BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M8

''Two Phase Commit''

by

Chih-Yung Huang

Two Phase Commit

Chih-Yung Huang

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

April 1992

Date 4/27/92

Stanley B. Zdonik

Stanley B. Zdonik
Advisor

Two Phase Commit

Chih-Yung Huang

April, 1992

1 Introduction

In a distributed database system consists of a collection of sites, each of which maintains its local database and may participate in the execution of transactions that access data at one site, or several sites. In order to ensure atomicity which keeps the whole database consistent, all the sites in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all sites or abort at all sites.

Lets elect one site of all sites which get involved with the execution of T as coordinator and the other sites as participants. An atomic commitment protocol(ACP) is an algorithm for the coordinator and participants such that either the coordinator and all participants commit T or they all abort it. The two-phase commit protocol(2PC) is one of the simplest and most widely used ACP.

2PC is not a new algorithm at all; actually, it has been part of some commercial products in the world. Why bother doing this project? By abstracting this piece of code away from the correctness algorithm, we come up with a few advantages which make this package unique. First, the code will be easier to maintain because the complicated issues of distribution are isolated from the sophisticated topics of correctness. Moreover, when the underlying hardware of communication platform changes, the effects will be localized. And the most obvious advantage is that such a facility allows independent experimentation with synchronization and correctness algorithms. Also, C++ plays a big role in the contribution since this project not only makes any module easily specialized by using the object oriented language, but it also explores some ways to integrate the new C++ programs with the existed library functions written in C.

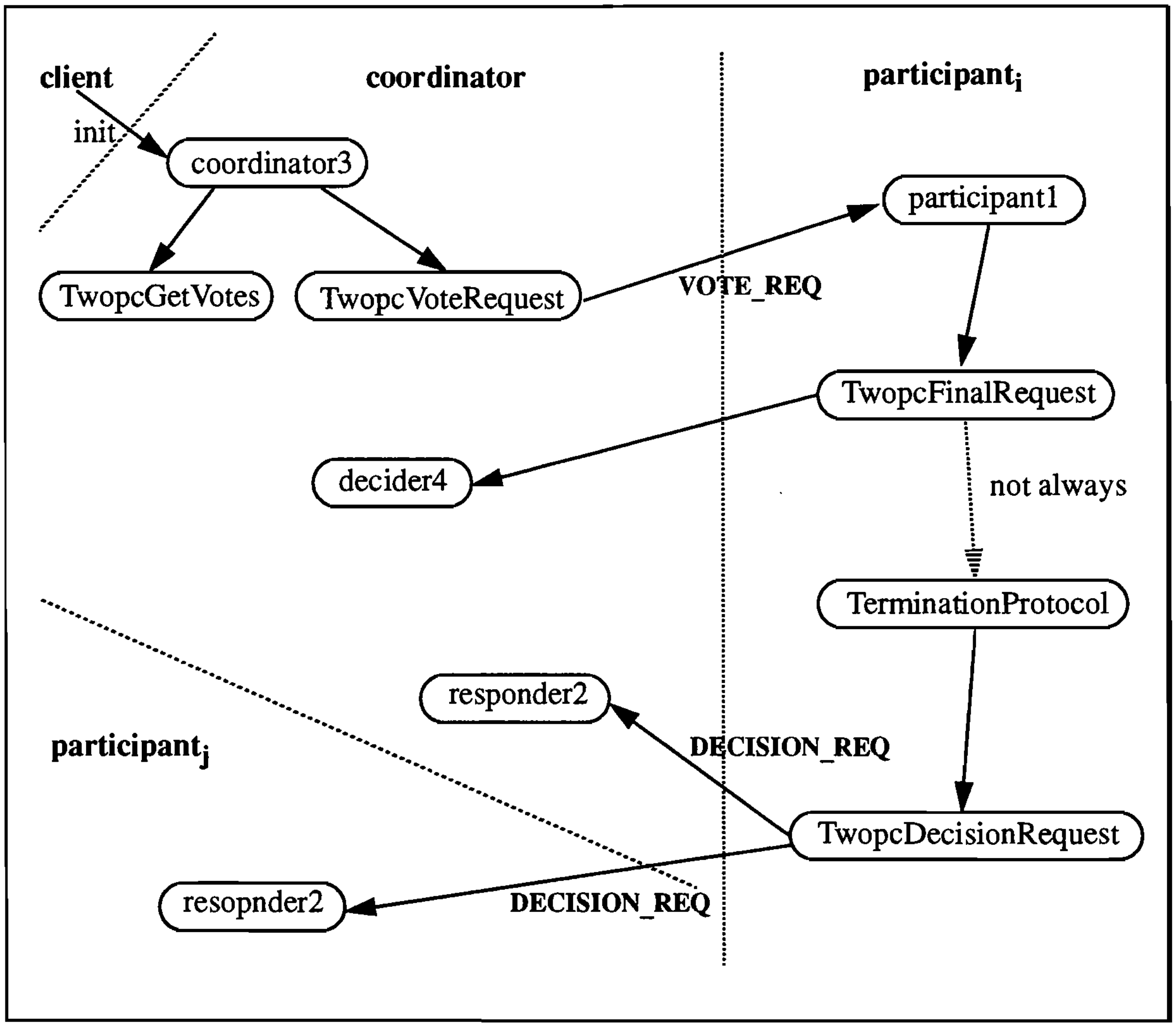
In this paper, I first outline several key modules of the system and the interaction between each other. Next, I discuss the 2PC algorithm and the corresponding components in my implementation. Then, some features are shown together with the ways of using them. The implementation issues are also included. Finally, there is some future work which is nice to be done.

2 Modules

The system consists of two parts, **tserver** and **tclient**. The later simply takes simulation data from test files and starts the commit protocol by sending the initial message to coordintor3 of the later. That is, tclient is like the front end of tserver, so I upgrade it into a graphical user interface(GUI), which is covered in section 4.

The tserver is composed of a few key modules; namely, participant1, responder2, coordinator3, decider4, TwopcVoteRequest, TwopcGetVotes, TwopcFinalRequest, TwopcDecisionRequest and TerminationProtocol. More details are presented in Fig.2.1. Those arrows cross dotted lines indicate where RPC calls take place.

Fig. 2.1 Module Specification



3 Algorithm

In this section, I put the standard 2PC algorithm in the left hand side and the relative component of my program in the right hand side. The purpose is to remind the reader of the algorithm as well as to go into some depth of the implementation details. The algorithm is composed of four parts; namely, coordinator's, participant's, initiator's and responder's. Figure 3.1 .. 3.4 present all the algorithms, respectively. By reading the last and this section, the reader should have a clear idea about how each part coordinates with one another.

The names in the implementation part are the exact function names in the program, while the two arrows indicate message flow between different modules in the server. Specifically, "A -> B" means A sends a message to B, and "A <- B" means A request a message from B.

Fig. 3.1 Coordinator's Algorithm

Algorithm	Implementation
send **VOTE_REQ** to all participants	TwopcVoteRequest coordinator3 -> participant1
write **START_2PC** in log	
wait for vote(**YES** or **NO**) from all participants	TwopcGetVotes
on timeout begin	
let S_Y be all sites from which **YES** came	
write **ABORT** in log	
send **ABORT** to all sites in S_Y	TwopcFinalRequest participant1 <- decider4
return	
end	
if all votes were **YES** and I vote **YES** begin	TwopcGetVotes
write **COMMIT** in log	
send **COMMIT** to all participants	TwopcFinalRequest participant1 <- decider4
else begin	
let S_Y be all sites from which **YES** came	
write **ABORT** in log	
send **ABORT** to all sites in S_Y	TwopcFinalRequest participant1 <- decider4
end	
return	

Fig. 3.2 Participant's Algorithm

Algorithm	Implementation
wait for **VOTE_REQ** from coordinator	participant1
on timeout begin	
write **ABORT** in log	
return	
end	
if I vote **YES** begin	
write **YES** in log	
send **YES** to coordinator	coordinator3 <- participant1
wait for decision(**COMMIT** or **ABORT**)	TwopcFinalRequest
from coordinator	participant1 <- decider4
on timeout initiate initiator's algorithm	call TerminationProtocol
write decision in log	
end	
else begin	
write **ABORT** in log	
send **NO** to coordinator	coordinator3 <- participant1
end	
return	

Fig. 3.3 Initiator's Algorithm

Algorithm	Implementation
start: send **DECISION_REQ** to all sites	TerminationProtocol
	participant1 -> responder2
wait for decision from any site	TwopcDecisonRequest
on timeout goto start // blocked	
if decision is **COMMIT** then	
write **COMMIT** in log	
else	
write **ABORT** in log	
return	

Fig. 3.4 Responder's Algorithm

```
Algorithm                                                   Implementation

wait for DECISION_REQ from any site $s_i$
if responder hasn't voted YES or has decided to ABORT
    send ABORT to $s_i$                                     participant1 <- responder2
else if responder has decided to COMMIT
    send COMMIT to $s_i$                                    participant1 <- responder2
else // responder is in its uncertainty period
    skip
return
```

4 System Overview

In order to show the user what this package does and how it works, I offer the system overview by presenting the following four parts. The libraries I utilized are described first, followed by the directory hierarchy and a brief description of the contents of those directories. Then, I show the effects of the system by three sample simulations. Finally, the GUI is discussed.

4.1 Libraries

I build my program on top of three existed libraries developed in C. They are SUN RPC, Brown Threads package and Brown Augmented Utilities for Motif(BAUM).

4.1.1 Remote Procedure Call

To use RPC or go lower to the transport level interface programming(TLI) was the main concern during the first phase of this project since 2PC is a communication-prone algorithm. Certainly, there is some way to implement it by TLI because it supercedes the socket-based interprocess communication mechanisms as the standard means of gaining direct access to transport services. Also we can get good performance by using the lower level TLI as long as enough time is spent for experiment.

In the other hand, RPC keeps us away from all those tedious details in transport services as well as provides a lot of nice routines such as xdr_*. It means much less work by taking RPC. After some study and investigation, we were positive to the RPC approach although nobody could expect the performance yet.

Basically, there are two components, client and server, in RPC model. The server is usually a background process which just sits there and waits for request from the client. The client makes a procedure call which sends requests to the server as necessary. When these requests arrive, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

In my model, things are not so simple. First, The tserver is not a standard server which only takes requests. It is a client as well, i.e., it also makes requests to other tserver's in the network. Besides, there are multiple tserver's in the whole system although there is only one tclient. Therefore, a lot of requests may arrive a tserver from other tserver's. We cannot ignore any request since it breaks the algorithm. Neither can we force clients to wait because this approach deteriorates performance. The decision we made is to create as many threads as possible for every single tserver, and use share data together with monitor to achieve the correctness. The number of threads within a tserver is not fixed, each thread is started as needed. In the other words, various threads are started based on the coming requests.

To make all these happen, I provide TwopcMySvcRun instead of using the svc_run routine provided by SUN. Nevertheless, they are very similar except the former allows multiple threads to coexist in a tserver without conflict. The other change I made to threads package is the header file, thread.h, due to the conflict between the old style definition and the new C++ program. The new header file resides in my local source directory.

4.1.2 Brown Threads Package

This package is a system for the efficient support of concurrency. The idea is that a number of concurrent threads are executing in a single shared address space and share a common view of which files are open. Thus threads may communicate very efficiently through this shared memory and all threads may participant in I/O on any file.

In the implementation, I create a thread and make it independent on its parent whenever the service required might take a while to finish so that I reduce the chance which makes the client wait to minimum. One of the typical example is the execution of TerminationProtocol, which may cause the server to be blocked if it cannot get decision from any other sites. By creating a thread to deal with this specific function, the server can keep receiving requests and offering services even though that TerminationProtocol thread gets blocked. Lots of other functions, which never result in blocked server but takes some time to complete are handled by independent threads.

Shared data are needed because different threads may try to reach some common data such as the log. However, serious problems would have been seen if we did not monitor those share data. Imagine that two threads try to write something into log simultaneously, then we cannot expect what the result will be, let alone the correctness. By making enough data sharable and enforce the monitor appropriately, we get the best performance. And this is where the trick is.

4.1.3 Brown Augmented Utilities for Motif

BAUM is a library of C++ classes intended to simplify interface design. This set of classes use inheritance to parallel and enhance the widgets provided by Motif. Actually, it encapsulates the Motif widget hierarchy into a C++ class hierarchy.

A C++ programmer who uses Motif must has noticed that the callback function cannot be the member function of a class that he created - it has to be a regular global C-style function. In the mean time, we always expect everything to be object-oriented(OO) while programming in an OO language like C++. How do I elegantly integrate those callbacks with my beautiful OO design? The solution is to ask the widget to pass a pointer to my class structure to the global C-style function which calls the appropriate function in the class that really handles the event. Thus, the C-style function is just a "passthrough" point so that I get a clean OO design.

How about the interface to RPC and Threads package? They are similar to Motif in the sense that they are all written in C before any OO language really became popular. The other similar characteristic is they all break OO style at some point. For instance, the function argument of THREADcreate cannot be anything else but a global C-style function. The same situation happens in the XDR routine argument of svc_getargs provided by SUN RPC. Therefore, it will be a big distribution to offer the class library for RPC and Threads so that the OO programmers can have a really clean OO design.

4.2 Directory Hierarchy

There are five directories; namely, src, bin, log, data and doc, related to this project. Currently, they can be found in "/u/cyh/work/masters/current/C".

4.2.1 src Directory

All the source codes, including C++ files and header files, reside here. Two kinds of header files exist. They are old style C header ending by ".h" and new style C++ header ending by ".H". A list of all files and a brief explanation of their functions are followed.

Makefile	
thread.h	: the header file for threads package in C++ application.
msg.h	: definition of message format passed through network by RPC.
defs.h	: language extensions.
twopc.H	: class definitions for the whole thing except GUI.
gui.H	: all definitions for gui.C including class, resources, menus, callbacks and more.
tools.C	: all member functions of the class TwopcTools, the utility for all other classes.
rpc.C	: all member functions of two classes, TwopcLog and TwopcRpc, the parent of TwopcRpcClient and TwopcRpcServer.
client.C	: all member functions of the class TwopcRpcClient, the RPC client.
clientinit.C	: the simulation front end in Text UI mode.

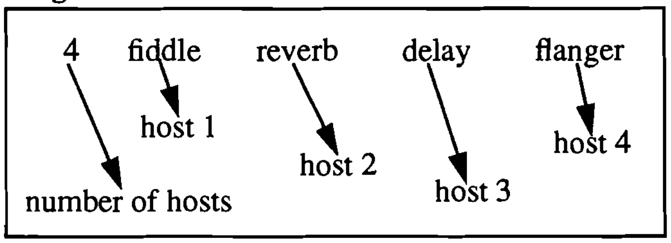
server.C	: all member functions of the class TwopcRpcServer, a RPC server as well as client.
verify.C	: the program to check the consistency of all logs.
gui.C	: the graphical user interface.

4.2.2 bin Directory

All the executable files are here together with a resource file, **hosts**, and a convenient file, **myalias**. The names of all hosts participating in the distributed database system should be in the file, **hosts**. It begins with the total number of hosts and follows by a series of host names. A sample file is shown in Fig.4.1.

There ought to be exactly one **tserver** running in every host, and the **tclient** starts in whichever host to trigger the simulation process. The user may start **gui** and follow the online instruction instead of **tclient**. In **myalias**, I define a shorthand, **cl**, for **tclient**. The last file, **verify**, can be started at any time in whichever host to check whether there is any inconsistent or undecided transactions in logs.

Fig.4.1 hosts file



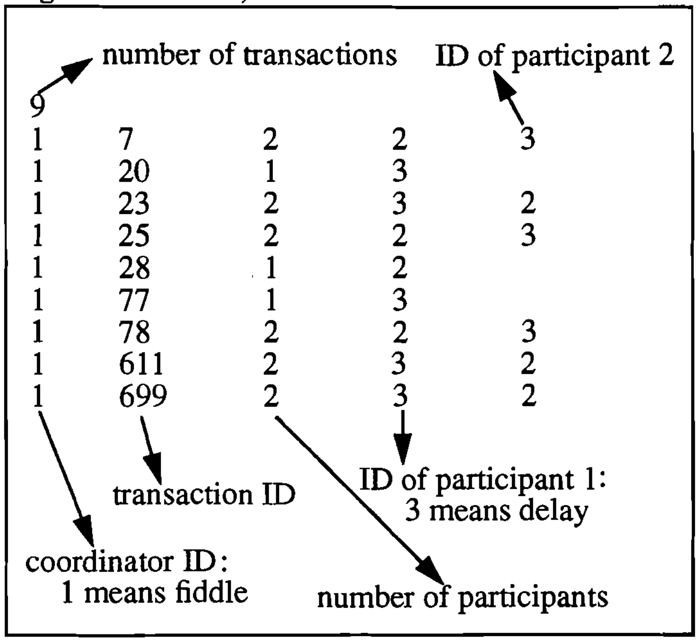
4.2.3 log Directory

All logs belonged to the hosts defined in hosts file are here. They can be easily distinguished by the corresponding host name. A log is a sequence of entries recording the history of transactions. Each entry is composed of two parts, transaction ID and status. Only five different statuses are found there: **START_2PC**, **YES**, **COMMIT**, **ABORT** and **CHECK_PT**. The last status always comes together with the transaction ID 0. Some sample logs will be shown in section 4.3.

4.2.4 data Directory

There are three test files available: test.Fiddle, test.Delay and test.Reverb. Lets take test.Fiddle as an example. A sample file, test.Fiddle, is shown in Fig.4.2. The coordinator ID and participant ID are given according to the hosts file in bin directory.

Fig.4.2 A test file, test.Fiddle



```
9
1    7     2    2    3
1    20    1    3
1    23    2    3    2
1    25    2    2    3
1    28    1    2
1    77    1    3
1    78    2    2    3
1    611   2    3    2
1    699   2    3    2
```

4.2.5 doc Directory

The document of this project is created by framemaker. twopc.doc is the filename.

4.3 Simulations

There are two ways to start a simulation according to the test files. Here I describe the fully supported method first, and the other method controlled by gui is covered in next section. Due to the effectiveness of explanation by examples, I presents three sample log results reflecting different situations. The log files can be found in the subdirectory of the log directory.

The first sample happened when all three servers, fiddle, reverb and delay, kept alive during the whole simulation process which is fired by a sequence of test files one after another. Only a single host was used to start tclient. The logs are in the subdirectory sample1.

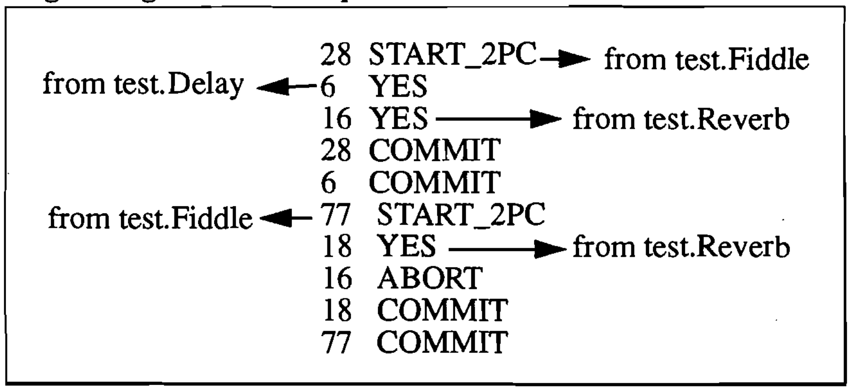
The second sample occurred without any server failure, too. However, I started three simulation from three distinct hosts almost simultaneously, so we found transactions belonging to those three test files interleave one another in logs. The name of a host which started a tclient running a test file reflects the test file name. A snap shot is in Fig.4.3 and the whole logs are in the subdirectory sample2.

Finally, I intentionally made one of the host failed and started it again to show the strength of recovery. Three tserver's were initiated as usual, then test.Fiddle and test.Reverb were started almost at the same time from the machines, fiddle and reverb, respectively. Before finishing that, I killed the server in delay. At this point, there was no decision for transaction 7 in log.delay. Now I started the tserver in delay again, and it checked the log, sent out the needed requests to the other servers, and got the decision for transaction 7 eventually. I execute another tclient in delay to fire those transactions in test.Delay anyway. The result is everything in logs is consistent and there is no undecided transaction in any log. Fig.4.4 shows a snap shot of the log states right before and after delay failed. Detailed can be found in the subdirectory sample3.

Two things are beyond these three sample simulations. First, if too many, say 30, requests come to a single tserver almost simultaneously, so that the server cannot handle all requests even though a lot of threads are created. In this circumstance, some transactions will be aborted but everything in the logs will still be consistent. I made this decision because the consistency is the key point of 2PC while good performance should be gained whenever possible. The other reason is that heavy traffic will happen some time no matter how well you distribute the load.

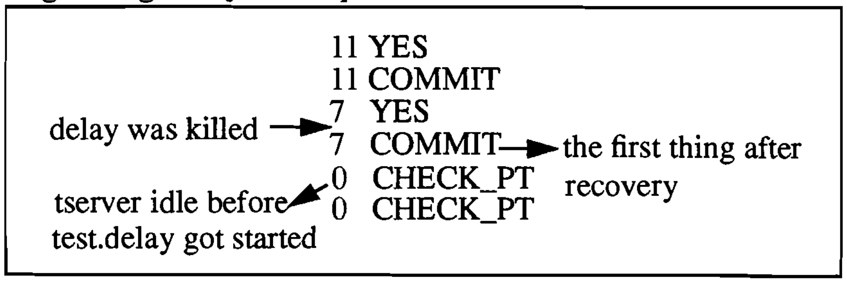
The other situation is the only weakness of 2PC, and the motivation of creating Three Phase Commit Protocol(3PC). That is, some server S in the system may fail while processing some transaction T such that there is no decision for T in its local log yet. In the mean time, there is no other server who has gotten a decision for T. After S recovers from failure, it sends out requests in order to get a decision for T; however, the thread which deals with this Termination-Protocol gets blocked since there is none decision for T in the whole system. I did notice the occurrence of this problem. Fortunately, it is scarce.

Fig.4.3 log.fiddle in sample2



```
28 START_2PC
6  YES
16 YES
28 COMMIT
6  COMMIT
77 START_2PC
18 YES
16 ABORT
18 COMMIT
77 COMMIT
```

Fig.4.4 log.Delay in sample3



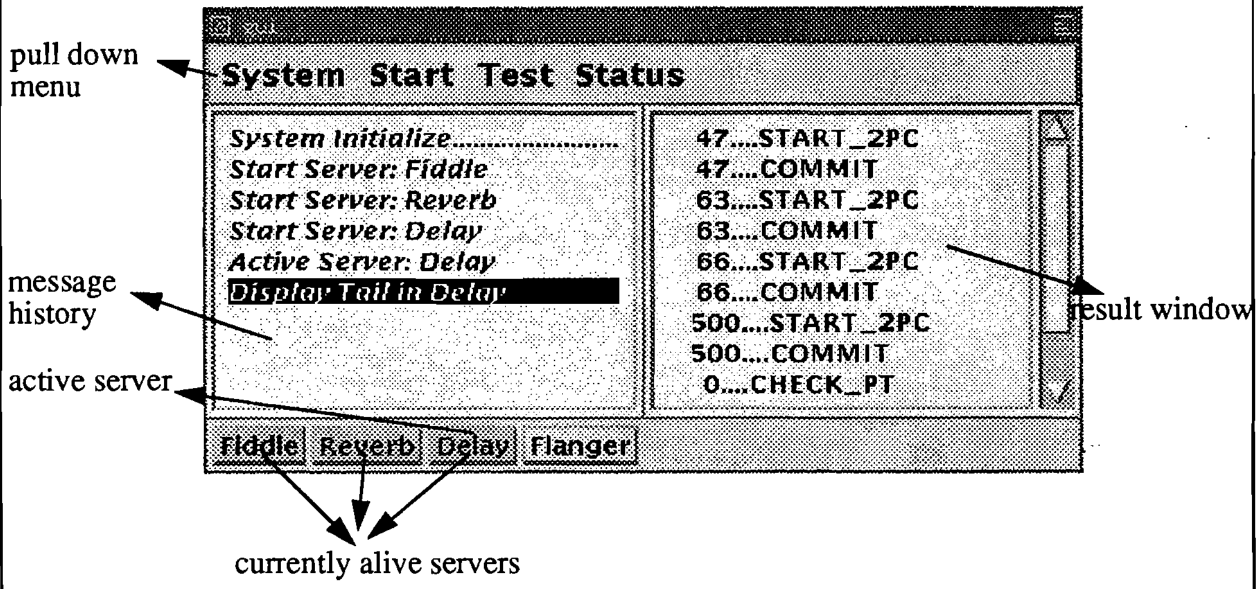
```
11 YES
11 COMMIT
7  YES
7  COMMIT
0  CHECK_PT
0  CHECK_PT
```

delay was killed
the first thing after recovery
tserver idle before test.delay got started

4.4 Graphical User Interface

The user can do whatever mentioned in the previous simulation section by this GUI except starting several test files from various machines at the same time. GUI is pretty easy understood once you try it, so I would only present two snap shots in the following figures. Fig.4.5 shows the tail portion of log.delay. The tail portion of a log is a collection of entries after the last active **CHECK_PT** which is the **CHECK_PT** immediately before the last effective **CHECK_PT** which is either the end of the log or a **CHECK_PT** after which there is nothing but **CHECK_PT**. In Fig.4.6, I show all entries which has transaction ID 7 in all logs.

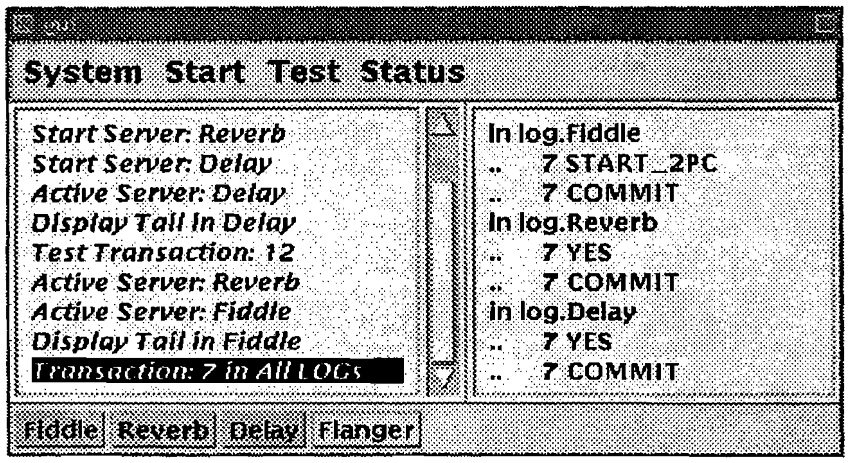
Fig. 4.5 GUI state 1



System Initialize........................
Start Server: Fiddle
Start Server: Reverb
Start Server: Delay
Active Server: Delay
Display Tail in Delay

```
47....START_2PC
47....COMMIT
63....START_2PC
63....COMMIT
66....START_2PC
66....COMMIT
500....START_2PC
500....COMMIT
 0....CHECK_PT
```

Fig. 4.6 GUI state 2



Start Server: Reverb
Start Server: Delay
Active Server: Delay
Display Tail in Delay
Test Transaction: 12
Active Server: Reverb
Active Server: Fiddle
Display Tail in Fiddle
Transaction: 7 in All LOGs

```
In log.Fiddle
..   7 START_2PC
..   7 COMMIT
In log.Reverb
..   7 YES
..   7 COMMIT
In log.Delay
..   7 YES
..   7 COMMIT
```

5 Future Work

There are a few things which will make life easier. First, detect the point where 2PC breaks and terminate the blocked thread by aborting the relative transaction and roll it back. This idea may lead to 3PC; however, the sophistication of 3PC is what we try hard to avoid. Thus, it would be wonderful if we can come up with something between those two protocols.

Second, I notice that it's so great to program in BAUM instead of Motif, and it results in a very clean C++ program for the GUI. How nice it would be if we build similar kind of C++ interface for both threads package and RPC, and write our program on top of them.

Last but not least, a fully supported GUI should be built. Two important features are needed to make it complete. One is the ability to start up several test files from different hosts simultaneously. The other is to start a tserver by executing a remote command in the specified host and keep track of the life span of all tserver's. This idea raised the issue of modelling a distributed system by a single GUI running in a single machine in the system. This kind of GUI, or system monitor, will benefit two groups of people a lot. The programmers who are developing distributed software can use it to debug their programs. And the system administrators of LAN may figure out the system performance and do some tuning.

6 Acknowledgments

Thanks to Dave Langworthy for helping me start up, to professor Tom Doeppner for all the prompt answers to my questions, and to all the people around me who offered technical opinions and experiences. Thanks to my family for all the support and to my girl friend, wen-chih, who makes my life rich.