High Fidelity Simulation of Distributed Applications

by

Vijay Kalpathi Ramanathan

B.E (Electronics and Communication Engineering),

Bharathiar University, Coimbatore, India

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____
Dr. Jerry James, Chair

____
Dr. Douglas Niehaus, Member

____
Dr. Susan Gauch, Member

____
Date Project Accepted

To my parents and my brother.

Acknowledgements

I would like to express my sincere gratitude to Dr. Jerry James, my faculty advisor and committee chair, for his guidance, support and encouragement throughout my research work. It has been a great pleasure working with him. I am extremely grateful to him for having provided me with an opportunity to work on this project.

I would like to thank Dr. Douglas Niehaus for providing me with valuable guidance and help during the course of the project. I would also like to thank Dr. Susan Gauch for serving on my masters committee.

I thank my project team members Rajiv, Radha and Dushyanth for their co-operation and support.

Special thanks to my parents and my brother for their support, encouragement and love that they have been giving me all these days.

Abstract

Developing distributed applications has always been challenging. Controlling all aspects of a distributed system is very difficult, thereby making it difficult to debug and test them. Simulating distributed applications contributes to a solution of the problem by making debugging and testing manageable, since a simulated environment offers more capability to control a wide range of parameters that affect the performance of the system.

This project presents a novel approach to simulating distributed applications based on the Reactor pattern. The simulation environment has the capability to adapt network models to simulate various kinds of networks. Using the simulation environment, we built (1) a Token Ring network; and (2) a network implementing the Bully Election algorithm. These were run in both simulated and distributed modes. The standard output generated in both the modes has been verified to be the same.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Debugging and testing distributed systems has always been challenging. This is because of the tremendous difficulty involved in controlling all aspects of a distributed system. Identifying the source of a particular problem is complex, thereby making debugging difficult. The behavior of the system is non-deterministic, due to effectively random context switches. The programmer has no control over the concurrency exhibited by the system. Certain errors might occur due to race conditions (timing-based bugs), which means that they will not occur every time the application runs. One solution is to provide the capability of running the application in exactly the same manner a second time, so that such errors can be replayed and such timing-based bugs can be tracked down. The lack of reproducibility of concurrency scenarios in a distributed system hinders testing.

There are many issues involved in building software for distributed systems, such as transparency, synchronization, concurrency control, fault-tolerance, etc. Software that manages all these is bound to be large, which in turn complicates debugging and testing. Hence, it is easier to solve the problem by trying to simulate the distributed applications prior to actually developing the real one. A simulated environment offers more capability to control the performance of distributed systems. Simulations make debugging and testing manageable, as it is much easier to identify the source of a problem. Therefore, the real application can be developed after the simulated system is completely debugged and tested.

1.2 Proposed Solution

The goal of this project is to build a simulated environment for developing, debugging and testing distributed applications. In the simulated environment, distributed applications are run as a single process. Each node in the distributed application is run as a separate thread in the process. This requires the presence of a thread library which provides the capability of scheduling threads. BThreads [1], a POSIX thread library for Linux, is used for this purpose. It is based on the BERT Reactor, a software pattern for event demultiplexing and dispatching. BERT is designed to facilitate debugging of distributed applications. Handlers for specific types of events are registered with the Reactor. The Reactor reacts to the events, by detecting them and calling the appropriate event handlers. Thus, the debugging information of the application can be obtained from the Reactor by getting the state of the different event handlers.

A virtual timeline is maintained in the simulation environment. Messages are not delivered to recipient nodes as soon as they are sent. They are delayed to account for network latencies. Messages are delivered only after the simulated time that represents the network latency has passed. Scheduling events based on a virtual timeline also helps to ensure that the sequence of events that takes place in the simulated environment is exactly the same as that in the distributed mode. Network models are used to simulate different kinds of networks, so that messages are delivered with delays appropriate to the real network.

1.3 Organization

The remainder of the document is organized as follows. Chapter 2 discusses some of the related work in simulating distributed systems using a Reactor pattern. Chapter 3 discusses the design of our system. It also describes in detail the implementation of the various classes designed to build a simulation environment. Chapter 4 discusses a Token Ring Network and a network that implements the Bully election algorithm, both built using the simulation environment. It also compares the results obtained to the real distributed system. Chapter 5 concludes the document by discussing future work in this area of research.

Chapter 2

Related Work

Numerous mechanisms are employed to simulate distributed systems. Among them, the mechanisms discussed in the following sections are most closely related to this project.

2.1 KU PNNI Simulator

KU PNNI is a simulator for describing, testing and instrumenting any network topology where the participating ATM switches employ the Private Network-to-Network Interface (PNNI) protocol. The main objective of the simulator is to test the performance of the PNNI protocol. The simulator was developed on Bellcore's Q.port software [2]. The KU PNNI Simulator is similar to ours, in that it is based on a reactor. The reactor is responsible for registering and dispatching multiple timer events and also for posting multiple *Q.Port* internal messages. Scheduling of events is done by maintaining virtual time. Events are scheduled based on two levels of priority. The user specifies the parameters to setup the network and run tests. "The main advantage of the KU PNNI Simulator is that it is based on the real ATM switch software and hence it is mostly assumption free" [3]. However, our solution provides the following features that are not present in the KU PNNI Simulator:

- Generic Application Simulation

  It can be used to simulate any distributed application that can be instantiated as an object, including the PNNI network performance analysis code. The KU PNNI simulator can simulate only PNNI networks.

- Reproducibility of execution sequence

  Our design presents a framework that supports replaying the execution sequence of events in exactly the same manner a second time. This makes it easier to debug distributed applications.

2.2 MONARC Distributed System Simulation

MONARC was developed at Caltech for CERN, to perform realistic simulation and modeling of distributed computing systems, customized for specific physics data processing.

> It models the behavior of the system of site facilities and networks, given the assumed physical structure of the computer systems and the usage patterns, including the manner in which hundreds of physicists will access LHC data. The hardware and networking costs, and the performance of a range of possible computer systems, as measured by their ability to provide the physicists with the requested data in the required time, are the main metrics that will be used to evaluate the models [4].

This is similar to our project, in that they use a process-oriented approach for discrete event simulation. MONARC is based on Java and it uses the built-in thread support for concurrent processing. It uses time dependent response functions to describe the behavior of all active components in the system. It also has a network package that models LAN/WAN networks. The main advantage of MONARC is that it allows certain parts of the system to be simulated and other parts running the real application. However, the following features that our simulation environment provides are absent in MONARC.

- Generic Application Simulation

  It can be used to simulate any distributed application that can be instantiated as an object. MONARC is specifically designed to simulate physics data processing. Our environment can be used if the physics data processing code of MONARC can be instantiated as an object.

- Reproducibility of execution sequence

2.3 SimJava

SimJava was developed at the Institute for Computing Systems Architecture, University of Edinburgh. It is a process based discrete event simulation package for Java [5] that simulates complex systems. It provides a visual representation of the objects during the simulation. Since it is based on Java, it can also be incorporated into web pages. It has three packages that provide the basic functionality. SimJava simulates static networks. The package also provides functions that enable the objects to communicate with one another. It maintains a virtual timeline to schedule events. There are a lot of projects developed using SimJava such as GridSim, Distributed SimJava and SIMPROD, to name a few.The advantages of our simulation environment are the following.

- Reproducibility of execution sequences.

- Consistent Application code, which enables the application to be run in distributed mode as well.

2.4 MPISim

This was developed at the University of California, Los Angeles. It is used to predict the performance of large parallel programs by discrete-event simulation [6]. The simulation helps to predict the performance of parallel computation-, communication- and I/O-intensive programs written using the Message Passing Interface (MPI) library. Existing MPI programs have to be modified to support multi-threading so as to be run as a single process in the simulation. It uses queues for communication and maintains a virtual clock based on which pending events in the simulation are sorted. MPI communication calls are translated to non-blocking calls in the simulation. It produces metrics relating to the simulation such as predicted performance of the application program in terms of the execution time, number and type of I/O operations, performance of the simulator and performance of simulated communication. The advantage of our simulation environment is the ability to reproduce execution sequences

2.5 DaSSF

Dartmouth SSF (DaSSF) is a synchronized parallel simulator mainly employed for Computer Network Simulations [7]. It was developed by Dartmouth College, USA. DaSSF is a C++ implementation of the Scalable Simulation

Framework (SSF). DaSSF has the capacity to simulate very large network models. There is a virtual timeline maintained in the simulation. The performance of DaSSF does not degrade in spite of having more entities & events added to the system. The API of DaSSF simplifies the expression of models. It uses a process-oriented simulation approach, which simplifies the modeling effort. DaSSF also provides a set of C++ class libraries. The principal classes are Entity, Process, Event, inChannel, and outChannel. Network models can be built by writing a C++ program that consists of classes derived from these principal classes. “These five base classes provide a truly generic and maximally compact interface that is sufficient to model not only telecommunication networks, but also many other domains” [8]. DaSSF can be run over a wide variety of architectures such as SGI IRIX, SUN Solaris, DEC OSF, Linux and Windows. Parallelism is achieved on these platforms by employing shared memory. The advantages of our simulation environment over DaSSF are the following.

- Generic Application Simulation.

  It can be used to simulate any application, not just networks.
- Reproducibility of execution sequences

Chapter 3

Implementation

The BERT Reactor [9] is a software pattern for event demultiplexing and dispatching that provides an environment for building distributed applications. BThreads [1], is a POSIX thread library for Linux based on the BERT Reactor. Our project uses BThreads to run distributed applications as a single process and to maintain the virtual timeline. Moreover, since BThreads is a user-level thread library, each node runs as a user-level thread and the entire distributed application is actually run as a single kernel-level thread.

The simulation environment is implemented in C++. Each application object that is using the simulation environment must inherit a base class called *Application*. This class has all the information about the objects that is necessary to facilitate simulation. Communication between the different nodes is achieved using the *SimComm* class. The interface for this class is the same for distributed mode as for simulated mode. However, the implementation differs. In distributed mode the functions are just wrappers to the Socket API. In simulated mode the data communication is achieved using queues. The *Network* class is responsible for simulating different kinds of networks and determines the delay involved in message transmission. Each application object runs as a separate user-level thread. These threads are scheduled to run based on their virtual time. An object of *SetUp* class is created and is responsible for setting up the simulation. The functionality of the different classes is explained in detail below.

3.1 BThreads

Our project uses BThreads [1] to control thread execution sequences. Moreover, since this thread library is based on the BERT Reactor, all events that affect concurrency can be recorded and later replayed. These events include I/O, signals and timers. The simulation environment is thus based on an event-driven framework that acts as the controlling point for all events that affect concurrency since all such events can be captured at this point. It is then possible to record such events so that they can be used later to replay execution sequences.

This project uses BThreads to run distributed applications as single-threaded processes. It runs each of the different instances of the application as a single user-level thread using the BThreads library. Also, BThreads provides an option to set our own scheduling function. Thus, using BThreads helps to schedule the threads based on the virtual time.

3.2 Design

The primary design of the simulation is shown in figure 3.1. *SetUp* creates as many instances of the distributed application as specified for the simulation. It then spawns a user-level thread for every application object. It also creates another thread for an object of the *SignalThreads* class. The total number of objects that participate in the simulation is passed to the constructor function of the application objects. It is necessary that the constructor function of the application be designed to pass this to the parent class *Application.* The scheduler has two functions, one to enqueue an application object thread to the ready queue and the other to dequeue it. Threads are enqueued in the ready queue based on their virtual time. The ready queue is thus a priority queue where the thread with the lowest virtual time is at the head and the highest is at the tail. When threads have the same virtual time, they are enqueued based on a FIFO discipline. Threads are dequeued from the head. The virtual time of each process is incremented by its elapsed execution in every time slice, before enqueuing the process on the ready queue. The *SimComm* class maintains a set of queues for data communication. When a node sends a message, it is timestamped with the receive time. The receive time of any message is determined by adding the virtual time of the sender to the delay obtained from the *Network* class.

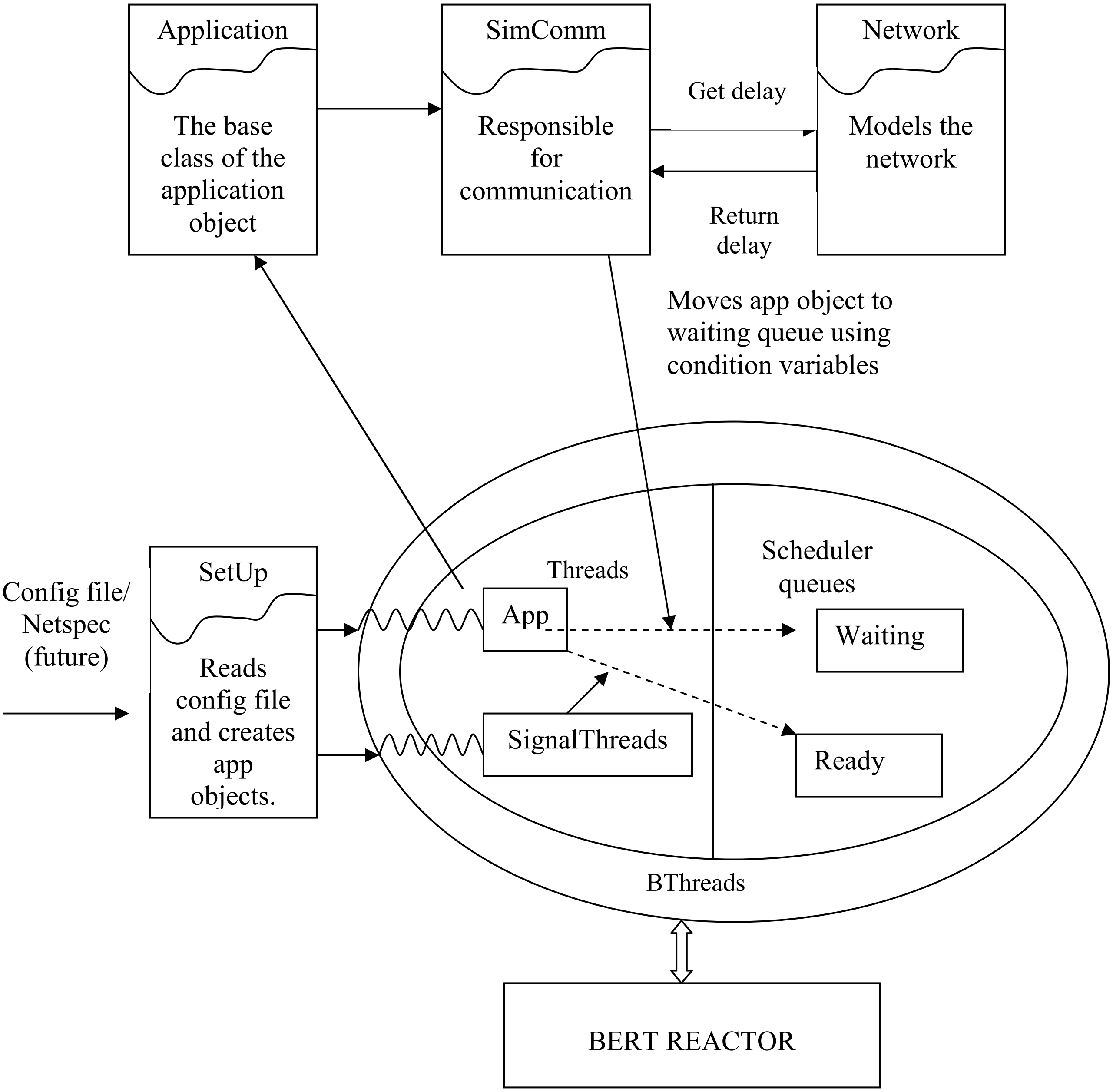

Application
The base class of the application object
SimComm
Responsible for communication
Network
Models the network
Get delay
Return delay
Moves app object to waiting queue using condition variables
Config file/ Netspec (future)
SetUp
Reads config file and creates app objects.
Threads
App
SignalThreads
Scheduler queues
Waiting
Ready
BThreads
BERT REACTOR

Figure 3.1: Simulation Design

Whenever a process tries to receive a message, it will block if the sender has not yet sent the message. Each thread has a condition variable associated with it. When a thread receives a message, its virtual time has to be more than the receive time stamped on the message. If not, the thread's state is changed to waiting, and it waits on the condition variable. At this time, the thread is moved from the ready queue to the waiting queue. When the *SignalThreads* object runs it determines the waiting thread with the lowest virtual time and increments its virtual time to that of the next highest thread. It then signals the waiting thread, which is added to the ready queue from the waiting queue. If the virtual time is still less than the receive time of the message, the thread waits again. This repeats until the virtual time is increased to a value high enough to receive the message. The *SignalThreads* object runs as long as there is at least one application object running, after which it exits. Thus, distributed applications are simulated as a single-threaded process running on a virtual timeline, incorporating network delays.

3.3 Description of Classes

3.3.1 Simulated Mode

3.3.1.1 Setup

Setup is responsible for starting the simulation. The simulation parameters are specified in a configuration file. Eventually, *Netspec* [10] will supply these parameters. *SetUp* creates as many application objects as specified by the configuration file with the given parameters. It then creates the user-level threads for these objects and the *SignalThreads* object.

SetUp forms the core part of the simulation by creating the data structures necessary for the simulation. These include pointers to application objects, a mapping of the application object to its thread ID and condition variables and mutexes for the application objects.

3.3.1.2 Application

This class is the base class that each application object must inherit. It is responsible for maintaining all information about the objects necessary for the simulation. It maintains the virtual time and the state (waiting or not) of the application objects. It also has wrapper functions for the communication functions provided by *SimComm*. The interface for this class is shown in figure 3.2.

SetUp determines the set of objects that participate in the simulation. The application object class must have a constructor function that accepts this value. SetUp passes this value to the application objects while instantiating them. The application in turn passes this to its parent, the *Application* class. *Application* has a global pointer to an instance of the *SimComm* class. This ensures that all application objects access the same instance of the *SimComm* class. *Application* has a constructor function that takes the count of the total objects participating in the simulation and passes this value to the *SimComm* class. It provides functions that are used by *SimComm* and *SignalThreads* to determine the state of an application object and to modify it. It also provides functions that are used by *SimComm*, *SignalThreads* and the scheduler to determine the virtual time of the application objects and modify them. The Application class also provides wrapper functions to those implemented by *SimComm*. It accesses these functions through the global *SimComm* object. Thus, whenever an application object needs to perform the functions associated with communication, it just needs to make calls to these wrapper functions

```
class Application
{
  public:
    Application();
    Application(int);
    long long getClock();
    void setClock(long long);
    void setBlocked(bool);
    bool getBlocked();
    void Send(int,void **,int);
    int Recv(int,void **,int);
    int Close(int);
  private:
    long long clock;
    bool blocked;
};
```

Figure 3.2: *Application* Class Interface

3.3.1.3 SimComm

An essential requirement for a simulation environment is the ability to provide functions that enable communication between the various parts of the distributed application. The *SimComm* class provides this functionality. The interface for this class is the same for both the distributed and simulated modes. This helps to keep the application code consistent. However, the implementation differs for the two modes. The application is unaware of the existence of this class. It accesses the functions of this class through the wrapper functions that the *Application* class provides. The interface is shown in figure 3.3.

Communication between the various nodes is achieved using queues. Queues allow unidirectional flow of data. Two queues are created per process pair. Hence, if there are N objects involved in the simulation, N^2 queues are created. Since, queues are always allocated in pairs, the index of a queue an object reads from is always the 1's complement of the index of the queue it writes to. For example, queue pairs will have indices (0, 1), (2, 3), etc. These queues are stored in an array. The index to this array of queues is the connection descriptor that represents the connection between any two nodes.

The *SimComm* class maintains a table of allocated queues. Queues are allocated in pairs. Before a queue is allocated its status is invalid. Whenever an object requests a connection with another object, the table is first checked to determine if the queue has already been allocated. If so, the index of the allocated queue is returned to the calling object. If not, two queues are allocated and the index of one of them is returned. The other index will be returned to the other object when it requests a connection with the first. The status of the two queues is then changed to valid.

When a node sends data, the status of the queue to which the data is to be sent is checked to determine if it is valid. If so, the receive time of the message is computed by adding the virtual time of the sender to the delay obtained from the *Network* class. The data to be sent is timestamped with this receive time and is added to the queue. *Send* is a blocking call. Hence, a message can be sent only if the queue is empty. If the queue has data, the sender has to block until this data is received.

When a node attempts to receive a message, the status of the queue is checked. If it is valid, the index of the queue it reads from is determined by computing the 1’s complement of its connection descriptor. If the queue has no data, the receiving object blocks until data arrives. If the data is present, the virtual time of the object must be more than the receive time of the message in order to maintain the virtual timeline. If so, it receives the message. If not, it waits on a condition variable and later gets signaled by the *SignalThreads* object.

After the application object performs all communication, it closes the connection descriptor, which changes the status of the corresponding queue to closed.

```
class SimComm
{
  public:
    SimComm();
    void setTotalObj(int);
    int getConnection(struct sockaddr_in *,
                      struct sockaddr_in *);
    int Send(int,void **,int,void *);
    int Recv(int,void**,int,void *);
    int Close(int);
  private:
    int totalObj;
    vector<string> Queue_ID_tbl;
    queue <void*> * Queues;
    string* QueueStat;
};
```

Figure 3.3: *SimComm* Class Interface

This class has four functions that enable the various nodes to communicate with one another.

getConnection

The first step involved in building a communication channel between any two parts of the distributed application is establishing a connection between them. The *getConnection* function of the class connects two nodes of the distributed system. The prototype of the function is given below.

```
int SimComm::getConnection(struct sockaddr_in * my_ID,struct
sockaddr_in * peer_ID)
```

where,

my_ID is the address of the node requesting the connection

peer_ID is the address of the node to which the connection must be established

Return Value:

On successful completion *getConnection* returns a descriptor that is used for further communication. Otherwise it returns –1.

Send

The *Send* function of the class sends a message from one node to another. The prototype of the function is given below.

```
int SimComm::Send(int ID,void ** data, int size,void * app_ptr)
```

where,

ID is the connection descriptor

data is the message that is to be sent

size is the size of the message to be sent in bytes

and app_ptr is a pointer to the node that is sending the message.

Return Value:

On successful completion *Send* returns the number of bytes that were sent. Otherwise it returns –1.

Recv

The *Recv* function of the class enables a node to receive a message. The prototype of the function is given below.

```
int SimComm::Recv(int ID,void ** data,int size,void * app_ptr)
```

where,

ID is the connection descriptor

data is the buffer to hold the received message

size is size of the message to be received in bytes

and app_ptr is a pointer to the node that is receiving the message.

Return Value:

On successful completion *Recv* returns the number of bytes that is received. Otherwise it returns –1.

Close

After all communication is complete the established connection must be closed. The *Close* function of the API fulfills this purpose. The prototype of the function is given below.

```
int SimComm::Close(int ID)
```

where,

ID is the descriptor of the connection to be closed.

Return Value:

On successful completion *Close* returns 0. Otherwise it returns –1.

3.3.1.4 Network

The *Network* class is responsible for modeling the network that is being simulated, by ascertaining the delay associated with message transmission from one node to another. Presently, this class does not model any particular network. Whenever SimComm contacts it, it returns a constant delay after which the message is to be received. The application object is not aware of this class. This class is in a rudimentary form; it has just one function. The section on future work discusses how to make it model real networks. The interface is shown figure 3.4.

```
class Network
{
  public:
  Network();
  long long getDelay();
};
```

Figure 3.4: *Network* Class Interface

3.3.1.5 SignalThreads

When a node tries to receive a message and its virtual time is less than the receive time stamped on any message sent to it, it waits on a condition variable associated with the message queue. This causes the application object to be removed from the ready queue and added to the waiting queue. *Setup* creates a thread for an object of the *SignalThreads* class. This class inherits from *Application*, so that it also has access to the virtual time. This class is responsible for signaling blocked threads. The *SignalThreads* object runs as long as there are application objects still running. The interface of this class is shown in figure 3.5.

This class has access to pointers of all application objects running in the simulation. The *Unblock* function signals waiting threads. This function first sorts the application objects based on their virtual times so that the ordering of objects is the same as the ready queue (with the exception that waiting objects will be missing in the ready queue). It then determines the first blocked object in this list and increments its virtual time to that of the next highest object. It then signals the object, causing it to be moved from the waiting queue to the ready queue. The application object receives the message if the increased virtual time is greater than the receive time of the message. Otherwise the object waits again, until *SignalThreads* increases its virtual time high enough to receive a message.

```
class SignalThreads: public Application
{
  public:
  SignalThreads();
  void Unblock(void *, int);
};
```

Figure 3.5: *SignalThreads* Class Interface

3.3.2 Distributed Mode

3.3.2.1 SetUp

The functionality provided by SetUp in distributed mode is minimal compared to the simulated mode. It creates just one instance of the application object by reading the configuration file. Also there are no threads spawned to run this object. *SetUp* runs the application object in the main thread and exits.

3.3.2.2 Application

Application objects must inherit from *Application* for the distributed mode also. In this mode, the *Application* class just provides the wrapper functions for *SimComm*. The other functions are needed only for the simulation. The Application class provides a constructor function that takes as an argument the count of the total objects participating in the simulation. In distributed mode this is equal to one, since each node now runs as a separate process. However, this constructor function does not do anything meaningful. It is present only to maintain consistency of the application code between simulated and distributed modes.

3.3.2.3 SimComm

The *SimComm* class uses the Socket API in order to provide communication in distributed mode. Since the application calls the *getConnection* function when it requests a connection with another node, *SimComm* has to ensure that one node blocks on the *accept* call and the other blocks on the *connect* call until the connection is established. This is achieved by representing the address in an integer format. The node with a lower value blocks on the *connect* call while that with a higher value blocks on the *accept* call. The *getConnection* function returns a socket to the calling node. The *Send* and *Recv* functions read and write respectively to the corresponding sockets. *Close* is called to close a socket. These functions return the values that the Socket API returns.

Chapter 4

Testing

The design of the simulation environment was described in the previous section. Its performance was tested by (1) simulating a Token Ring network and (2) simulating a network where the nodes elect a leader based on the Bully Election algorithm. These applications were run in both distributed and simulated modes. The standard output generated in both modes were compared.

4.1 Token Ring Network

Each Token Ring application object represents a node in a Token Ring network. Each node establishes a connection with its neighboring nodes. It waits until it receives the token from the previous node, prints the token and then passes it to the next node. The user specifies the number of times the token should loop around the network. The code for the Token Ring application remains the same in both modes.

4.1.1 Design

Each Token Ring application object needs to know the following information: its ID in the network, the local address of the host machine to which it binds, the port number on which it runs, the address and the port number on which its previous node and next node run, and the number of times the token should loop around the network. It also needs to know if it is the node that generates the token or not. The information about the port number and IP address is used to create sockets in distributed mode, and is used to create an identifier to associate the node with its queue in simulated mode.

The first step is to set up the network using the *Setup* function of the *TokenRingApp* class. This function establishes a connection between each node of the network with its neighbors. The node that generates the token establishes a connection first with its next node and then with the previous node. Each other node establishes a connection with its previous node before connecting with its next node. This does not matter for simulated mode. However, if this was not done in distributed mode, application objects might block or the socket descriptors for the previous and next node may be swapped due to the order in which connection requests arrive. For example, consider a network with 5 nodes numbered 1 through 5. If all nodes try to establish a connection with the previous node and then with the next node, then node 5 blocks on an *accept* call from node 4, node 4 from node 3, and so on. However node 1 will block on a connect call to node 5 (its previous node). Since node 5 is already blocked on an *accept* call, it accepts the connection from node 1, assuming that it is node 4. Hence, node 5 misinterprets the socket it has established with node 1 as a socket it established with node 4. In simulated mode this does not happen since there are no *accept* or *connect* calls.

Once the connection has been established with neighboring nodes, the starter node generates the token. All other nodes read the token from the previous node. After the token has been generated or read, the node prints its ID, the number of times the token has looped around the network and its value. After the token has looped around a sufficient number of times, the node closes connections with its neighbors and exits.

4.1.2 Configuration file

A configuration file specifies the structure of the Token Ring network. The layout of the file is shown in Table 4.1.

Field Name
Node ID
Next node pointer
Previous node pointer
Address
Port number

Table 4.1: Layout of Configuration file for Token Ring network

The first field is the ID of the node. It is a string that uniquely represents a node of the Token Ring. Next node pointer is the ID that corresponds to the next node. Previous node pointer is the ID that corresponds to the previous node. The next field is the local address of the host to which the node binds. The last field is the port number on which to listen. The fields in the configuration file are space delimited. The configuration file used for testing the simulation environment is shown in figure 4.1.

```
ID01 ID05 ID02 kermit.ittc.ku.edu 10001
ID02 ID01 ID03 diannao.ittc.ku.edu 10002
ID03 ID02 ID04 scooter.ittc.ku.edu 10003
ID04 ID03 ID05 waldorf.ittc.ku.edu 10004
ID05 ID04 ID01 marcus.ittc.ku.edu 10005
```

Figure 4.1: Configuration file used in simulating Token Ring network

4.1.3 Distributed mode

The user supplies the path to the configuration file and the number of times the token should loop around the network as command-line arguments. *SetUp* then reads the configuration file. The first node as specified in the configuration file is the starter of the network. The application objects are created with the specified parameters and are run as separate processes.

In the example used for testing, a Token Ring with 5 nodes is created. The token is specified to loop around the network twice. The standard output from the various nodes is shown in figure 4.2. Each node prints its ID, the number of times the token has already looped around the network and the token.

Since the node with ID “ID01” is the first node specified in the configuration file, *SetUp* designates it as the starter. So it thus generates the token. The token passing does not begin until all nodes are running and the connection is established. After that, ID01 generates the token and prints it. It then blocks on the *Recv* call waiting for the token to be passed to it by ID05 since the token must loop twice. At this point, ID02 reads the token from ID01 and passes it to ID03. Each node reads the token from the previous node, print its value and passes it to the next node. The nodes pass the token for as many times as the user specifies. The nodes remain blocked in the *Recv* call until the previous node writes to them. After completion, the nodes close the connection and exit.

```
kermit [22] % SetUp ID01 config.txt 2

Machine ID ID01
LOOP COUNT 1
Token: TOKEN

Machine ID ID01
LOOP COUNT 2
Token: TOKEN

diannao [119] % SetUp ID02 config.txt 2

Machine ID ID02
LOOP COUNT 1
Token: TOKEN

Machine ID ID02
LOOP COUNT 2
Token: TOKEN

scooter [19] % SetUp ID03 config.txt 2

Machine ID ID03
LOOP COUNT 1
Token: TOKEN

Machine ID ID03
LOOP COUNT 2
Token: TOKEN

waldorf [4] % SetUp ID04 config.txt 2

Machine ID ID04
LOOP COUNT 1
Token: TOKEN

Machine ID ID04
LOOP COUNT 2
Token: TOKEN

marcus [11] % SetUp ID05 config.txt 2

Machine ID ID05
LOOP COUNT 1
Token: TOKEN

Machine ID ID05
LOOP COUNT 2
Token: TOKEN
```

Figure 4.2: Standard output of the Token Ring network for Distributed mode

4.1.4 Simulated mode

In simulated mode, the user specifies the path of the configuration file and the number of times the token should loop around the network. *SetUp* creates the application objects with the specific parameters.

It parses the configuration file shown in figure 4.1 and creates five Token Ring objects and runs them in a separate user-level thread. An object of *SignalThreads* is also created and is run in another thread. The standard output of the simulation is shown in figure 4.3. Each node prints its ID, the number of times the token has already looped around the network and the token.

In simulated mode also, node ID01 is designated to be the starter. It generates the token, prints it and passes it to node ID02. It then waits for the token to be passed to it by ID05. As seen from the output, each node receives the token from the previous node, prints its value and passes it over to the next node. As seen from figures 4.2 and 4.3 the standard output generated in both modes is the same.

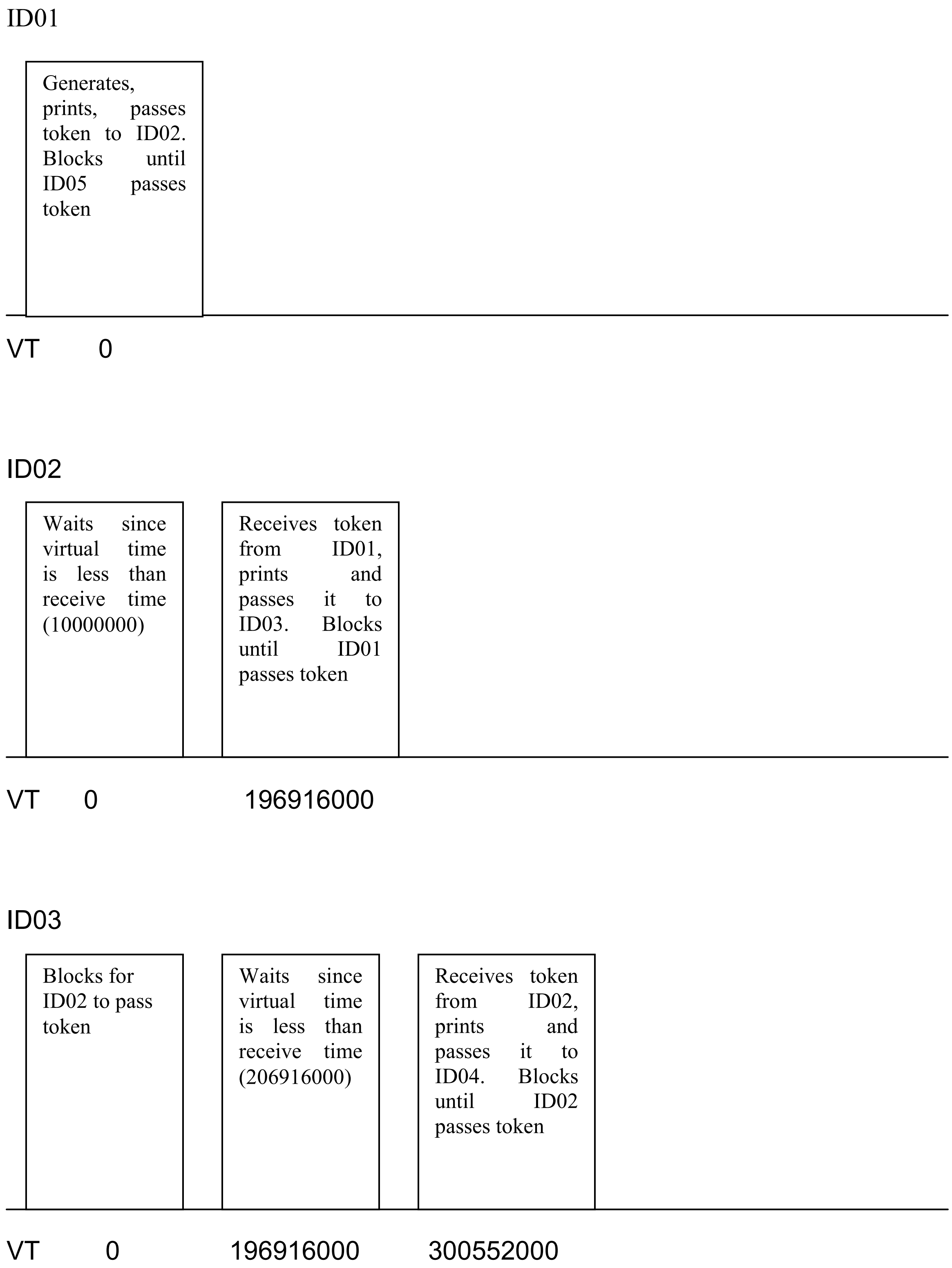
```
diannao [2] % SetUp config.txt 2

Machine ID ID01
LOOP COUNT 1
Token: TOKEN

Machine ID ID02
LOOP COUNT 1
Token: TOKEN

Machine ID ID03
LOOP COUNT 1
Token: TOKEN

Machine ID ID04
LOOP COUNT 1
Token: TOKEN

Machine ID ID05
LOOP COUNT 1
Token: TOKEN

Machine ID ID01
LOOP COUNT 2
Token: TOKEN

Machine ID ID02
LOOP COUNT 2
Token: TOKEN

Machine ID ID03
LOOP COUNT 2
Token: TOKEN

Machine ID ID04
LOOP COUNT 2
Token: TOKEN

Machine ID ID05
LOOP COUNT 2
Token: TOKEN
```

Figure 4.3: Standard output of the Token Ring network for Simulated mode

In order to better understand the output of simulated mode, the concept of a virtual timeline must be understood. The exact sequence of events that occur is shown in figure 4.4. For testing purposes, the *Network* class returns a constant delay of 10 milliseconds. Since at the start of the simulation all objects have virtual time 0, they are on the ready queue in FIFO order. Thus, when the scheduler dequeues the objects, ID01 is the first node to run.

As seen in figure 4.4, at virtual time 0, ID01 generates the token, prints its value and passes it to ID02. It then waits for ID05 to pass it back to it for the next loop. At this point, ID03, ID04 and ID05 are blocked for the previous node to pass the token to them. However, ID02 has to wait until the network delay has expired though ID01 has passed the token. Thus, it is waiting for the delay to expire, while other nodes are waiting for their respective previous nodes to pass the token to them. The difference is that ID02 is in the waiting queue while the other nodes are in the ready queue. This is because ID02 waits on a condition variable, while the other nodes wait on a loop that breaks when the token is passed.

When the *SignalThreads* object runs, it checks for waiting objects and increments their virtual times to that of the next highest node. At this time, ID02 is the only waiting object. *SignalThreads* increases its virtual time to 96.927 milliseconds, that of ID01, the next highest node and signals it. This causes ID02 to be added to the ready queue. The increased virtual time however, is less than the receive time of the message, so it waits again. After sufficient time has elapsed, *SignalThreads* increases the virtual time of ID02 to 196.916



Generates, prints, passes token to ID02. Blocks until ID05 passes token

Waits since virtual time is less than receive time (10000000)

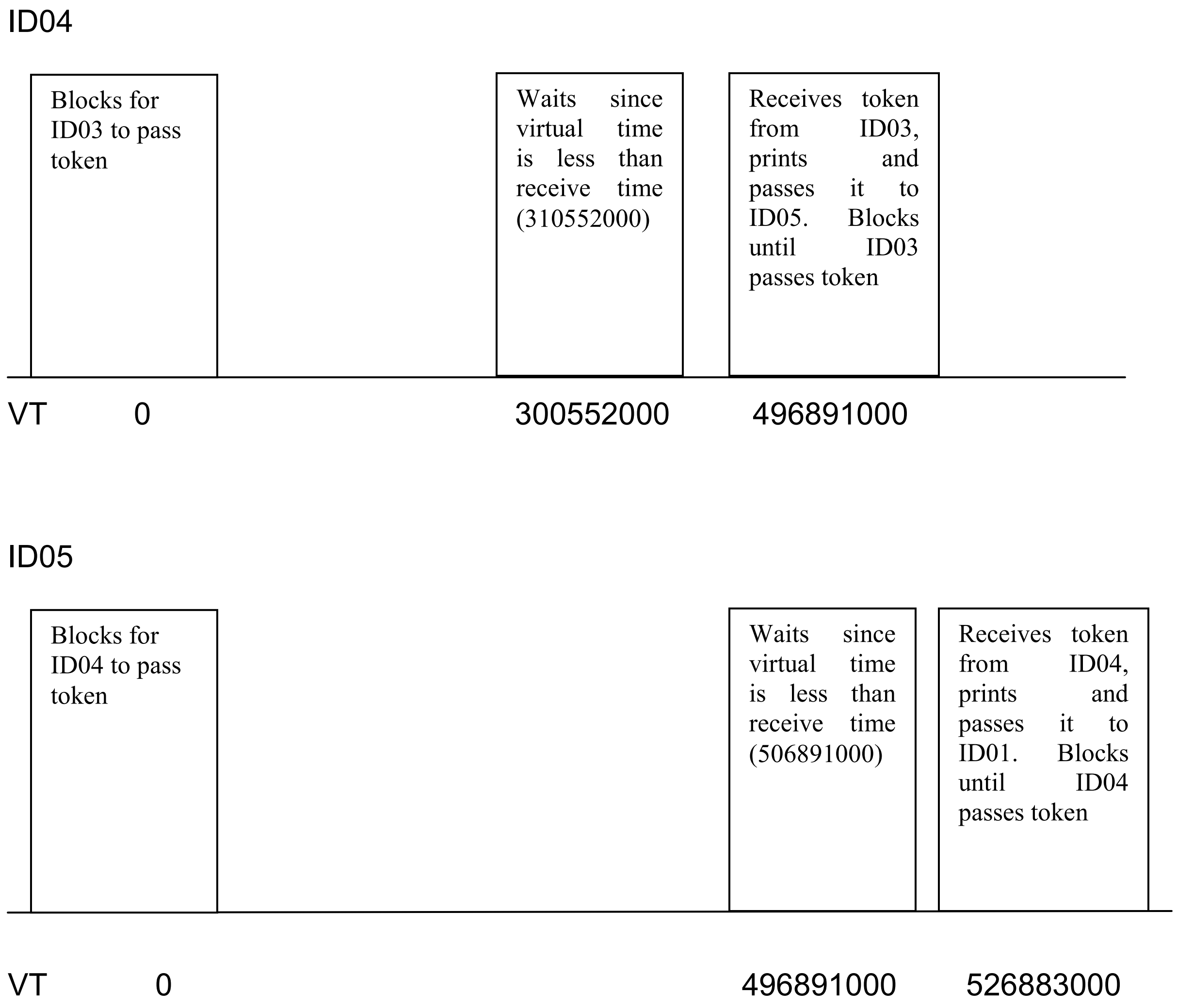
Receives token from ID01, prints and passes it to ID03. Blocks until ID01 passes token

Blocks for ID02 to pass token

Waits since virtual time is less than receive time (206916000)

Receives token from ID02, prints and passes it to ID04. Blocks until ID02 passes token

Figure 4.4: Virtual timeline showing the sequence of events for the Token Ring network.



Blocks for ID03 to pass token

Waits since virtual time is less than receive time (310552000)

Receives token from ID03, prints and passes it to ID05. Blocks until ID03 passes token

Blocks for ID04 to pass token

Waits since virtual time is less than receive time (506891000)

Receives token from ID04, prints and passes it to ID01. Blocks until ID04 passes token

Figure 4.4 (Continued)

milliseconds, a value high enough to receive the message. ID02 then receives the token, prints it and passes it to ID03. It then waits for ID01 to pass the token back to it for the next loop.

At this point, ID03 no longer waits for the token to be passed. However, it will wait for the network delay to elapse before it can receive the token. The virtual time of ID03 is increased in the same manner as ID02. Each node blocks until the previous node sends the token, waits for its virtual time to be high enough to receive the token and passes the token to the next node. Thus, in the simulated mode, the sequence of events that happens is exactly the same as those that happen in the distributed mode.

4.2 Bully Algorithm

Each application object represents a node in an arbitrary network that implements the bully election algorithm to elect a leader. The purpose of this application is to stress the simulation environment by making it handle intense communication between the application objects, since communication between nodes in the Token Ring network is trivial. Moreover, there is a greater amount of concurrency involved in this application. Hence, it tests the performance of the simulation environment better than the Token Ring application. The code for this application also remains the same in both modes.

4.2.1 Design

Each application object needs to know the following information: its ID in the network, the IP address and port number on which it runs and the IP address and the port number on which all other processes in the network run, their IDs and the total number of nodes in the network. The first step is to set up connections with all other nodes in the network. The *GetConnection* function of the *Bully* class takes care of this. Connection descriptors to nodes with a lower ID are stored in one array and to nodes with a higher ID in another array. This is done because each node tries to receive election messages from all the lower nodes. Since *Recv* is a blocking call, we must ensure that a node does not try to receive a message from any of the higher nodes. Maintaining two arrays for storing connection descriptors identifies connections on which to receive election messages. Once the connection is established, each node attempts to receive a message from all nodes with a lower ID. If the message is an election message, it sends back an acknowledgement. After it has received and acknowledged messages from all nodes with a lower ID, it sends an election message to all nodes with a higher ID. After that, it attempts to receive a message from all nodes with a higher ID. If it receives an acknowledgement message from at least one node with a higher ID, it loses the election. The leader is thus the node with the highest ID. The leader prints the result to standard output. The connections with all nodes are closed and the application exits.

4.2.2 Configuration file

The configuration file specifies information about the nodes of the network. The layout of the file is shown in Table 4.2.

Field Name
Node ID
IP address
Port number

Table 4.2: Layout of Configuration file for network implementing Bully Algorithm

The first field is the ID of the node. It is an integer that uniquely represents a node in the network. The next field is the local address of the host to which the node binds. The last field is the port number on which to listen. The fields in the configuration file are space delimited. The configuration file used for testing the simulation environment in figure 4.5

```
0001 diannao.ittc.ku.edu 10001
0002 marcus.ittc.ku.edu 10002
0003 kermit.ittc.ku.edu 10003
0004 scooter.ittc.ku.edu 10004
0005 waldorf.ittc.ku.edu 10005
```

Figure 4.5 Configuration file used in simulating a network implementing Bully Algorithm

4.2.3 Distributed mode

The user supplies the path to the configuration file as a command-line argument. *SetUp* reads the configuration file and creates the application objects with the specified parameters and runs them as separate processes.

As seen from the configuration file shown in figure 4.5, a network with 5 nodes is created. The standard output from the various nodes is shown in figure 4.6. The leader prints its ID and the message that it won the election.

```
diannao [42] % SetUp 0001 config.txt
diannao [43] %

marcus [21] % SetUp 0002 config.txt
marcus [22] %

kermit [27] % SetUp 0003 config.txt
kermit [28] %

scooter [24] % SetUp 0004 config.txt
scooter [25] %

waldorf [9] % SetUp 0005 config.txt
My ID is 0005. I have won the election.
```

Figure 4.6 Standard output of the Bully Algorithm for the Distributed mode

The election does not begin until all nodes are running and connections are established. After that, each node blocks on the *Recv* call waiting for nodes with a lower ID to send an election message. Upon receiving the election message, an acknowledgement is sent. The node then sends an election message to all nodes with a higher ID. It loses if it gets an acknowledgement message back from at least one of them. The winner is thus the node with the highest ID. As seen from figure 4.6, node 0005 is elected to be the leader. The winner prints its ID and the message that it won the election. After completion, the nodes close connections with other nodes and exit.

4.2.4 Simulated mode

In simulated mode, the user specifies the path of the configuration file as a command-line argument. *SetUp* creates the application objects with the specific parameters. It parses the configuration file shown in figure 4.5 and creates ten application objects and runs them in a separate user-level thread. An object of *SignalThreads* is also created and is run in another user-level thread. The standard output of the simulation is shown in figure 4.7. The winner prints its ID and the message that it won the election. As seen from figure 4.7, node 0005 is elected to be the leader.

```
diannao [29] % SetUp config.txt
My ID is 0005. I have won the election.
```

Figure 4.7: Standard output of the Bully Algorithm for the Simulated mode

It can be seen that the communication and concurrency involved with the Bully algorithm is much more than that with the Token Ring. Hence, it clearly tests the simulation environment more vigorously. The virtual timeline for the bully algorithm is not shown since the number of objects involved is much greater than that in the Token Ring, thereby increasing the difficulty to represent it. The concept however remains the same.

Chapter 5

Conclusions and Future Work

This project presents a novel approach to simulating distributed applications. It makes use of a user-level threading library working on top of a Reactor pattern. This framework supports replaying the execution sequence of events in exactly the same manner a second time, making it easier to debug distributed applications.

The simulation environment requires applications to use the communication interface (*SimComm*) for sending and receiving messages. The application code for both the simulated and the distributed mode remains the same. Thus, the applications can be built using the simulation environment and without any changes can be run in distributed mode.

To use the simulation, distributed applications must be designed to be instantiated as C++ objects. A class is implemented to provide for the communication between the various objects in the simulation. Network models can be incorporated to deliver messages with delays appropriate to real networks. The simulation also maintains a virtual timeline that enables sequencing of events in a manner exactly the same as that of distributed systems.

The simulation environment was also tested by simulating a Token Ring network with five nodes. The resulting output conforms to that obtained by running the application in distributed mode. Also, objects wait for network delays to expire before receiving messages. This confirms the ability of the simulation environment to adapt network models.

5.1 Future Work

- Since the BERT Reactor is the core of the simulation, it can be used to record all debugging information. The simulation can be extended to replay execution sequences.

- Presently, the network model returns a constant delay. It can be extended to make it simulate real networks. The delay can be calculated based on the message size, bandwidth, source node, and destination node.

Bibliography

[1] Penumarthy Sreenivas Sunil. “Design and Implementation of a User-Level Thread Library for Testing and Reproducing Concurrency Scenarios”. Master’s thesis, The University of Kansas, 2002.

[2] Information and Telecommunication Technology Center, The University of Kansas. “KU PNNI Simulator User’s Manual Version 1.2”, January 2000.

[3] Sandeep Bhat, Doug Niehaus, Victor Frost. “A Performance Evaluation Architecture for PNNI”. Master’s thesis, The University of Kansas, 1998.

[4] Iosif C. Legrand. “MONARC Distributed System Simulation”. June 1999.

http://monarc.web.cern.ch/MONARC/

[5] Fred Howell and Ross McNab. "SimJava: a Discrete Event Simulation Package for Java with Applications in Computer Systems Modeling", First International Conference on Web-based Modeling and Simulation, San Diego CA, Society for Computer Simulation, Jan 1998.

[6] S.Prakash and R.L.Bagrodia. “MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs”. 1998 Winter Simulation Conference (WSC98), 1998.

[7] Jason Liu, David M. Nicol, Brian J. Premore and Anna L. Poplawski. “Performance Prediction of a Parallel Simulator”. Proceedings of the Parallel and Distributed Simulation Conference (PADS'99), Atlanta, GA 1999

[8] “Dartmouth Scalable Simulation Framework”

http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/

[9] Rajkumar Girimaji. “Reactor, a Software Pattern for Building, Simulating, and Debugging Distributed Systems”. Master’s thesis, The University of Kansas, 2002.

[10] “NetSpec: A Tool for Network Experimentation and Measurement”

http://www.tisl.ukans.edu/netspec/