Design and Implementation of a Two-Phase Commit Protocol Simulator

Toufik Taibi[1], Abdelouahab Abid[2], Wei Jiann Lim[2], Yeong Fei Chiam[2], and Chong Ting Ng[2]
[1]College of Information Technology, United Arab Emirates University, UAE
[2]Faculty of Information Technology, Multimedia University, Malaysia

Abstract: *The Two-Phase Commit Protocol (2PC) is a set of rules, which guarantee that every single transaction in a distributed system is executed to its completion or none of its operations is performed. To show the effectiveness of 2PC, a generic simulator is designed and implemented to demonstrate how transactions are committed in a safe manner, and how data consistency is maintained in a distributed system with concurrent execution of randomly generated transactions. Several possible failure cases are identified and created in the system to test its integrity, thus showing how well it responds to different failure scenarios, recovers from these failures, and maintains data consistency and integrity. The simulator was developed using Java Remote Method Invocation (RMI), which is particularly powerful in developing networking systems of such scale, as it provides easy remote method calls without the need to handle low-level socket connection.*

Keywords: *2PC, transaction coordinator, transaction manager, data manager, locking manager, failure recovery, RMI.*

Received July 9, 2004; accepted December 17, 2004

1. Introduction

The world of computing is moving towards a trend where tasks are performed in a distributed manner. This is especially relevant in distributed transaction processing systems used by financial institutions, where a single transaction could result in significant changes in other parts of the system. In a distributed system, a transaction often involves the participation of multiple sites and access of shared data in remote locations. A failure of one site in committing its part of the transaction could cause the entire system to be inconsistent. Thus, some form of control is necessary to ensure that concurrent execution of transactions in a distributed environment does not jeopardize the integrity of the system as well as its data consistency. The Two-Phase Commit Protocol (2PC) is a set of rules, which guarantee that every single transaction is executed to its completion or none of its operations is performed at all [8]. This is especially important in a distributed environment, which requires atomic updates. A concurrency control mechanism is also applied to ensure synchronized access to shared data and their replica by many concurrently running transactions.

To show the effectiveness of 2PC, a generic simulator is designed and implemented to demonstrate how transactions are committed in a safe manner, and how data consistency is maintained in a distributed system with concurrent execution of randomly generated transactions.

Several possible failure cases are identified and created in the system to test its integrity, thus showing how well it responds to different failure scenarios, recovers from these failures and maintains data consistency and integrity.

2PC is of prime importance to many distributed transaction processing applications used by financial institutions and other applications that fall within the spectrum of enterprise computing. These types of applications are increasingly being used to harness the availability of commodity processing power scattered in many sites of medium-to-large scale organizations.

The simulator was developed using Java Remote Method Invocation (RMI) [6, 7], which is particularly powerful in developing networking systems of such scale, as it provides easy remote method calls without the need to handle low-level socket connection. In order to provide a standard documentation, Unified Modeling Language (UML) [2] is used in modeling and designing the simulator.

The rest of the paper is organized as follows. Section 2 gives an overview of 2PC and how failure and concurrency control are handled. Section 3 describes the system architecture of the simulator and its software architecture, which includes all the UML diagrams. Section 4 describes the implementation of the simulator, while section 5 concludes the paper.

2. Overview of the Two-Phase Commit Protocol

Atomicity is ensured when either all the operations associated with a program unit are executed to

completion, or none are performed. Ensuring atomicity in a distributed system requires a *transaction coordinator*, which is responsible for the following [8]:

- Starting the execution of the transaction.
- Breaking the transaction into a number of sub-transactions, and distributing these sub-transactions to the appropriate sites for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

The execution of 2PC is initiated by the *coordinator* after the last step of the transaction has been reached. When the protocol is initiated, the transaction may still be executing at some of the local sites. The protocol involves all the local sites at which the transaction executed. Let T be a transaction initiated at site S_i and let the transaction coordinator at S_i be C_i.

Phase 1 of 2PC is usually called "*Obtaining a decision*". The following are the actions performed during this phase [3]:

- C_i adds <*prepare T*> record to the log.
- C_i sends <*prepare T*> message to all sites.
- When a site receives a <*prepare T*> message, the transaction manager determines if it can commit the transaction.
- If no: Add <*no T*> record to the log and respond to C_i with <*abort T*>.
- If yes: Add <*ready T*> record to the log, force *all log records* for *T* onto stable storage and transaction manager sends <*ready T*> message to C_i.
- The Coordinator collects responses from all sites. If all respond "ready", the final decision is *commit*. If at least one response is "abort", the final decision is *abort*. If at least one participant fails to respond within a time out period, the final decision is *abort*.

Phase 2 of 2PC is usually called "*Recording the decision in the database*". The following are the actions performed during this phase [3]:

- The coordinator adds a decision record <*abort T*> or <*commit T*> to its log and forces the record onto stable storage.
- Once that record reaches stable storage it is irrevocable (even if failures occur).
- The coordinator sends a message to each participant informing it of the decision (*commit* or *abort*).
- Participants take appropriate action locally.

Site failure in 2PC is handled in the following manner [5]:

1. If the log contains a <*commit T*> record, the site executes *redo (T)*.
2. If the log contains an <*abort T*> record, the site executes *undo (T)*.
3. If the log contains a <*ready T*> record, consult C_i. If C_i is down, site sends *query-status T* message to the other sites.
4. If the log contains no control records concerning *T*, the site executes *undo (T)*.

Coordinator failure in 2PC is handled in the following manner [5]:

1. If an active site contains a <*commit T*> record in its log, the *T* must be *committed*.
2. If an active site contains an <*abort T*> record in its log, then *T* must be *aborted*.
3. If some active site does *not* contain the record <*ready T*> in its log then the failed coordinator C_i cannot have decided to *commit T*. Rather than wait for C_i to recover, it is preferable to *abort T*.
4. If all active sites have a <*ready T*> record in their logs, but no additional control records, there is a need to wait for the coordinator to recover.
5. Blocking problem – *T* is blocked pending the recovery of site S_i.

As for synchronizing access to shared data and their replicas, we have chosen to use a centralized approach. A single lock manager resides in a single chosen site, all lock and unlock requests are made at that site. This implementation is simple but there is a possibility for the lock manager to become a bottleneck.

3. System and Software Architectures

The proposed simulation system consists of the following components (Figure 1):

1. *Transaction Manager*: Coordinates (as a coordinator) or executes (as a participant) atomic transactions. A Transaction Manager can act as coordinator and participant at the same time.
2. *Data Manager*: Manages data transfer between its replica and other sites.
3. *Locking Manager*: Synchronizes access to the data and updates data to replicas. There will be only one *Locking Manager* in the system as discussed in section 2.

TransactionManagers, *Data Managers* and *Locking Managers* are shown as different sites in Figure 1. In reality, a site may contain a combination of *Transaction Managers*, *Data Managers* and *Locking Managers*.

Figure 2 shows the use case diagram. There is only one main use case, which is "*Perform Two-Phase Commit Protocol*". However, the use case contains other sub-use cases: "*Execute Transaction*", "*Redo*" and "*Undo*".

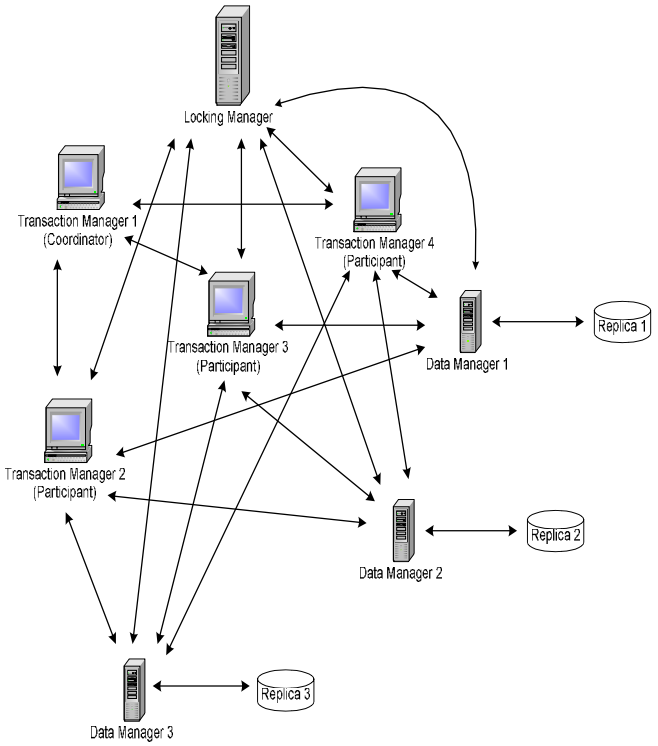


Figure 1. Overall system architecture.

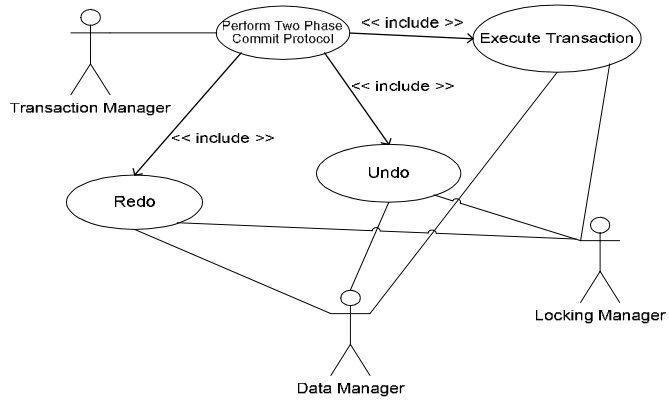


Figure 2. Use case diagram.

The software architecture of the simulator contains 3 packages as shown in Figure 3:

1. *TwoPhaseCommit*: Contains *Transaction Manager*, *Coordinator*, *Participant* and other classes that help to carry out 2PC.
2. *DataLocking*: Contains *Locking Manager*, *Data Semaphore* and other classes that are related to synchronization of access to shared data.
3. *Data Storage*: Contains *Data Manager* class that manages data transfer and *Data* class which is an abstraction of data.

Figure 4 shows the class diagram of *TwoPhaseCommit* package. *TransactionManager* is the main class in the *TwoPhaseCommit* package. Its main function is to initiate *Coordinator* and *Participant* and to keep a list of them when a transaction occurs. It contains an instance of *CoordinatorLog*, which is referred by all instances of *Coordinator* belonging to the *TransactionManager*, an instance of *ParticipantLog* and an instance of *TransactionDataLog*, which is referred by all instances of *Participant* belonging to the *TransactionManager*. *Transaction Manager* also receives messages from *Coordinator* and *Participant* of other *TransactionManager* and forwards the messages to the corresponding *Participant* or *Coordinator*.

The *Coordinator* thread coordinates a *Transaction*. It carries out the proper procedures of the 2PC as coordinator. It initiates *Participants* on other *TransactionManager* by sending them their corresponding *SubTransaction.* It logs the status of the transaction to a *CoordinatorLog*. It also contains recovery procedures to deal with failures.

The *Participant* thread will execute, redo or undo of a *SubTransaction.* It carries out the proper procedures of the 2PC as participant by following instructions from *Coordinator*. It logs the status of the transaction to a *ParticipantLog*. It also contains recovery procedures to deal with failures.

The *Transaction* class stores a transaction ID, which uniquely identifies a *Transaction* and an array of *SubTransaction*. The *SubTransaction* class mainly keeps the data that is needed to be read or updated, and new values for data that are needed to be updated during the execution of *SubTransaction*. This class also contains the address of the coordinator and other participants that are involved in the transaction.

The *TwoPCLog* is a parent class that contains common functions and variables of *CoordinatorLog* and *ParticipantLog*. It implements a remote interface, *LogQueryListener*.

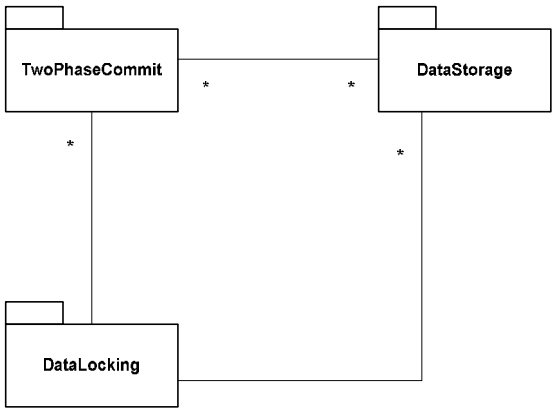


Figure 3. Package diagram.

The *CoordinatorLog* class reads from or writes to stable storage the status of all *Coordinators* of a *TransactionManager*. This class also replies to queries about the log.

The *ParticipantLog* class reads from or writes to stable storage the status of all P*articipants* of a *TransactionManager*. This class also replies to queries about the log.

The *LogQueryListener* is an interface, which extends from *java.rmi.Remote*. It contains a remote

function for querying a *TwoPCLog*. This interface is implemented by *CoordinatorLog* and *ParticipantLog*. The *TransactionDataLog* class logs to stable storage all the necessary information to redo or undo a sub transaction.

The *MessageDispatcher* interface extends from *java.rmi.Remote*. It contains remote functions, which are implemented by *TransactionManager*. The *ParticipantMessage* class is an abstraction of a message sent by a *Participant*. It contains the address of the sender, a transaction ID and the message status. The *CoordinatorMessage* class is an abstraction of a message sent by a *Coordinator*. It contains a transaction ID and the message status. The *MessageConstant* structure stores a list of constants that is used as the message status.

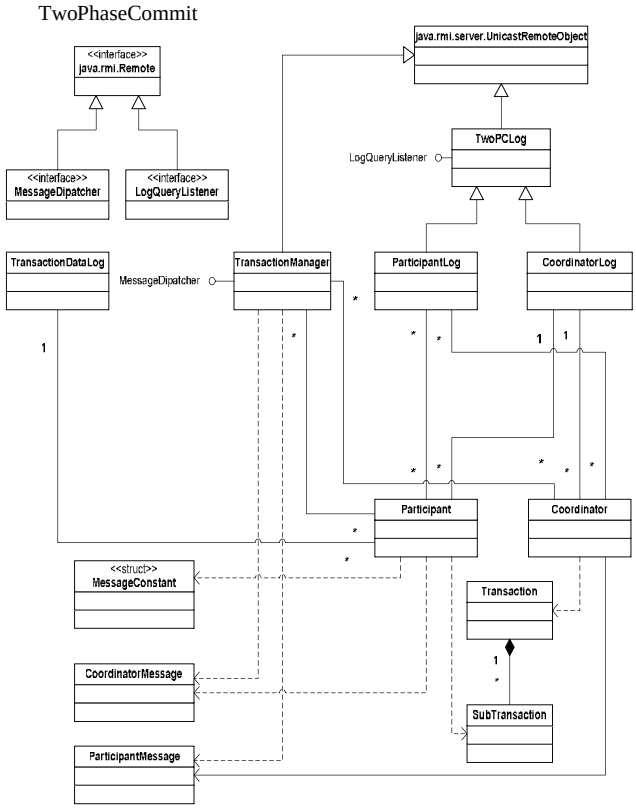


Figure 4. Class diagram of TwoPhaseCommit package.

As shown in Figure 5, *LockingManager* is the main class of *DataLocking* package.
It contains a list of *DataSemaphore*. Its main function is to listen for *LockRequest* and *UnlockRequest* from *DataManager*, and respond correspondingly. It implements a Java RMI remote interface (*LockHandler*) which listens to all remote RMI calls made by *DataManager*. Requests to add or remove *DataManager* (and its replicas) from *dataList* (data member of *LockingManager* class) are also handled here.

The *DataSemaphore* class contains a data ID that uniquely identify data, and a list of replicas of the data. It also contains functions and variables necessary to synchronize access to the data. Lock and unlock of data is handled separately for each read and write operation.

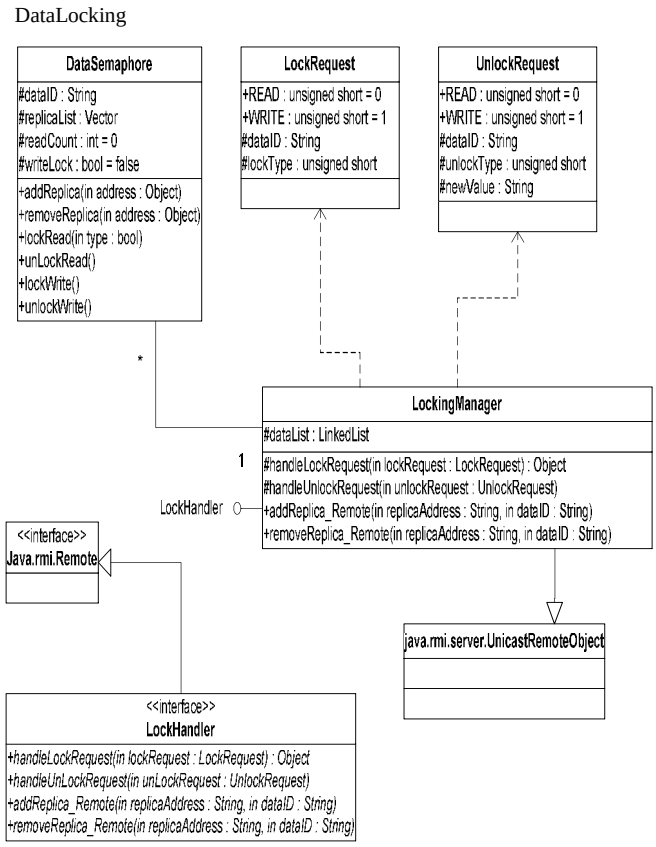


Figure 5. Class diagram of DataLocking package.

The *LockRequest* class contains a data ID and a lock type (READ or WRITE). It is sent by *DataManager* to *LockingManager* to obtain a lock. The *UnlockRequest* class contains a data ID and an unlock type (READ or WRITE). It is sent by *DataManager* to *LockingManager* to release a lock. If a write lock is released, it contains a new data value. The *LockHandler* is a *Java.rmi.Remote* interface that handles lock and unlock requests as well as add or remove data replica list requests from *DataManager*. It is implemented in *LockingManager*. As shown in Figure 6, *DataManager* is a *Java.rmi.Remote* interface to read and write operations. It is implemented in *DataManagerImpl*.

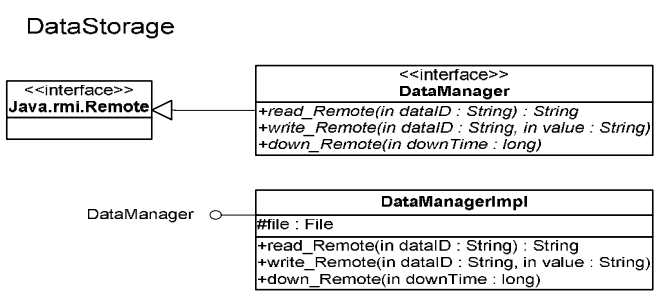


Figure 6. Class diagram of DataStorage package.

The *DataManagerImpl* class implements the *DataManager* interface. Read and write operations on data are handled here. Data are retrieved from an

external *file* and stored back upon successful update of that data.

It is to be noted that the application we developed is a real implementation of 2PC, as the communications between distributed nodes is not simulated, but actually implemented using Java RMI. What is simulated is the transactions and their sub-transactions and the data they share, replicate and update.

4. Coding and Implementation

4.1. Remote Method Invocation

Remote Method Invocation (RMI) is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine (i. e., method invocation in distributed environment).

Any object that can be invoked in this way must implement the *java.rmi.Remote* interface and extends *java.rmi.server.UnicastRemoteObject.* Remote object can be bound or registered to RMI registry by using *java.rmi.Naming.rebind(String name, Remote object)* or *java.rmi.Naming.bind (String name, Remote object)*, where the name is in URL form: "*rmi://host/name*" and object is the object to be bound. RMI registry must be available on the host. For example, (referring to the class diagrams):

```
Naming.rebind ("rmi://" + rmiHost + "/" + name +
"/CoordinatorLog", coordinatorLog);
```

To obtain references for a remote object *java.rmi.Naming.lookup(String name)* is used.
For example, (referring to the class diagrams):

```
LogQueryListener logQueryListener =
(LogQueryListener) Naming.lookup("rmi://" +
rmiHost + "/" + coordinatorAddress +
"/CoordinatorLog");
```

Below is a list of remote objects in the 2PC simulator:

- *CoordinatorLog, ParticipantLog.*
- *TransactionManager, DataManagerImpl.*
- *LockingManager, TimerImpl. SimulationEventHandler, NewTransactionHandler.*
- *SimulatorServerImpl.*

Simplicity and availability of RMI package in Java SDK 1.4 Standard Edition are the main drivers behind the use of RMI in the 2PC simulator.

4.2. Extensible Markup Language

Extensible Markup Language (XML) is a simple and flexible language to keep data as text string in file. In our project, *TransactionDataLog, CoordinatorLog, ParticipantLog* and *DataManagerImpl* store these data as XML documents. Classes for processing XML documents, which are available in Java SDK 1.4 Standard Edition provided us with an easy means to create and edit XML documents and to transform XML documents between file stream and Document Object Model (DOM), which is a structural form of XML in system memory [4]. In additon, data in XML document can be displayed in any XML browser by applying an Extensible Stylesheet Language - Transformations (XSLT) stylesheet to the XML document. Packages used in this project for processing XML documents are *javax.xml.parsers* and *javax.xml.transform*.

4.3. Timing

To control the speed of the simulation, a timer is needed to be referred by *Coordinator* and *Participant* threads as the central clock. Delays and timeouts of the threads are based on the central clock. The timer, called *TimerImpl* is a remote object that implements a remote interface *Timer*, that contains one remote function, *getTime()* which returns the time in milliseconds of the central clock. The timer is started when the simulation is started and the speed is adjustable through the user interface.

4.4. Unique ID Generator

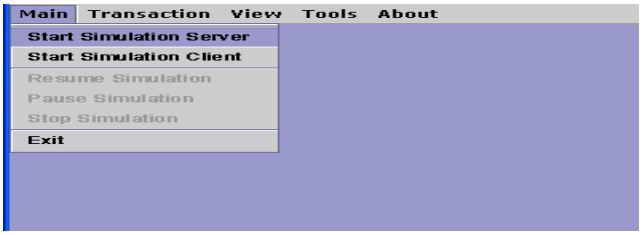
Every transaction must have a unique ID for the simulator to run properly. A class called *UniqueIDGenerator* is designed to generate a unique ID, which is the concatenation of date and time of the moment the ID is generated and a string of 4 characters, which are randomly generated separately. For example: 07/03/04 01:12:10 AGKq.

4.5. Implementation Process and Results

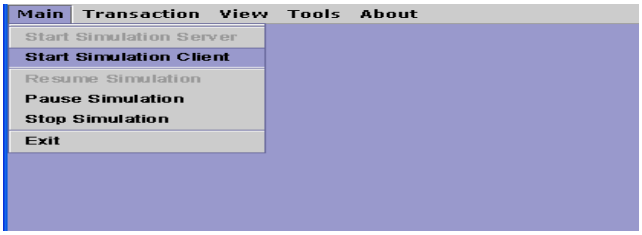
A Graphical User Interface (GUI) has been developed to clearly show the transaction of 2PC, failure injection, and failure recovery. The GUI consists of a main page that includes a menu for setting and controlling the simulation processes, and a panel that shows the status of *Data Manager* and *Transaction Manager*. It also contains a list of generated transactions and allows end-users to inject failures.

The simulation starts by setting up the server as shown in Figure 7-a. When the simulation server is initiated, it is ready to receive connection from any remote or local simulation client that can be configured and started as in Figure 7-b. Once the simulation server and client are set, transactions can be configured to be started as shown in Figure 7-c. An *Initial Transaction* has to be selected. A simulation will be created for each of the transactions specified in the *Initial Transaction.* Depending on whether random transaction is *Disabled* or *Enabled*, transactions will be randomly generated according to the *minimum time between each occurrence of a new transaction (in millisecond)* and the *probability of spawning a new*

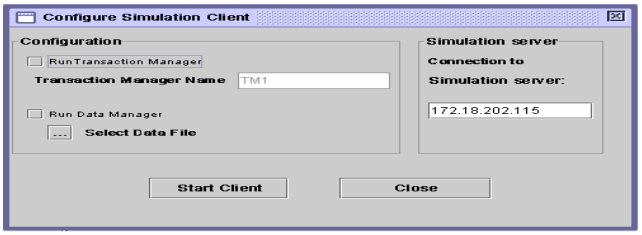
transaction (in percentage) specified as shown in Figure 7-d. Otherwise, transactions have to be created manually. Once the *OK* button is clicked, a number of pop-up transaction window(s) will appear depending on the number of *Initial Transactions* specified, as shown in Figure 7-e. Each transaction has a unique ID.



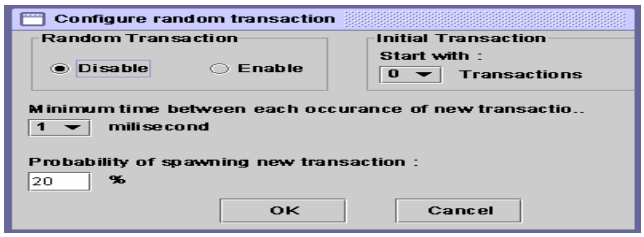
a) Starting the simulation server.



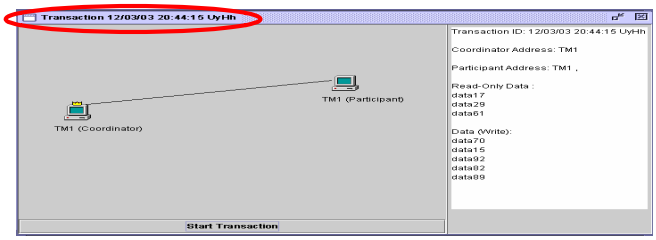
b) Starting the simulation client.



c) Configuring the simulation client.



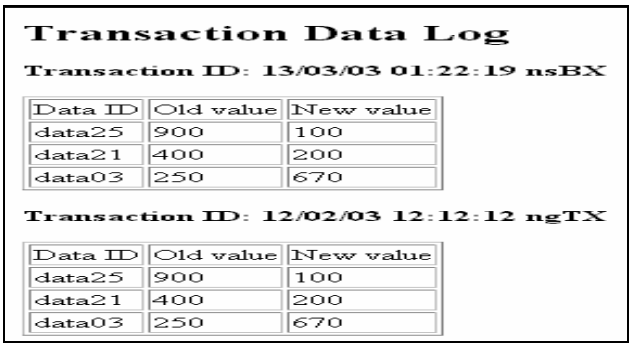
d) Configuring random transactions.



e) A running simulation.

Figure 7. The 2PC simulator.

Coordinator Log, *Participant Log*, *and Transaction Data Log* files can be displayed in any Internet browser using XSLT as the example shown in Figure 8.



Transaction Data Log

Transaction ID: 13/03/03 01:22:19 nsBX

Data ID	Old value	New value
data25	900	100
data21	400	200
data03	250	670

Transaction ID: 12/02/03 12:12:12 ngTX

Data ID	Old value	New value
data25	900	100
data21	400	200
data03	250	670

Figure 8. Transaction data log file.

For a better view of the simulation process, a user can control the overall simulation speed of the front-end GUI without affecting the 2PC algorithm actual speed in the back-end of the simulator as shown in Figure 9-a.

Transaction Manager and *Data Manager*'s down (failure) time can be generated randomly or manually. The user can set the down time for both Transaction Manager and Data Manager, usually in the range of seconds as shown in Figure 9-b.

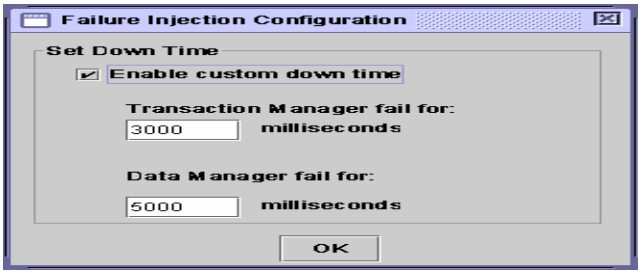
Moreover, the user can inject failures on transaction manager or data manager by clicking on the button next to each *Transaction Manager* or *Data Manager* respectively. The circled red icon shows a failed transaction manager as shown in Figure 9-c. If the user set the failure time while configuring the failure injection, the transaction manager will recover after an amount of time as specified in Figure 9-b. Otherwise, the transaction manager will be recovered after a random amount of time determined by the simulator.

The 2PC statistics can be viewed in the window shown in Figure 9-d. The statistics window gives details on the number of participants and availability of data replicas, as well as different portions of read and write operations on a set of data. The statistics show the Transaction ID, number of participants, number of *Data Managers*, total number of data access, number of read-only data, number of write-only data and elapsed time. The elapsed time shows the time taken for the specified transaction to complete.

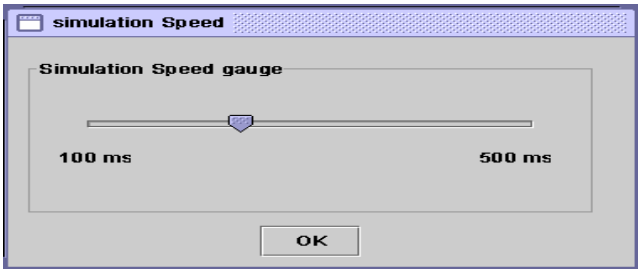
5. Conclusion

This work was mainly centered on the simulation of the 2PC for ensuring atomicity in distributed transactions. RMI was used in our message-passing and communication model, instead of using socket to handle communication. Some other considerations related to this protocol are also taken into account and improved upon in order to construct an optimized simulation. In our implementation, a generic system is simulated in a distributed environment to represent the real world scenario more vividly. This topic deserves research due to the fact that distributed transaction processing systems are widely used in many different organizations of varying size, as well as the nature of

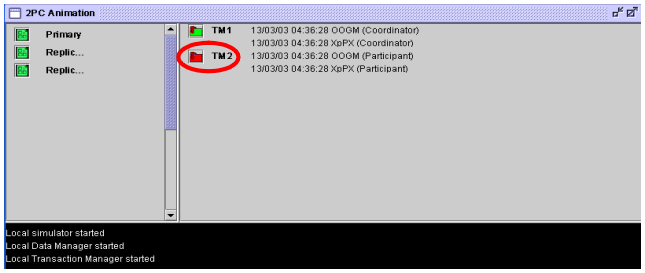
task distribution in a networking environment. In the coming future, this generic 2PC package can be further improved and enhanced [1] to be as an excellent solution for any distributed transaction systems.



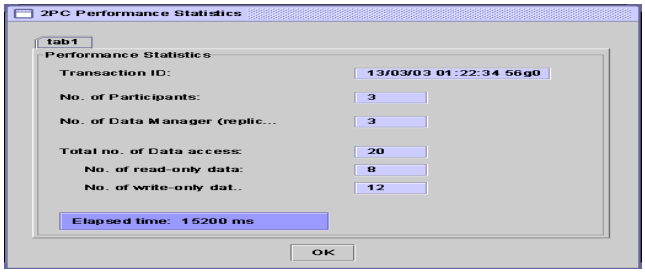
a) Failure injection.



b) Setting the simulation speed.



c) Failure recovery.



d) Simulation statistics.

Figure 9. Different options of the simulator.

References

[1] Attaluri G. K. and Salem K., "The Presumed-either Two-phase Commit Protocol," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1190-1196, 2002.

[2] Booch G., Jacobson I., and Rumbaugh J., *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

[3] Boutros B. S. and Desai B. C., "A Two-Phase Commit Protocol and its Performance," *in Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, pp. 100-105, 1996.

[4] Goldfarb C. and Prescod P., *XML Handbook*, 5th Edition, Prentice Hall, 2003.

[5] Liu M. L., Agrawal D., and El Abbadi A., "The Performance of Two-Phase Commit Protocols in the Presence of Site Failures," *in Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pp. 234-243, 1994.

[6] Oberg R., *Mastering RMI: Developing Enterprise Applications in Java and EJB*, John Wiley & Sons, 2001.

[7] Pitt E. and McNiff K., *Java.rmi: The Remote Method Invocation Guide*, Addison-Wesley, 2001.

[8] Silberschatz A., Galvin P. V., and Gagne G., *Operating Systems Concepts*, 6th Edition, John Wiley and Sons, 2001.

Toufik Taibi is an associate professor at the College of Information Technology of The United Arab Emirates University, UAE. He holds a BSc, MSc, and PhD in computer science. He has more than 8 years of teaching and research experience in Malaysia. His research interests include formal specification of design patterns, distributed object computing, object-oriented methods, and component-based software development.

Abdelouahab Abid is a lecturer at the Faculty of Information Technology of Multimedia University, Malaysia. He holds BSc in mathematics, MSc in IT, and is currently pursuing a PhD in IT at Multimedia University. He has more than 7 years of teaching, research and industry experience including a consultancy work for Telecom Malaysia. His research interests include designing networking protocols and e-commerce applications.

Wei Jiann Lim was a BSc student in information technology majoring in software engineering at the Faculty of Information Technology of Multimedia University, Malaysia. Currently, he is working as software developers in different Malaysian companies.

Yeong Fei Chiam was a BSc student in information technology majoring in software engineering at the Faculty of Information Technology of Multimedia University, Malaysia. Currently, he is working as software developers in different Malaysian companies.

Chong Ting Ng was a BSc student in information technology majoring in software engineering at the Faculty of Information Technology of Multimedia University, Malaysia. Currently, he is working as software developers in different Malaysian companies.