A Fault-Tolerant Collaborative Tools Development System

by

Miranda W.S. Ko

B.Sc. (Hons), The University of British Columbia, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 1998

Abstract

Collaborative tools (multi-user applications) are very popular. It is desirable to have an application-independent system which provides the basic elements that are necessary for developing any kind of collaborative tools. This thesis presents a system named Collaborative Tools Development System (**CTDS**) for developing these tools. It not only facilitates the building of collaborative software from stand-alone (single-user) software, but it also eases the development of collaborative software from scratch. **CTDS** provides the communication and coordination services which are used by collaborative software. **CTDS** offers fault-tolerant collaboration. Failures of key components in **CTDS** do not have any impact on collaboration. **CTDS** also offers many other features for more effective and efficient collaboration.

Contents

List of Tables

List of Figures

Acknowledgements

Many thanks to Dr. Peter Cahoon and Dr. Norm Hutchinson for supervising this thesis. Their suggestions and comments led to many improvements in this project. Thanks to Dr. Dave Forsey for his advice on different aspects of this project. I would also like to thank Roger Tam for his help and his input to this project. Lastly, thanks to Jean-Luc Duprat for his valuable input on the design of the system in this project. This research was supported by the Natural Sciences and Engineering Research Council.

MIRANDA W.S. KO

The University of British Columbia

August 1998

This thesis is dedicated to Charles, my daddy, and Lisa, my mommy.

Chapter 1

Introduction

In the past few years, white board software has become popular. It allows widely separate users to collaborate on documents, to participate in presentations, etc. However, the sharing is not real-time; sharing is limited to static images or screen snapshots from programs' output. This kind of sharing is unacceptable in some situations. Consider the sharing of a morphing process from an ape to a human being. Apparently, one can understand better how an ape morphs to a human being by looking at the entire animation as compared to looking at snapshots of the animation. This is due to information lost in-between the snapshots. This inadequacy of white board software leads to the development of "application sharing" software. Application sharing software allows two or more users running the same application to work collaboratively and simultaneously on the shared model/data. The meaning of the shared model/data is application-dependent. For instance, the shared model/data is a document if the shared application is a document editor. On the other hand, the shared model/data is a 3-D object in a 3-D application sharing environment. Application sharing software offers real-time sharing. Application sharing software, which allows multi-user collaboration, is also called *collaborative tools*. In most cases, *collaborative tools* are developed from stand-alone (single-user) applications. Users of *collaborative tools* are called *participants*. *Participants* are said to be participating in a *session*. *Collaborative tools* are useful in many different fields. In the field of medicine, it opens up the possibility to perform a surgery without the presence of the necessary specialists in the operating room. By sharing the patient's model in an appropriate application, specialists can give advice to the surgeons. Specialists can also give instructions to the surgeons by manipulating the patient's model. *Collaborative tools* also make interactive distance learning possible in the field of education. With a shared model between the instructor and the students, together with video conferencing software, a virtual classroom is formed. The instructor can explain the lecture material clearly to the students by referring to the relevant model and through verbal and visual communication. Students can also clarify their questions by manipulating the corresponding model. One last example of *collaborative tools* is Internet games. Internet games allow users from different parts of the world to play a single game session on the Internet.

Evidently, collaborative applications need to communicate with other instances of themselves. For example, changes made to the shared model by any of the participants must be seen by other participants. In addition, a coordination mechanism must exist to ensure state consistency among all participants. State consistency among participants is guaranteed only if all participants execute the same set of changes to the shared model in exactly the same order (total atomic ordering). In the field of distributed systems, many techniques have been developed to handle problems encountered in designing collaborative tools. For instance, there are techniques to ensure replica consistency and techniques to preserve ordering of requests from different clients. Most of the collaborative tools that have been developed build the code for communication and ordering into the applications themselves. In fact, every kind of collaborative tool requires similar communication and ordering mechanisms. Thus, it is desirable to build an application-independent system which provides the communication and coordination services. This not only allows reuse of code, but also facilitates conversion of a stand-alone application into a collaborative one. The collaborative version of an application can be built by reusing existing application code with any modifications required by the system. Such a system for developing collaborative tools can be classified into two main categories:

1. **view-level sharing systems**

2. **model-level sharing systems**

View-level sharing systems require no modification to the application code in converting a stand-alone application to a collaborative one. It offers collaboration by exchanges of window events, e.g. XEvents. XEvents that are generated as a result of changes to shared models are sent to all participants to achieve application sharing. The application is said to be collaboration-unaware since the same piece of code is used for running the application in stand-alone mode and collaborative mode. On the contrary, **model-level sharing systems** require application code to be modified in building a collaborative version of a stand-alone application. However, **model-level sharing systems** provide sharing at a higher level of semantics. In **model-level sharing systems**, sharing is achieved by exchanging information of the shared model, e.g. rotations to the shared model. A good model-level sharing system should require as little change to the application code as possible. Collaboration-aware software is produced by **model-level sharing systems**.

It is also important that collaborative tools provide fault-tolerant services. There are two aspects of fault-tolerance in application sharing. The first one is the isolation of failures of participants. Failure of any of the application instances must not affect the others from functioning correctly. The other aspect of fault-tolerance is the recoverability of a session. The meaning of the recoverability of a session depends on the design of the system that provides the communication and coordination services. In general, a system that provides the communication and coordination services consists of a certain number of components or processes. A session is recoverable if failure of any component in the system does not affect the continuity of the session. The implementation of fault-tolerant services differs from system to system, depending on the design of the system. If a system fulfills the fault-tolerance requirements in the two aspects discussed, there should be no failure at all from the point of view of a participant. This is because the application instance that a participant is running dies only if the participant quits or terminates his/her own application instance.

To summarize, a system for developing collaborative tools should provide the communication and coordination services, which are necessary in application sharing. Moreover, it should provide fault-tolerant application sharing. Failures of any of the participants or any of the components in the system should have no effect on the continuity of a session. Lastly, a model-level sharing system should require as little code change to the original application code as possible.

This thesis presents a fault-tolerant approach in implementing a system for developing collaborative tools. The system developed offers model-level sharing. It allows any number of users sharing the same application at geographically dispersed locations. The system developed is called the **Collaborative Tools Development System** (**CTDS**). The design and implementation of this system targets the three major requirements of a system for developing collaborative tools. It employs a client-server architecture. The application instances together with session management processes provided by the system form the clients. The server is a separate process provided by the system to give the communication and coordination services. The server is also called *arbiter* because of its important role in the system. **CTDS** requires the *arbiter* running on a computer with **inetd (Internet Daemon)** running. Code of a stand-alone application is modified by adding necessary function calls to communicate with the *arbiter* and the session management process, and by adding state information which is collaboration-specific. Application sharing is then achieved by having each of the participants running the modified application on his/her machine. Fault tolerance is achieved by automatic recovery of the *arbiter*.

Chapter 2 of this thesis presents the model of **CTDS**. Each of the components in the system is described briefly. A dataflow diagram is also presented to illustrate the process to commit a transaction/request. Some important terminologies are also introduced in this chapter. This chapter also presents other models which are commonly used in building collaborative tools development systems. Comparisons and contrasts are done between **CTDS** model and these models.

Chapter 3 talks about the implementation of the system. It gives a detailed description of the implementation of session initiation and session termination. In addition, the algorithm of joining a session and that of leaving a session are also discussed. Some internal details of the core component in **CTDS** – the arbiter – are also uncovered.

Chapter 4 presents the main features of **CTDS**. It introduces each of the key features, followed by an in-depth discussion of its implementation.

Chapter 5 discusses related work. Research work which employs models similar to that used by **CTDS** as well as research work which uses the other models introduced in Chapter 2 are presented.

Chapter 6 presents the conclusions and some ideas on enhancing the system.

Chapter 2

Model

This chapter presents the model used in developing the system (**CTDS**) that we have implemented in this project. Each of the components in the system is described in detail, along with a discussion on its functionality. Moreover, interactions between the system components are also discussed. This chapter then gives an overview on how application sharing is achieved in **CTDS**. Lastly, two other models, which are used in building collaborative tools development systems in the research field are presented. Comparisons and contrasts between these models and the **CTDS** model are also given.

2.1 Terminologies

Before presenting the model used in developing **CTDS**, it is crucial to know the definitions of some terms that are used throughout this document. Some of the terms are introduced previously. We will repeat them here for the completeness of this section.

- *session*

  - the environment formed by multiple shared instances of a single application. All interactions occurring in any of the shared instances are seen by all other instances.

- *arbiter*

  - server in **CTDS** which provides the communication and coordination services to support collaboration. There is one arbiter per session.

- *participant*

  - the user of a shared application instance. The user is said to be participating in a session.

- *shared model*

  - a generic term used to describe all the objects shared by the participants in a session.

- *client*

  - the shared application instance and the session management process associated with that shared application instance run by a participant. (Since **CTDS** offers model-level sharing, the application instance is a modified one.)

  Note: *participant* and *client* are used interchangeably in this document based on the justification that an application instance is run by a single participant.

- *master*

  - a participant/client who has permission to perform interactions on the shared model in a session. Subject to the capability and limitations of a system, the number of masters in a session can be greater than one.

- *slave*

  - a participant/client who does not have permission to perform interactions on the shared model in a session. Obviously, there is no limit on the number of slaves in a session.

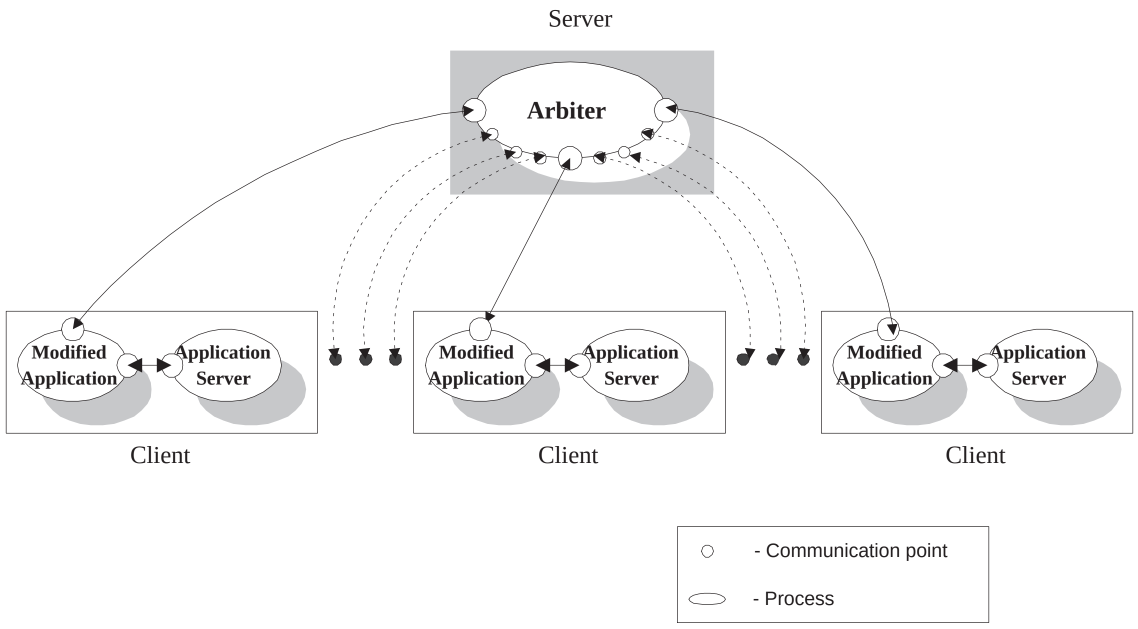


Figure 2.1: The Client-Server Architecture in **CTDS**

- *initiator*

  - the participant/client which initiates a session. In other words, the initiator is the first client in a session.

2.2 System Architecture

Figure 2.1 shows the client-server architecture used in developing **CTDS**.

2.2.1 Server

The server is called the *arbiter*. It is part of **CTDS**. It provides the communication and co-ordination services to clients in a session. The *arbiter* provides total ordering by assigning sequence numbers to transactions on the shared model. In other words, the sequencing and ordering job is done by the *arbiter*. The *arbiter* works with any application. It is started automatically when the initiator starts a session, and it runs on a initiator-specified machine.

2.2.2 Client

The client consists of two processes, the modified application program and an application server associated with it. Both processes run on the same machine, which can be different from the *arbiter*'s machine.

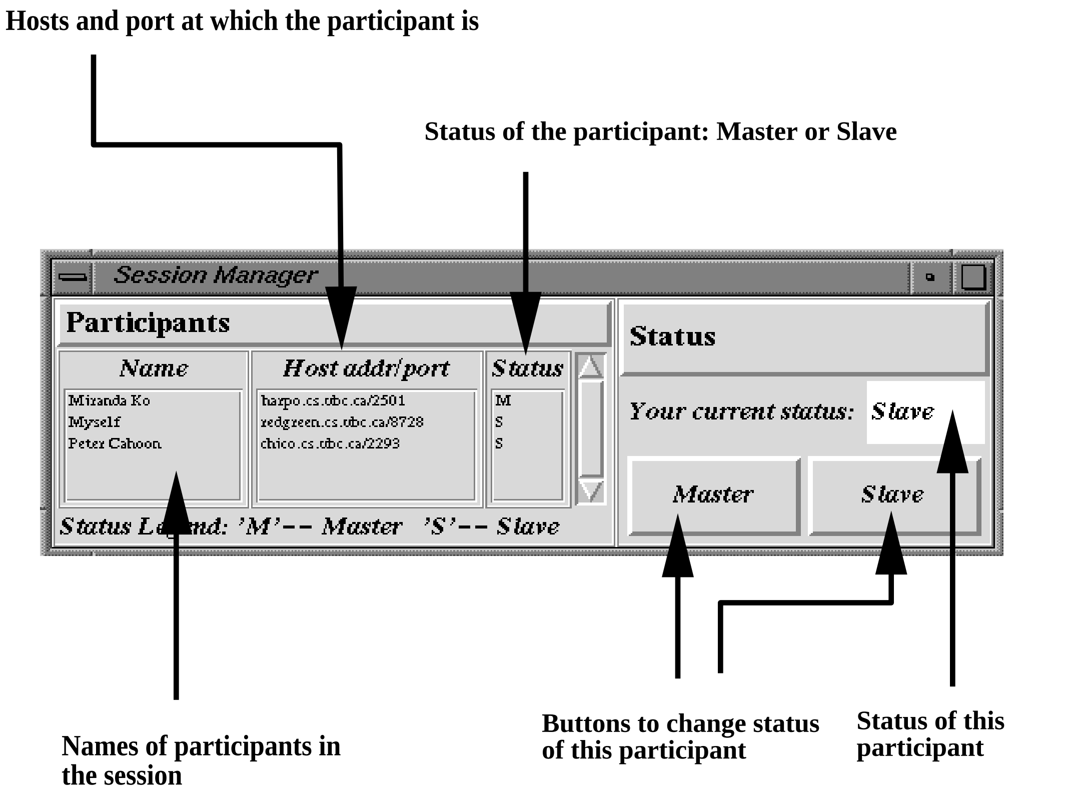
Modified Application

Since **CTDS** provides model-level sharing, the application code has to be modified to work with **CTDS**. Connection-establishment procedures have to be added to the application code for the application program to communicate with other components in **CTDS**. In addition, communication procedures have to be added to the application code at appropriate places to achieve application sharing. All the modifications are made based on a library of functions provided to any applications by **CTDS**. Refer to *Appendix A* for details on the integration of **CTDS** into an application. The user interface to the stand-alone version of the application is preserved even after integration with **CTDS**. Thus, participants will not have any difficulty in using the collaborative version of the software.

Note: In the rest of this document, the modified application will be referred to as the application.

Application Server

The application server is part of **CTDS**. It provides the additional user interface for controlling collaboration-specific parameters, and thus, preserving the interface of the original application. It also handles all interactions with participants which result from collaboration. The application server always comes with the application program. It is started automati-


Hosts and port at which the participant is
Status of the participant: Master or Slave
Session Manager
Participants
Status
Your current status: Slave
Master
Slave
Status Legend: 'M'-- Master 'S'-- Slave
Names of participants in the session
Buttons to change status of this participant
Status of this participant

Name	Host addr/port	Status
Miranda Ko	harpo.cs.ubc.ca/2501	M
Myself	redgreen.cs.ubc.ca/8728	S
Peter Cahoon	chico.cs.ubc.ca/2293	S

Figure 2.2: Application Server

cally when the modified application is run. The application server allows a participant to switch between *master* and *slave*. In addition, it shows a list consisting of the information of all participants in the corresponding session. The information of a participant includes his/her name, the host and the port number on which the participant's application is, and the status (*master/slave*) of the participant. *Figure 2.2* shows the user interface provided by the application server, along with an annotation for each piece of information.

2.3 Communication

Figure 2.3 shows the types of communication used between each process in **CTDS**. Note that communication only exists between the *arbiter* and clients. Clients do not communicate

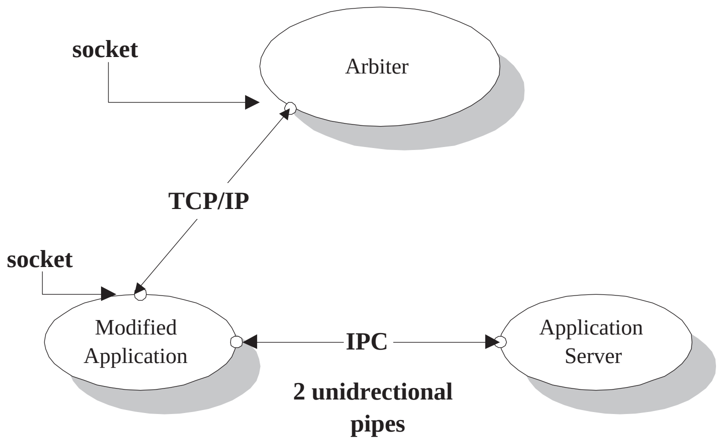


Figure 2.3: Communications Between Each Process in **CTDS**

with each other.

2.3.1 Communication Between Application Server and Application Program

Any kind of IPC (Interprocess Communication) can be used for the communication between the application server and the application program. In **CTDS**, two pipes are used for this purpose. Since pipes provide one-way flow of data, two pipes are used to provide the bi-directional communication desired. For a detailed discussion on pipes, refer to [Ste90].

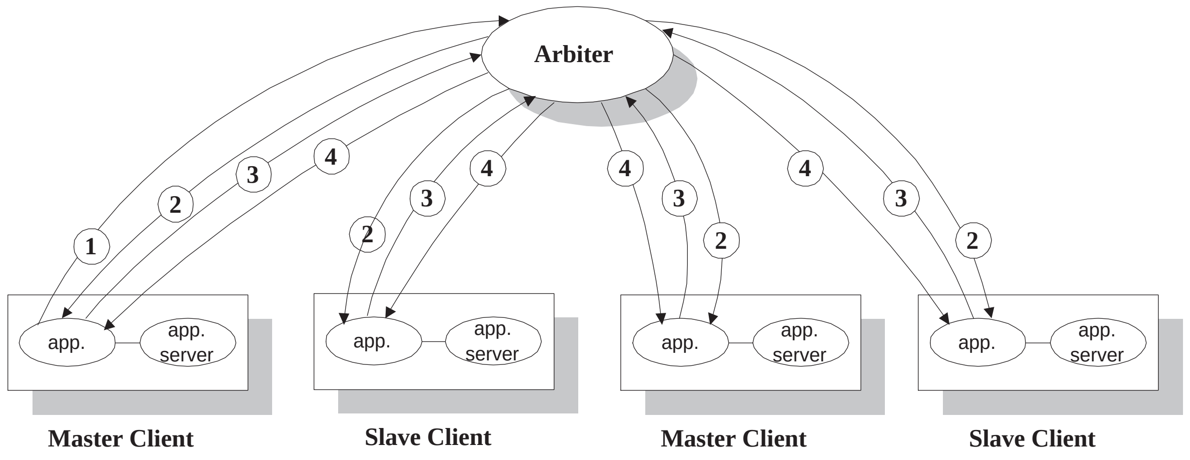
The communication between the application server and the application program is necessary upon situations in which the information displayed by the application server has to be updated or the application server has to provide collaboration-specific interactions. These situations include joining of new participants and leaving of existing participants. When a new participant joins, the *arbiter* will notify the application program, which in turn notifies the corresponding application server of the joining. Upon notification of the joining, the application server updates the list of participants. The application server also handles all interactions in the joining process (Section 3.1.2). Similarly, when an existing participant leaves, the *arbiter* will notify the application program, which in turn notifies the application server of the leaving. Again, the application server will update the participant list. Moreover, the application server also needs to communicate with the application program when a participant switches status (*master/slave*) using the buttons provided. If a participant switches status, the application server informs the application program of the request to change status. The application program then consults the *arbiter*, which determines whether the participant can make the status change. The rule for determining the validity of status change is discussed in Section 4.1. After consulting the *arbiter*, the application program will reply to the application server saying whether the request is approved or not. The application server will then update the status if the requested status change is approved.

The communication between the application server and the application program is indispensable in many other situations which will be discussed in the presentation of the implementation of the *arbiter* and in the discussion of the main features of **CTDS**.

2.3.2 Communication Between Application Program and Arbiter

Since the arbiter does not necessarily run on the same machine as any of the clients, some kind of network communication is necessitated between the application programs and the *arbiter*. In **CTDS**, TCP/IP communication is used for this purpose. TCP/IP is chosen for simplifying the implementation of total ordering since it guarantees in-order delivery of data packets.

The communication channel between an application program and an *arbiter* not only makes the communications necessitated from the scenarios described in the previous section (joining or leaving of participants, requests and approvals of status changes) feasible, but it also plays a very important role in achieving application sharing. In fact, the communication channel is mainly used for transmitting transactions, which are changes to the shared model and transaction-related requests and replies, in either direction. Details of data flow in **CTDS**



1. Master sends a new transaction A, which is state information of the shared model or update to the state of the shared model, to the arbiter.
2. Arbiter assigns a sequenece number to transaction A => transaction A w/ seq. # a. Then, arbiter multicasts the sequenced transaction to all clients.
3. Client replies "ready to commit" sequence number a, i.e. transaction A.
4. Arbiter, upon receiving "ready to commit" from all clients, sends "commit seq. # a" to all clients.

Figure 2.4: Data Flow Diagram of One Transaction in **CTDS**

are discussed in the following section.

2.4 Data Flow

In **CTDS**, application sharing is achieved by multicasting changes to the shared model to all participants. Application programmers have control over which changes should be multicasted and which should not. Atomic ordering provided by a 2-phase commit protocol [CDT94], working with the total ordering provided by TCP/IP and the *arbiter*, ensures state consistency among all clients. Masters send changes to the shared model as a transaction to the *arbiter*. The *arbiter* then assigns unique sequence number to this transaction and multicasts it to all clients, including the originator of the transaction. The multicasting is done by point-to-point TCP/IP communication to every client. Upon the receipt of a transaction, a client checks whether it is “ready-to-commit” the transaction by comparing the sequence number of the received transaction and the sequence number of the transaction it just committed. If the difference between the sequence number of the transaction just committed and that of the received transaction is greater than 1, the received transaction is not ready to be committed. If the transaction received is ready to be committed, the client replies to the *arbiter*. Otherwise, it stores the transaction and waits until it is “ready-to-commit” the transaction before replying to the *arbiter*. When the *arbiter* receives replies from all clients for a particular transaction, it will inform all clients to commit the transaction by sending “commit” to them. For details on the algorithm of 2-phase commit protocol, refer to [CDT94]. *Figure 2.4* summarizes the data flow of a single transaction in **CTDS**.

2.5 Other Models

In addition to the architecture used in building **CTDS**, there are two other main architectures for implementing collaborative tools development systems which are commonly used in the research field. We will introduce both of them. Some comparisons and contrasts between them and the **CTDS** model will also be presented.

2.5.1 Sequencer-based Model

The first model we are going to introduce is what we described as the *sequencer-based distributed application architecture*. In this architecture, one of the application instances, which is called the *sequencer*, will do the sequencing and ordering job. Masters send transactions to the *sequencer*. Acting like the *arbiter* in **CTDS**, the *sequencer* will carry out the 2-phase commit protocol for the transactions after assigning sequence numbers to them. *Figure 2.5* shows the *sequencer-based distributed application architecture*.

Apparently, the application code has to be modified in order to communicate with

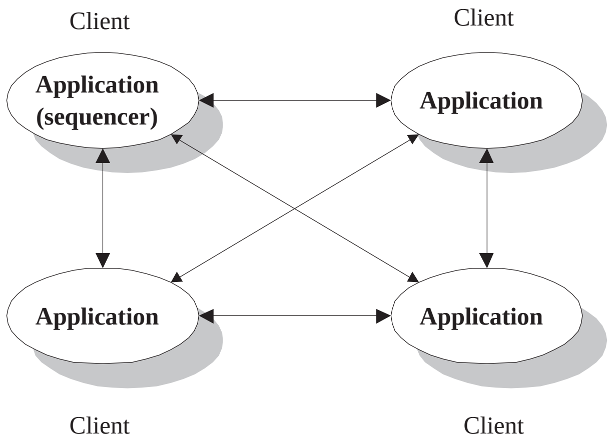


Figure 2.5: Sequencer-based Distributed Application Architecture

other application instances, as well as to perform the sequencing and ordering job. Note that not only the sequencer, but also all application instances in the session need to have the sequencing and ordering procedures incorporated. This is because the provision of fault-tolerant service requires that every application instance be willing to become the sequencer. If the current sequencer dies or the corresponding participant quits, an election has to be called to elect a new sequencer for the session to continue. This is done by some kind of election algorithm, which are well-developed in the field of distributed systems. Since any of the application instances can become the sequencer, communication channels have to be established between each pair of application instances despite of the fact that application instances only communicate with the *sequencer*.

The architecture employed in building **CTDS** has advantages over the *sequencer-based distributed architecture* in several aspects. Firstly, less communication channels are required to be established in the **CTDS** architecture. The number of communication channels in the **CTDS** architecture increases linearly with the number of participants, whereas the number of communication channels in the *sequencer-based architecture* increases with

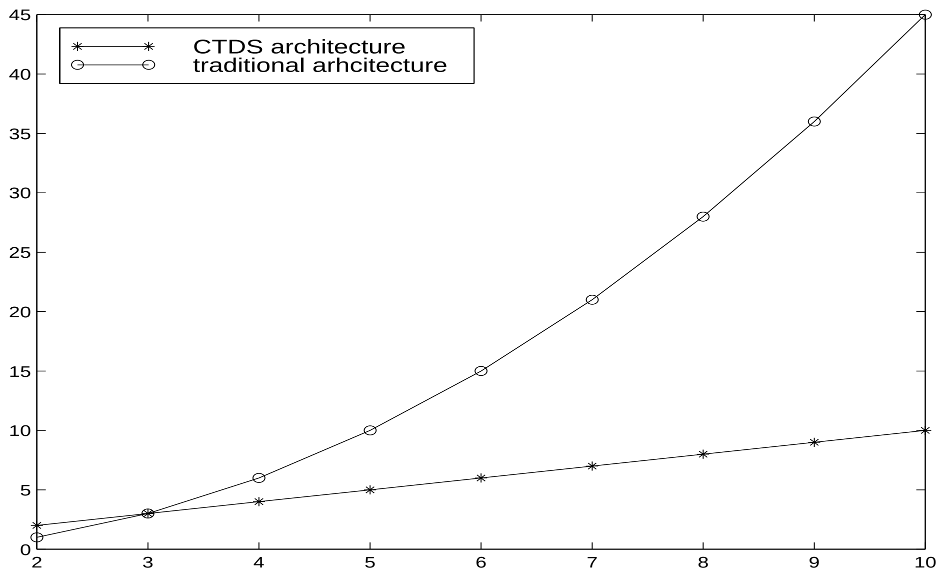


Figure 2.6: Number of Communication Channels Required in *Traditional* vs. **CTDS**

the square of the number of participants. *Figure 2.6* gives the numbers of communication channels required for some chosen numbers of participants in both architectures. The larger the number of participants, the bigger the difference between the number of communication channels required to be established (*Table 2.1*).

Worst of all, most of the communication channels established in the *sequencer-based architecture* are not active. Application instances do not need to communication with each other, except with the sequencer. Consider the session shown in *Figure 2.5*, only three of the six communication channels established are actually used. On the contrary, all the established communication channels are utilized in **CTDS**. Thus, the *sequencer-based architecture* is wasting computer resources in some sense. *Table 2.2* shows the percentage of unused communication channels in the *sequencer-based architecture*.

Secondly, in the **CDTS** architecture, the collaboration has less impact on the interac-

Number of Participants	**Communication Channels Required**		**Communication Channels Difference**
	Traditional	**CTDS**	
2	1	2	-1
3	3	3	0
4	6	4	2
5	10	5	5
6	15	6	9
7	21	7	14
8	28	8	20
9	36	9	27
10	45	10	35

Table 2.1: Information on Communication Channels Required in *Sequencer-based* and **CTDS**

Number of Participants	**Percentage of Unused Communication Channels**
2	0%
3	33.3%
4	50.0%
5	60.0%
6	66.7%
7	71.4%
8	75.0%
9	77.8%
10	80.0%

Table 2.2: Percentage of Unused Channels in *Sequencer-based architecture*

tivity of the application instances. In the *sequencer-based architecture*, the sequencer, which is one of the application instances, has to do the sequencing job in addition to handling the normal application interactions. This not only deteriorates the interactivity of the sequencer, but also makes the 2-phase commit protocol carried out by the sequencer not as effective as that carried out by the *arbiter* in **CTDS**. As a result, the performance of the entire session is affected. Thirdly, the **CTDS** architecture requires no election algorithm since it is always the *arbiter* which does the sequencing job. Even though election algorithms are well-developed in the area of distributed systems, they are usually hard to implement. **CTDS** uses a clean, simple method to take care of the failure of the *arbiter*. The method is described later in the

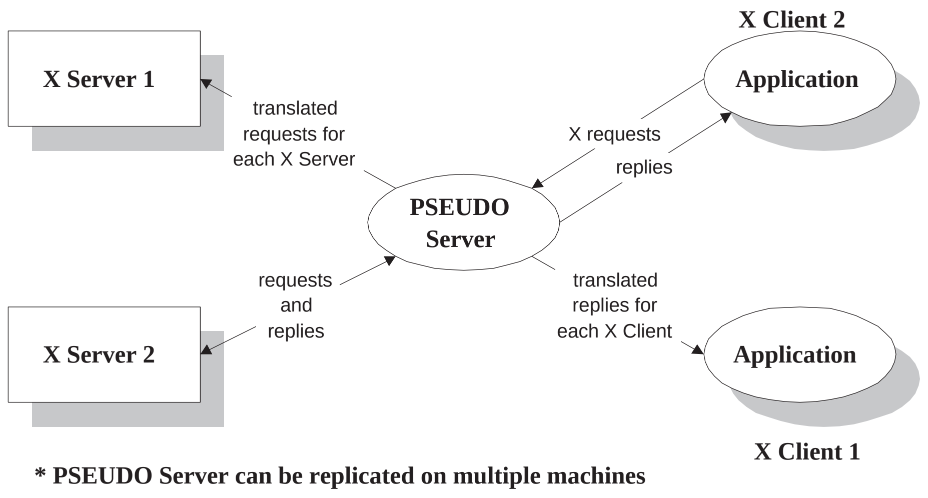

X Client 2
X Server 1
Application
translated requests for each X Server
X requests
replies
PSEUDO Server
requests and replies
translated replies for each X Client
X Server 2
Application
X Client 1
* PSEUDO Server can be replicated on multiple machines

Figure 2.7: *PSEUDO Server Architecture*

discussion of the fault-tolerant feature of **CTDS** (Section 4.3). Lastly, and most importantly, the sequencing code is totally separate from the application code in the **CTDS** architecture. In **CTDS**, the sequencing code is located in the *arbiter*. As discussed before, the sequencing code is completely application independent. Thus, **CTDS** allows the sequencing code to be written only once for the *arbiter* but works for all applications.

2.5.2 PSEUDO Server Architecture

PSEUDO server architecture is a popular architecture used in providing application sharing. *Figure 2.7* shows the basics of the *PSEUDO server architecture*. There are many variants of the *PSEUDO server architecture*. All the variants provide application sharing based on exchanges of some kind of window events, mostly XEvents. In the rest of this discussion, we will assume an X window environment, and hence, XEvents are exchanged to provide application sharing. This section serves as an overview on how the *PSEUDO server architecture* provides application sharing. Details can be found in many papers discussed in Chapter 5.

In the *PSEUDO server architecture*, a PSEUDO server is introduced in between the

X Clients, which are the applications in a session, and the corresponding X Servers. This PSEUDO server assumes the role of a X Server when interacting with X Clients and the role of a X Client when interacting with X Servers. Its main responsibility is the translations and modifications of X requests and replies in order to make them meaningful to a particular X Server or X Client. The translations and modifications are necessary because different resources with different identifiers are associated with each X Server and X Client connection. When a X Client generates a X request, it is sent to the PSEUDO server. The PSEUDO server forwards the request to the local X Server, as well as translating and modifying the request for X Servers associated with other X Clients in the session. The PSEUDO server then sends the translated request to the X Servers of all other participants. Variants of the *PSEUDO server architecture* include replicated or centralized PSEUDO servers, and replicated or centralized storage of application data, replicated or centralized applications.

The *PSEUDO server architecture* give rise to collaboration-unaware software. It provides view-level sharing. Thus, it requires no modification to the application. However, as mentioned before, it results in a lost of semantics. This is due to the exchanges of XEvents, in contrast to exchanges of model-level events, between application instances. View-level events are not as meaningful as model-level events to human users. The *PSEUDO server architecture* also introduces complications into providing fault-tolerant application sharing since there are more points of failures as compared to **CTDS**. For instance, in a replicated *PSEUDO server architecture*, mechanisms have to be defined to take care of the failure of any of the PSEUDO servers. This may include having some of the X Clients connect to another PSEUDO server when failure of the corresponding PSEUDO server happens. Moreover, systems using *PSEUDO server architecture* are limited to a particular window environment. This results from the sharing of window events. And the PSEUDO server is also hard-coded to perform the translations and modifications of window-system-dependent events. **CTDS**, on the other hand, does not have any restriction on the underlying window environment. In **CTDS**, only the application server has a user interface. Tcl/Tk, a programming system which consists of a basic programming language (Tcl) and a toolkit of widgets (Tk), is used to implement the application server so that the application server can run on the two main window environments: Windows, and X Window. In addition, implementing the request translation algorithm in PSEUDO servers requires thorough understanding of the window environment of interest. On the contrary, **CTDS** does not require any knowledge of the underlying window environment. Lastly, the *PSEUDO server architecture* only allows sharing of graphics calls which are directed to X Servers. This causes difficulty in some 3-D X Windows applications that utilize direct hardware access which bypasses X Servers.

Chapter 3

Implementation

This chapter discusses the implementation details of **CTDS**. It presents the details on the initiation and termination of a session. Implementations of joining a session and leaving a session are also discussed. **CTDS** consists of the application server and the *arbiter*. *Arbiter* is the core component in **CTDS**. The application server, which is the other component, is mainly for provision of a user interface for session management resulted from collaboration. Thus, only the *arbiter* is discussed in detail in this chapter. Internal details of the *arbiter* are presented.

3.1 Session

3.1.1 Session Initiation

A session is started by having the initiator running the application program. There are several parameters associated with a session in **CTDS**. All of them are required for initiating a session:

1. *Server Host*

   - the host name of the machine on which the *arbiter* is going to run.

2. *Server Port*

   - the port number (communication point) on the *server host* through which the *arbiter* is going to communicate with clients.

3. *Session Number*

   - an unique identifier to identify a session. This is indispensable in identifying a session if there are more than one *arbiter* running on the same machine.

4. *Maximum Number of Masters*

   - the maximum number of masters allowed in the session.

In addition to these parameters, every participant has to give his/her own name to start the application. The name is part of the participant information displayed in the application server.

As mentioned earlier, the *arbiter* is started automatically when a session is initiated. This is done by registering the *arbiter* as a new service provided by **inetd** (**Internet Daemon**). **inetd** is a daemon process running on a typical UNIX server machine which listens for all kinds of requests and invokes the appropriate server to handle the request based on the type of the request received. For details on **inetd**, refer to [Ste90]. This means that the *arbiter* has to run on a machine with **inetd** running. When the initiator's application is started, it will talk to **inetd** on the given *server host*. **Inetd** will then spawn off an *arbiter* as requested by the initiator's application. The *arbiter* then binds to the given *server port*. Failure in binding to the *server port* results in immediate termination of the session initiation process. The initiator has to pick another *server port* and try to establish the session again. Finally, the initiator's application establishes a connection with the *arbiter* on the *server port*. Mean-

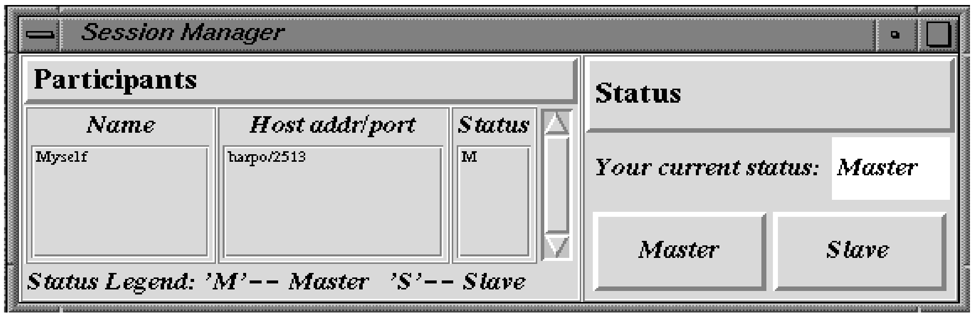


Figure 3.1: Initiator's Application Server

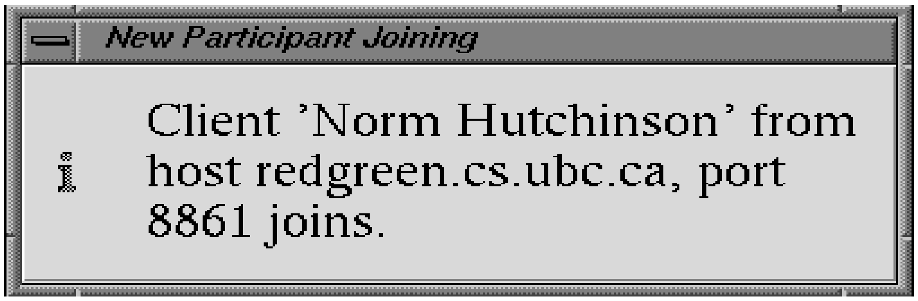
while, the application server will also be invoked and it will show the initiator as the only participant in the session. The initiator is always a master unless an explicit status change is requested through the application server after the session is successfully established. *Figure 3.1* shows the application server of the initiator right after a session is started.

3.1.2 Session Joining

To join a session, a participant is required to specify three of the four parameters that are mandatory in initiating a session. These three parameters are used to correctly identify the desired session. They are:

1. *Server Host*
2. *Server Port*
3. *Session Number*

Every time an application is started, it will talk to **inetd** on the *server host*. As in the initiator's case, **inetd** will spawn off an *arbiter* as requested. However, this *arbiter* will fail in binding to the given *server port* since the initiator has already started an *arbiter* on the given *server port* for the specified session. This *arbiter* will inform the application of the



Client 'Norm Hutchinson' from host redgreen.cs.ubc.ca, port 8861 joins.

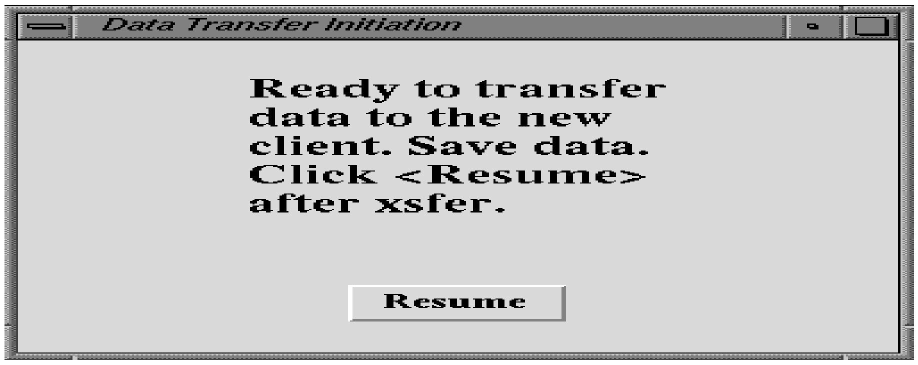
Figure 3.2: Join Notice Message Box

failure in binding. Upon receiving the notice of the failure, the application will establish a connection directly with the existing *arbiter* started by the initiator.

All participants currently in the session will be informed of the joining by the *arbiter*. The information is in the form of a message box as illustrated in *Figure 3.2*. Interactions on the shared model will be suspended until the new participant successfully joins or dies in the joining process.

The suspension of interactions is enforced to ensure that the shared model is in a consistent state for retrieval by the new participant. After suspending interactivity, all outstanding transactions, which have not completed 2-phase commit protocol, will be processed as usual, bringing the shared model to a final state. To complete the joining process, all participants currently in the session will be prompted to transfer the shared model to the new participant (*Figure 3.3*).

Any of the participants can push the "**Resume**" button after transferring the shared model to the new participant. Since only one of the participants will do the transfer, the dialog box shown in *Figure 3.3* at all other participants' sites will disappear once the "**Resume**" button is pushed by the participant who does the transfer. The new participant not only needs the shared model, but also needs the information of all the existing participants.



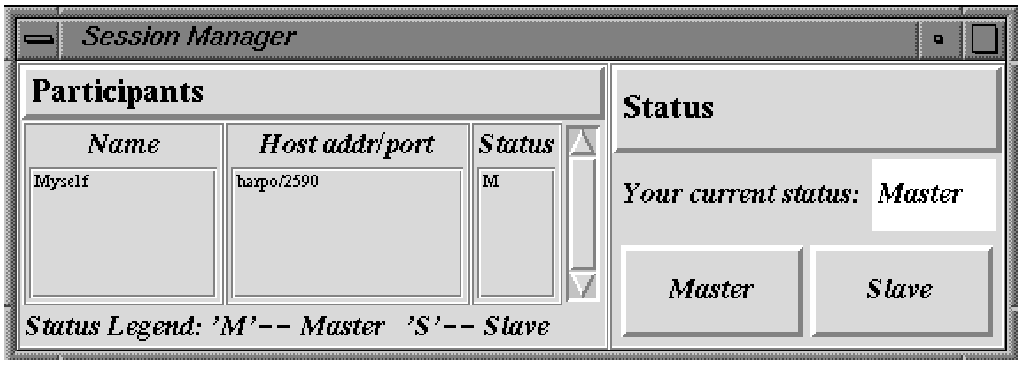
Ready to transfer data to the new client. Save data. Click <Resume> after xsfer.

Figure 3.3: Transfer and Resume Dialog Box

After the “**Resume**” button is pushed by one of the participants, the *arbiter* transfers information of all existing participants to the new participant. Then, interactivity is resumed and the session continues. Application servers of all the old clients will reflect the joining of the new participant while the application server of the new participant shows a complete list of all the participants currently in the session, including himself/herself. *Figure 3.4* presents the application server before and after the joining of a new participant. If, anytime during the joining process, the new participant dies, all the current participants will be informed by an appropriate message box (*Figure 3.5*). Interactivity will be resumed automatically afterwards.

3.1.3 Session Leaving

A participant can leave anytime during a session. Leaving under special circumstances, including recovery of *arbiter* (Section 4.3), the process of joining of a new participant, the process of requesting to join, are supported. A participant leaves a session by either closing the application server or by quitting the application program.

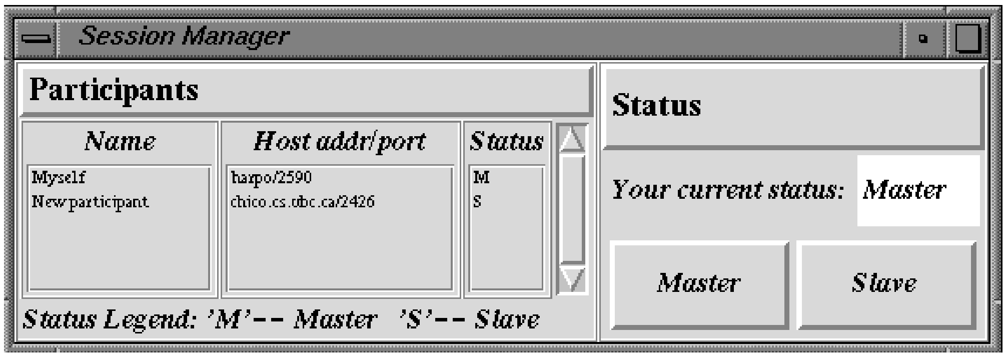


Name	Host addr/port	Status
Myself	harpo/2590	M

Status Legend: 'M'-- Master 'S'-- Slave

Your current status: Master

Application Server before a new participant joins



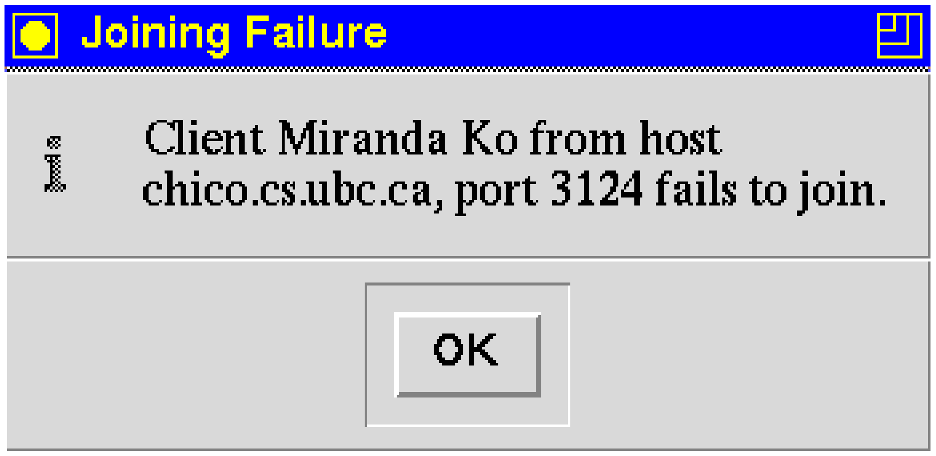
Name	Host addr/port	Status
Myself	harpo/2590	M
New participant	chico.cs.ubc.ca/2426	S

Status Legend: 'M'-- Master 'S'-- Slave

Your current status: Master

Application Server after the new participant with name "New participant"joins

Figure 3.4: Application Server Before and After New Participant Joins


Joining Failure

Client Miranda Ko from host chico.cs.ubc.ca, port 3124 fails to join.

Figure 3.5: Failure in Joining Message Box

3.1.4 Session Termination

A session is terminated automatically when all participants leave. If there are no participants left, the corresponding *arbiter* will be terminated.

3.2 Arbiter

This section uncovers the state information stored in the *arbiter* as well as the implementation details on several aspects of the arbiter's functioning.

3.2.1 State

Section 2.4 outlines how the *arbiter* provides coordination and communication services to collaborative applications. In order to support these two services, the *arbiter* has to keep different kinds of information. The information that the *arbiter* has to keep is described as the *state* of the *arbiter*. The *state* of the *arbiter* consists of two main components:

1. Information of all participants in the session

2. Sequenced transactions received from all clients, along with a 2-phase commit protocol status for each of the participants for every transaction

Information of all participants includes the participants' names, the addresses of the hosts and the port numbers at which the clients are, the clients' status (*master/slave*). Information of the participants is kept for providing multicast as well as for validating status change requests.

Different participants can be at different stages in the 2-phase commit protocol. For instance, participants at slower machines may still be processing transactions which were received earlier when a new transaction is sent to them by the *arbiter*. Hence, those participants are not "ready-to-commit" the new transaction yet. On the contrary, participants at faster machines, who have processed all earlier transactions, are "ready-to-commit" the new transaction. And they will send "ready-to-commit" to the *arbiter* for the new transaction. Besides, some participants may have received a particular transaction while some others have not. This is due to the usage of multiple point-to-point communications in implementing multicast. Thus, for every transaction, the *arbiter* has to store a 2-phase commit protocol status for each participant. The *arbiter* also stores the sequence numbers of transactions. As discussed in Section 2.4, the 2-phase commit protocol uses sequence numbers of transactions and 2-phase commit protocol status of each participant to provide atomic ordering. Therefore, the storage of transactions' sequence numbers and 2-phase commit protocol status is essential for proper implementation of 2-phase commit protocol. For every transaction, there are four possible 2-phase commit protocol status:

1. *NONE*

   - The initial status for every participant. When the *arbiter* is in the process of multicasting a transaction and a participant has not been sent the transaction yet, the status for that participant is *NONE*.

2. *MULTICASTED*

   - The transaction has been sent to the corresponding participant.

3. *READY TO COMMIT*

   - "ready-to-commit" has been received from the corresponding participant.

4. *COMMIT*

   - "commit" has been sent to the corresponding participant. This status only appears after all participants have replied "ready-to-commit".

Transactions with 2-phase commit protocol status being *COMMIT* for all participants are said to be committed from the point of view of the *arbiter*. Indeed, the *arbiter* has no way to tell whether or not a transaction is actually committed by clients based on the 2-phase commit protocol status it stores. Clients may fail to perform the transaction which the arbiter asks all clients to commit. As a result, there are two interpretations of "committed transactions". The first interpretation results from the point of view of the *arbiter*. And the second one results from the point of view of a client. In order to distinguish between the two interpretations, *committed*$_{arbiter}$ is used to describe the committed transactions from the point of view of the *arbiter* and *committed*$_{client}$ is used to describe the committed transactions from the point of view of clients.

The sequence number of a transaction, the transaction, and the corresponding set of 2-phase commit protocol status are stored in a fixed-size list. This list is named a *transaction list* (*Figure 3.6*).

There are two important pointers to the list:

1. *current*

   - pointer to the first empty slot in the list. The next transaction received will be stored

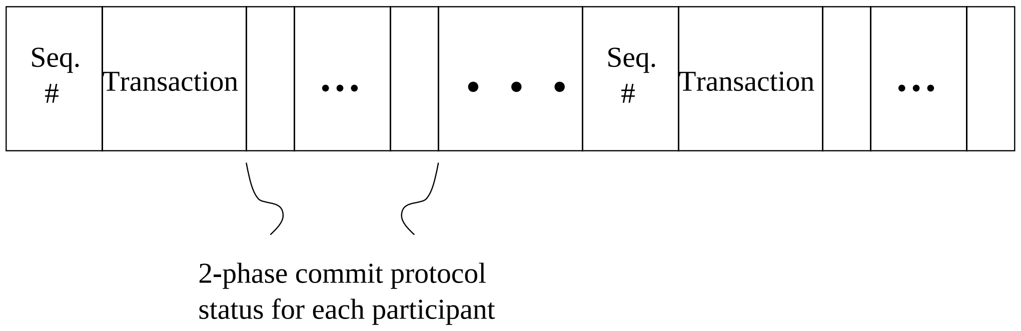


Figure 3.6: *Transaction list* Structure

in the slot being pointed to. This pointer will be incremented to point to the next slot every time a new transaction is received and stored.

2. *start*

   - pointer to the first transaction stored in the list.

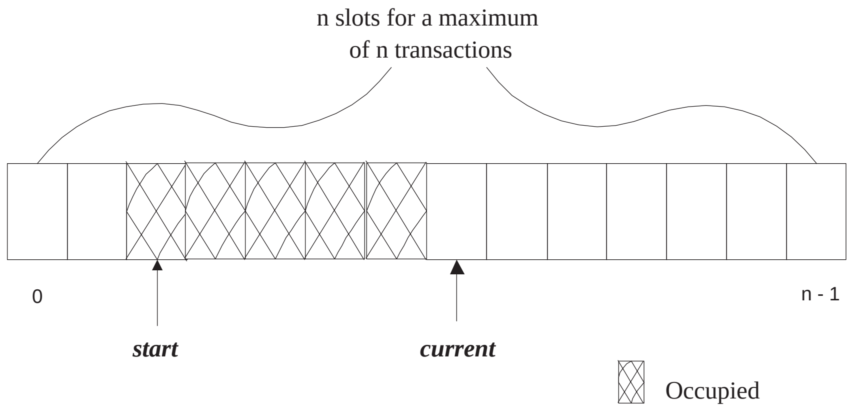


Figure 3.7: *Transaction list* in the *arbiter*

Transaction list is actually a circular list. The *start* pointer can point to any slot in the list (*Figure 3.7*). If all slots from *start* to n-1 are occupied, the next transaction received will be stored in slot 0. Slots in the list are recycled by a *flushing* mechanism discussed in

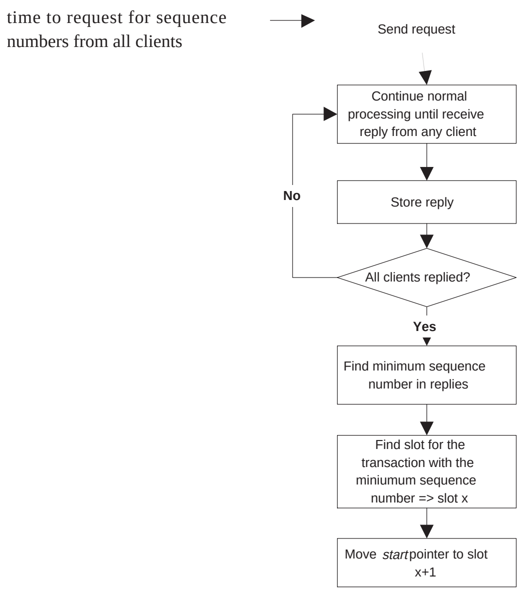
Section 3.2.2.

3.2.2 Flushing

Due to the limited number of slots in the *transaction list*, a mechanism to free slots has to be established to avoid overflow in the list. The mechanism used in **CTDS** is known as *flushing*. Only slots containing *committed*$_{client}$ transactions can be recycled.

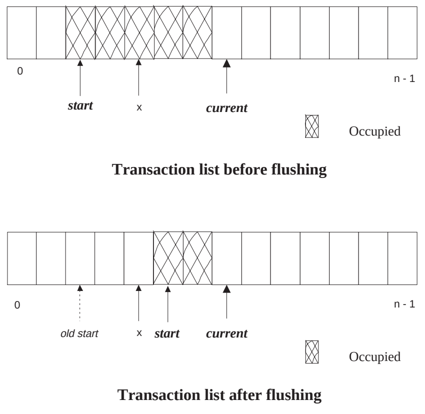
The *flushing* mechanism composes of a periodic polling of all clients for the largest sequence number they have committed so far, finding a minimum from all these sequence numbers, and a recycling of slots in the *transaction list* up to and including the transaction with the minimum sequence number found. Every certain period of time (can be customized), the *arbiter* requests a sequence number from all clients. Every client, upon receipt of this request (*request*$_{flushing}$), replies to the *arbiter* with the sequence number of the last committed transaction. After the *arbiter* collects replies from all clients, it finds the minimum of the sequence numbers in the replies. This sequence number gives the latest transactions that are *committed*$_{client}$ to all clients. Thus, every transaction up to and including this transaction stored in the *transaction list* can be removed safely as no more clients need them. *Figure 3.8(a)* presents the flow chart for the *flushing* mechanism.

An example: Suppose the transaction with the minimum sequence number found is stored in slot `x`. The *arbiter* recycles the slots starting from *start* to `x` by moving the *start* pointer to slot `x+1`. In other words, slots *start* to `x` are appended to the end of the *transaction list* for reuse. *Figure 3.8(b)* illustrates a flushing with the transaction of the minimum sequence number found stored in slot `x`.



time to request for sequence numbers from all clients
Send request
Continue normal processing until receive reply from any client
Store reply
No
All clients replied?
Yes
Find minimum sequence number in replies
Find slot for the transaction with the miniumum sequence number => slot x
Move *start* pointer to slot x+1

(a)



(b)

Figure 3.8: (a) Flushing Flow Chart and (b) Flushing Mechanism

3.2.3 Processing Priority

Priority of processing comes into place when an *arbiter* receives transactions from more than one client at the same time.

Every time the *arbiter* is ready to read requests, it forms a priority list by first assigning the highest priority to joining requests from new participants. The *arbiter* then scans the *transaction list* for the first non-*committed*$_{arbiter}$ transaction. For every transaction starting from the first non-*committed*$_{arbiter}$ transaction, the *arbiter* checks which clients have not replied "ready-to-commit" and the *arbiter* will assign the next highest priority to these participants if they have not been assigned a priority yet. The underlying theory of this is that we want to recycle the slots at the beginning of the *transaction list* as soon as possible. And by having "ready-to-commit" for all participants, the *arbiter* can send "commit" immediately to all clients, which will make the clients commit the transaction. Lastly, the clients without priority assigned after the scanning process will be assigned equal priority.

Whenever there are multiple inputs to the *arbiter*, the *arbiter* will process the inputs according to the priority list generated at that time. The priority list will be re-generated everytime when the *arbiter* reads input.

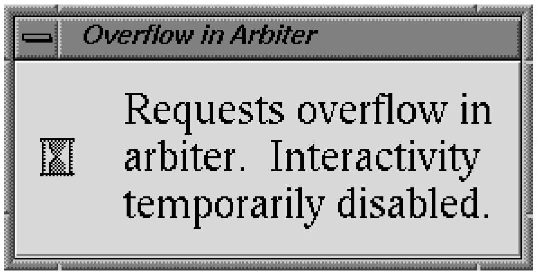
3.2.4 *Transaction List* Full

Despite of the *flushing* mechanism, there is a possibility of overflowing the *transaction list*, especially with interactive client applications. This is because interactive applications are transaction-intensive. Thus, the *arbiter* has to be able to deal with overflow in the *transaction list*.

In **CTDS**, the *arbiter* raises a warning if the number of empty slots in the *transaction list* drops below a certain threshold. The warning forces the *arbiter* to assign the highest priority to the clients who have not replied *request*$_{flushing}$. This is because *flushing* frees slots in the *transaction list*. If the *transaction list* does become full, the *arbiter* will inform all clients to stop sending new transactions. In other words, interactivity of clients will be temporarily suspended until the number of empty slots in the *transaction list* becomes reasonable again. *Figure 3.9(a)* shows the message box used to inform clients of a *transaction list* overflow. Unprocessed transactions being sent to the *arbiter* before the issue of the list full notice will be stored in a temporary list. After suspending interactivity, the *arbiter* continues carrying out 2-phase commit protocol for the transactions in the *transaction list* and continues *flushing* the *transaction list*. When the number of empty slots in the list, in view of the number of unprocessed transactions stored temporarily, becomes acceptable again, the *arbiter* will process the unprocessed transactions. The *arbiter* assigns sequence numbers to these unprocessed transactions and carries out 2-phase commit protocol on them. Then, the *arbiter* resumes interactivity of all clients.

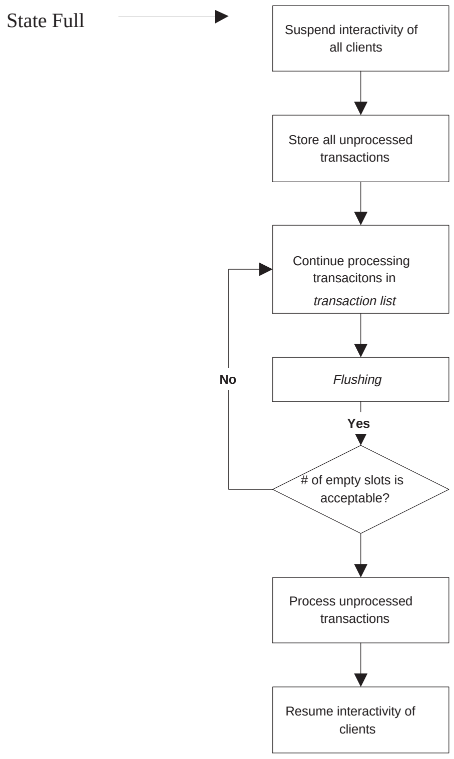
3.2.5 Failure or Leaving of Clients

This is one of the fault-tolerance requirements discussed earlier. In order to satisfy this requirement, the failure or leaving of clients should not affect the continuity of a session. In **CTDS**, leaving or failure of clients are handled in the same fashion. In **CTDS**, the *arbiter* monitors all clients. Whenever the *arbiter* detects failure of a client, it immediately updates its own state by removing the information of that client. The *arbiter* then checks whether there is any clients left in the session. If there are none, the *arbiter* quits to terminate the session. Otherwise, the *arbiter* informs all other clients of the leaving of that client so that the corresponding application server can update the participant list accordingly. In updating the state, the *arbiter* not only has to delete the information of the dead client, it also has to remove the 2-phase commit protocol status of that client for every transaction in the *transaction list* and to remove the reply to *request*$_{flushing}$ from that dead client. Immediate update


Overflow in Arbiter

Requests overflow in arbiter. Interactivity temporarily disabled.

(a)


State Full
Suspend interactivity of all clients
Store all unprocessed transactions
Continue processing transacitons in *transaction list*
Flushing
Yes
No
# of empty slots is acceptable?
Process unprocessed transactions
Resume interactivity of clients

(b)

Figure 3.9: (a) Transaction List Full Message Box and (b) Handling of *Transaction List* Full Flow Chart

of the *arbiter*'s state is crucial to ensure that the *arbiter* does not send latter transactions to the dead client, which may cause unexpected results, including termination of the *arbiter*. Nested client failures may occur when the *arbiter* informs other clients of the failure of a particular client. In this case, failures encountered later are ignored until handling of the current failure is completed.

Since the update of the *arbiter*'s state affects the 2-phase commit protocol status stored, which may make the first non-*committed*$_{arbiter}$ transaction in the *transaction list* "ready-to-commit", the *arbiter* has to check against the first non-*committed*$_{arbiter}$ transaction after updating its state. If, after the failure or leaving of a client, the 2-phase commit protocol status of all remaining clients for the first non-*committed*$_{arbiter}$ transaction are "ready-to-commit", the *arbiter* has to inform all clients to commit that transaction. Moreover, the *arbiter* also has to check whether, after the failure or leaving of a client, all the remaining clients have replied to the *request*$_{flushing}$ sent if any. If the *arbiter* finds replies to *request*$_{flushing}$ for all remaining clients, it carries out *flushing* described in Section 3.2.2.

Note: Either the failure of an application program or the failure of the corresponding application server results in a failure of a client in **CTDS**.

Figure 3.10 summarizes the handling of a client's failure or leaving.

3.2.6 Recovery of State

In addition to handling clients' failure, the other fault-tolerance requirement states that the failure of the *arbiter* should have no impact on the continuity of a session. In order to satisfy this requirement, a new *arbiter* has to be started after the previous one dies. However, restarting an *arbiter* is by no means sufficient to continue a session. The new *arbiter* has to have the same state of the previous *arbiter* in order to continue providing services to clients. Thus, we need to recover the state of the previous *arbiter*. In other words, everytime an *ar-*

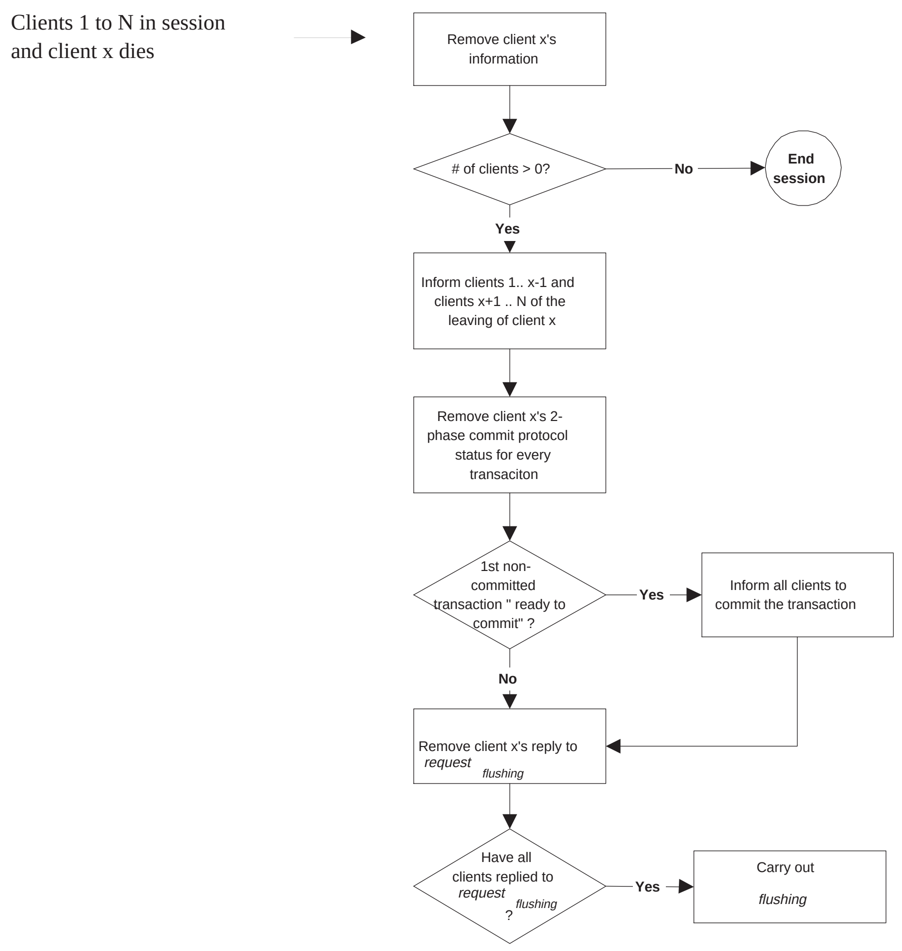

Clients 1 to N in session and client x dies
Remove client x's information
# of clients > 0?
No
End session
Yes
Inform clients 1.. x-1 and clients x+1 .. N of the leaving of client x
Remove client x's 2-phase commit protocol status for every transaciton
1st non-committed transaction " ready to commit" ?
Yes
Inform all clients to commit the transaction
No
Remove client x's reply to *request*_{*flushing*}
Have all clients replied to *request* *flushing* ?
Yes
Carry out *flushing*

Figure 3.10: Handling of Client's Failure or Leaving Flow Chart

biter is started, it has to figure out whether it is a replacement of a dead *arbiter* or it is the first *arbiter* of a session. If it is a replacement of a dead *arbiter*, it has to recover the state of the dead *arbiter* before entering normal service mode. Details on recovering the state of the previous *arbiter* are given in Section 4.3.

Chapter 4

CTDS Features

This chapter presents the key features of **CTDS**. It gives detailed descriptions of the features and the motivations behind them. Implementation of the features are also discussed.

4.1 Multiple Masters

CTDS supports more than one master in a session. A session with a single participant being a master and all others being slave is often called a *presentation*. In a *presentation*, only the *presenter*, who is the master, is allowed to perform interactions on the shared model. All participants take turn to be the *presenter*. A single-master session makes implementation of the underlying collaborative tool development system easier since transactions only come from a single source. As a result, no ordering of transactions is necessary. In a multiple-master environment, the underlying collaborative tool development system not only has to order transactions originating from different masters, but it also has to take into account the possible conflicting actions performed by different masters. In most systems which support multiple masters, either multiple resources are introduced or a "soft protocol" is required to avoid conflicting actions. "Soft protocol" refers to the human coordination between different masters in order to avoid conflicting actions. Because every master is allowed to interact with every part of the shared model, there is a possibility that more than one master is trying to change the shared model at the same time. These simultaneous actions may lead to conflicts. For instance, in a two-master session with a dragon as the shared model, both master A and master B intend to move the tail of the dragon. Master A wants to move the tail to the left hand side, whereas master B wants to move the tail to the right hand side. Masters A and B do the changes at about the same time before they see each other's action. The result of this scenario depends on the amounts of movement performed by masters A and B. The actions of masters A and B can completely cancel out each other, resulting in no movement of the tail. Other possible outcomes include a movement of the tail to the left if the amount of A's movement is bigger than that of B's, and a movement to the right if the amount of B's movement is bigger than that of A's. Unfortunately, none of the three outcomes is desired by either A or B. Therefore, a "soft protocol" is desired. The implementation of a "soft protocol" requires some kind of human communication between the masters. The communication can be done by means of video conferencing or telephone conferencing.

Another way to avoid possible conflicting actions in a multiple-master session is the introduction of multiple collaborative resources into a session. In a multiple-resource and multiple-master session, the shared model is partitioned into several parts. Each part is called a resource. Each resource is "owned" by a single master. That is, only a designated master is allowed to interact with a resource. Thus, no conflicting actions will happen with the provision of multiple-master support. **CTDS** supports multiple-master and multiple-resource sessions. Details on the multiple-resource aspect of a session in **CTDS** are given in the next section.

In **CTDS**, there is a limit on the maximum number of masters in a session. The maximum number of masters is specified by the initiator of a session as discussed in Section

3.1.1. The maximum number of masters is in effect anytime during a session. The application server shows the status of each of the participants so that a participant knows the current number of masters in a session. Switching of status is done using the buttons provided by the application server (*Figure 2.2*). A participant can request to be a master anytime during a session. The request is sent to the *arbiter* for approval. The approval is necessary to ensure the limit on the maximum number of masters is honored. Upon receiving a request to be a master, the *arbiter* checks the current number of masters against the maximum allowable number of masters. If the current number of masters has not reached the limit yet, the *arbiter* grants permission to the originator of the request by informing all participants of his/her status change. Otherwise, the request is declined and the *arbiter* continues servicing as if it has not received such a request. Requests to be slaves can also be made anytime during a session. Requests to be slaves are sent to the *arbiter*. The *arbiter* always approves requests to be slaves. Upon receiving a request to be a slave, the *arbiter* immediately informs all participants, including the originator of the request, of the status change of the originator. All application servers will update their displays afterwards to reflect the new status of the originator of the request. If every participant requests to be a slave, there will be no master in the session. This is allowed in **CTDS**. And we say that all the control is at the *arbiter* in such scenario. Participants can claim control from the *arbiter* anytime afterwards. *Figure 4.1* summarizes the processing of a request of status change.

4.2 Multiple Resources

CTDS supports multiple resources. Resources make up the shared model in a session. In other words, a resource is part of the shared model. Some systems simply divide the workspace in which the shared model sits into a number of fixed-size 2-D squares or 3-D cubes and make these squares or cubes as resources. However, this method of generating resources

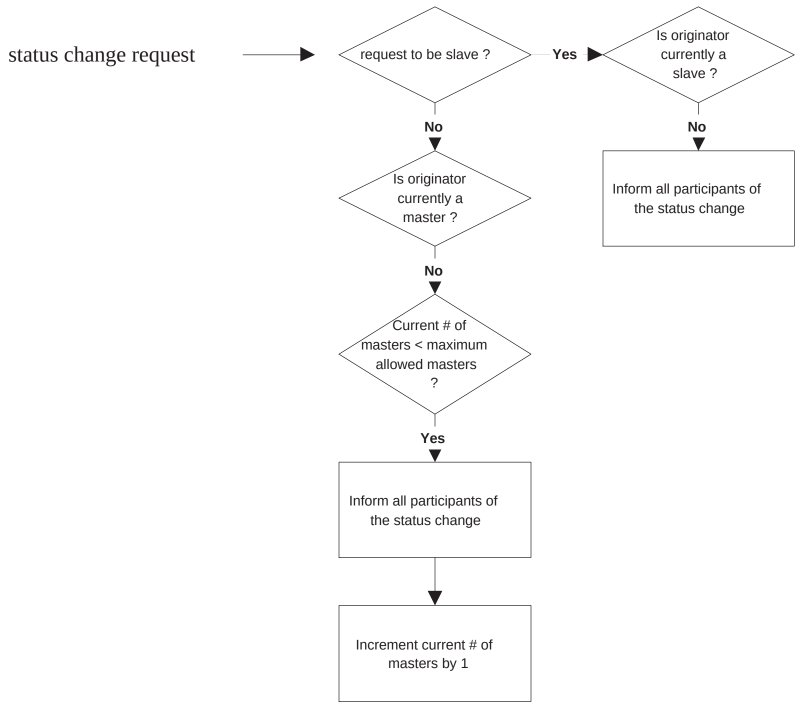

status change request
request to be slave ?
Yes
Is originator currently a slave ?
No
Inform all participants of the status change
No
Is originator currently a master ?
No
Current # of masters < maximum allowed masters ?
Yes
Inform all participants of the status change
Increment current # of masters by 1

Figure 4.1: Processing of Status Change Request Flow Chart

may not be appropriate in some applications. Consider the example used before in which the shared model is a 3-D dragon. In this case, if we follow the method we just described in generating resources, we partition the 3-D workspace into a number of cubes. And each cube can be owned by a master. However, a cube may not mean anything to the corresponding master since the cube may contain part of the tail and part of the dragon's body. Some of the cubes may even contain part of a foot of the dragon but nothing else! Because different applications have different numbers of resources and different ways of partitioning shared models into sensible multiple resources, **CTDS** requires application programmers to define resources on their own. The definition of resources is done by assigning each part of the shared model a *resource ID*. **CTDS** keeps track of which resource is owned by which master and which resource is not owned by any of the masters. Each resource is owned by one master. **CTDS** provides the flexibility to application programmers to determine whether or not a master is allowed to own more than one resource. By supporting multiple resources and enforcing one master per resource, collaboration becomes more effective from the point of view of participants. This is because masters can work on different parts of the shared model simultaneously without worrying about interfering with others' work.

After a participant becomes a master, he/she can start claiming ownership of available resources. Claiming or releasing resources and checking availability of resources are easy and simple in **CTDS**. **CTDS** offers three functions to client applications for doing so:

1. *release_resource(resource ID)*
   - releases the resource of *resource ID*

2. *grab_resource(resource ID)*
   - claims ownership of the resource of *resource ID* if available

3. *resource_available(resource ID)*
   - checks the availability of resource of *resource ID*

release_resource/*grab_resource* both notify the *arbiter* of the release/taking of a resource. The *arbiter* then informs all participants of the change of ownership of the resource of interest. *grab_resource* also checks the availability of the resource of interest before notifying the *arbiter* in order to save communication costs. *resource_available* is provided to application programmers for implementing some other application-specific features. For instance, the example application presented in the next section utilizes *resource_available* for coloring different resources depending on their status. Resources owned by a master are automatically released when the master switches to be a slave or the master dies. Ownership of these resources are given back to the *arbiter*.

4.2.1 Example Application - Robot

The **Robot** application is an interactive graphical application that allows a participant to manipulate a robot, which is the shared model. The robot is partitioned into six different resources: body, head, left arm, right arm, left leg, and right leg (*Figure 4.2*).

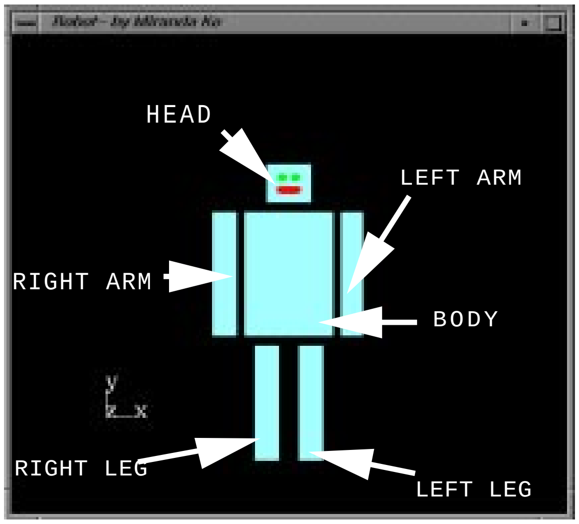


Figure 4.2: **Robot** Application

The original **Robot** application is a stand-alone application. It allows the user to pick any of the body parts and manipulate it. Picking is done by clicking the mouse over the desired body part. There can only be one active body part (picked body part) at any time. After picking a body part, the user can manipulate it by moving the mouse while holding any of the mouse buttons. The name of the current active body part is shown at the lower left hand corner of the application window. The lower right hand corner of the application window also displays the name of a body part. This name belongs to the body part which is currently pickable by the mouse at its current position. For instance, if the mouse cursor is positioned over the left arm with the body being the current active body part, the lower left hand corner shows “BODY” and the lower right hand corner shows “LEFT ARM”. Release of a body part is done by clicking the mouse over the same or another body part. The former results in no current active body part while the latter results in a new active body part. **Robot** employs a coloring scheme to color body parts differently basing on their status. The current active body part is colored red, the pickable body part is colored yellow, and all other body parts are colored light blue.

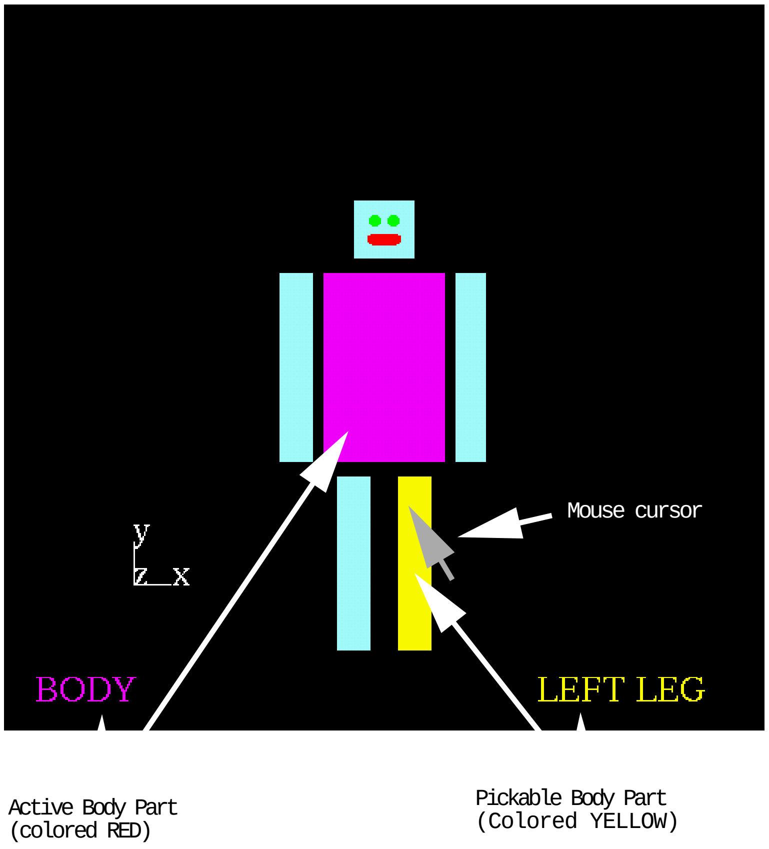


Figure 4.3: Snapshot of the **Robot** Application with Active and Pickable Body Parts

In order to run **Robot** in collaborative mode using **CTDS**, several changes have to be made in the implementations of the picking/releasing of resources and in that of the coloring scheme. Similar to the stand-alone version, the collaborative **Robot** only allows one resource per master. However, picking in the collaborative version is more complicated than that in the stand-alone version. Whenever a participant picks a body part, the resource is not picked right away. That is, the body part will not be colored red immediately, which is not the case in the stand-alone version. Instead, the application program calls *grab_resource* with the ID of the body part as the argument and waits for a reply from the *arbiter*. The reply is used for ensuring that all participants are aware of the change of ownership of the resource. After receiving the reply from the *arbiter*, the corresponding resource can be colored red. Similar procedures are carried out in releasing a resource. The application program does not color the body part light blue immediately upon a releasing action. Instead, the application program calls *release_resource* with the ID of the resource as the argument and waits for a reply from the *arbiter*. After getting the reply from the *arbiter*, the application program colors the corresponding body part appropriately. Changes also have to be made to the coloring scheme employed by the stand-alone version. A new color is required in coloring the body parts owned by other masters so that a master knows what resources are available to be picked. Moreover, resources owned by other masters must not be shown as pickable. In the collaborative **Robot** implemented in this project, only a single color is used to color all resources owned by other masters. Therefore, one cannot tell a resource is owned by which master if there are more than two masters in a session. The new color that collaborative **Robot** uses is dark blue. With the addition of a couple of tests, the rendering procedures used in the stand-alone **Robot** are modified to suit the needs of the collaborative version. Before rendering a resource, a test is performed to check the availability of the resource using *resource_available*. If the result returned is negative, the resource is colored dark blue. An additional test is also introduced into the coloring of the pickable body part. In coloring the pickable body part, the application program ensures that the body part is not owned by other masters utilizing *resource_available* again. *Figure 4.4* shows the scenario in which the pickable body part is owned by another master.

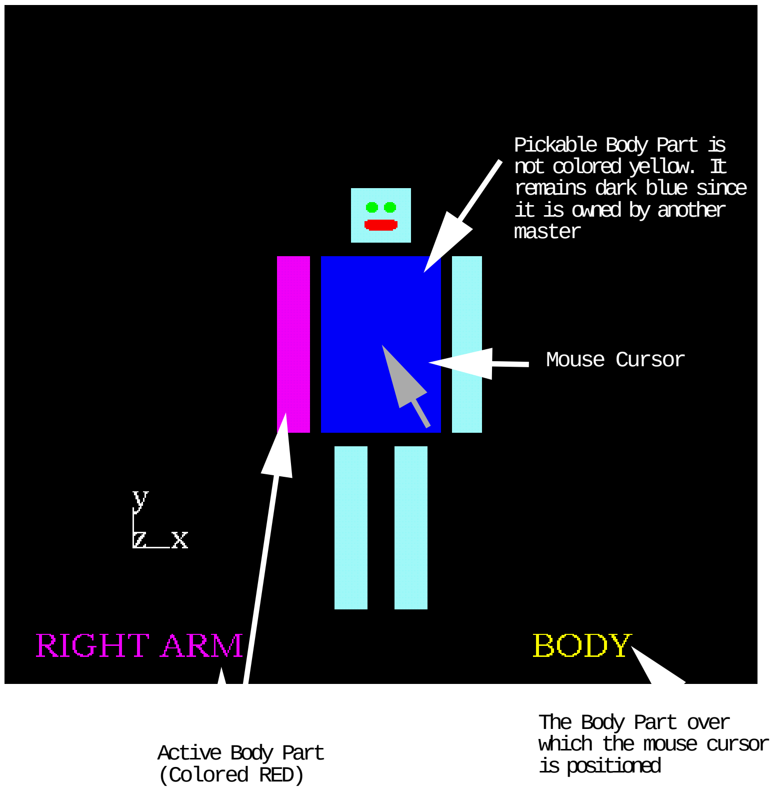

Pickable Body Part is not colored yellow. It remains dark blue since it is owned by another master
Mouse Cursor
y
z x
RIGHT ARM
BODY
Active Body Part (Colored RED)
The Body Part over which the mouse cursor is positioned

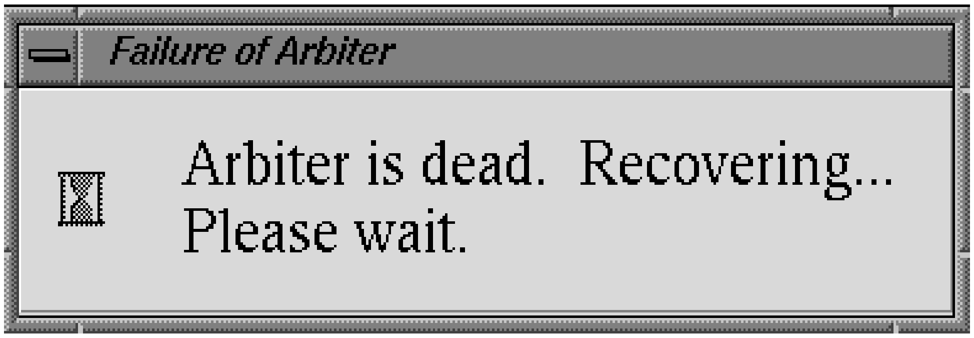
Figure 4.4: No Pickable Body Part in **Robot**

4.3 Fault-Tolerance

CTDS is a fault-tolerant system. The *arbiter* is recoverable. In other words, a session can continue after the corresponding *arbiter* recovers from failure. The ability to sustain the failure of a server is one of the fault-tolerance requirements discussed earlier. The recovery of an *arbiter* refers to the starting of a new *arbiter* as a replacement of the dead one, and the recovering of the state of the dead *arbiter* in the new *arbiter*.

4.3.1 Algorithm

The fundamental question is: who or which process is responsible for starting a new *arbiter* as a replacement of the dead one. One simple solution is a manual restart of the *arbiter*. This solution is not desired due to two main reasons. Firstly, this solution implies that at least one of the participants must monitor the *arbiter* all the time during a session in order to catch any failure of the *arbiter* and to restart it as soon as possible. As a result, the system is not automatic. Secondly, none of the participants may have the authority to start a new process on the *server host*. **CTDS** automates the restart of an *arbiter*. The automatic restart of an *arbiter* is done by the first client which notices the failure of the *arbiter*. Similar to the detection of clients' failure, the failure of an *arbiter* is detected by having all clients in a session monitoring the *arbiter*. The rationale behind this method of restarting an *arbiter* is that an *arbiter* does not have to be restarted if there is no client left in a session. This is because the session should have been terminated if there is no client left. Thus, there is always a client which detects the failure of the *arbiter* whenever the *arbiter* has to be restarted. Upon detection of the failure of the *arbiter*, the application server notifies the participant of the temporary service interruption by a message box (*Figure 4.5*). Interactivity is temporarily suspended until the recovery of the *arbiter* completes. Depending on whether it is the first client which detects the failure, the client then carries out procedures similar to those in session initiation (Section 3.1.1) or to those in session joining (Section 3.1.2) to start a new *arbiter*. The former ones being carried out by the first client which detects the failure while the latter ones being carried out by all other clients which detect the failure. Once a client detects the failure of the *arbiter*, it increments the *server port* by a certain number. The client then requests the *arbiter* service from **inetd** on the *server host*. A new *arbiter* will be spawned by **inetd**. This new *arbiter* binds to the incremented *server port*. Finally, the client establishes a connection with the new *arbiter*. As in session initiation and session joining, only the first client which



Arbiter is dead. Recovering...
Please wait.

Figure 4.5: Notification of Failure of *Arbiter* Message Box

talks to **inetd** will succeed in creating a new *arbiter* at the incremented *server port*. Upon receiving the notice of the new *arbiter*'s failure in binding to the incremented *server port*, the client connects directly to the existing *arbiter*. The existing *arbiter* is the new *arbiter* spawned off by the first client which detects the failure of the previous *arbiter*. *Figure 4.6* illustrates the process of restarting a new *arbiter*. The purpose of incrementing the *server port* is to ensure that failure of binding to the *server port* is a result of an existing *arbiter*, and it is not a result of the hold time imposed on a port. If the new *arbiter* tries to bind to the same *server port* as the dead one did, it may fail since the port is not released yet due to the hold time.

The new *arbiter* has to have the same state as the dead *arbiter* in order to continue servicing clients in the session. The recovery process requires total recovery of the state of the dead *arbiter*. The state of the dead *arbiter* includes information of all clients, and the *transaction list*. Information of all clients (names of clients, hosts and ports of clients, status of clients) can be acquired through the establishments of connections with clients. However, the new *arbiter* still lacks the data stored in the *transaction list* of the dead *arbiter*. Because there exists only two kinds of processes in a session: client and *arbiter* and, as discussed earlier, no recovery of an *arbiter* is needed if there is no client left in a session, the content

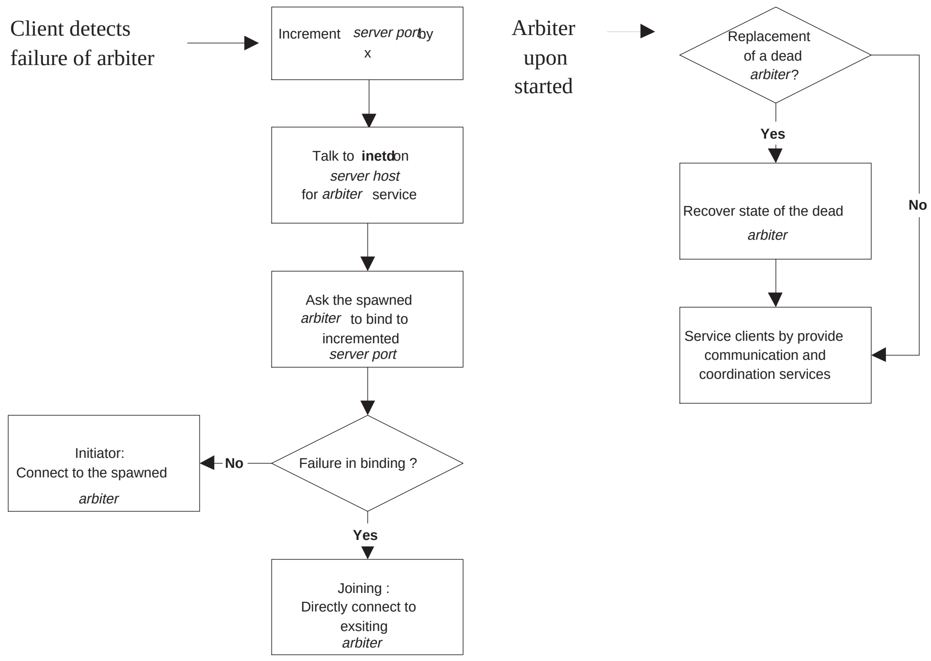

Client detects failure of arbiter
Increment server port by x
Talk to inetd on server host for arbiter service
Ask the spawned arbiter to bind to incremented server port
Failure in binding ?
No
Initiator: Connect to the spawned arbiter
Yes
Joining : Directly connect to exsiting arbiter
Arbiter upon started
Replacement of a dead arbiter?
Yes
Recover state of the dead arbiter
No
Service clients by provide communication and coordination services

Figure 4.6: Restart of *Arbiter* Flow Chart

of the *transaction list* can be recovered from all clients in **CTDS**. The recovery process is successful only after the *transaction list* of the dead *arbiter* is totally recovered and all processing on the recovered *transaction list* are completed. Recovery of the *transaction list* is discussed in details in later sections. When an *arbiter* is in the process of recovering state, it is said to be in recovery mode. After the recovery process, the message box in *Figure 4.5* will disappear and interactivity will be resumed.

Every time an *arbiter* is started, every connected client will send a message to it. This message is for telling the *arbiter* whether it is a replacement of a dead one or it is a server of a new session. If the *arbiter* realizes that it is a server of a new session, it begins servicing clients immediately. Otherwise, the *arbiter* recovers the state of the dead *arbiter* before entering service mode. *Figure 4.6* shows the procedures that an *arbiter* follows when it is first started.

When an *arbiter* dies during recovery, a *nest failure* occurs. **CTDS** handles *nest failure* and also any occurrences of *nest failures* within a *nest failure*. *Nest failures* are handled by having the clients to go through the procedures of starting a new *arbiter* again. The monitoring of an *arbiter* by clients starts immediately after connections are established with the *arbiter*. During a recovery process, if failure of the *arbiter* is detected, a client will carry out exactly the same procedures of starting a new *arbiter* as in the non-*nest failure* case. In other words, regardless of the mode of an *arbiter* (service/recovery), the same set of procedures (*Figure 4.6*) is followed by a client upon the detection of the failure of the *arbiter*. In the case of *nest failure*, the recovery process involves the recovery of the state of the last "functional" *arbiter* instead of the dead *arbiter*. A "functional" *arbiter* is defined as an *arbiter* which is in service mode.

Failure of clients during recovery will not affect the recovery process. The information of the dead client as well as the *transaction list* information sent to the *arbiter* by that client will be erased at once.

4.3.2 Implementation

Client

The recovery of the *transaction list* of an *arbiter* relies on clients. Recall that the *transaction list* of an *arbiter* is an array of sequence numbers of transactions, the transactions, along with a set of 2-phase commit protocol status associated with every transaction. In order to recover the *transaction list* of an *arbiter*, clients need to store the transactions received from the *arbiter*. The data structure used to store the transactions is very similar to the *transaction list* of the *arbiter*. The only difference is that the client's data structure only stores the client's own 2-phase commit protocol status for each of the transactions received from the *arbiter*, instead of an array of 2-phase commit protocol status for every participant as on the *arbiter* side. During recovery of the *transaction list*, the new *arbiter* recovers the 2-phase commit protocol status of all clients by having all clients send to it their own status with sequence numbers attached to them. The client's data structure is named *Buffer*$_{processed}$ (*Figure 4.7(a)*) since this data structure stores information of all transactions that are processed (being assigned sequence numbers) by the *arbiter*. The 2-phase commit protocol status used on the client side are slightly different from those used on the *arbiter* side (Section 3.2.1). This is due to different meanings of a 2-phase commit protocol stage resulted from the two points of view of a sender and a recipient: when an *arbiter* sends, a client receives. For every transaction, there are three possible 2-phase commit protocol status on the client side:

1. *RECEIVED*

   - The multicast of the transaction is received. This status corresponds to the status *MULTICASTED* used on the *arbiter* side.

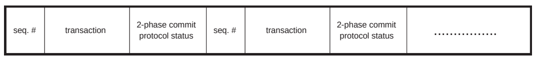
2. *RCOMMIT*

   - The transaction is ready to commit. In other words, "ready-to-commit" has been sent to the *arbiter*. This status corresponds to the status *READY TO COMMIT* used on the *arbiter* side.

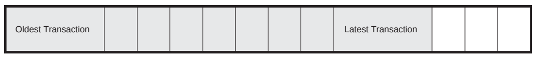
3. *COMMITTED*

   - The transaction is *committed*$_{client}$. This status only appears after receiving "commit" from the *arbiter*. This status corresponds to the status *COMMIT* used on the *arbiter* side.

In fact, clients need to store another type of data to ensure no data is lost due to the failure of an *arbiter*. This type of data consists of the transactions that are generated by clients but have not been processed by the *arbiter*. These transactions have already been


seq. #
transaction
2-phase commit protocol status
seq. #
transaction
2-phase commit protocol status
....

(a)



(b)

Figure 4.7: (a) *Buffer*$_{processed}$ and (b) *Buffer*$_{unprocessed}$

sent to the *arbiter* but they may be sitting in the *arbiter*'s socket buffer or they haven't arrived at the *arbiter* yet when the *arbiter* dies. If clients do not store these transactions, none of the clients or the new *arbiter* will have the information of the transactions after the old *arbiter* dies. As a result, these transactions will simply disappear in the recovered session, resulting in a data lose. These unprocessed transactions do not have sequence numbers. And they do not exist in the *transaction list* of the dead *arbiter*. The data structure used in storing these transactions in clients is named *Buffer*$_{unprocessed}$ (*Figure 4.7(b)*). Every time a client generates a transaction, the transaction is first stored in *Buffer*$_{unprocessed}$. Then, the transaction is sent to the *arbiter* for assignment of a sequence number and for notifying other clients of this transaction. The transaction is kept in *Buffer*$_{unprocessed}$ until the multicast of the transaction is received from the *arbiter*. After receiving the multicast, the transaction is moved to *Buffer*$_{processed}$ with the sequence number assigned by the *arbiter* and with a 2-phase commit protocol status *RECEIVED*.

During the recovery process, clients not only have to send data in *Buffer*$_{processed}$ for recovering the *transaction list* of the dead *arbiter*, but they also have to send data in *Buffer*$_{unprocessed}$ for precluding data lose. Since transactions in *Buffer*$_{unprocessed}$ do not have sequence numbers, clients have to adopt a mechanism to preserve the order in which the transactions are generated so that the *arbiter* knows which comes first upon receiving them in the recovery process. Clients use “fake sequence numbers” to ensure the ordering of the transactions in *Buffer*$_{unprocessed}$ is honoured by the *arbiter*. “Fake sequence numbers” are similar to the sequence numbers used by the *arbiter*. However, “fake sequence numbers” are negative integers. They run from -1 to the maximum allowable negative integer. The oldest transaction in *Buffer*$_{unprocessed}$ will be assigned -1, the next one will be assigned -2, so on so forth. As a result, the ordering of the transactions in a client’s *Buffer*$_{unprocessed}$ is preserved. There is no ordering relationship between transactions in *Buffer*$_{unprocessed}$ of different clients. If there exists an ordering relationship between two transactions A and B, which are generated by different clients, and both of them are not processed by the *arbiter* yet, only the earlier one will appear in *Buffer*$_{unprocessed}$ of one of the clients. Suppose transaction A from client A is generated earlier than transaction B from client B. By assumption, transaction A has not been processed by the *arbiter* yet. Thus, it has not been multicasted to client B. In other words, client B has not committed transaction A yet. Hence, transaction B, which should occur after transaction A, cannot be generated by client B and be put into *Buffer*$_{unprocessed}$ of client B.

Recovery Process

The recovery process includes collecting recovery information (data in *Buffer*$_{processed}$ and *Buffer*$_{unprocessed}$) from all clients, reorganizing the recovery information collected, as well as completing the 2-phase commit protocol for all transactions received. Completing the 2-phase commit protocol for all received transactions prepares a clean new *arbiter* for the recovered session.

As discussed before, clients send data in *Buffer*$_{processed}$ as well as data in *Buffer*$_{unprocessed}$

after establishing connections with the new *arbiter*. Upon receiving a transaction, the new *arbiter* stores it into the *recovery list*. The *recovery list* has identical data structure as the *transaction list* (*Figure 3.6*). It can be viewed as the *transaction list* used in recovery mode. The 2-phase commit protocol status used in the *recovery list* are exactly the same as those used on the client side with the additional of 2 new statuses:

1. *NA*

   - the default status.

2. *SENT*

   - the status for transactions from *Buffer*$_{unprocessed}$.

In storing a transaction with a sequence number (from *Buffer*$_{processed}$) into the *recovery list*, the *arbiter* first checks for the existence of that transaction in the list. A record for that transaction may have already been created by another client. If no record for the transaction is found, the *arbiter* stores the sequence number as well as the 2-phase commit protocol status sent by the client into a new record and appends the record to the end of the *recovery list*. Otherwise, the *arbiter* extracts the 2-phase commit protocol status in the data received and stores it into the status field corresponding to the sender of the data in the record found. Transactions without sequence numbers (from *Buffer*$_{unprocessed}$) from clients are also stored in the *recovery list*. However, no searching of records is required before storing these into the *recovery list*. This is because the sender of a transaction in *Buffer*$_{unprocessed}$ is the only client which carries information of that transaction. Thus, a record is always created for a transaction without sequence number. And the record will be appended to the *recovery list*. The *arbiter* will assign *SENT* as the 2-phase commit protocol status for transactions received without sequence numbers.

After getting recovery information from all clients, the *arbiter* starts organizing and processing the information received. It organizes the information by sorting the *recovery list* in ascending order of sequence number. Then, the *arbiter* starts processing transactions with positive sequence numbers in ascending order of sequence number, followed by processing those with negative sequence numbers in descending order of sequence number. Processing of transactions mainly involves carrying out 2-phase commit protocol for the transactions. Transactions in the *recovery list* are processed sequentially during recovery. That is, unlike in normal service mode, the *arbiter* does not start processing the next transaction in the *recovery list* until it finishes the entire 2-phase commit protocol for the current one. The way that the *arbiter* processes a transaction in the *recovery list* is based on the combination of the 2-phase commit protocol status of all clients. *Table 4.1* presents all possible combinations of the 2-phase commit protocol status of clients, the causes of the combinations, and the ways the *arbiter* processes the transactions.

Table 4.1: Processing of *Recovery List* in Recovery Process

Case	**Characteristics of Combination**	**Cause of Combination**	*Arbiter*'**s Processing**
1	1 *SENT*, all other *NAs*	*Arbiter* dies before receiving the transaction. The transaction has a negative sequence number.	Assign a sequence number to the transaction. If this is the first transaction with negative sequence number, the sequence number to be assigned is the sequence number of the last processed positive-sequence-number transaction + 1. Otherwise, the sequence number to be assigned is the sequence number of the last processed transaction + 1. Then, carry out 2-phase commit protocol for the transaction by first multicasting the transaction to all clients.
2	>= 1 *RECEIVED*, all other *NAs*	*Arbiter* dies in the middle of multicasting the transaction. The originator and some other clients has been sent the multicast.	Send the transaction to *NA* clients and continue with 2-phase commit protocol (wait for "ready-to-commit" from all clients).
3	>= 1 *RCOMMIT*, all other *NAs*	*Arbiter* dies in the middle of multicasting the transaction and the clients which has been sent the multicast are already "ready-to-commit" the transaction.	Send the transaction to *NA* clients and continue with 2-phase commit protocol (wait for "ready-to-commit" from *NA* clients).
4	1 *SENT*, >= 1 *RECEIVED*, all other *NAs*.	*Arbiter* dies in the middle of multicasting the transaction. And originator of the transaction has not been sent the multicast.	Send the transaction to *SENT* and *NA* clients and continue with 2-phase commit protocol (wait for "ready-to-commit" from all client).
5	1 *SENT*, >= 0 *RECEIVED*, >= 1 *RCOMMIT*, all other *NAs*.	*Arbiter* dies in the middle of multicasting the transaction. The originator of the transaction has not been sent the multicast. And some clients which . have been sent the multicast are "ready-to-commit" the transaction.	Send the transaction to *SENT* and *NA* clients and continue with 2-phase commit protocol (wait for "ready-to-commit" from *NA*, *SENT*, *RECEIVED* clients).

Case	Characteristics of Combination	Cause of Combination	Processing
6	>= 1 *RECEIVED*, >= 1 *RCOMMIT*, all other *NAs*.	*Arbiter* dies in the middle of multicasting the transaction. Originator and some other clients have been sent the multicast. Some of these clients are "ready-to-commit" the transaction.	Send the transaction to *NA* clients and continue with 2-phase commit protocol (wait for "ready-to-commit" from *NA*, *RECEIVED*, clients).
7	>= 1 *RCOMMIT*, >= 1 *COMMITTED*.	*Arbiter* dies in the middle of multicasting "commit". Some clients which have been sent "commit" have already committed the transaction.	Send "commit" for the corresponding transaction to *RCOMMIT* clients.
8	All *COMMITTED*.	The transaction is committed by all clients. However, the transaction has not been *flushed* yet.	Proceed to process next transaction in the *recovery list*.

After sending data in *Buffer*$_{processed}$ and in *Buffer*$_{unprocessed}$ to a new *arbiter*, a client waits for data from the *arbiter* to complete the recovery process. As seen from *Table 4.1*, data from the *arbiter* is the data used in carrying out 2-phase commit protocol. Thus, the client behaves normally as if the new *arbiter* is already in service mode. The client sends appropriate 2-phase commit protocol responses to the *arbiter* upon receipt of data from the *arbiter*. The only difference between clients' processing in recovery mode and that in service mode is that the clients are not generating new transactions in recovery mode since interactivity is temporarily suspended. Clients only receive new transactions from the *arbiter* if any exist.

The recovery process is completed after the *arbiter*, with the cooperation of clients, finishes 2-phase commit protocol for every transaction in the sorted *recovery list*. Interactivity of clients will be resumed after the completion of the recovery process, and the *arbiter* will begin servicing clients.

4.4 Interactive Applications Enhanced

Interactive applications demand quick response time. The response time of a transaction refers to the time required for the effect of the transaction to take place after the transaction is generated. In **CTDS**, the time spent on communications between the *arbiter* and the clients introduced by 2-phase commit protocol lengthens the response time of a transaction significantly. Because of the 2-phase commit protocol, a transaction experiences a total of two round-trip time before it is committed (*Figure 4.8*).

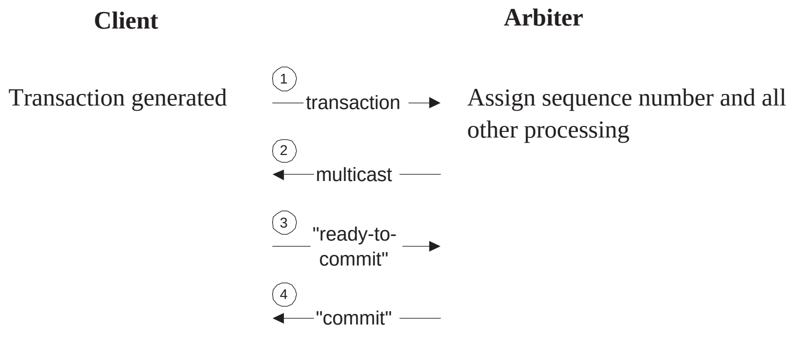


Figure 4.8: Two Round-trip Time as a Result of Using 2-phase Commit Protocol

Unfortunately, there is no way to reduce the additional response time introduced by the 2-phase commit protocol since the communications are necessary to implement the 2-phase commit protocol, and the 2-phase commit protocol is required to ensure atomic ordering. On the contrary, the way that clients send transactions to the *arbiter*, which can adversely affect the response time of client applications, is improvable. Suppose a client sends transactions to the *arbiter* immediately after they are generated. *Figure 4.9* illustrates the adverse effect on transactions' response time if clients send transactions to *arbiter* right after they are generated.

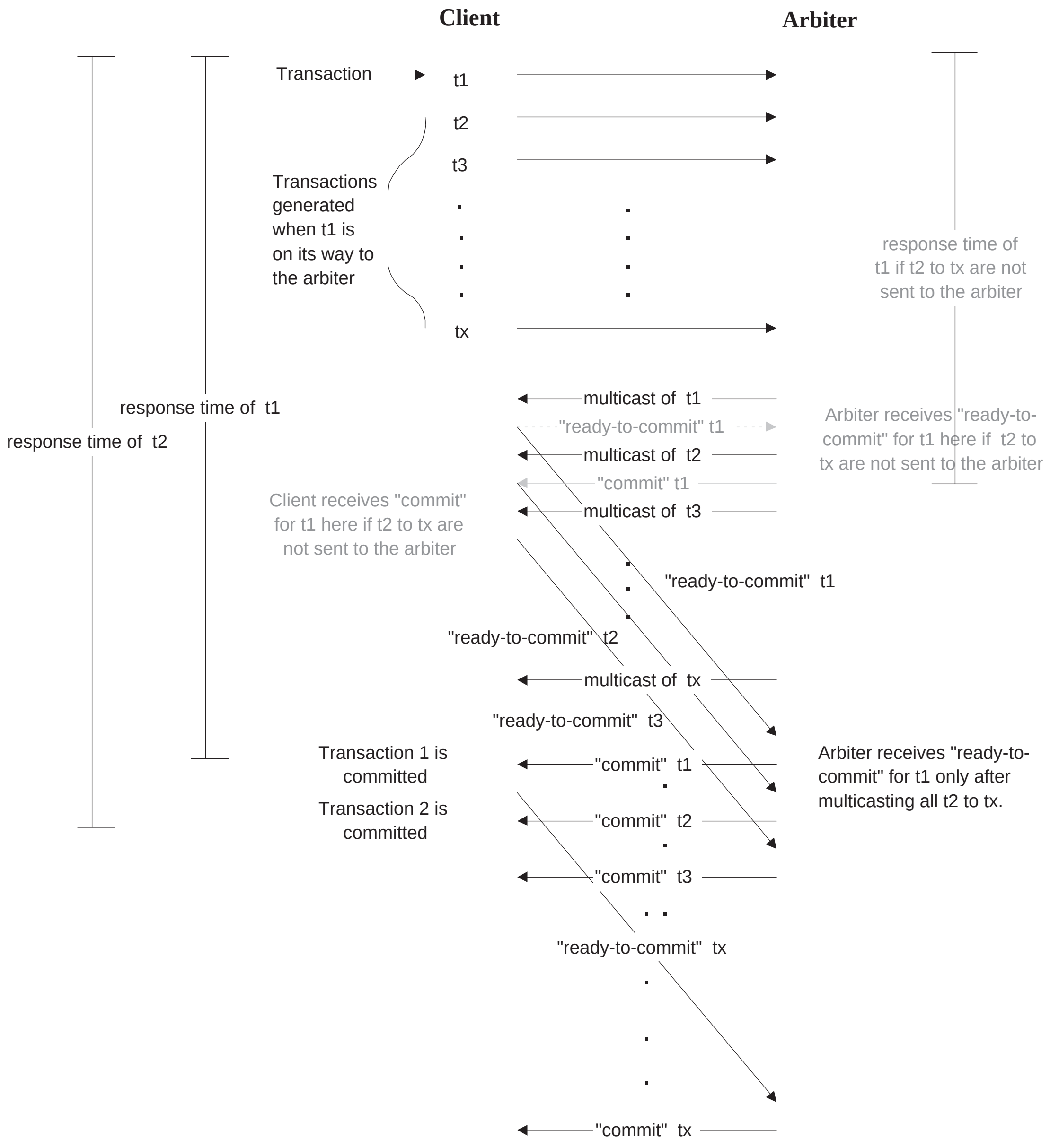

Client
Arbiter
Transaction
t1
t2
t3
tx
Transactions generated when t1 is on its way to the arbiter
response time of t1 if t2 to tx are not sent to the arbiter
multicast of t1
"ready-to-commit" t1
multicast of t2
"commit" t1
multicast of t3
response time of t1
response time of t2
Arbiter receives "ready-to-commit" for t1 here if t2 to tx are not sent to the arbiter
Client receives "commit" for t1 here if t2 to tx are not sent to the arbiter
"ready-to-commit" t1
"ready-to-commit" t2
multicast of tx
"ready-to-commit" t3
Transaction 1 is committed
"commit" t1
Arbiter receives "ready-to-commit" for t1 only after multicasting all t2 to tx.
Transaction 2 is committed
"commit" t2
"commit" t3
"ready-to-commit" tx
"commit" tx

Figure 4.9: Adverse Effect on Response Time If Generated Transactions are sent Immediately

Because the communication time required for data to be exchanged between a client and the *arbiter* is significantly larger than that required to generate a transaction, many transactions can be generated (`t2` to `tx`) and be sent to the *arbiter* before the *arbiter* receives the first transaction (`t1`). Upon receiving `t1`, the *arbiter* multicasts `t1` to all clients after assigning a sequence number to it. Assuming all clients have committed all previous transactions, clients respond "ready-to-commit" `t1` right after receiving the multicast. However, the clients do not receive "commit" `t1` as the next piece of data from the *arbiter* even all clients have sent "ready-to-commit" `t1` to the *arbiter*. This is because the "ready-to-commit" `t1` responses from all clients are not processed by the *arbiter* until the *arbiter* has finished multicasting `t2` to `tx`. After multicasting `t2` to `tx`, the *arbiter* will multicast "commit" for `t1`. Then, `t1` can be committed by all clients. On the contrary, if transactions `t2` to `tx` were not sent to the *arbiter* yet, `t1` will be committed earlier since the time for the *arbiter* to multicast `t2` to `tx` is saved (*Figure 4.9* shows this scenario in gray). This kind of scenario happens very frequently in interactive applications since transactions are generated at a high rate.

In **CTDS**, this adverse effect on response time is reduced by synchronizing the rate at which transactions are generated and are sent to the *arbiter* with the rate at which transactions are committed on the client side. Recall that everytime a transaction is generated, it is stored into *Buffer*$_{unprocessed}$. Instead of sending transactions to the *arbiter* right after storing them into *Buffer*$_{unprocessed}$, transactions generated are simply stored into *Buffer*$_{unprocessed}$ without being sent to the *arbiter* until the previous transaction is *committed*$_{client}$. When a transaction is committed by the client application, the oldest transaction in *Buffer*$_{unprocessed}$ will be sent to the *arbiter*. Thus, the response time of transaction `t1` will not be affected by the transactions `t2` to `tx`, which are generated while `t1` is being sent to the *arbiter*.

To further improve transactions' response time, **CTDS** combines adjacent transactions of the same kind in *Buffer*$_{unprocessed}$ into a single transaction. Application programmers define rules to combine transactions. For each piece of data in every kind of transactions, application programmers define a way to combine it with the same piece of data in an adjacent transaction of the same kind. Currently, two ways of combining data are supported by **CTDS**:

1. *Overwrite*

   - Data is combined by replacing the data in the older transaction with that in the later one.

2. *Add*

   - Data is combined by adding the data in the older transaction and that in the later one. Currently, only additions of real numbers are supported.

Application programmers also specify whether a particular type of transaction is combinable or not. If a type of transaction is combinable, the corresponding combining rule will be applied to that type of transaction. Otherwise, adjacent transactions of that type in *Buffer*$_{unprocessed}$ will not be combined into one transaction. *Figure 4.10* gives an example on combining transactions in *Buffer*$_{unprocessed}$ in the **Robot** application.

Due to the combination of the first, the second, and the third *Body Rotation* transactions, response time of the second and the third transactions is improved. This is because the effects of these two transactions now happen earlier after being integrated with the first transaction. The combination of transactions also lowers the probability of overflowing *Buffer*$_{unprocessed}$ on the client side and that of overflowing the *transaction list* on the arbiter side. The only disadvantage of the combination of transactions is that large state changes may happen to the shared model. For instance, the combination of *Body Rotation* transactions in *Figure 4.10* can result in a jaggy rotation of the robot. This is because rotations of the robot suddenly

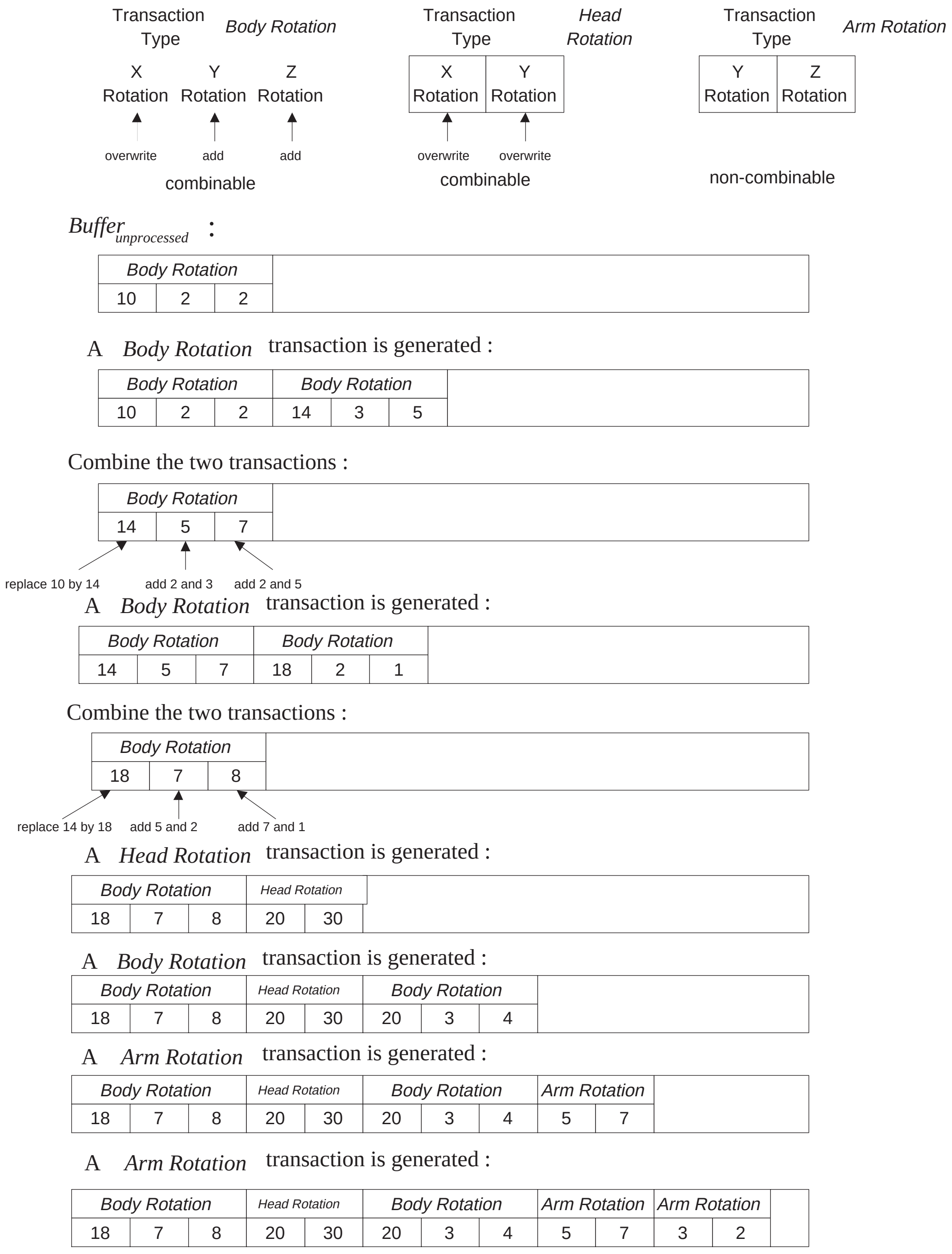

Transaction Type
Body Rotation
X Rotation
Y Rotation
Z Rotation
overwrite
add
add
combinable
Transaction Type
Head Rotation
X Rotation
Y Rotation
overwrite
overwrite
combinable
Transaction Type
Arm Rotation
Y Rotation
Z Rotation
non-combinable
Buffer unprocessed :
Body Rotation
10 2 2
A Body Rotation transaction is generated :
Body Rotation | Body Rotation
10 2 2 14 3 5
Combine the two transactions :
Body Rotation
14 5 7
replace 10 by 14
add 2 and 3
add 2 and 5
A Body Rotation transaction is generated :
Body Rotation | Body Rotation
14 5 7 18 2 1
Combine the two transactions :
Body Rotation
18 7 8
replace 14 by 18
add 5 and 2
add 7 and 1
A Head Rotation transaction is generated :
Body Rotation | Head Rotation
18 7 8 20 30
A Body Rotation transaction is generated :
Body Rotation | Head Rotation | Body Rotation
18 7 8 20 30 20 3 4
A Arm Rotation transaction is generated :
Body Rotation | Head Rotation | Body Rotation | Arm Rotation
18 7 8 20 30 20 3 4 5 7
A Arm Rotation transaction is generated :
Body Rotation | Head Rotation | Body Rotation | Arm Rotation | Arm Rotation
18 7 8 20 30 20 3 4 5 7 3 2

Figure 4.10: An Example of Combining Transactions in *Buffer*$_{unprocessed}$

jump from (2,2) to (7,8) in the Y, Z directions.

Chapter 5

Related Work

This chapter presents a number of projects developed to provide application-sharing environments. Some of the projects handle fault-tolerance. However, they are not as robust as **CTDS** in the fault-tolerant aspects. Most of the systems built only allow one master per application. Some systems provide more generic application sharing environments in which more than one application can be shared.

As discussed in Section 2.5.2, the usage of a *PSEUDO server architecture* is an easy way to provide application sharing without requiring modifications to stand-alone applications. Most of the researches done in the field use *PSEUDO server architecture* for sharing X applications. *XTV* by Abdel-Wahab *et al.* [AWF91], *COMIX* by Babadi [Bab93], and *Share* by Greenberg [Gre90] are systems based on the *PSEUDO server architecture* for sharing X applications. The PSEUDO servers in these three systems are made up of several processes. In *XTV*, some of these processes run both locally and remotely, whereas others run only remotely. All processes in *COMIX* and *Share* only have one running instance. Nevertheless, the processes making up the PSEUDO server in the corresponding systems provide similar functionality. Each of the processes making up the PSEUDO server has its own responsibility in the system. For instance, the packet translator process in *XTV*, the comix server in *COMIX*, and the view manager in *Share* all handle the X requests and replies translations; the token manager in *XTV*, the comix-control process in *COMIX*, and the chair manager in *Share* regulate access to applications. All three systems allow only one master per application. However, *XTV* supports more than one application in a session. It is unclear as to whether *COMIX* and *Share* support multiple applications in a session. Abdel-Wahab *et al.* go into details of the translation of X resources IDs, X client requests, and X server replies. They also mention the possible failure of applications due to failures of key centralized processes in the system (processes running on a single machine). A replicated approach, i.e. running key processes in the system on multiple machines, is suggested as a possible solution. No recovery of key processes is supported by all *XTV*, *COMIX*, and *Share*. Ahuja *et al.* explore a couple of variants of the *PSEUDO server architecture* by having the shared application running on a different number of machines [AEL90]. One variant is the *single-site* approach. In this approach, the shared application is run on the machine of one of the participants. The other variant is named *multi-site* approach. The shared application is run on several participants' machines. An analysis is done on the pros and cons of the two variants. The performance of the two different variants is also discussed.

Studies such as the *Amoeba* distributed operating system [MVRT$^+$90, TVRVS$^+$90, KT92] and the telemedicine system by Gomez *et al.* [GdPA$^+$96] use the *sequencer-based model* for group communication. In *Amoeba*, all participants in a session form a group. Group communication is achieved by having all participants to send messages/transactions to the sequencer in the corresponding group. The sequencer then assigns a sequence number to the message before it multicasts the message to the group. The sequencer in *Amoeba* is integrated into the kernel, but it is not the application itself. An election for a new sequencer will be called if the current sequencer fails [CDT94, KT92]. [GdPA$^+$96] built a telemedicine system for remote cooperative medical imaging diagnosis. In the telemedicine system, collaborative toolkits, which provide coordination service, are integrated into applications. There are also group communication modules being integrated into applications. However, no details is given on the data flow in the system. Thus, it is unclear as to whether a single application instance acts as the sequencer or multiple application instances cooperate to do the sequencing job. One major weakness of this telemedicine system is the required modifications to the interface of stand-alone applications. Modifications to interface of applications may lead to inefficient collaboration due to unfamiliarity of the new interface.

An event-capturing mechanism is proposed by Hao *et al.* [HLJ96, HJ96] to provide application sharing. Instead of intercepting traffic between X clients and X servers as in the *PSEUDO server architecture*, the system developed (*RES-AP*) captures relevant input events on a shared window. *RES-AP* then orders and groups the input events before sending them to other application instances. The authors claim that capturing and processing of input events reduce the communication traffic as compared to the *PSEUDO server architecture*.

A number of papers discuss generic collaborative tool development environments. The generic environment allows sharing of more than one application. Participants can invoke any X applications, which they want to share, in the environment and collaborate with other participants. Maly *et al.* [MAWO$^+$97] present a Interactive Remote Instruction System for interactive distance learning. The system provides a virtual classroom for geographically dispersed students. The architecture of the system is a combination of the PSEUDO X server architecture and the client-server architecture. There are several servers which provide specific types of services, e.g. class information service, multicast service. In addition, the system also incorporates *XTV* [AWF91], an X Windows tool-sharing engine, which is *PSEUDO server architecture* based. Fault-tolerant service is also addressed by the system. However, the system only handles application failures. It does not handle server failures. In contrast with **CTDS**, this system protects server processes from crashing if application fails. Applications are restarted after failure. Another multi-application sharing environment is discussed in [JJ96]. This environment is different from the one developed by Maly *et al.* in the sense that the shared model is made up by joining 3-D objects from different applications, in contrast to each application has its own shared model. This shared 3-D environment model is targeted for existing applications which use a scene graph model for display geometry, e.g. Open Inventor applications. *DEEDS*, a prototype distributed multitasking environment, developed by Liang *et al.* [LLC+94], uses a 3-layer architecture. *DEEDS* consists of a groupware server that possesses similar functionality as the *arbiter* in **CTDS**. The groupware server provides the coordination and communication services to applications. There is an application server for each shared application to take care of coordination and other needs associated with the execution of the application. For instance, partitioning of the 2-D workspace in a 2-D paint program into several resources and the access to each of the resources are handled by the application server associated with the 2-D paint program. In **CTDS**, an application server is associated with each application instance. The application server in DEEDS sits in between the groupware server and the application.

Chapter 6

Conclusions and Future Work

This thesis presents a system for developing collaborative tools (multi-user applications). The system is named **CTDS**. It facilitates the development of collaborative tools by providing the communication and coordination services, which are required in multi-user applications, to collaborative-application programmers. Application programmers not only can use **CTDS** to develop multi-user applications from scratch, but they can also use **CTDS** to easily convert stand-alone applications (single-user applications) to collaborative ones. In converting a stand-alone application to a collaborative one using **CTDS**, application programmers have to modify the source code of the stand-alone application. However, changes to the stand-alone application are limited to a few function calls. **CTDS** offers a library of functions to client applications for integration with the system.

CTDS employs a client-server architecture as the model. **CTDS** consists of two main components: *arbiter* and application server. The *arbiter* is the server which provides the communication and coordination services. The application server together with the modified application form the client. There is an application server associated with every instance of the shared application. It provides the additional Graphical User Interface required for session management and gives information of the session. Thus, the interface of the shared application remains intact. **CTDS** provides total and atomic ordering of transactions by usage of sequence numbers (assigned by the *arbiter*) and two-phase commit protocol respectively.

CTDS is designed to target at satisfying the two main fault-tolerance requirements on collaborative tool development systems. Failure of any of the participants and failure of the *arbiter* are not going to affect the continuity of a session. Participants can leave or fail any time during a session with a notice being sent to all other participants by **CTDS**. **CTDS** recovers the *arbiter* of a session by restarting a new *arbiter* and recovering the state of the dead *arbiter*. The recovery process of an *arbiter* is achieved by having clients to monitor the *arbiter* and by having clients to store information which allows the new *arbiter* to reconstruct the state of the dead *arbiter* completely. In addition to the fault-tolerant features, **CTDS** offers other attractive features which are desired in the development of most collaborative applications. With the capability of handling multiple collaborative resources, **CTDS** supports multiple masters in a session without requiring "soft protocol" (coordination between participants on interactions on the shared model). This results in more efficient and more effective collaboration. Moreover, **CTDS** is enhanced for building interactive collaborative tools. Because of the relatively long time required to commit a transaction using 2-phase commit protocol, the rate at which transactions are generated is much higher than the rate at which transactions are committed in interactive applications. It is found from experiments that the higher the rate at which transactions are generated, the longer the response time of transactions. **CTDS** employs two strategies to improve the response time of transactions in interactive applications. Firstly, it holds onto generated transactions without sending them to the *arbiter* until the previous transaction being sent is committed. Secondly, **CTDS** combines transactions of the same kind that are being held onto.

There are still a number of improvements that can be made to **CTDS**. **CTDS** currently treats every participant equally. In other words, every participant has the same level of access to resources. Every master can interact with every resource. This may be inadequate in some situations. For instance, in a session with both instructors and students as participants, it is undesirable to grant write (interaction) permissions on some parts of the shared model to students. **CTDS** can be modified to take a user level and a password when a participant joins a session. Based on the user level, **CTDS** checks the given password against the password associated with the given user level. If the given password is correct, the participant is allowed to join the session. And **CTDS** also determines the level of access to each resource based on the user level. Some participants may not be able to become masters on certain subsets of the resources. The provision of different levels of access to resources requires application programmers to specify all user levels with their corresponding levels of access on each resource.

The transfer of the shared model when a new participant joins a session can be automated. **CTDS** currently requires one of the participants in a session to manually save and transfer the shared model to the new participant before resuming interactivity of the session. To make the transfer of the shared model to the new participant automatic, one of the participants have to send the shared model to the *arbiter*. And the *arbiter* then forwards the shared model to the new participant. The *arbiter* is involved in the process because there does not exist any connection between clients in **CTDS**. Only the *arbiter* can communicate with clients. Clients cannot communicate with each other. Transfer of the shared model from one of the participants to the *arbiter* is necessary because the *arbiter* does not carry any information of the shared model. The *arbiter* only stores the transactions being applied to the shared model. Thus, the state of the shared model has to be transferred from one of the participants to the *arbiter* before the *arbiter* can send the shared model to the new participant.

Bibliography

[AEL90] S.R. Ahuja, J.R. Ensor, and S.E. Lucco. A comparison of application sharing mechanisms in real-time desktop conferencing systems. *Sigois Bulletin*, 11(2 and 3):238–248, 1990.

[AWF91] H.M. Abdel-Wahab and M.A. Feit. Xtv: A framework for sharing x window clients in remote synchronous collaboration. In *Proceedings of the IEEE Conference on Communications Software: Communications for Distributed Applications and Systems (TRICOMM) 1991*, pages 159–167, Chapel Hill, NC, 1991.

[AWGN88] H.M. Abdel-Wahab, S.U. Guan, and J. Nievergelt. Shared workspaces for group collaboration: An experiment using internet and unix interprocess communications. *IEEE Communications*, 26(11):10–16, November 1988.

[Bab93] A. Babadi. Comix: A tool to share x applications. In *Proceedings of the Seconf Workshop on Enabling Technologies for Collablorative Enterprises*, pages 192–196, Morgantown, WV, 1993.

[BRPS94] R. Bentley, T. Rodden, Sawyer P., and I. Sommerville. Architectural support for cooperative multiuser interfaces. *Computer*, 27(5):37–45, May 1994.

[CDT94] G. Coulouris, J. Dollimore, and Kindberg T. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishers Ltd., 1994.

[CKT91] E. Chang, R. Kasperski, and Copping T. Group coordination in participant systems. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 589–599, Kauai, HI, 1991.

[CVB92] M.S. Chen, H.M. Vin, and T. Barzilai. Designing a distributed collaborative environment. In *Global Telecommunications Conference. Conference Record., GLOBECOM'92. Communication for Global Users.*, volume 1, pages 213–219, Orlando, FL, 1992.

[GdPA$^+$96] E.J. Gomez, F. del Pozo, M.T. Arredondo, H. Rahms, M. Sanz, and P. Cano. A telemedicine system for remote cooperative medical imaging diagnosis. *Computer Methods and Programs in Biomedicine*, 49(1):37–48, January 1996.

[Gre90] S. Greenberg. Sharing views and interactions with single-user applications. *SIGOIS Bulletin*, 11(2 and 3):227–237, 1990.

[Gre91] S. Greenberg. Computer-supported cooperative work and groupware: An introduction to the special issues. *International journal of man-machine studies*, 34(2):133–141, February 1991.

[HJ96] M.C. Hao and Sventek J.S. Collaborative design using your favourite 3d application. *HP Laboratories Technical Report*, 96(51), April 1996.

[HLJ96] M.C. Hao, D. Lee, and Sventek J.S. A light-weight application sharing infrastructure for graphics intensive applications. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, pages 127–131, 1996.

[JJ96] B. Jasnoch, U.and Anderson and H. Joseph. Shared 3-d environments within a virtual prototyping environment. In *Proceedings of the WET ICE '96. IEEE 5th Workshop on Enabling Technologies*, pages 274–279, Stanford, CA, 1996.

[KC95] M. Ko and P. Cahoon. A shared 4-d workspace. *The University of British Columbia Department of Computer Science Technical Report*, 95(19), August 1995.

[Kis96] O. Kiselyov. Handling multiple tcp connections in c++. *C/C++ Users Journal*, 14(5):17–23, May 1996.

[KT92] M.F. Kaashoek and A.S. Tanenbaum. Efficient reliable group communication for distributed systems. *Part of Ph.D. Thesis "Group Communication in Distributed Computer Systems, Vrije Universiteit, Amsterdam*, 1992.

[LLC$^+$94] T.P. Liang, H. Lai, N.S. Chen, H. Wei, and M.C. Chen. When client/server isn't enough: Coordinating multiple distributed tasks. *Computer*, 27(5):73–79, May 1994.

[MAWO$^+$97] K. Maly, H. Abdel-Wahab, C.M. Overstreet, J.C. Wild, A.K. Gupta, A. Youssef, E. Stoica, and E.S. Al-Shaer. Interactive distance learning over intranets. *IEEE Internet Computing*, 1(1):60–71, January and February 1997.

[MVRT$^+$90] S.J. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[RCHS97] I. Rhee, S.Y. Cheung, P.W. Hutto, and V.S. Sunderam. Group communication support for distributed collaboration systems. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 43–50, Baltimore, MD, 1997.

[Ste90] W. R. Stevens. *UNIX Network Programming*. Prentice Hall PTR, Englewood Cliffs, New Jersey, 1990.

[TBE$^+$94] I. Tou, S. Berson, G. Estrin, Y. Eterovic, and E. Wu. Prototyping synchronous group applications. *Computer*, 27(5):48–56, May 1994.

[TBK$^+$96] A. Thiel, J. Bernarding, M. Krauss, S. Schulz, and T. Tolxdorff. Distributed medical services within the atm-based berlin regional testbed. In *Proceedings of the SPIE. The International Society for Optical Engineering V2711. Society of Photo Optical Instrumentation*, pages 32–43, 1996.

[TVRVS$^+$90] A.S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. Van Rossum. Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, December 1990.

Appendix A

CTDS User Guide

This appendix serves as a guide on the integration of an application with **CTDS**. Application programmers are referred to different files which come with **CTDS** in the discussion. Application programmers are advised to look into the suggested files to get a better understanding on the usage of **CTDS**.

A.1 Header Files

There are four header files in **CTDS** which should be included into an application:

1. *client.h*
2. *client_const.h*
3. *client_state.h*
4. *const.h*

Both *client_const.h* and *client.h* should be modified to suit the needs of an application. These two header files contain definitions of constants which are used by **CTDS**. There are **THREE** constants in these two header files which **MUST** be modified for an application:

- *ARBITER_PATH_NAME (const.h)*

  - the directory in which the **CTDS** *arbiter* executable is.

- *RESOURCES (client_const.h)*

  - the number of resources in the application.

- *NO_XSACTIONS (client_const.h)*

  - the number of types/kinds of transactions in the application.

A.2 Modules

Since **CTDS** is an application-independent system, application programmers have to write a number of application-specific modules for **CTDS** to work with a particular application.

A.2.1 Xsaction_types

Xsaction_types.h

This header file should contain all possible types of transactions in the application as well as the definitions of the structure of the data parts in the types of transactions. Every transaction generated by an application in **CTDS** has the structure shown in *Figure A.1*. The transaction structure is called Xsaction.

For every transaction type, the transaction data part is a record composing of a number of fields, each of which stores a piece of information in that type of transaction.

The following example illustrates the organization of *Xsaction_types.h*. Refer to *Xsaction_types.h*, which is coded for **Robot** application, in the **CTDS** package.

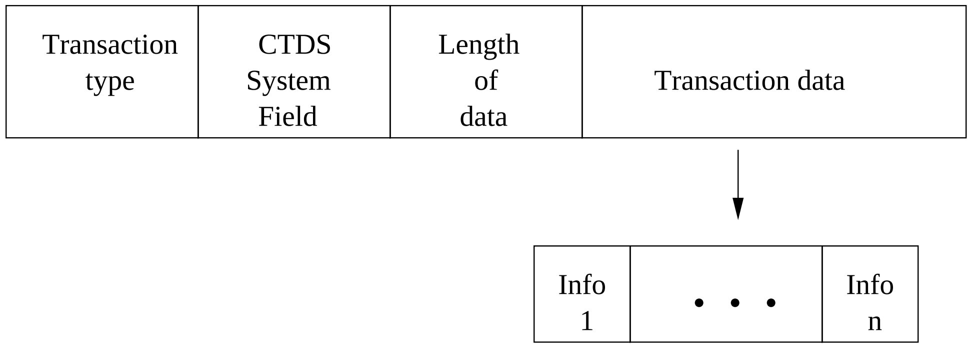


Figure A.1: Structure of a Transaction in **CTDS**

```
// All possible types of transactions
typedef enum
{
   type_1,/* application-dependent
   type_2,
   type_3,*/
   .
   .
   .
} XsactionType;

// Record structure for transaction data of transaction type_1
typedef struct
{
   type_1_field_1_type field_1; /* application-dependent
   type_1_field_2_type field_2; */
} type_1_data;

// Record structure for transaction data of transaction type_2
typedef struct
{
   type_2_field_1_type field_1; // application-dependent
} type_2_data;

// No transaction data of transaction type_3
```

Xsaction_types.cpp

This file is optional. However, application programmers are suggested to put functions which initialize the transaction data structures defined in *Xsaction_types.h*. These initialization functions can be used later when **CTDS** reads transactions (Section A.2.3). *Xsaction_types.cpp* in the **CTDS** package contains initialization functions for **Robot** application.

A.2.2 event_dispatcher

event_dispatcher.h

This header file should not be modified by application programmers. It contains the prototypes of two application-dependent functions used by **CTDS**.

event_dispatcher.cpp

Definitions of two application-dependent functions are in this file. Application programmers are required to give the definitions of these two functions. The two functions are named dispatch_Xsaction and redraw_scene. dispatch_Xsaction tells **CTDS** how to carry out transactions upon receiving "COMMIT" from the *arbiter*. And redraw_scene tells **CTDS** how to refresh the application window. There is a *event_dispatcher.cpp* in the **CTDS** package which is coded for **Robot** application. The following piece of code is a template for the function dispatch_Xsaction. Note that the number of arguments to the functions and the arguments should not be modified. They should be kept the same as those in *event_dispatcher.cpp* in the **CTDS** package.

```
void dispatch_Xsaction(struct Xsaction *t)
// Transaction to be committed is stored in the t
{
   // Define behaviour on every possible transaction type
   switch(t->type)
   {
      case type_1: //  Transaction type is application-dependent

         /* Carry out the transaction of type_1 in an
            application-specific way */
         process_type_1(...);
         break;

      case type_2: // Transaction type is application-dependent

         /* Carry out the transaction of type_2 in an
            application-specific way */
         process_type_2(...);
         break;

      /* Do the same thing for every transaction type */
   }   /* End switch */
}
```

A.2.3 app_Xsaction

app_Xsaction.h

This header file should not be modified by application programmers. It contains the prototype of an application-dependent function used by **CTDS**.

app_Xsaction.cpp

This file contains the definition of a function which tells **CTDS** how to read transactions defined for an application. The function is named read_Xsaction. *app_Xsaction.cpp* in the **CTDS** package is coded for **Robot** application. The following piece of code is a template for the function read_Xsaction. Note that the number of arguments to the function and the arguments should not be modified.

```
int read_Xsaction(int sockfd, struct Xsaction *t)
// Transaction is to be read from socket sockfd to t
{
   int amt_read = 0; /* Total number of bytes read from
```

```
                                sockfd */
int n; // Temporary integer

// Define way to read every type of transaction
switch(t->type)
{
   struct type_1_data *data1;        /* Structures for storing
                                        transaction
   struct type_2_data *data2;           data for every kind of
   .                                    transactions --
   .                                    application-dependent */
   .

   case type_1:

      data1 = new type_1_data;       /* Allocate space for data of
                                        transaction type_1 */

      /* Initialize data1 using the initialization function
         defined in Xsaction_types.cpp */

      t->dynamic = 1;                /* REQUIRED for freeing of
                                        the space dynamically
                                        allocated to data1 */
      // Read data from sockfd to data1 with error checking
      if ((n = smart_read(sockfd, (char *)data1, t->length)) == 0)
         return 0;

      // Error checking
      if (n < 0)
         return n;

      // Increment the number of bytes read from sockfd
      amt_read += n;

      t->data = data1;               /* REQUIRED for storing
                                        the data read to the
                                        transaction structure
                                     */
      break;

   case type_2:

      data2 = new type_2_data;       /* Allocate space for data of
                                        transaction type_2 */

      /* Initialize data1 using the initialization function
         defined in Xsaction_types.cpp */
      t->dynamic = 1;                /* REQUIRED for freeing
                                        of the space
                                        dynamically allocated
                                        to data2 */

      // Read data from sockfd to data1 with error checking
      if ((n = smart_read(sockfd, (char *)data2, t->length)) == 0)
         return 0;

      // Error checking
      if (n < 0)
         return n;

      // Increment the number of bytes read from sockfd
      amt_read += n;

      t->data = data2;               /* REQUIRED for storing
                                        the data read to the
                                        transaction structure
                                      */
      break;

   case type_3:

      /* No transaction data for transaction type type_3 =>
```

```
            nothing to be read */
         t->dynamic = 0;                // No dynamic allocation
         t->data = NULL;                // No transaction data
         break;

      /* Do the same thing for every type of transaction */
   } /* End switch */
   return amt_read;                     /* Return number of bytes
                                           read */
}
```

Note: *smart_read(socket, buffer, size)* is a function provided by **CTDS** to read data of *size* bytes from *socket* to *buffer*. It is similar to the function *read* in UNIX. However, *smart_read* will not return until it has read *size* bytes from *socket* even if *size* is bigger than the TCP segment size (usually 1460 bytes). Besides, *smart_read* is capable of resuming reading after signal interruption.

A.3 The Application

In addition to defining modules required by **CTDS**, application programmers also need to modify the application itself to work with **CTDS**. Application programmers are advised to refer to *robot.cpp* in the **CTDS** package for details on the integration of an application with **CTDS**.

A.3.1 Setting Up a Session

In order to establish or join a session, a number of function calls has to be placed at the beginning of the application. The following list contains the functions which are necessary in establishing or joining a session. These functions must be put into the application in exactly the same order as in this list.

1. define_Xsaction_acc_rules()

   - This function defines the rules for combining transactions in *Buffer*$_{unprocessed}$. Application programmers are required to define this function as the combining of transactions are application-specific (Section A.3.4).

2. session_init(int argc, char *argv[])

   - This function initializes all state variables in the session. It takes two arguments. The first one is the number of mainline arguments to the application program. The second one is an array of string which contains the mainline arguments to the application program.

3. setup_app_server()

   - This function sets up the application server associated with the application instance.

4. connect_to_arbiter()

   - This function connects the application instance to the *arbiter* if an *arbiter* exists in the session. Otherwise, it creates an *arbiter* for the session and connects the application instance to it.

A.3.2 Communications with Application Server and *Arbiter*

To communicate with the *arbiter* and the associated application server, the application should make use of two functions provided by **CTDS**:

- process_sockets()

  - This function checks whether there is data from the application server and the *arbiter*. It reads and processes the data if any.

- send_Xsaction(struct Xsaction *t)

  - This function sends a transaction generated to the *arbiter*.

A.3.3 Multiple Resources

After defining the number of resources in *client_const.h*, application programmers need to associate resource IDs to resources in the application code. This is done by a number of #define statements. For instance, #define BODY 1 associates the resource ID 1 to the body of the robot in **Robot** application. Resource IDs must start from 0 to *RESOURCES* (in *client_const.h*) - 1.

In order for the application to access information of the resources, the application needs to include the following statement in the code as a global variable:

extern my_resources[RESOURCES];

A.3.4 Transaction Accumulation/Combination

CTDS is enhanced for interactive applications by combining adjacent transactions of the same type into a single transaction. Since different applications have different kinds of transactions, and hence, different methods to combine transactions, **CTDS** requires application programmers to define rules to combine each type of transactions. **CTDS** requires a function named define_Xsaction_acc_rules to be defined. An array of rule-storage structures is offered to an application. Each element in the array should contain the rules for combining all data fields in the corresponding type of transaction. This array is called xar. *Figure A.2* shows the rule-storage structure used to store a combination rule. This structure is defined in *client.h* and is called Xsaction_acc_rule.

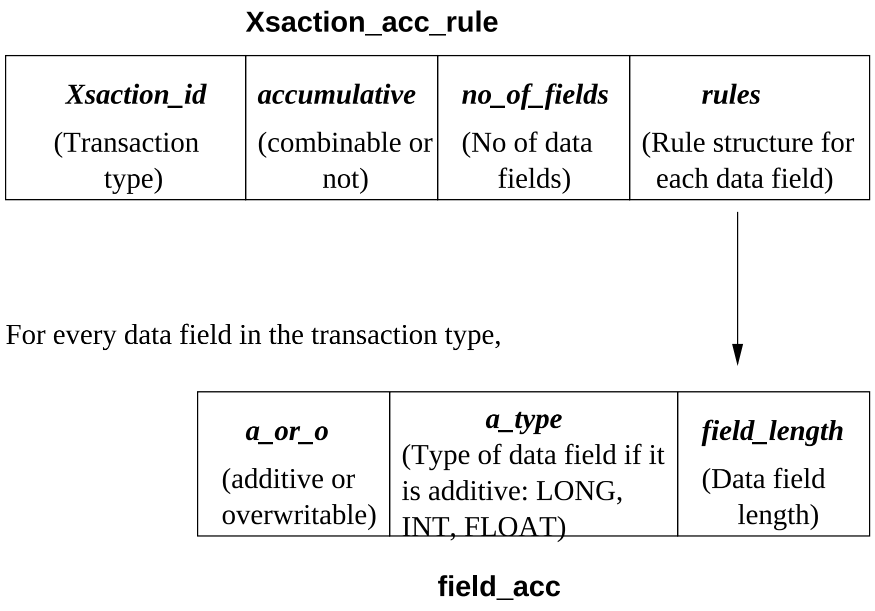


Figure A.2: Structure of a Combination-rule-storage Record

The following is an example of define_Xsaction_acc_rules. Refer to *robot.cpp* in the **CTDS** package for details. Note that the number of arguments to the function and the arguments should not be modified.

```
void define_Xsaction_acc_rules()
{
   struct Xsaction_acc_rule *a;

   /* Rule for transaction type_1 to be stored in xar[0].
      type_1 is accumulative/combinable and it has two
      data fields.  The first data field should be
      combined using the overwrite method and the second
      one should be combined using the additive method
      on float. */
   a = &xar[0];
   a->Xsaction_id = type_1;
   a->accumulative = 1;
   a->no_of_fields = 2;
   a->rules = new field_acc[a->no_of_fields];
   a->rules[0].a_or_o = overwritable;
   a->rules[0].a_type = ADD_TYPE_NONE;     // Not additive
   a->rules[0].field_length = 10;
   a->rules[1].a_or_o = additive;
   a->rules[1].a_type = FLOAT;
   a->rules[1].field_length = sizeof(float);

   /* Rule for transaction type_2 to be stored in xar[1].
      type_2 is not combinable. */
   a = &xar[1];
   a->Xsaction_id = type_2;
   a->accumulative = 0;

   /* Do the same thing for every type of transaction */
}
```