July 2006, Vol.21, No.4, pp.529–536 J. Comput. Sci. & Technol.

Automatic Transaction Compensation for Reliable Grid Applications

Fei-Long Tang[1] (唐飞龙), Ming-Lu Li[1] (李明禄), and Joshua Zhexue Huang[2] (黄哲学)

[1] *Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200240, P.R. China*

[2] *E-Business Technology Institute, The University of Hong Kong, Hong Kong Special Administration Region, P.R. China*

E-mail: {tang-fl,li-ml}@cs.sjtu.edu.cn; jhuang@eti.hku.hk

Revised May 20, 2006.

Abstract As grid technology is expanding from scientific computing to business applications, service oriented grid computing is aimed at providing reliable services for users and hiding complexity of service processes from them. The grid services for coordinating long-lived transactions that occur in business applications play an important role in reliable grid applications. In this paper, the grid transaction service (GridTS) is proposed for dealing with long-lived business transactions. We present a compensation-based long-lived transaction coordination algorithm that enables users to select results from committed sub-transactions. Unlike other long-lived transaction models that require application programmers to develop corresponding compensating transactions, GridTS can automatically generate compensating transactions on execution of a long-lived grid transaction. The simulation result has demonstrated the feasibility of GridTS and effectiveness of the corresponding algorithm.

Keywords service grid, long-lived transaction, compensating transaction, algorithm

1 Introduction

Grid technology enables people to utilize computing and storage resources transparently. By providing service oriented computation and data infrastructures, grid technology is expanding from scientific computing to business applications[1−4].

Business applications require highly reliable support from a computation platform. As an effective and widely-used means, transaction technology can help people to create reliable applications and provide application developers with multiple transparencies on location, replica, concurrency and failure[5]. In service grid, a transaction is defined as a set of operations that execute on different grid services. The transaction service is responsible for coordination of these services to keep the system consistent and free from various failures. It shields users from the complex recovery process. Owing to the autonomous, dynamic and heterogenous properties of grid services, however, the existing transaction technologies are not directly applicable to service grid.

A long-lived transaction is associated with a business process that lasts for a long time that can be a few hours or even a few days. A long-lived transaction often consists of a set of sub-transactions that have to be executed to complete the transaction. Transaction compensation is an appropriate method to release grid resources being held by sub-transactions as early as possible. For example, in an e-shopping process, an enterprise E orders a set of machines from service A, applies for a shipment from service B, and reserves a storage from service C. If a sub-task fails, other two sub-tasks have to be cancelled. On the other hand, the user E may apply for more orders for the shipment and then selects the "best" (e.g., the cheapest) one and cancels others. In the above two situations, cancellation of submitted sub-task(s) (sub-transactions) should use compensating transactions. In existing transaction models, however, application programmers have to develop compensating transactions, which is impracticable in the grid environment because service providers need to setup special compensating rules.

In this paper, we present the grid transaction service (GridTS) for dealing with long-lived business transactions and a coordination algorithm that enables users to select results from committed sub-transactions. Especially, we focus on how to automate the generation of compensating transactions for service grid. Our motivation is to provide a transaction service based on automatic compensation for grid applications. Our simulation result has demonstrated the feasibility of GridTS and effectiveness of the corresponding algorithm.

The remainder of this paper is organized as follows. In Section 2, we review related work. The GridTS and the coordination algorithm are presented in Section 3 and Section 4 respectively. In Section 5, we investigate how to automatically generate compensating transactions. The comparison and experimental result are reported in Section 6. Finally, Section 7 concludes the paper with the discussion of our future work.

2 Related Work

Existing models for long-lived transactions were generally built on compensating transaction that was first proposed by Gray[6,7]. The typical implementation of

Regular Paper

This work is supported by the National Basic Research 973 Program of China (Grant No. 2002CB312002), the National Natural Science Foundation of China (Grant Nos. 60473092 and 90612018), Natural Science Foundation of Shanghai Municipality of China (Grant No. 05ZR14081), and ShanghaiGrid from Science and Technology Commission of Shanghai Municipality (Grant No. 05DZ15005).

compensating transactions is the Sagas model that is widely used in many extended transaction models[8−11].

Sagas[8] is a classical transaction model for handling long-lived transactions, based on transaction compensation. In Sagas, a transaction is called a "Saga", which consists of a set of sub-transactions with ACID (atomicity, consistency, isolation, durability) properties $T = \{T_1, T_2, \ldots, T_n\}$, and a set of associated compensating transactions $C = \{C_1, C_2, \ldots, C_n\}$, where each sub-transaction T_i associates with a compensating transaction C_i that can semantically undo the effect caused by the commit of T_i. Sub-transactions in Sagas independently commit and immediately release resources accessed in the execution of the sub-transactions in order to reduce the duration of resource lock and improve the system efficiency. In Sagas, all the committed sub-transactions must be undone if a subsequent sub-transaction fails, which causes waste of a lot of valuable work already finished.

ACTA[11] is a comprehensive transaction framework that permits a transaction modeler to specify the effects of extended transactions on each other and on objects in the database. ACTA allows to specify interactions between transactions in terms of relationships and transactions' effects on objects' state and concurrency status. ACTA provides a reasoning ability more powerful and flexible than Sagas through a series of variations to the original Sagas.

ConTracts[12] is a mechanism for grouping transactions into a multi-transaction activity. It consists of a set of predefined actions called steps, and an explicitly specified execution plan called a script. In case of a failure, the ConTract state must be restored and its execution may continue.

The above long-lived transaction models require application programmers to provide compensating transactions beforehand for all sub-transactions. These models are database-centric and primarily aimed at preserving the consistency of shared data. Thus, they are generally not applicable for applications that comprise of loosely coupled, Web-based business services[13]. In the Business Transaction Protocol (BTP)[14] and Web Services Transaction (WS-Transaction)[15], the use of compensation for coordination of long-running activities was proposed, but no details are given on how to provide compensating transactions.

3 Grid Transaction Service

3.1 Layered Architecture

The architecture of grid transaction is divided into three layers (see Fig.1). The middle layer, GridTS, is a special grid service responsible for management of long-lived grid transactions. It consists of the following main components.

Coordinator and Participant. They cooperatively coordinate a transaction for an application and grid services respectively. The coordinator and participant themselves do not execute actual application operations.

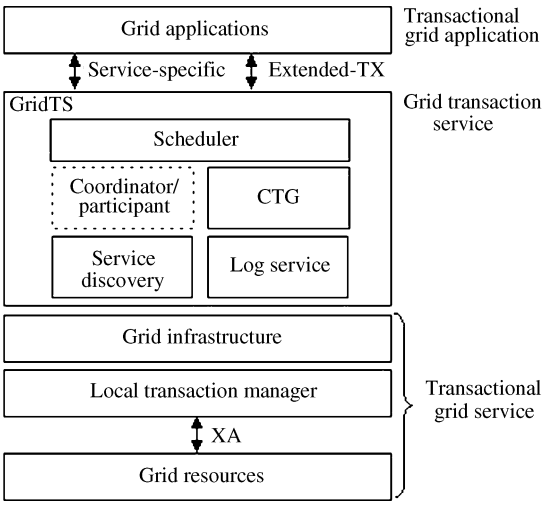

Fig.1. Architecture of grid transaction.

Scheduler. This component takes charge of (1) creating a coordinator and a coordination context (CC) on the application side and participants on the service side, and (2) scheduling the Service Discovery module.

Compensating Transaction Generator (CTG). If a predefined event occurs, the component first queries the corresponding compensating rule(s), and then dynamically generates a compensating operation. Finally, it encapsulates the generated compensating operations into a compensating transaction when the sub-transaction commits.

Log Service. This component records the coordination operations and the state information for recovery of transactions from failures.

Service Discovery. This component dynamically discovers qualified grid services according to users' requirements, such as cost, quality and availability, to complete specified sub-transactions. Further information can be found in [16–19].

Interfaces. GridTS provides grid applications with two types of APIs: the extended TX interfaces for transaction management, and the service-specific interfaces for management of the GridTS service instances and discovery of grid services to execute application operations in sub-transactions.

3.2 How to Use GridTS

GridTS is a special grid service and possesses all properties of a grid service. Interfaces of GridTS are encapsulated in TX portType of grid services by defining each interface, corresponding input and output parameters as operation, input and output messages. The interface definition is exemplified in Fig.2.

GridTS ensures the reliability for grid applications through the following ways.

1) *Public transaction service.* GridTS is published in the public registration center. Transactional applications discover and invoke the GridTS. The advantage in this way is flexible and convenient, which means that users may share reliability support without installing the GridTS.

2) *Private transaction service.* The GridTS locates on the application-side and service-side nodes. The strength of this method is efficiency and the weakness is less flexibility.

```
⟨gwsdl: portType name = "TX" extends = "ogsi:GridService" ⟩
  ⟨operation name = "Begin" ⟩
    ⟨input message = "tns:txType"/⟩
    ⟨output message = "tns:CoordinationContext"/⟩
    ⟨fault name = "Fault" message = "ogsi:FaultMessage"/⟩
  ⟨/operation⟩
  ...
⟨/gwsdl:portType⟩
```

Fig.2. Definition of interface begin in GridTS.

4 Coordination of Long-Lived Grid Transaction

A long-lived grid transaction (LGT) T consists of a set of sub-transactions that execute on different grid services, formally described as $T = \{T_i | T_i \in T,\ 1 \leqslant i \leqslant n,$ n is the number of sub-transactions involved in $T\}$. As the name suggests, an LGT takes a relatively long time to finish, even without the interference from other concurrent transactions, so the LGT relaxes the atomicity and isolation properties. Grid services that join an LGT independently commit sub-transactions after receiving the pre-commit message, and then immediately release the held resources.

The coordination algorithm of an LGT (CALGT), as shown in Fig.3, allows users to confirm or cancel committed sub-transactions according to their own requirements. The algorithm consists of two parts, the coordinator algorithm ActionOfCoordinator and the participant algorithm ActionOfParticipant, where t is the system time, CC is a coordination context, and T_{valid} is the valid time before which a coordinator must send a confirmation or cancellation decision and participants must report their commit states. Otherwise, if a coordinator does not confirm or cancel a sub-transaction before T_{valid}, the corresponding participant automatically undoes the committed sub-transaction by the compensating transaction. On the other hand, a coordinator presumes that a participant has failed if the participant does not return the commit result before T_{valid}. The state diagrams of a Coordinator and a Participant in an LGT are respectively depicted in Figs.4(a) and 4(b), where messages close to solid lines and break lines come from a Coordinator and a Participant respectively.

```
ActionOfCoordinator
Input: references of all participants, time parameters;
Output: global transaction results or failure;
{step 1: initiate an LGT
Scheduler creates a Coordinator;
completed=false;
while (t ⩽ T_valid and not completed) {
 Scheduler sends CC to Participants;
 wait for Response messages;
step 2: enroll participants
 send Enroll to Participants;
step 3: confirm/cancel participants
 wait for and record incoming messages;
 if (message is Enrolled)
  if (user selects some)
   send Confirm to them;
   wait for Comfirmed;
  } else {
   send Cancel to them;
   wait for Cancelled; }
 if (LGT completes successfully)
  completed=true;
} }
```

(a)

```
ActionOfParticipant
Input: CC and time parameters;
Output: T_i' results or failure;
{step 1: join in the transaction
Scheduler creates a Participant after receiving CC;
send Response to Coordinator;
step 2: commit sub-transaction
wait for Enroll from Coordinator;
if timeout exit;
   allocate resources;
   record commit information in log;
   commit and generate compensating transaction;
   release resources;
//nested transaction, call ActionOfCoordinator;
step 3: confirm/compensate
if (commit successfully) {
 send Enrolled to Coordinator;
 while (t ⩽ T_valid) {
  wait for incoming messages;
  if (message is Cancel) {
   call its compensating transaction;
   send Cancelled;
  } else {
   if (message is Confirm)
    send Confirmed;}}}}
```

(b)

Fig.3. Coordination algorithm of an LGT (CALGT). (a) Coordinator algorithm. (b) Participant algorithm.

5 Automatic Generation of Compensating Transaction

The LGT uses compensating transactions to undo committed sub-transactions. In the existing compensation-based long-lived transaction models, application programmers generally have to define and implement compensating transactions, which is impracticable in service grid. There are two major reasons: (1) owing to the autonomy of grid services, some compensating actions are service-specific and it is difficult for application programmers to know special compensating policies of services discovered dynamically; (2)

it imposes heavy burden and programming complexity on application development. On the other hand, there are common compensating rules with which each transaction complies. We define the common compensating rules according to the types of operations, while allowing service providers to add and modify their own rules.

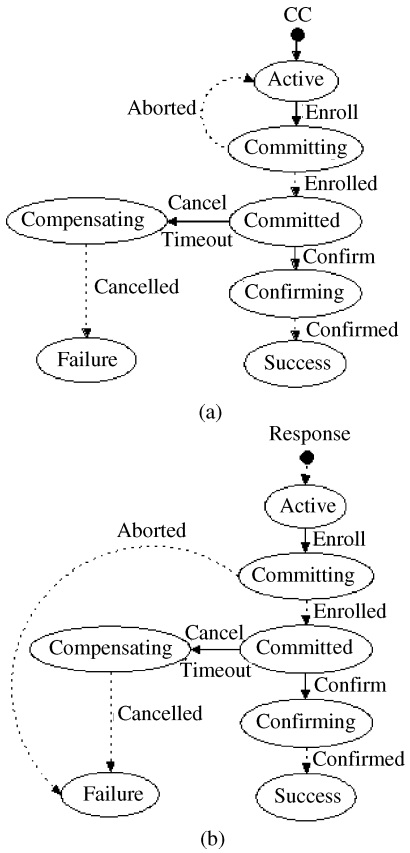

Fig.4. State diagrams of an LGT. (a) Coordinator. (b) Participant.

5.1 Key Technologies for Automatic Transaction Compensation

Compensating actions are closely related to system states. The states describe current properties and possible further action(s) of a transaction system. For example, we can describe the state of an airline booking system as $S = \{reservation, available\}$, where *reservation* is the number of available tickets, and *available* indicates whether the system can accept new reservations or not. If *reservation* is greater than 0, *available* becomes true; otherwise, *available* becomes false.

States are changed by operations. However, not all operations affect system states. For example, the update $(d1, d2)$ operation changes the data value from $d1$ to $d2$ and the Enroll message transfers the Participant state from Active to Committing, but reading a data value does not affect the system state.

Definition 1. *A compensating transaction* (CT) *is the transaction that rollbacks the operations taken by a committed transaction* T *and undoes semantically the effects from the commit of original transaction* T.

A compensating transaction mainly involves in two aspects. One is to undo the effects of the original operations, and the other is to recover the system consistency. The key technologies to generate automatically compensating transactions include:

- definition of compensating rules,
- generation of compensating operations in the execution of a long-lived transaction, and
- generation of a compensating transaction at the commit of a sub-transaction.

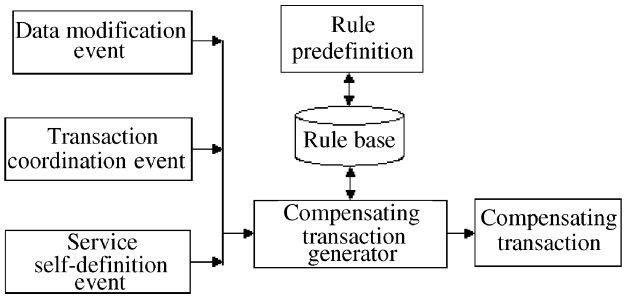

Fig.5. Architecture of generating compensating transactions.

5.2 Set Compensating Rules

Generation of a compensating transaction is event-driven. Compensating rules indicate how to undo the effects from events that change system states. We divide these events into three types: data modification event, transaction coordination event and service self-definition event (see Fig.5). Compensating rules for the first two types of events are provided by GridTS while rules for service self-definition events are set by service providers through the following interfaces:

setCompensatingRule (): sets compensating rules for grid services;

getCompensatingRule (): gets compensating rules of grid services.

5.2.1 Data Modification Event

Currently, most companies store their information in relation databases. Data modification operations in an LGT mainly consist of insertion, deletion and replacement of records in databases.

Definition 2. *A data modification event refers to insertion, deletion or modification of data in a databases. Let* $e_{T_i}[p(d)]$ *be a data modification event from which transaction* T_i *modifies data* d *using operation* p, *where* $p \in OT$ *belongs to one of operation types. Furthermore,* DE_{T_i} *is a set of data modification events caused by* T_i *and* $e_{T_i}[p(d)] \in DE_{T_i}$.

For a relation database, OT={update, insert, delete}. We mainly analyze how to compensate these three data modification operations.

Let S_i and S_{i+1} be the states before and after T_i commits respectively, CT_i a compensating transaction

of T_i, and T_j ($j \neq i$) a dependent transaction that executes between T_i and CT_i. If data accessed by T_i is not modified by T_j, CT_i simply executes a reversed action for each operation in T_i. Otherwise, CT_i undoes the committed transaction T_i, but may not change the results of the dependent transaction T_j. For example, the cancellation of Alice's airline ticket reservation cannot affect Bob's reservation. Compensating rules for update, insert and delete operations are set as follows.

(1) *Update*

Let $op_i = \text{update}(d1, d2)$ be an operation in T_i that replaces $d1$ with $d2$. How to compensate op_i depends on the data modification operation op_j in T_j.

① An insert operation $op_j = \text{insert}(d)$ in T_j does not affect the result of op_i. The compensating operation for op_i is $cop_i = \text{update}(d2, d1)$.

② A delete operation $op_j = \text{delete}(d2)$ in T_j will delete the result of op_i. As a result, it is not necessary to compensate op_i.

③ An update operation $op_j = \text{update}(d2, d3)$ in T_j will change the result of op_i. The compensating operation of op_i depends on the type of the replacement operation.

Relevant replacement $S_{i+1} = f(S_i, T_i)$. It means that the state S_{i+1} is relevant to the state S_i. cop_i has to remove the effect of op_i. For example, if $op_i = \text{update}(d1, d2)$ and $d2 = d1 + n$, the corresponding compensating operation is $cop_i = \text{update}(d3, d4)$, where $d4 = d3 - n$.

Irrelevant replacement $S_{i+1} = f(T_i)$, where S_{i+1} is irrelevant to S_i, e.g., op_i = update("Monday", "Tuesday") and op_j=update("Tuesday", "Wednesday"). Such a replacement need not be compensated.

(2) *Insert*

Let operation $op_i = \text{insert}(d1)$ in T_i insert a record with value $d1$. cop_i is also relevant to data modification operations op_j in T_j.

① $op_j = \text{insert}(d2)$ does not affect the result of op_i so that the compensating rule for op_i is $\text{cop}_i = \text{delete}(d1)$.

② $op_j = \text{delete}(d1)$ will delete the result of op_i, however, op_i need not be compensated in order to keep the result of op_j.

③ $op_j = \text{update}(d1, d2)$ is compensated as follows.

If ($d2 \neq d1$)
 If (relevant replacement) {
  temp=change caused by $op_j = \text{update}(d1, d2)$;
  insert (*temp*);
  delete ($d1$); }
 else
  do nothing;

(3) *Delete*

A delete operation $op_i = \text{delete}(d)$ in T_i deletes a record with the value d. Any operation in T_j cannot affect the result of op_i so that the compensating operation for op_i is simply a reversed operation $cop_i = \text{insert}(d)$.

5.2.2 Transaction Coordination Event

Definition 3. *A transaction coordination event denotes that a sub-transaction receives messages from a coordinator. The set of transaction coordination events involved in a sub-transaction T_i is $TE_{T_i} \subset \{CC, Enroll, Confirm, Cancel\}$.*

Each transaction coordination event changes the state of a transaction system. The GridTS sets the compensating rules for the transaction coordination event in the following way.

1) For *CC* message, it records the original transaction identifier and input parameters.

2) For *Enroll* message, it encapsulates compensating operations in delimiters Begin and Commit, and stores the compensating transaction CT_i in a database.

3) For *Cancel* messages, it invokes CT_i stored in the database.

4) For *Confirm* message, it deletes CT_i in the database because CT_i will be useless after the sub-transaction is confirmed.

5.2.3 Service Self-Definition Event

Definition 4. *A service self-definition event refers to the actions that a service provider takes according to the states of a business process, which depends on the special business model of a service provider.*

The compensating rules for the service self-definition event are defined by service providers. They typically focus on:

- subsequent activities after undoing operations in an original transaction, e.g., sending an email to notify the user of new available services;
- economic compensation, for example, if a user cancels a committed sub-transaction which has finished a transportation order, the transportation company typically requires amends from the user.

5.3 Generate Compensating Operations

In the execution of an LGT, the Compensating Transaction Generator (CTG) of GridTS monitors events, such as a delete operation or an Enroll message. Once predefined events occur, CTG examines whether the conditions for a rule are satisfied. If so, it extracts the type and parameters of the operation, queries corresponding compensating rules of the operation, generates a compensating operation, and records input parameters. For example, when a sub-transaction deletes a record from the database, the Delete event will generate a compensating operation to insert the record.

5.4 Generate and Invoke Compensating Transactions

The Enroll message enables the CTG to generate delimiters Begin and Commit, and combines the compensating operations into a transaction. If the sub-

transaction fails, all compensating operations generated previously are abandoned. A compensating transaction is stored in a database, and deleted from the database when GridTS receives a Confirm message from the Coordinator.

In an LGT, both the Cancel message and a timeout signal, which is generated after the transaction deadline T_{valid}, can start the corresponding compensating transaction.

5.5 Handle Noncompensable Transaction

A transaction is compensable if effects from its commit can be semantically undone by another transaction, i.e., the corresponding compensating transaction. Otherwise, the transaction is noncompensable.

Actual enterprise applications are complex. Currently, our automatical compensation mechanism can work in a restricted environment. One condition is that an original transaction has to be compensable. Another is that data modification operations in a transaction do not cause successional processing that changes other data, such as cascade delete.

A compensating transaction consists of a set of compensating operations. A transaction T is compensable, if and only if each operation $\mathrm{OP}_i \in T$ has a corresponding compensating operation COP_i. Some transactional grid applications comprise noncompensable operations so that these transactions are noncompensable. Generally, noncompensable operations can be divided into two types:

1) difficult compensating operations such as the sale of stocks bought previously, which means that the execution of these compensating operations may cause unexpected results;

2) unable compensating operations, which refer to the operations that cannot be compensated. For example, it is impossible to compensate a launched missile.

Noncompensable operations often generate effects on outside activities so that, in general, their effects are not allowed visible out of these applications. Thus, GridTS does not allow such a sub-transaction to commit in the pre-commit phase if it cannot find compensating rule(s) for an operation. Instead, we handle noncompensable transactions with the following policies.

• GridTS imposes commit dependence between the sub-transaction and the global transaction, which indicates that the sub-transaction actually commits only if the global transaction commits.

• GridTS rollbacks operations taken previously but returns the Committed message to the coordinator. After receiving the Confirm message, it redoes and commits the sub-transaction.

• GridTS rollbacks the executed operations and reports a commit exception to a user. The latter decides how to handle the exception.

6 Evaluation of the Coordination Algorithm CALGT

6.1 Comparison with Other Related Work

We compare our work with other long-lived models in the following two ways.

1. Generation of compensating transactions. Existing long-lived transactions models were generally built on compensating transactions, both in the traditional distributed system and in the Web Services environment. The compensating transaction was first implemented in Sagas that requires application programmers to provide compensating transactions before a transaction execution. BTP and WS-Transaction[14,15] only mentioned to use compensation to reverse the effects of completed business tasks, but they did not propose how to generate compensating transactions.

Different from these work, the GridTS proposed in this paper can automatically generate compensating transactions. We predefine common compensating rules for data modification operations and transaction coordination messages, and allow service providers to modify and add compensating rules. In the execution of an LGT, the GridTS dynamically generates and stores a compensating transaction for each subtransaction based on the compensating rules. On receiving a Confirm message, indicating that the result(s) of the sub-transaction will not change from then, the GridTS deletes the generated compensating transaction from the database.

2. Number of messages. BTP has clearly influenced heavily and positively the development of Web services transactions[13]. It uses Cohesion to model long-running business activities and defines a set of messages to manage a Cohesion transaction. Currently, no influential Grid transaction model can be found. We compare the number of messages that are issued by our CALGT with the BTP. An LGT may proceed even if some subtransactions fail by reselection of other services. Therefore, we only consider the number of messages in case that a transaction successfully commits.

In execution of the BTP protocol, a committed transaction requires three messages by each subordinate (enrolled, prepared and confirmed/cancelled) and four messages by the coordinator (context, enroll, prepare and confirm/cancel) to each subordinate. Therefore, let n be the number of sub-transactions in a transaction that has successfully finished. The number of messages that cross the network in the BTP is

$$M_{\mathrm{BTP}} = 7n.$$

In case that a broadcast communication mechanism is available, the $4n$ messages sent by the coordinator are substituted by four broadcast messages. The number of messages is

$$M_{\mathrm{BTP}}^{(b)} = 3n + 4.$$

Our coordination algorithm CALGT, in the absence of a broadcast facility, requires two messages from each participant to a coordinator (Enrolled and Confirmed/Cancelled) and three messages from the coordinator to each participant (CC, Enroll and Confrim/Cancel). Thus, the number of messages for a successful LGT is

$$M_{\mathrm{CALGT}} = 5n.$$

If broadcasting of messages is available, the coordinator only sends three messages for an LGT so that we have

$$M^{(b)}_{\mathrm{CALGT}} = 2n + 3.$$

The number of messages in CALGT, which are expressed as a fraction of the messages exchanged in BTP or point-to-point messaging, is

$$K = \frac{M_{\mathrm{LGTCA}}}{M_{\mathrm{BTP}}} = \frac{5}{7}.$$

If messages are exchanged in a broadcast way, we have

$$K^{(b)} = \frac{M^{(b)}_{\mathrm{LGTCA}}}{M^{(b)}_{\mathrm{BTP}}} = \frac{2n+3}{3n+4}.$$

An important observation is the fraction $K^{(b)} \in (2/3, 5/7]$.

From the above analysis, we can find that the number of messages exchanged in our algorithm CALGT is less than that in BTP, no matter in point-to-point messaging or broadcast way.

6.2 Performance Testing

To validate the performance of the CALGT, we have developed a prototype system. The system was built in a small scale intra-grid. GridTS and the associated grid service that actually executes application operations in a sub-transaction are installed on each node of the system. GridTS is a persistent service, while the latter is implemented as a transient service. GridTS creates a Coordinator and a Participant upon receiving a request to initiate a transaction and a CC message respectively. The created Coordinator and Participant then interact a set of coordination messages to control the outcome of an LGT.

GridTS provides long-lived transactional grid applications with the following interfaces:

- Begin(): initiates an LGT;
- Enroll(): notifies a grid service that has joined the transaction for committing a sub-transaction;
- Confirm(): confirms the commit of the grid service;
- Cancel(): requires committed sub-transactions to execute the corresponding compensating transactions;
- GetTransactionStates(): queries states of a transaction.

The system randomly generates transactions to simulate requests from different users. The system workload is modeled as the maximal number of concurrent transactions, which are created randomly within the interval 60ms. The number of application operations to modify data in a sub-transaction is represented in Tran_Size.

We measure the performance of CALGT as the average response time (ART), referring to the average response time of all committed transactions in a given time.

$$ART = \frac{1}{n}\sum_{i=1}^{n} RT_i, \tag{1}$$

where RT_i is the response time of a finished LGT T_i, i.e., the interval from T_i's starting to coordinator's receiving all Confirmed messages.

For each T_i, the response time RT_i consists of messaging delay $t_{delay}(T_i)$ and transaction processing time $t_{processing}(T_i)$ within which the transaction coordination and application operations finish, that is

$$RT_i = t_{delay}(T_i) + t_{processing}(T_i). \tag{2}$$

In Grid, the messaging delay $t_{delay}(T_i)$ is affected by the network status. In our experiment, $t_{delay}(T_i)$ was fixed and the change of RT_i mainly resulted from $t_{processing}(T_i)$. We were focused on testing how $t_{processing}(T_i)$ changed with the system workload.

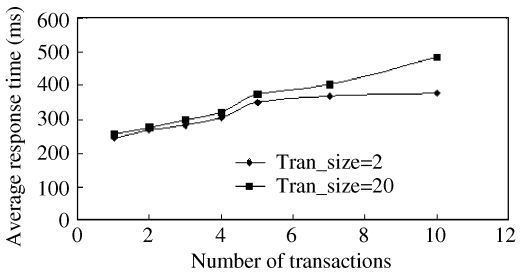

Fig.6. Performance of the coordination algorithm CALGT.

An LGT finishes if all the confirmed sub-transactions return Confirmed messages. We tested the average response time of two groups of transactions, in which Tran_Size was set to 2 and 20 respectively, as shown in Fig.6.

The response time of two groups of transactions was similar to each other when the maximal number of concurrent transactions was less than 7. However, in the group whose Tran_Size was 20, the average response time increased rapidly as the maximal number of concurrent transactions exceeded 7.

7 Conclusions and Future Work

We have proposed a transaction service and a coordination algorithm for management of long-lived business activities in service grid, based on automatic compensation. Our work has three advantages. Firstly, it can automate the generation of compensating transactions

for reliable grid applications. Next, it allows users to select committed results. Finally, it is extensible because it is built on top of a series of open standards, technologies and infrastructures. Simulation result shows the effectiveness of the coordination algorithm.

We plan to integrate security measures with the transaction service GridTS. Grid Security Infrastructure (GSI) will be used because it provides abilities for authentication, authorization and communication protection, based on the public-key mechanism, and is the de facto standard authentication method with the "single sign-on" property. Moreover, we plan to investigate the mechanism for combining the transaction management with the resource scheduling and management to enhance system efficiency.

References

[1] Foster I, Kesselman C (eds.). The Grid: Blueprint for a Future Computing Infrastructure. Morgan Kaufmann Publisher, USA, 1999.

[2] Foster I, Kesselman C, Nick J, Tuecke S. Grid Services for Distributed System Integration. *Computer*, 2002, 35(6): 37–46.

[3] Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001, 15(3): 200–222.

[4] Foster I. Service-oriented science. *Science*, May 2005, 308(5723): 814–817.

[5] Traiger I, Gray J, Galtieri C, Lindsay B. Transactions and consistency in distributed database systems. *ACM Trans. Database Systems*, Sept. 1982, 7(3): 323–342.

[6] Gray J. The transaction concept: Virtues and limitations. In *Proc. the 7th International Conference on Very Large Data Bases*, Cannes, France, Sept. 1981, pp.144–154.

[7] Gray J. Notes on database operating systems. Operating systems: An advanced course. *Lecture Notes in Computer Science* 60, Berlin, Springer-Verlag, 1978, pp.393–481.

[8] Garcia-Molina H, Salem K. SAGAS. In *Proc. The 1987 ACM SIGMOD International Conference on Management of Data*, California, United States, May, 1987, pp.249–259.

[9] Liang D, Tripathi S. Performance analysis of long-lived transaction processing systems with rollbacks and aborts. *IEEE Transactions on Knowledge and Data Engineering*, Oct. 1996, 8(5): 802–815.

[10] Garcia-Molina H, Gawlick D, Klein J *et al.* Modeling long-running activities as nested sagas. *Bulletin of the IEEE Technical Committee on Data Engineering*, 1991, 14(1): 14–18.

[11] Chrysanthis P, Ramamriham K. ACTA: The SAGA Continues. Chapter 10 of Transactions Models for Advanced Database Applications. Morgan Kaufmann, 1992.

[12] Wachter H, Reuter A. Contracts: A Means for Extending Control Beyond Transaction Boundaries. Advanced Transaction Models for New Applications, Morgan Kaufmann, 1992.

[13] Dalal S, Temel S, Little M *et al.* Coordinating business transactions on the Web. *IEEE Internet Computing*, 2003, 7(1): 30–39.

[14] Ceponkus A, Cox W, Brown G *et al.* Business transaction protocol V1.0. 2002, http://www.oasis-open.org/committees/download.php.

[15] Cabrera F, Copel G, Coxetal B. Web Services Transaction (WS-Transaction), August 2002, http://www.ibm.com/developerworks/library/ws-transpec.

[16] Tang F L, Li M L, Cao J. A transaction model for grid computing. In *Proc. The 5th International Workshop on Advanced Parallel Programming Technologies*, Lecture Notes in Computer Science 2834, Sept. 2003, pp.382–386.

[17] Li M L, Wu M Y, Li Y *et al.* ShanghaiGrid: An information services grid. *Concurrency and Computation: Practice and Experience*, Jan. 2006, 18(1): 111–135.

[18] Li M L, Liu H, Tang F L *et al.* ShanghaiGrid in action: The first stage projects towards digital city and city grid. *International Journal of Grid and Utility Computing*, 2005, 1(1): 22–31.

[19] Tang F L, Li M L, Joshua Huang Z X *et al.* Real-time transaction processing for autonomic grid applications. *Engineering Applications of Artificial Intelligence*, 2004, 17(7): 799–807.

Fei-Long Tang received his Ph.D. degree in computer science and technology from Shanghai Jiaotong University in 2005. From May 2004 to June 2005, he researched on grid computing and e-business in the E-Business Technology Institute, the University of Hong Kong. His research interests include grid computing, Web services, computer network and distributed computing, especially on grid transaction and reliability analysis.

Ming-Lu Li is a full professor and deputy director of Department of Computer Science and Engineering, Shanghai Jiaotong University, China. He also is a director of Grid Computing Center of Shanghai Jiaotong University, Grid expert of Ministry of Education, P.R. China, and expert-in-chief of ShanghaiGrid, an influential Grid project in China. His research interests mainly include grid computing, Web services, service computing and multimedia computing.

Joshua Zhexue Huang is assistant director at the E-Business Technology Institute (ETI), honorary professor at the Department of Mathematics of the University of Hong Kong (HKU) and visiting professor at the School of Computer Science of Harbin Institute of Technology of China. He has contributed to the development of a series of k-means type algorithms in data mining, including k-modes, fuzzy k-modes and k-prototypes which are being widely used in research and real world applications. His current research interests include data mining algorithms, text mining, parallel data mining and business intelligence service grid.