A Hybrid Scheme for Controlling Transactional Composite Services

Xinfeng Ye Yi Chen
Department of Computer Science
Auckland University
Auckland, New Zealand
xinfeng@cs.auckland.ac.nz yche486@aucklanduni.ac.nz

Abstract—**This paper proposes a hybrid concurrency control scheme for transactional composite services. The scheme uses the information gathered from the workflow specifications of the composite services to reduce the overhead in detecting cycles in the serialization graph. The scheme carries out run-time analysis of the SQL statements used by the composite services to determine whether the clients that execute composite services depend on each other more accurately. As a result, it reduces the response time to some users. The proposed scheme also tackles the repeated rollback problem facing many concurrency control schemes.**

Keywords- concurrency control; composite services

I. INTRODUCTION

Business processes can be modeled as *composite services* consisting of many tasks implemented by Web Services. The workflow of a composite service can be specified using BPEL4WS [4]. Since different composite services might access the same data concurrently, it is necessary to ensure the correctness of the concurrent execution of the composite services.

The ACID-relaxed concurrency control schemes, e.g. [1, 5, 6], have been used for long-running applications. In these schemes, a composite service releases its lock on a service provider after its call to the service provider is completed. That is, locks on the service providers are released before the composite service terminates. These schemes use a protocol similar to the distributed Serialization Graph Testing protocol [12] to detect violation to the serializability of the system. If a violation is detected, the composite services are rolled back and re-executed.

A problem with the ACID-relaxed schemes is that, if a composite service is re-executed for serializability violation, there is no guarantee that the re-execution can be carried out successfully. As a result, some composite services might be rolled back repeatedly. For example, assume that (a) a client c_1 accesses two service providers sp_1 and sp_2 in sequence while client c_2 accesses sp_2 and sp_1 in sequence, and, (b) c_1 and c_2 start their execution concurrently. It can be seen that c_1 would access sp_1 first while c_2 would access sp_2 before c_1. As a result, a cycle involving c_1 and c_2 is formed in the serialization graph. Hence, the operations of c_1 and c_2 are re-executed. However, during the re-execution, the above scenario might reoccur again. Thus, cs_1 and cs_2 might be rolled back again.

In ACID-relaxed schemes, clients depend on each other if they operate on the same set of data. Many existing schemes regard two clients depend on each other if they access the same service provider. This approach might create a lot of false dependency amongst the composite services. This is because the clients might pass different parameters to the service provider. Thus, the service provider might access different sets of data while being activated by different clients. Since locks on the service providers are released before the clients terminate, if a client aborts, the clients that depend on it also need to be aborted. Thus, to prevent releasing results to the users prematurely, a client can only terminate after all the clients that it depends on have terminated. False dependency would make clients wait for each other to terminate unnecessarily. Hence, it would increase the response time of the clients.

This paper proposed a concurrency control scheme for executing transactional composite services. It uses the static analysis technique to collect information about the composite services off-line. As the workflows of composite services are known in advance, by analyzing the workflows of the composite services, the service providers accessed by the composite services as well as the composite services that are likely to form cycles in the serialization graph can be known in advance. This information is used by the concurrency control scheme (a) to reduce the overhead on checking the serializability of the system, and, (b) to avoid the repeated rollback problem described above. The scheme analyses the SQL statements used to manipulate data at run-time to reduce the amount of false dependency amongst the clients. To minimise the response time, under normal circumstance, the scheme allows clients to carry out their calls to the service providers without waiting for other clients to terminate. For the clients that are re-executed due to serialization violation, the scheme requires the clients to reserve execution positions before they start their execution to avoid repeated rollback.

The rest of the paper is organized as follow. Related works are described in section II. The proposed scheme is given in section III. Section IV compares the performance of the scheme with some existing schemes. Conclusions are given in section V.

II. Related Work

The scheme proposed by Ye et al. [18] requires a composite service to reserve an execution position on each of the service providers to be accessed by the composite service before the execution starts. The position reservation scheme ensures that no cycles can be formed in the serialization graph. As the execution of the composite services is ordered according to their reserved positions, sometimes a composite service has to wait for others to carry out their execution despite the execution of the composite service does not cause serializability violation. Hence, the response time to the composite service might be unnecessarily increased. The scheme in this paper only requires the set of composite services whose executions might violate the serilizability of the system to reserve their execution positions and to execute in the order of their reserved positions. Thus, it causes less unnecessary delay to the execution of the composite services.

Haller et al. [6] described a decentralised serialization graph testing protocol to carry out concurrency control in peer-to-peer environments. In [6], each peer keeps the information on precedence relations amongst the peers. The precedence information is passed amongst the peers when necessary to discover non-serializable execution of the peers. The approaches proposed by Alrifai et al. [1] and Choi et al. [5] use the same principle as Haller's approach in detecting non-serializable executions. Microsoft BizTalk Server [16] uses the saga model [17] to manage long-lived processes. All these schemes suffer a drawback as they might repeatedly roll back some composite services.

Optimistic concurrency control protocols [8, 9] validate the serializability of the system at the end of the execution of each transaction. Instead of waiting until the end of a composite service's execution, the scheme in this paper rolls back a client as soon as it is discovered that the serializability of the system is violated. Thus, it avoids wasting resources on executing operations that will be rolled back later as the scheme initiates the rollback as soon as it is needed.

Yang et al. [14] proposed a scheme for consistently updating replicated data in Internet-based applications. Yang's scheme only concerns with the ordering of a replicated operation. Since multiple service providers are accessed in a composite service, unlike Yang's scheme, the scheme in this paper considers the ordering of multiple operations.

Li et al. [15] described an approach for deriving the transactional properties of composite web services. Unlike this paper, instead of discussing concrete scheduling schemes, they focused on studying whether a given composite service is schedule-able.

Many works have been carried out on query containment problem [7]. They focused on determining whether the constraints of one query are included in another query. Different to these works, the run-time analysis of the SQL statements in this paper focuses on checking whether the data set operated on by the SQL statements intersect with each other.

III. The Scheme

A. System Model

It is assumed that the workflows of the composite services are known in advance and clients only execute these composite services[1]. It is assumed that (a) each service provider provides a compensation operation to undo the effect of its operation, and, (b) a client re-executes the composite service after it is rolled back.

According to the serilizability theory [2], two clients are regarded as *conflict* if they access the same set of data and at least one of them modifies the data. Two clients become conflict with each other through invoking the same service provider (i.e. the service provider modified the data shared by the two clients). It is assumed that the data held by a service provider are stored in a database and the granularity of data sharing is at the database's row level. The service providers access the databases using SQL statements. For two clients, say c_1 and c_2, if (a) they become conflict with each other while accessing a service provider, and, (b) c_1 accessed the service provider before c_2, it is said c_2 *depends* on c_1 or c_1 *precedes* c_2 (denoted as $c_2 \rightarrow c_1$). A serialization graph is a directed graph (N, E) where "$N = \{c \mid c \text{ is a client}\}$", and, "$E = \{<c_i, c_j> \mid (i \neq j) \wedge (c_i, c_j \in N) \wedge (c_i \rightarrow c_j)\}$". N is the set of nodes and E is the set of edges. According to the serializability theorem [2], the execution of the composite services is regarded as correct if the serialization graph is acyclic.

B. An Overview

Figure 1 is a conceptual diagram of the system. An execution coordinator resides on each service provider's site. Each execution coordinator provides the same operation as its corresponding service provider. On the client side, the execution engine of the composite service binds the operations in the composite service to the operations provided by the execution coordinators. Thus, calls to the service providers are processed by the execution coordinators first. Each execution coordinator keeps an execution queue to hold the execution requests from the clients. To ensure execution order, the requests in the queue are executed one at a time. When it is the turn for a client's request to be executed, the execution coordinator invokes the corresponding operation on the service provider. The result produced by the service provider is sent back to the client through its execution coordinator.

The workflow of each composite service is analyzed off-line by a workflow analyzer. From the analysis, the service providers that might be accessed in a composite service are known. The workflow analyzer checks the composite

[1] This assumption is reasonable since in an enterprise environment users are only allowed to run pre-configured applications, e.g. accounting, billing, etc.

services to find out the sets of composite services whose executions might form cycles in the serialization graphs. The information collected by the analyzer is disseminated to the execution coordinators involved in the execution of the relevant composite services. Thus, each execution coordinator is aware of (a) all the composite services that access it, (b) the composite services that have the potential to generate non-serializable execution, and, (c) for each composite service, the other execution coordinators that are involved in the execution of the composite service.

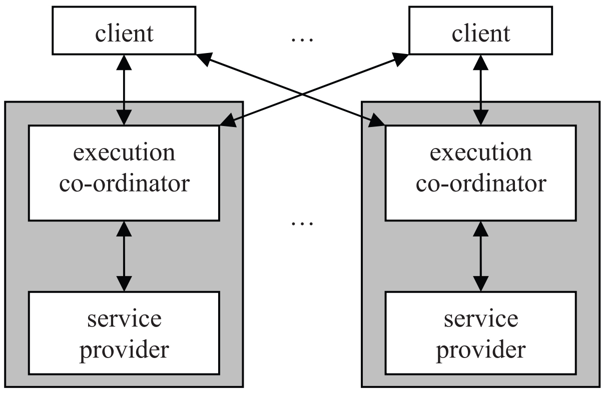

Figure 1. A Conceptual Diagram of the System

When a client executes a composite service, the client *terminates* if it aborts or commits all its operations. Like other ACID-relaxed schemes, for two clients, say c_1 and c_2, such that $c_2 \rightarrow c_1$, c_2 is allowed to access the service providers shared by c_1 and c_2 before c_1 terminates. If c_1 aborts later, c_2 has to abort as well since it might have accessed some values written by c_1. Thus, a client can only terminate if all the clients that it depends on have terminated. The analyzer inserts some operations at the end of each composite service's workflow to determine whether a client can terminate. These operations ask each of the service providers accessed by the client to find out whether the client's preceding clients have all terminated.

A server code analyser is used to analyse the code implementing the service providers off-line. The analyser inserts some statements into the service providers' code to record the SQL statements being executed and the parameters being passed to the SQL statements while the service providers execute the clients' requests. After a service provider completes the execution of a client's request, the service provider passes the information back to the execution coordinator. The execution coordinator analyses the SQL statements executed for different clients to determine whether the statements operate on any common data set. This allows the execution coordinator to determine whether the clients conflict with each other. The checking for conflicting clients can be carried out at the same time as the next client's request (if any) is executed by the service provider. Hence, the checking does not block the operation of the service provider.

If two clients conflict with each other, the execution coordinator checks whether the composite services executed by the two clients might form a cycle according to the information provided by the workflow analyser. If they might form a cycle, the execution coordinator runs a distributed cycle-chasing algorithm [12] to check whether there is a cycle in the serialization graph. The algorithm passes probes along the edges in the serialization graph. Each execution coordinator only needs to know the dependency relations between the clients on its own site. It does not need to know the complete serialization graph of the system. Since a client might access multiple service providers, when a client receives a cycle-chasing probe, the probe is propagated to all the service providers accessed by the client to ensure that all the relevant paths in the serialization graph are checked. If a cycle is discovered, all the clients in the cycle are aborted and re-executed.

To prevent the repeated rollback problem, before a client is re-executed, the execution coordinators involved in executing the client are asked to reserve a position in their execution queues for executing the client's requests. The execution coordinators ensure that the positions reserved for the client allow the client's requests to be carried out in the same relative order in terms of the other re-executed clients on all the service providers. For example, if two clients, say c_1 and c_2, both need to access two service providers, say sp_1 and sp_2, then c_1's reserved position should precede c_2's position on both sp_1 and sp_2 or vice versa. By reserving positions, the scheme prevents cycles from forming in the serialization graph.

C. *The Workflow Analyzer*

The cycle detection algorithm needs to exchange messages amongst the execution coordinators. To minimize the overhead, the algorithm should only be executed when there is a potential for the clients to form a cycle. By checking the workflows of the composite services, the workflow analyzer can determine whether a composite service might form a cycle with other composite services.

The workflow analyzer constructs an undirected *workflow relationship graph* according to the workflows of the composite services[2]. Each node in the graph is a composite service. If two composite services access at least one common service provider, an edge is created between the two nodes. Finding all possible cycles in the graph is equivalent to the problem of finding the bi-connected components in an undirected graph. The analyser uses Tarjan's Algorithm [11] to find all the cycles in the graph. For each composite service in the cycles, the analyzer creates a *potential conflict set* to record the composite services that connect to the composite service in the cycles. The potential conflict set for a composite service, say i, is denoted as PC_i. PC_i also includes i itself. This is because

[2] There are only a limited number of composite services. Thus, the graph is a finite graph.

two clients executing the same composite service that contains concurrent access to service providers (e.g. BPEL4WS's parallel flow activity) might forms a cycle in the serialization graph. Since a composite service, say i, can only form cycles with the composite services in PC_i, the cycle-chasing algorithm only needs to propagate the cycle-chasing probes to the clients executing the composite services in PC_i.

For the workflow relationship graph in Figure 2, there are three cycles, i.e. CS_1-CS_2-CS_3, CS_1-CS_2-CS_3-CS_4, and, CS_1-CS_3-CS_4. The potential conflict sets are $PC_{CS1}=\{CS_1, CS_2, CS_3, CS_4\}$, $PC_{CS2}=\{CS_1, CS_2, CS_3\}$, $PC_{CS3}=\{CS_1, CS_2, CS_3, CS_4\}$, $PC_{CS4}=\{CS_1, CS_3, CS_4\}$ and $PC_{CS5}=\{CS_5\}$. CS_2 and CS_4 are not in each other's potential conflict sets since they are not connected to each other in the cycles. The potential conflict sets are given to the relevant execution coordinators. Assume that (a) clients c_1, c_2 and c_3, are executing CS_1, CS_2 and CS_5 respectively, and, (b) $c_2 \rightarrow c_1$ and $c_3 \rightarrow c_2$. Since CS_1 and CS_2 might form a cycle (i.e. $CS_1 \in PC_{CS2}$), the execution coordinator initiates cycle detection to check whether there is a cycle in the serialization graph after it is known that c_2 depends on c_1. However, since CS_2 and CS_5 cannot be in a cycle (i.e. $CS_2 \notin PC_{CS5}$), there is no need to carry out cycle detection when it is determined that c_3 depends on c_2. For the same reason, when an execution coordinator receives a cycle-chasing probe from a client, say c, it only propagates the probe to the clients that are executing the composite services that are in PC_c.

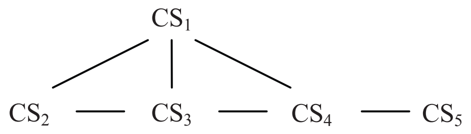

Figure 2. A Workflow Relationship Graph

D. Detecting Conflicting Clients at Run Time

Clients accessing the same service providers might not access the same set of data. This is because, as clients might pass different inputs to the service provider, this might result in different sets of SQL statements being executed. Also, even if the same set of SQL statements are executed, if the parameters given to the SQL statements are different, different sets of data might be manipulated by the SQL statements. Thus, by comparing the SQL statements that are executed by the service provider for different clients, it is possible to determine whether the clients access the same set of data. Determining whether two clients conflict based on whether they share data instead of whether they access the same service provider can eliminate some of the false conflict amongst the clients.

According to [2], two operations conflict if they access the same data and at least one of them modifies the shared data. There are four types of statement for data manipulation in SQL, i.e. SELECT, UPDATE, INSERT and DELETE statement. The SELECT statement can be regarded as the read operation while the other three types of statements can be regarded as the write operation. Thus, two SQL statements are regarded as conflicting with each other if (a) at least one of the statements is a DELETE, INSERT or UPDATE statement, and, (b) they might operate on the same data.

A service provider might execute multiple SQL statements. Two clients conflict with each other if any of the SQL statements executed on behalf of different clients conflict with each other. The following steps are used to determine whether two SQL statements conflict.

1) Determine the types of the SQL statements: As at least one of the statements must be a write operation, it is necessary to check whether at least one of the SQL statements is a DELETE, INSERT, or, UPDATE command.

2) Determine the columns that the SQL statements operate on: If two SQL statements access the same data, they must operate on at least one common column of a database table. Each command has some elements specifying the columns that the command operates on. By comparing these elements, it can be determined whether two SQL statements access any column in common. For example, assume there are two SQL statements, i.e. SQL_1="SELECT Name FROM Persons", SQL_2="UPDATE Persons SET year=5". SQL_1 and SQL_2 do not operate on any shared data as they handle different columns in the table.

3) Compare the constraints for selecting rows in a table:

- The "WHERE" clause in a SQL statement decides the rows that will be operated on by the SQL statement. First, the Boolean expression contained in the "WHERE" clause is converted to a disjunctive (OR) normal form. A disjunctive normal form consists of one or more disjuncts, each of which is a conjunction (AND) of one or more literals [10]. Each literal is an expression specifying the constraint that the value of a column should satisfy.
- Let the "WHERE" clauses of two SQL statements are $where_1 = \vee_{i=1..n} item_i$ such that $item_i = \wedge_{p=1..s_i} subitem_p$ and $where_2 = \vee_{j=1..m} item_j$ such that $item_j = \wedge_{q=1..t_j} subitem_q$. To determine that two SQL statements do not operate on shared data, it is necessary to show that the "WHERE" clause of the two SQL statements cannot be evaluated to true at the same time. That is "$where_1 \wedge where_2$" (i.e. $\vee_{i=1..n,j=1..m}(item_i \wedge item_j)$) is false. Thus, it needs to check whether all "$item_i \wedge item_j$" are false. "$item_i \wedge item_j$" evaluates to false if the constraints expressed in "$item_i \wedge item_j$" contradict each other, e.g. "*Name*=`Jones` AND *Name*=`Owen`". If the value of a Boolean expression cannot be determined, e.g. "$Name = 'Jones' \wedge Year = 5$", the two SQL statements will be regarded as operating on some common rows.

- The INSERT and UPDATE statements need special consideration as they might create and modify rows that satisfy the "WHERE" clause of other statements. The INSERT statement inserts a new row into the database. To check whether the new row satisfies a constraint, say "$\vee_{i=1..n} item_i$", it needs to check whether "$(\vee_{i=1..n} item_i) \wedge (\wedge_{j=1..m} c_j = value_j)$" is true (where c_j is the column name and $value_j$ is the value assigned to c_j in the INSERT statement). For example, assume that there are two columns, i.e. *Name*, and, *Year*, in table *Persons*. To check whether the row inserted by "INSERT INTO *Persons* VALUES (`*Owen*`, 6)" satisfies constraint "$\vee_{i=1..n} item_i$", it needs to check whether "$(\vee_{i=1..n} item_i) \wedge ((Name = 'Owen') \wedge (Year = 6))$" is true.
- When checking whether an UPDATE statement might operate on the same data as a SQL statement that is executed after the UPDATE statement, if the "WHERE" clause of the UPDATE statement includes some constraints specified in terms of the columns updated by the UPDATE statement, the values assigned to the columns by the UPDATE statement should be used in the checking. For example, assume that (a) there are two SQL statements, i.e. SQL_1="UPDATE *Persons* SET *Year*=7 WHERE *Year*=6", SQL_2="SELECT * FROM *Persons* WHERE *Year*=5", and, (b) SQL_2 is executed after SQL_1. Since the rows operated on by SQL_1 have changed their value of column *Year* to 7, when comparing whether SQL_1 and SQL_2 operate on the same set of data, the constraint used for SQL_1 in the checking should be "*Year*=7" instead of "*Year*=6",. Thus, condition "(*Year*=7) $\wedge$ (*Year*=5)" is checked.

The server code analyzer can automatically insert code into the service provider's implementation to collect the information about the SQL statements at run-time. The example below shows the code inserted into the service provider implemented using C# and ADO.net. It shows how to capture the SQL statement that updates a table. Lines 2 to 9 are inserted by the server code analyzer. The even handler *OnRowUpdated* is executed whenever the contents of a table is changed. The handler captures the SQL command being executed (line 5) as well as the values given to the parameters (lines 6 to 8). The captured information is stored in a record (line 9) to be passed to the execution coordinator later.

```
1  SqlDataAdapter adapter =
    new SqlDataAdapter(queryString, connectionString);
2  adapter.RowUpdated +=
   new SqlRowUpdatedEventHandler(OnRowUpdated);

3  private void OnRowUpdated(object sender,
                               SqlRowUpdatedEventArgs e) {
4    string tempWriteCommand = "";
5    tempWriteCommand = e.Command.CommandText;
6    for (int i = 0; i < e.Command.Parameters.Count;
                                                 ++i) {
7      tempWriteCommand.Replace(
            e.Command.Parameters[i].ParameterName,
            e.Command.Parameters[i].Value.ToString());
8    }
9    AddWrite(tempWriteCommand);
   }
```

E. The Execution Coordinator

In the following discussion, the execution queue maintained by an execution coordinator is denoted as *EQ*. *elem(EQ)* denotes the set of all the elements in execution queue *EQ*. *e.client* denotes the client whose operation is held in element *e* in *EQ*.

When an execution coordinator receives a client's request for invoking a service provider, the execution coordinator places the request at the end of the queue. The state of the element holding *ER* is set to *active* to indicate that the operation is available for execution (line 1 of R1).

Set *re-execute* records the clients that are re-executed as a result of breaking cycles in the serialization graph. If a client is re-executed, in order to avoid repeated rollback, a position is reserved for the client on all the service providers that will be accessed by the client (line 3 of R1). *c.SP* denotes the set of service providers accessed by client *c*. The scheme uses the ABCAST protocol [3] to deliver the reserve position request (*RPR*) messages. The ABCAST protocol ensures the total order of multicast messages. Total order means multicast messages, say m_1 and m_2, are always delivered to all their recipients in the same relative order even if the senders of the messages are different. Thus, the *RPR* messages for different re-executed clients will arrive at the relevant execution coordinators in the same relative order. The execution positions reserved for the re-executed clients are the same as the order in which the multicast *RPR* messages are received by the execution coordinators. Thus, the positions reserved for the re-executed clients are in the same relative order in all the execution coordinators' execution queues. This guarantees that the re-executed clients do not form a cycle in the serialization graph.

On receipt of a *RPR* message (R2), an execution coordinator creates a position for the client at the end of its execution queue. Since the execution coordinator has not received the request from the client to invoke the service provider, the execution coordinator sets the state of the element holding the reserved position to *dormant* to indicate that the expected operation has not been received from the client.

When the execution coordinator receives an execution request from a re-executed client and an execution position has been reserved for the client, the state of the element holding the reserved position for the client is set to *active* to indicate that the request is waiting to be executed (line 2 of R1).

```
R1  when receive an execution request $ER$ from
        a client, say $c$:
1   if $\neg\exists e : elem(EQ).(e.client = c)$ then
        $e \leftarrow ER$; $e.state \leftarrow active$; $EQ \leftarrow EQ$^<e>;
2   else //i.e. $\exists e : elem(EQ).(e.client = c)$
        $e \leftarrow ER$; $e.state \leftarrow active$
    end-if
3   if $c \in re\text{-}execute$ then
        ABCAST reserve position request $RPR(c)$
        to all the execution coordinators in $c.SP$
    end-if
```

```
R2  when receive a reserve position request $RPR(client)$
        from an execution coordinator, say $ec$:
1   create an element $e$;
    $EQ \leftarrow EQ$^<e>;  $e.state \leftarrow dormant$;
```

R3 specifies the condition under which a client's request can be executed. In R3, *hd(EQ)* denotes the first element in *EQ*. Predicate *front(e',e)* means element *e'* precedes element *e* in *EQ*. If an operation is at the head of an execution queue and is ready to be executed (i.e. "$e = hd(EQ)$" and "$e.state = active$" are both true in R3's condition), the operation can be carried out immediately. When a client completed its access to a service provider, the state of the element holding the client's operation is set to *completed* (line 1 of R3). As other ACID-relaxed schemes, a client can access a service provider as soon as the clients that precede it have completed their access to the service provider ($front(e',e)$ and "$e'.state = completed$" are both true in R3's condition). For the re-executed clients, their requests need to be executed in the order determined by their reserved positions to avoid repeated rollback problem explained in section I. However, other clients (i.e. "$e \notin re-execute$" in R3's condition) do not need to wait for the re-executed clients to complete their requests if the re-executed clients' requests have not been received (i.e. "$e'.state = dormant$" in R3's condition). Thus, if the request of a client, say *c*, is ready to be executed while all the requests preceding *c*'s requests are in *dormant* state (i.e. the preceding requests are the elements for holding the execution positions of the re-executed clients), *c*'s request can be executed first. Thus, the re-executed clients do not delay the execution of the other clients.

```
R3  when  $\exists e : elem(EQ).((e.state = active) \wedge ((e = hd(EQ))$
              $\vee (\forall e' : elem(EQ).(front(e',e)$
                  $\wedge ((e'.state = completed)$
              $\vee ((e'.state = dormant) \wedge (e \notin re-execute))))))$
1   execute $e.op$;   $e.state \leftarrow completed$
```

As described in section III.*B*, after the service provider completes the execution for a client, say *c*, the service provider returns the SQL statements executed on behalf of *c*. The execution coordinator compares the statements with the statements executed on behalf of the clients that precede *c* in the execution queue. If *c* conflicts with other clients, a cycle detection algorithm is initiated to find possible cycles in the serialization graph. In R4 and R5, $conf\_set_x$ denotes a set that holds the IDs of the clients that conflict with *x* and precede *x* in *EQ*. A cycle-chasing probe, say *msg*, records the set of IDs of the clients that have been reached by the *msg*. The set of IDs is denoted as *msg.visited*. Initially, *msg.visited* is set to empty (line 1 in R4). The cycle-chasing message is passed to the other clients using *SendCycleChasingMsg*. Parameter *sender* is the ID of the client that forwards the cycle-chasing message to the other clients. Parameter *msg* is the message being forwarded along the edges of the serialization graph. *msg* is only forwarded to the clients that are likely to form cycles with *sender* (i.e. the clients that are executing the composite services in *sender*'s potential conflict set). In line 4 of R4, *c.cs* denotes the composite service executed by client *c*. As a client might access multiple service providers, the message needs to be propagated to all the service providers visited by the client to ensure that all possible paths are checked (line 7 of R4).

```
R4  when receive the result of executing client c's
        request:
1   create a message $msg$;  $msg.visited = \phi$
2   SendCycleChasingMsg($c$, $msg$, $conf\_set_c$)

3   SendCycleChasingMsg($sender$, $msg$, $conf\_set_{sender}$)
4   for_each $c$ in $conf\_set_{sender}$ s.t. $c.cs \in PC_{sender.cs}$ do
5       let $m$ be a copy of $msg$
6       $m.visited = m.visited \cup \{sender\}$ ;  $m.receiver = c$;
7       send $m$ to all the sites in $c.SP$
8   end_for_each
```

When a cycle-chasing message is received, if it is found that the receiver of the message has already been reached previously (line 1 in R5), it means that a cycle has been detected. As a result, the clients involved in the cycle are rolled back (lines 2-4 in R5). If no cycle has been formed yet, as explained in R4, the cycle-chasing message is propagated to the clients that have the potential to form cycles (lines 5-7 in R5).

```
R5  when receive $cycle\_chasing$ message $m$:
1   if $m.receiver \in m.visited$ then
2       for_each $client \in m.visited$ do
3           send rollback message $RB(client)$ to
                all the execution coordinators in $client.SP$
4       end_for_each
5   else
6       $m.visited = m.visited \cup \{m.receiver\}$;
7       SendCycleChasingMsg($m.receiver$, $m$, $conf\_set_{m.receiver}$)
    end_if
```

R6 specifies how the rollback is propagated through the execution queues. When a rollback message is received, the operation of the rolled back client is compensated and the client is notified through the raised exceptions (line 2 of R6). The rolled back clients are recorded in *re-execute* for preventing repeated rollback as explained in R1. All the operations that have been executed after the rolled back operations (line 3 of R6) need to be rolled back. The rollback message needs to be propagated to all the service providers accessed by the client (lines 4-5 of R6).

```
R6  when receive a rollback message RB(client):
1   find an element e in EQ s.t. e.client = client;
2   compensate(e);  raise exception to client;
    add client to re-execute;
3   let $rb\_set = \{e' \mid (e' \in elem(EQ)) \wedge front(e, e')\}$
4   for_each $x \in rb\_set$ do
5       send rollback message RB(x.client) to all
            the execution coordinators in (x.client).SP;
    end_for_each
```

As described in section III.*C*, the workflow analyser inserts some operations at the end of each composite service for checking the termination of the composite service. On receipt of the *check_termination* message, the execution coordinators involved in executing the client's operations will determine whether the client can terminate. In R7, *OP* is a set that includes all the operations of client *c*. *EQS* is the set that contains all the execution queues in the system. As explained in section III.*B*, a client can only terminate once all the clients that it depends on have completed (i.e. the condition in line 1 of R7). When a client terminates, all its operations are removed from the execution coordinators' queues (line 2 of R7).

```
R7  when receive a check_termination message from
        a client, say c:
1   wait until the following condition becomes true
    $\forall e : OP. \forall eq : EQS.((e \in elem(eq)) \rightarrow$
        $(\forall e' : eq.(front(e', e) \rightarrow \neg conflcit(e, e'))))$
2   remove c's operations from their execution queues
    remove c from all re-execute sets in c.SP (if needed)
```

IV. Performance Evaluation

Experiments were carried out to compare the performance of the proposed scheme with some related schemes. In the experiments, the execution coordinators and the service providers are implemented using WCF [13]. The database being used is SQL-Server 2005.

The first experiment compares the *strict two-phase locking scheme* (S2PL) [2] and the scheme in this paper. To make the experiment condition more favorable to S2PL, two composite services are regarded as conflicting with each other if they access at least one common service provider. The conflict rate is defined as "(*the number of composite services that conflict with at least one other composite service*)/(*the total number of composite services*)". The experiments measured how the two schemes perform by varying the rate of the conflict amongst the clients. In each experiment, each composite service accesses three service providers that are chosen randomly. It is assumed that the system has sufficient resource to handle the client's execution. That is, a client is only blocked due to concurrency control. The improvement rate is defined as "(*the response time to client under the S2PL scheme - the response time to client under the proposed scheme*)/ *the response time to client under the S2PL scheme*". From Figure 3, it can be seen that the scheme in this paper performed better than S2PL even if the conflict rate is low (i.e. an improvement of 11.5% when the conflict rate is 10%). When the conflict rate is high, the proposed scheme outperforms the S2PL by a large margin (i.e. an improvement of 36.2% when the conflict rate is 50%).

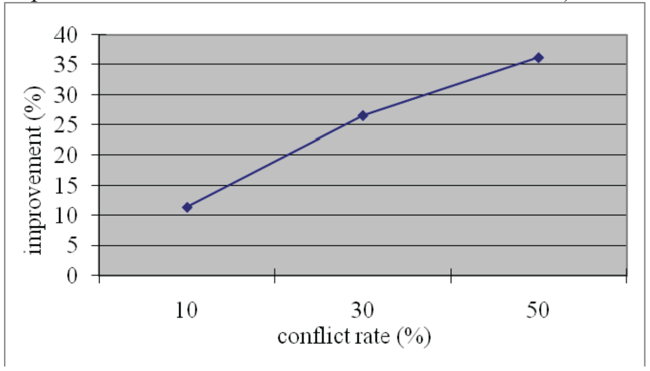

Figure 3. Comparison with the S2PL Scheme

The second set of experiments compares the the proposed scheme with the schemes in [1, 18]. In these experiments, it is assumed that two clients share at least one service provider. For the schemes in [1, 18], two clients conflict with each other if they access the same service provider. As discussed in section III.*D*, for the scheme in this paper, two clients sharing a service provider do not necessarily conflict with each other. In the experiments, the conflict rate for the scheme in this paper is set to 50% (i.e. half of the clients that share service providers conflict with each other). Conflicts amongst the composite services can cause cycles being formed in the serialization graph. Breaking the cycles involves rolling back some of the operations. The experiment assumes that undo an operation takes the same amount of time as carrying out the operation. Thus, the amount of cycles influences the efficiency of the schemes. The experiments compare the proposed scheme with the ones in [1, 18] when 10% and 50% of the conflicting clients form cycles. Figure 4 shows the ratios between the response time of the schemes being compared, i.e. "*the response time to the client under the proposed scheme / the response time to the client under* [1, 18]". *xy10* and *xy50* denote the ratios when the percentage of the clients that form cycles in the serialization graph are 10% and 50% respectively.

The scheme in [18] prevents the forming of cycles at the cost of causing possible delays to the execution of the clients that do not conflict with each other as the clients are executed in the order of their reserved positions. From *yc10* and *yc50*, it can be seen that, when there are more cycles (i.e. for *yc50*), the performance of the proposed scheme becomes closer to the scheme in [18]. This is because the time used to roll back the clients offsets the gain obtained in avoiding the unnecessary delay in executing the clients.

As the scheme in this paper uses run-time analysis to eliminate some false dependency amongst the clients, compared with [1], the clients under the scheme in this paper have fewer conflicts with the other clients. As a result, fewer clients need to wait for other clients in order to terminate. Hence, as shown in *adn10* and *adn50*, the response time of the clients under the proposed scheme is shorter than the scheme in [1]. Cycles in the serialization graph can also be formed in [1]. From *adn10* and *adn50*, it can be seen that the response time of the two schemes is similar under different percentages of cycles. This is because both schemes have overheads for breaking the cycles.

The experiments also compared the number of cycle-chasing probes of the proposed scheme and the scheme in [1]. The ratios between the numbers of probes of the two schemes, i.e. "*the number of probes generated by the proposed scheme / the number of probes generated by* [1]" are shown as *msg-adn10* and *msg-adn50* in Figure 4. From *msg-adn10* and *msg-adn50*, it can be seen that the proposed scheme generates less probes. This is because the scheme in [1] propagates the probes to all the composite services that conflict with each other, while the proposed scheme only propagates the cycle-chasing probes to the clients if there is a possibility that a cycle might be formed by the clients.

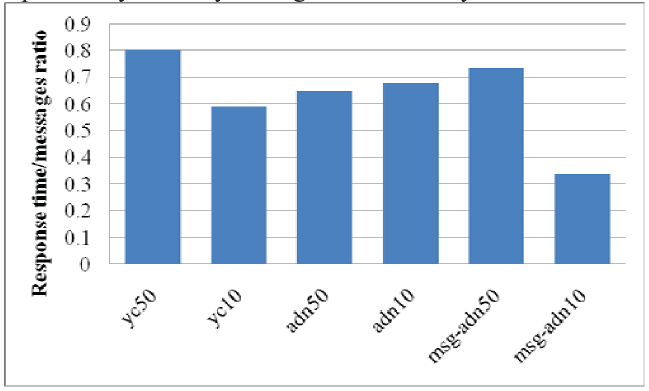

Figure 4. Comparisons with Other ACID-relaxed Schemes

V. Conclusion

The scheme in this paper ensures the serilizability of the concurrent execution of the composite services. As the detection of the cycles in the serialization graph is carried out concurrently with the execution of the clients, the detection does not slow down the execution of the composite services. The information gathered during the off-line analysis of the composite services' workflows can effectively reduce the overhead in detecting cycles in the serialization graph and avoid the repeated rollback problem. Using the run-time analysis of the SQL statements, the proposed scheme can decide whether the clients conflict with each other more accurately than other schemes. Consequently, it reduces the amount of unnecessary waiting amongst the clients. Hence, compared to the other schemes, the proposed scheme shortened the response time for clients and incurred less overhead.

References

[1] M. Alrifai, P. Dolog and W. Nejdl, Transactions Concurrency Control in Web Service Environment, Proc. of the European Conf. on Web Services, pp 109-118, 2006

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

[3] K. Birman and T. Joseph. Reliable communications in the presence of failures. ACM Trans. Comput. Syst., 5(1), 1987

[4] Business Process Execution Language for Web Services, version 1.1, http://download.boulder.ibm.com/ibmdl/pub/software/ dw /specs/ws-bpel/ws-bpel.pdf, accessed on 28/02/2010

[5] S. Choi, H. Jang, H. Kim, J. Kim, S. Kim, J. Song, and Y. Lee. Maintaining consistency under isolation relaxation of web services transactions. In Proc. of Web Information Systems Engineering, 2005

[6] K. Haller, H. Schuldt, and C. Türker. Decentralized coordination of transactional processes in peer to peer environments, Proc. of the 14th ACM Intl. Conference on Information and Knowledge Management, pp 36-43, 2005.

[7] T. Härder and A. Bühmann, Value complete, column complete, predicate complete, The VLDB Journal, 17(4):805-826, 2008

[8] H. Kung and J. Robinson. On optimistic Methods for Concurrency Control. ACM TODSs, 6(2), 1981.

[9] E. Levy, H. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In Proc. of ACM SIGMOD, pages 88-97, 1991.

[10] Mendelson, E. Introduction to Mathematical Logic, 4th ed. London: Chapman & Hall, p. 30, 1997.

[11] Tarjan, R.: Depth-first search and linear graph algorithms. In: SIAM Journal on Computing. Vol. 1 (1972), No. 2, P. 146-160

[12] G.Weikum and G.Vossen, Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. Morgan Kaufmann, 2001.

[13] Windows Communication Foundation, http://msdn.microsoft.com/en-us/library/ms735119.aspx, accessed on 28/02/2010

[14] J. Yang et al., Lock-Free Consistency Control for Web 2.0 Applications, Proc. of the 17th Intl. conference on World Wide Web, 2007.

[15] L. Li, C. Liu, J. Wang: Deriving Transactional Properties of Composite Web Services, Proc. Of the 2007 IEEE Intl. Conf. on Web Services, pp 631-638.

[16] Microsoft BizTalk Server, http://msdn.microsoft.com/en-us/library/bb905520.aspx, accessed on 28/02/2010

[17] H. Garcia-Molina and K. Salem, Sagas, Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, ACM Press (1987), pp. 249–259

[18] Ye, X. and Chen, Y., 'Concurrency Control for Transactional Composite Services', Proc. Of 2009 IEEE World Congress on Services, p.781-788