString(“DATE”);
   compensationRecords[i].col[2].type = “short”;
   compensationRecords[i].col[2].value = branchID;
   i++;
}
```

To compensate a transaction, the transaction manager builds a connection with the saved database URL. Then it retrieves all the compensation records associated with the transaction. By replaying each record, the state of the transaction is removed from the database. For the above example, the algorithm performs the following.

```
Connection con = DriverManager.getConnection(url, id, pw);
for (i = 0; i < numberOfRecords; i++) {
    PreparedStatement pstmt =
                      con.prepareStatement(compensationRecords[i].sqlString);
    pstmt.setInt(1, compensationRecords[i].col[0].value);
    pstmt.setString(2, compensationRecords[i].col[1].value);
    pstmt.setShort(3, compensationRecords[i].col[2].value);
    pstmt.executeUpdate();
}
```

5. Conclusion

E-services are federated, composite, and autonomous. Due to the nature of e-services, the traditional transaction processing mechanisms do not work well in the e-service world. We proposed in this paper a new approach called XIP to transaction processing on the Internet. XIP consists of two parts. The first part is an optimistic commit protocol to systematically enable the Internet transaction semantics (e.g., commit, abort, retry, cancel). Compared to other optimistic commit protocols, it addresses some unique aspects of e-service transactions. The second part is a transaction-reversing algorithm enabling transparent transaction compensation. With optimistic commit, XIP eliminates the resource-locking problem with TIP. With the transparent transaction compensation, XIP, unlike the existing approaches, does not require applications to provide compensation functions—the programming burden is greatly reduced—and this makes XIP a practical approach to transaction support for e-services.

References

1. Object Management Group. *The common object request broker. Architecture and specification.* Revision 2., July 1995. D. Rogers http://www.omg.org
2. 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.hp.com
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.
6. C. Wood. *OLE DB and ODBC Developer's Guide.* M&T Books, Foster City, CA., 1999.
7. S. While, et al. *JDBC API Tutorial and Reference, 2nd Edition.* Addison-Wesley, 1999.
8. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, 1987.
9. H. F. Korth, E.Levy, and A Silberschatz. . A formal approach to recovery by compensation transactions. In *Proceedings of ACM-SIGMOD 1991 International Conference on Very Large Databases*, pages95-106, 1990.
10. 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.
11. P. K. Chrysanthis. Transaction processing in mobile computing environment. In *Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems,* pages 77-82, 1993.
12. E. Pitoura and B. Bhargava. Building information systems for mobile environments. In *Proceedings of the 3rd International Conference on Information and Knowledge Management,* pages 371-378, 1994.
13. E. Pitoura and B. Bhargava. Maintaining consistency of data in mobile computing environments. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems,* June, 1995.
14. 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.
15. Hewlett-Packard Company. *Vantage Point Web Transaction Observer*. 1999.
16. XML at World Wide Web (WWW) Consortium http://www.w3c.org/xml
17. x/Open Compony Ltd. *Distributed Transaction Processing: The XA Specification*, 1991