Distributed synchronizers in network simulator (Ns) software

Wei Huang
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Huang, Wei, "Distributed synchronizers in network simulator (Ns) software" (2002). UNLV Retrospective Theses & Dissertations. 1428. https://digitalscholarship.unlv.edu/rtds/1428
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
DISTRIBUTED SYNCHRONIZERS IN NETWORK
SIMULATOR (NS) SOFTWARE

by

Wei Huang

Bachelor of Science
Fuxin Institute of Mining and Technology, China
1992

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2002
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

October 14, 2002

The Thesis prepared by
Wei Huang

Entitled
Distributed Synchronizers in Network Simulator (NS) Software

is approved in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Ajay Kunnuru Dalal
Examination Committee Chair

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Distributed Synchronizers in Network Simulator (NS) Software

by

Wei Huang

Dr. Ajoy K. Datta, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Distributed algorithms are designed for systems consisting of many interconnected processors that communicate with one another by exchanging messages through communication links. Distributed algorithms are used on a wide range of applications, from a VLSI chip to LAN, to the Internet. The advantages of distributed systems include information exchange, resource sharing, replication, parallelization, and modularization.

NS (Network Simulator) is an object-oriented, discrete event driven network simulator developed at USC/ISI written in C++ and OTCL. NS is primarily useful for simulating local and wide area networks. It produces one or more text-based output files that contain detailed simulation data. The data can be used for simulation analysis or as an input to a graphical simulation display tool, called Network Animator (NAM).

There are two approaches to designing distributed algorithms. In synchronous algorithms, the operation of each process is done in a lock-step behavior, whereas in asynchronous algorithms, the processes take steps in an arbitrary order and at arbitrary relative speeds. Synchronous algorithms are easier to write and prove. However,
asynchronous algorithms are easier to implement. Thus, an approach to designing distributed algorithms in asynchronous systems is to start with synchronous algorithms, then transform them into corresponding asynchronous versions by passing them through a special algorithm, called synchronizer. This allows one to use asynchronous systems to run the original synchronous algorithms. The synchronizer itself is an asynchronous algorithm.

In this research, we experiment with different types of synchronizers. We implement them by considering two applications: leader election and breadth-first search algorithms. The algorithms are implemented on arbitrary networks. We compare the algorithms in terms of communication complexity. We also discuss the suitability of NS as a platform to implement synchronous and asynchronous algorithms.
# TABLE OF CONTENTS

ABSTRACT ........................................ iii

LIST OF FIGURES ................................ vii

ACKNOWLEDGEMENTS ........................ viii

CHAPTER 1 INTRODUCTION ................ 1
  Synchrony and Distributed Synchronizer .. 1
  Implementation in the Network Simulator .. 2
  Related Work .................................. 2
  Our Contributions ........................... 3
  Outline of the Thesis ........................ 4

CHAPTER 2 PRELIMINARIES ............... 5
  Distributed Systems ......................... 5
  Distributed Algorithms ..................... 7
  Synchronous Networks ..................... 8
  Asynchronous Networks .................... 10
  Synchronizer ................................ 10
  Our Model and Topology ................... 12
  Programming Notations .................... 12

CHAPTER 3 SYNCHRONIZER ALGORITHMS .. 13
  A Simple Synchronizer ...................... 13
  Safe Synchronizer ........................... 15
    1. The Alpha Synchronizer .............. 16
    2. Beta Synchronizer .................... 16

CHAPTER 4 NETWORK SIMULATOR .......... 18
  Overview .................................... 18
  Features of NS .............................. 21
    1. A simple simulation example ........ 22
    2. Event Scheduler ...................... 22
    3. Packet .................................. 22

CHAPTER 5 BFS IN NS ....................... 25
  Synchronous BFS Algorithm .............. 25

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF FIGURES

4.1 View of NS ................................................................. 19
4.2 C++ and OTcl: The Duality ................................. 20
4.3 TCL Interpreter with Extensions .......................... 20
4.4 NAM ................................................................. 21
4.5 A Simple Simulation Example ......................... 23
4.6 Scheduler ......................................................... 24
4.7 Structure of Packet ............................................. 24
5.1 Alphabfs Header File: alphabfs.h .................... 27
5.2 Linkage between Tcl and C++ ......................... 28
5.3 Function ‘command’ and ‘recv’ ......................... 29
5.4 packet.h ......................................................... 30
5.5 ns_packet.tcl ................................................ 30
5.6 Procedure ‘count_safe’ in alphabfs.tcl .............. 31
5.7 A Screenshot of BFS in NAM ......................... 32
5.8 Procedure ‘recv_forward’ in asyncbfs.tcl .......... 34
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Dr. Ajoy K. Datta for his direction throughout the duration of this work. He has had a major role in shaping this thesis. From him, I have learnt most of all to be careful in my thesis. He has also been a great friend and mentor, and I have enjoyed our discussions very much, on both technical and non-technical matters. I would also like to thank Dr. John Minor, Dr. Wolfgang Bein, Dr. John Wang and Dr. Yingtao Jiang for serving on my committee. My special thanks goes to Dr. Sebastien Tixeuil (University of Paris, Sud, France) whose great ideas helped me start and improve my thesis. And I want to thank Mr. Chris Ulmer (Omnilux, Inc.) for revising all I had written.

I would like to thank all the faculty of the Computer Science Department of the University of Nevada, Las Vegas for their help and support during my graduate studies. This thesis is dedicated to my family. They have supported and encouraged me throughout my education.
CHAPTER 1

INTRODUCTION

In this chapter, we discuss the aims and goals of this thesis. We will briefly introduce the synchrony in the network and the distributed synchronizer to solve synchrony problems. We will also introduce Network Simulator, a tool used during our research. Section 1.4 outlines the remainder of this thesis.

Synchrony and Distributed Synchronizer

Distributed algorithms are used in a wide range of applications from VLSI chip design to LAN and Internet systems. Among these algorithms there are two types: synchronous algorithms and asynchronous algorithms. Algorithms for synchronous networks are easier to design, debug, analyze and understand than similar algorithms for asynchronous networks, because the possible behaviors of a synchronous system are more restricted than the possible behaviors of an asynchronous system [1]. However most real systems, like Internet, are at least somewhat asynchronous which makes their behaviors more difficult to model and analyze. Consequently, it is desired and would be helpful to develop a general simulation technique for transforming an algorithm for synchronous networks into an algorithm for general asynchronous networks. This technique will allow the user to design an algorithm for a synchronous network, analyze it, and then use the simulation technique to transform the synchronous algorithm into an asynchronous one and implement it in the asynchronous network.

This general approach for handling asynchrony is referred to as a distributed synchronizer. The concept of synchronizer was first introduced by Baruch Awerbuch in 1985 [2]. With the synchronizer methodology, efficient synchronous algorithms can
be easily implemented into actual asynchronous networks. This thesis will discuss several different types of synchronizers, including the Simple synchronizer, Alpha synchronizer, and Beta synchronizer.

Implementation in the Network Simulator

NS (Network Simulator) is an object-oriented, discrete event driven network simulator developed at USC/ISI written in C++ and OTcl (Tool Command Language with Object-Oriented extensions developed at MIT) [5]. NS is primarily useful for simulating local and wide area networks in an asynchronous environment. It produces one or more text-based output files that contain detailed simulation data. The data can be used for simulation analysis or as an input to a graphical simulation display tool, called NAM (Network Animator).

This thesis involves implementing some synchronous algorithms combined with some synchronizers and some asynchronous algorithms in NS. In the NS environment, by embedding our code and running it, we can get a visible topology of the network in NAM. We can also monitor the status of the real-time status of the nodes and links and the result of the computation of each node at any specified time. From the output trace file, we can also test and analyze the efficiency of the algorithm, even improve and optimize the algorithms.

NS is widely regarded as one of the best academic network simulators developed to date, yet no one has implemented the distributed algorithms in NS. Because of the difficult nature of modeling synchronous algorithms, this thesis addresses this challenge and describes results from a successful use of NS on synchronous algorithms.

Related Work

Several strategies were developed to solve the problem with asynchrony. The ABD (asynchronous Bounded-delay) synchronizer of Tel et al is used in ABD network. But
in this kind of network the execution of an event in a process takes zero time units; each process has a clock to measure physical time; the physical time between the sending and the receipt of a message is bounded [15]. The ABD synchronizer cannot be used in the fully asynchronous network. In 1985, Baruch Awerbuch designed Alpha, Beta, and Gamma synchronizers [2]. Some improved synchronizers have been suggested under various restrictions, e.g., on the network topology or the synchrony assumptions, for example, by Peleg and Ullman [10].

Our Contributions

No work has been reported on implementing distributed algorithms in network simulator software, such as NS. We will discuss the suitability and limitations of implementing a synchronizer algorithm in NS. We also reported some bugs of NS to NS developers.

One might argue that in the real asynchronous network in order to achieve an optimal asynchronous algorithm, it is necessary to program it directly in the environment in which it to be run. However, in spite of the overheads of the synchronizer, asynchronous algorithm transformed from synchronous algorithm by combining with synchronizer are sometimes more efficient than any regular asynchronous algorithm. This is mainly because it is very hard to analyze and monitor the behaviors of an asynchronous network, in which sometimes it is difficult to get a satisfied solution for a problem if we design the asynchronous algorithm directly in the environment. Even though many efficient synchronous algorithms have been developed, they are lacking in implementation because there exists no fully synchronous network for testing and using them.

This thesis will present the code for some efficient synchronizers in NS that enable any synchronous algorithm to run in any asynchronous network environment. Meanwhile we analyze the differences among different synchronizers and their efficiencies
as well as the synchronous algorithms from the output of NS and NAM.

Outline of the Thesis

The remainder of the thesis is organized as follows: In Chapter 2, we will give some general definitions for distributed systems and distributed algorithms and introduce the two models of the networks used in the thesis: synchronous model and asynchronous model, along with the definition of distributed synchronizer. Chapter 3 introduces the detail of the synchronizer algorithms, different versions of the synchronizer. NS, the utility we will use in our work as a platform for our algorithm running, is introduced in Chapter 4. As the applications of synchronizers the leader election and the construction of breadth-first search tree will be considered in Chapters 5 and 6. In Chapter 7, we also write a manual of how to design the code of distributed algorithm in NS and how to compile and run it. Conclusions will be discussed in Chapter 8.
CHAPTER 2

PRELIMINARIES

In this chapter we present a number of definitions for different models of the distributed system, and we give the topology and programming notations used in this thesis.

Distributed Systems

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages [8, 1, 11]. A network of interconnected computers, such as the ARPA net, is a distributed system [13]. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process [7, 4]. This very general definition encompasses a wide spectrum of modern day computer systems, from a VLSI chip, to a tightly coupled shared memory multiprocessor, to a local-area cluster of workstations to the wide-area communication networks [14].

One of the main parameters of this spectrum is the coupling level of different processors in a distributed system, which may vary significantly from one application to another.

1. In a tightly coupled system (e.g., a parallel machine), the processors typically work in tight synchrony. They share the computer bus, the clock, memory and peripheral devices, and have very fast and reliable communication mechanisms
between them. The processors use map memory system calls to gain access to regions of memory owned by other processes. Processes may exchange information by reading and writing data in these shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.

2. In contrast, in a loosely coupled distributed system (e.g., Internet), the processors are more independent and free. There is no shared memory or global clock and instead, all processors have their own memory. The processors communicate with each other through various communication lines, such as high-speed buses or telephone lines.

Definition 2.0.1 (Distributed System) A distributed system is an undirected connected graph, \( S = (V, E) \), where \( V \) is a set of nodes (\(|V| = n\)) and \( E \) is the set of edges. Nodes represent processors, and edges represent bi-directional communication links.

Another significant characteristic of a distributed system is that it is non-uniform. For instance, the processors participating in it may be (and often are) physically distributed in different geographic locations. All processors seem to be identical, but at least there should be something different among them, such as user ID.

This thesis focuses on systems at the more loosely coupled distributed system. We will use “nodes” and “processors” interchangeably in this thesis. Each processor is modeled as a state machine with state set \( Z_i \). Each state of processor contains the local variables and two sets of messages: \( outbuf[l] \) and \( inbuf[l] \), for every \( l \), \( l \) stands for particular link. A communication link \((p, q)\) exists iff \( p \) and \( q \) are neighbors.
Distributed Algorithms

In the previous chapter, we can see the reason for the use of distributed system. Consequently, the need to program these systems arises. The term of distributed algorithm covers a large variety of algorithms for a wide range of applications including telecommunications, distributed information processing etc. Obviously, it is important to write distributed algorithms correctly and efficiently. However, because behavior of the loosely coupled systems is typically harder to grasp and analyze, designing such algorithms can be extremely difficult. Here we discuss three fundamental difficulties.

1. Lack of knowledge of global state. Each node in the distributed system can only access to its own local state and not to the global state of the system. So the node cannot make a decision from the global state. In this case, we should use massage-passing model. Each node receives message containing other nodes information from other nodes. Then the node can perform the local computations and configurations based on the information received, finally, make a decision. The state of the communication is not directly viewable for the node so the nodes can learn this only by analyzing the messages received from other nodes.

2. Asynchrony. Compared to centralized (uniprocessor) computer systems, in distributed system, each node operates very independently. There is no central control unit to coordinates all the nodes to proceed concurrently. And there is no global time to make all nodes to work in a bounded time at each step.

3. Non-determinism. Each execution of a distributed system is usually non-deterministic, because of the possible differences in execution speed of the system components.

It is common to concentrate on two kinds of algorithms (synchronous algorithms and asynchronous algorithms) in extreme network models: the fully synchronous...
model and the totally asynchronous one. To avoid confusion we shall refer to algorithms in a synchronous system as synchronous algorithm, and algorithms in an asynchronous system as asynchronous algorithm.

Synchronous Networks

In a fully synchronous distributed network system, it is assumed that all delays are bounded; the operation of each process takes place in a sequence of discrete steps, called round, or pulse. In a round a process first sends zero or more messages, then receives zero or more messages, and finally performs local computations, then change the state. Messages are received in the same round as that in which they are sent; i.e., if node $p$ sends a message to node $q$ in its $i$th round, then the message is received by node $q$ in its $i$th round and before node $q$'s computation in that round.

The rounds or pulses can be thought of as the ticks ("pulses") of a global clock. Each node has an access to this clock. Computation is performed at the clock pulses, and a message sent at one pulse is guaranteed to be received before the next pulse. This globally synchronized operation is also referred to as a lockstep operation [6]. It is easy to implement lockstep operations if clocks are available and an upper bound on transmission delay is assumed.

Here we must mention, generally synchronous algorithm is not realistic and achievable in practical distributed network system, such as Internet, but is very convenient for designing synchronous algorithms, since a synchronous algorithm does not face too much uncertainty. Once a synchronous algorithm has been designed for this ideal timing model, it can be simulated with synchronizer methodology in more realistic model, as we shall see later. Here we present the definitions of the synchronous process and system [15].

**Definition 2.0.2** A synchronous process is a four-tuple $p = (Z, I, MG, \tau)$ where
1. $Z$ is a set of states,

2. $I$ is a subset of $Z$ of initial states,

3. $\mathcal{MG} : Z \rightarrow \mathcal{PM}$ is a message generation function, and

4. $\vdash$ is a relation on $Z \times \mathcal{PM} \times Z$. We write $(c, \mathcal{M}) \vdash d$ for $(c, \mathcal{M}, d) \in \vdash$.

A process starts its computation in an initial state $c_0 \in I$. When the process starts a round in state $c \in Z$, it sends the collection $\mathcal{MG}(c)$ of messages. Before the next round, it receives the collection of messages, say $\mathcal{M}$, sent to it in this round and enters a state $d$ such that $(c, \mathcal{M}) \vdash d$.

**Definition 2.0.3** The synchronous network system consisting of (synchronous) processes $\mathcal{P} = (p_1, \ldots, p_n)$ (where $p_i$ is the process $(Z_i, I_i, \mathcal{MG}_i, \vdash_i)$) is a transition system $S = (C, \rightarrow, I)$, where

1. $C = \{(c_1, c_2, \ldots, c_n) : \forall i, c_i \in Z_i\}$,

2. $\rightarrow$ is the relation defined by $(c_1, \ldots, c_n) \rightarrow (d_1, \ldots, d_n)$ if, for each $p_i \in \mathcal{P}$; $(c_i, \{m \in (\bigcup_j \mathcal{MG}_j(c_j)) : m \text{ has destination } p_i\}) \vdash_i d_i$;

3. $I = \{(c_1, c_2, \ldots, c_n) : \forall i, c_i \in I_i\}$

A synchronous computation of a system is a maximal sequence $\gamma_0, \gamma_1, \ldots$ of configurations, such that $\gamma_0$ is an initial configuration and, for all $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$.

The message complexity of this computation is the number of messages exchanged, and the time complexity equals the index of the last configuration (assuming the computation is finite).
Asynchronous Networks

The synchronous network is introduced in the previous section, which is an ideal model. But it is not achievable to get a fully synchronous model. In this section we will present a more realistic model, asynchronous network. A system is said to be asynchronous if there is no fixed upper bound on the message delivery time or no fixed upper bound on the gap between two consecutive steps of a processor [1]; all that can be said is that each step may take an arbitrarily long time. An example of an asynchronous system is the Internet, where messages (for instance, email) can take less than one second or very long time to arrive. There are usually upper bounds on message delays and processor step times, but sometimes these upper bounds are very large and can change over time. An algorithm that does not depend on the timing bound is named an asynchronous algorithm.

In an asynchronous model, there is no global memory or global clock shared by the processors; algorithms are message driven. Message sent from one node to another node arrives in a finite but unpredictable time. One cannot say that a message was not sent from a neighbor by a certain time or that the message was lost just according to the time elapsed. All messages cannot be guaranteed to arrive in the same order as it was sent due to different transmission speeds.

Synchronizer

From the previous two sections, we can say synchronous algorithms are easier to design, reason, and test than asynchronous algorithms, but it is not achievable to build a fully synchronous network. The behavior of asynchronous network is typically harder to handle and analyze, but asynchronous network is easier and realistic to build. So it is desirable to have a methodology for transforming an efficient algorithm for synchronous network into an asynchronous algorithm and use it in the asynchronous network. This general approach for handling asynchrony is known as a
The synchronizer should enable any synchronous algorithm to run on any asynchronous network. Given a synchronous algorithm \( \Pi_S \) and a synchronizer \( \nu \), it is possible to combine \( \Pi_S \) with \( \nu \) to generate a protocol \( \Pi_A = \nu(\Pi_S) \) that can be implemented in an asynchronous network. The simulation is correct that \( \Pi_A \)’s execution in a synchronous network should be “similar” to \( \Pi_S \)’s execution in a synchronous network [9].

The combined protocol \( \Pi_A \) is composed of two main components, which we refer to as the “synchronous component” and the “synchronizer component”. Each of these components has its own local variables and message types at every node. “The synchronous component” consists of the local variables and the messages of the synchronous protocol \( \Pi_S \), whereas the “synchronizer component” consists of local synchronization variable and synchronization messages.

A process is said to simulate a certain round when it sends the messages of that round. Each node is supposed to generate a sequence of “local rounds” = 1, 2, … for itself. These rounds are supposed to simulate the ticks of the clock in the synchronous setting. That is, under the combined protocol \( \Pi_A \), each node should participate during any simulated round same as to participate in each round of the synchronous algorithm \( \Pi_S \).

For all synchronizers in this thesis each node receives all messages of the synchronous algorithm, sent to that node from its neighbors in the current round; when this is the case all messages are processed and the node goes to the new state then the next round is started. All the actions are event driven, not based on the use of clocks for fully asynchronous networks. Because all asynchronous algorithms can be written in message-driven form, each process must receive a message to trigger the next round, which implies that at least \( N \) messages are needed to simulate each round.
Our Model and Topology

Here, we present an implementation of distributed algorithms in NS software in one network mode, asynchronous network instead of synchronous network. The asynchronous network is a generally point-to-point communication network in arbitrary topology, described by an undirected communication graph \((V, E)\) where the set of node \(V(1, 2, \ldots, n)\) represents nodes of the network and the set of Links \(E((1, 2), (2, 3), \ldots, (m, n))\) represents bi-directional communication channels operating between them. No memory is shared by the nodes, and each node has a distinct ID. Each node processes messages received only from its neighbors, performs local computations, and sends messages only to its neighbors. Local computation is assumed to take negligible time compared to message transmission. Each message sent by a node to its neighbor arrives within some finite but unpredictable time. Experiments run on these implementations have shown all synchronizers to function correctly and terminate successfully in stable networks without link failures.

Programming Notations

During a computation step, one of the following actions (local step) occurs on at least one process \(p\): (1) \(p\) receives a message; (2) \(p\) executes some internal actions; (3) \(p\) sends at least one message. Each node has a local state, which is defined by the ID of the node and some variables of the program. We define the global state as the union of the local state of all nodes.

In our actual program written in C++ and OTcl, we declared some global variables. We do so in order to make the program run correctly, since we simulate the algorithms in single software and in a single machine. But a process in an individual node can have access (read/write) only to its own variables including the ones received from its neighbors.
CHAPTER 3

SYNCHRONIZER ALGORITHMS

In this section we shall consider several synchronizers for fully asynchronous networks. All synchronizer implementations we describe are "local", in the sense that they only involve synchronization among neighbors in the network rather than among arbitrary nodes. The complexity of these synchronizers is measured by the following four parameters:

- $M_{\text{init}}$, the message complexity of the initialization round
- $T_{\text{init}}$, the time complexity of the initialization round
- $M_{\text{round}}$, the message complexity of simulating each round
- $T_{\text{round}}$, the time complexity of simulating each round

A synchronous algorithm with message complexity $M$ and time complexity $T$ can be simulated using $M_{\text{init}} + M + M_{\text{round}} T$ messages in $T_{\text{init}} + T_{\text{round}} T$ time units [15].

We present the complexities of the different synchronizers in Table 3.1.

A Simple Synchronizer

First we introduce the simplest synchronizer called "Simple synchronizer". Each process sends exactly one message to each neighbor in each round. If the synchronous

<table>
<thead>
<tr>
<th>Synchronizer</th>
<th>$M_{\text{init}}$</th>
<th>$T_{\text{init}}$</th>
<th>$M_{\text{round}}$</th>
<th>$T_{\text{round}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>0</td>
<td>Diam</td>
<td>2$</td>
<td>E</td>
</tr>
<tr>
<td>Alpha</td>
<td>0</td>
<td>Diam</td>
<td>2$</td>
<td>E</td>
</tr>
<tr>
<td>Beta</td>
<td>$\Theta(N \log N +</td>
<td>E</td>
<td>)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

Table 3.1: Complexity of Synchronizers
algorithm does not send a message in some round, the process still needs to send a message to each neighbor. If the synchronous algorithm needs to send more than one message in each round, all messages will be packed in one message. In every round, the process first sends a message to each neighbor, and when it receives a message from each neighbor for this round, then, it processes the messages and changes to a new state and starts a new round. See algorithm 3.0.1.

Algorithm 3.0.1 SimpleSynchronizer

```
var:  
round(p) : int
state(p) : \mathbb{Z}_p
missing(p): int /* initialize to |Nbr(p)| number of neighbors */
next.missing(p): int /* initialize to |Nbr(p)| number of neighbors */

/* initialize the algorithm */
begin  /* the node can start spontaneously or be activated by the 1st msg*/
  state(p) = initial state; round = 1 /* initialize to 1 */
  for all q \in Nbr(p) do
    begin
      \begin{align*}
        M_p[q] &:= \{ m \in MG_p(state(p)) : m \text{ has destination } q \}; \\
        \text{send } < \text{pack}, M_p[q], round(p) > \text{ to } q
      \end{align*}
    end
  end

/* A message < pack, (m_1, \ldots, i) > has received */
begin receive and store message;
  if i = round(p) then missing(p) := missing(p) - 1
  else next.missing(p) := next.missing(p) - 1;
  while missing(p) = 0 do
    begin  /* M is the set of received messages of this round */
      state(p) := d_p with (state(p), M) \rightarrow d_p;
      round(p) := round(p) + 1;
      missing(p) := next.missing(p);
      next.missing(p) := |Nbr(p)|;
      for all q \in Nbr(p) do
        begin
          \begin{align*}
            M_p[q] &:= \{ m \in MG_p(state(p)) : m \text{ has destination } q \}; \\
            \text{send } < \text{pack}, M_p[q], round(p) > \text{ to } q
          \end{align*}
        end
    end

'pack' is the packed message of the original simulated algorithm. Process p can
```
only receive the messages of current round $i$ and the following round $i+1$. All messages of earlier round have already been received and processed, and no neighbors of $p$ can send messages of round higher than $i+1$ before $p$ send its messages of round $i+1$. Therefore $p$ only maintains knowledge of two consecutive rounds. Caution: process $p$ may receive all messages of the following round before it receives all messages of current round, but we cannot process the messages of the next round prior to the current round. For actually coding, the messages of the last round of the algorithm must be handled specially in order to get correct result and terminate successfully. We will show this in the later chapters.

$M_{\text{init}} = 0$ since there is no additional message needed for initialization. $T_{\text{init}} = \text{Diameter}$, since maybe only one node starts spontaneously. If the last simulation of round $i$ occurs at time $t$, all messages of round $i$ have been received at time $t+1$. Consequently $T_{\text{round}} = 1$. Each round requires the exchange of exactly $2|E|$ messages. Consequently, $M_{\text{round}} = 2|E|$.

The generality of the simple synchronizer demonstrates that asynchronous network can simulate all synchronous algorithms. Hence, all problems, which can be solved by synchronous algorithms in synchronous networks, can also be solved in asynchronous networks.

Safe Synchronizer

In the Simple Synchronizer each process communicates with each of its neighbors in every round, leading to a message complexity of $\Theta(|E|)$ for the simulation of each round. It is impossible to reduce the time complexity of the Simple Synchronizer algorithm significantly, but it is possible to reduce the communication complexity if communication takes place via a sub-topology.

Some of the basic concepts, such as the notation of “safe” and “pulse” are quite similar to those mentioned in [12]. Node $p$ is said to be safe with respect to its current
round some time after the receipt of all messages sent by node $p$ in that round [15]. An acknowledgement is sent for each message of the synchronous algorithm, and when node $p$ has received an acknowledgement for each message, it becomes safe. Observe that the acknowledgments do not increase the message complexity, and each node learns that it is safe in a constant time. The synchronizer must now provide the inter-process communication by which a process learns that all its neighbors are safe. When node $p$ has verified that all its neighbors are safe for round $i$, it performs the local computations of round $i$, changes the state, and starts the next round $i+1$. Here we present two safe synchronizers in the two following subsections.

1. The Alpha Synchronizer

The most straightforward implementation of safe synchronizer is the Alpha synchronizer, which is very similar to the Simple synchronizer. A node $p$ sends messages to all neighbors or some of the neighbors for a certain round. (For some synchronous algorithms, the node does not need to send message to each neighbor, or no message to any neighbor at all for a certain round). If node $p$ receives an acknowledgement for each message or node $p$ has no message to send to any neighbor in the current round, we say the node $p$ is safe for the current round, then node $p$ send an safe message to each neighbor. A process can start next round when the process is safe itself and receive a safe message from each neighbor for the current round.

It's clear that $O(|E|)$ messages are sent by the synchronizer for each round. If the last simulation of pulse $i$ occurs at time $t$, all messages of pulse $i$ will be received by time $t+1$ and all acknowledges will be received by time $t+2$ and all safe messages will received by time $t+3$. Consequently $T_{round} = 3$. From the analysis above, we cannot get better solution from Alpha synchronizer than Simple synchronizer.

2. Beta Synchronizer

For Beta synchronizer we assume the existence of a rooted spanning tree $T$ in the network. A node sends 'tree safe' message to its parent in the spanning tree if
this node is safe itself and has received a 'tree safe' message from each child in its sub-tree. When the root of the spanning tree has received a 'tree safe' message from each of its children and the root is safe itself, the root sends 'new round' message to all the nodes via the edges of the tree to let each node know all nodes are safe for current round. When a node receives 'new round' message, it processes the messages received in the current round, changes state, and starts the new round. The nodes do not need to send the message to the neighbors in some certain rounds if there is no new message to send. This reduces the communication complexity.

This synchronizer needs an initialization phase to create a spanning tree. Once we got a spanning tree, we can repeat to use it later. We do not consider the communication complexity of creating the spanning tree when we evaluate the communication complexity of the synchronizer, since we assume there is a spanning tree already in the network. It is preferred to generate a spanning tree of small height for broadcast. In our program we just created a regular spanning tree for the beta synchronizer. But we also wrote three programs to create Breadth First Search tree by using synchronous algorithm combined with $\alpha$ and $\beta$ synchronizers and a purely asynchronous algorithm.

In addition to the acknowledgement, each round requires 'tree safe' and 'new round' messages to be sent along the edges of the spanning tree, to a total of $O(N)$ control messages. The time complexity is $\Omega(N)$ in the worst case.
CHAPTER 4

NETWORK SIMULATOR

In this chapter we simply introduce NS (Network Simulator) and some basic features of it. We give a simple simulation sample, from which one can get the basic idea of programming in NS.

Overview

NS (Network Simulator) [5] is an object-oriented, discrete event driven network simulator that simulates variety of IP networks. NS is primarily useful for simulating local and wide area networks. It implements some network protocols, like TCP and UDP, and some implementations like FTP, Telnet, some routing algorithms like Dijkstra, and so on. It produces one or more text based output files that contain detailed simulation data. The data can be used for simulation analysis or as an input to a graphical simulation display tool, called Network Animator (NAM). NS can help to debug problems in a controlled environment.

The NS has over 100K lines of C++, 70K lines of OTcl, 30K lines of test suite and 20K lines of documentation. It is quite difficult for new user to start to use NS, because there are few good and clear manuals. Even though there are some documentations such as The NS MANUAL, which the developer explain the simulator very deeply and professionally, but NS is updated day by day and most of the documentations are still out-of-date and not suitable for the new users.

Even though NS is not polished and finished software (with many bugs and limited features), it still provides the best substantial support for simulation of the algorithms over wired and wireless networks. It increases confidence in results of your algorithms.
As shown in Figure 4.1 [3], NS is an Object-oriented Tcl (OTcl) script interpreter with NS simulator library. The simulator library contains the Event Scheduler Objects, Network Component Objects and Network Setup Helping Modules. To implement an algorithm in NS, user should use OTcl script language to create an network topology by using Network Component Object and Network Setup Helping Modules, create the event scheduler by using Event scheduler Objects, then the source of traffic will know when to send packets, stop or cancel events from the setting of the scheduler.

![Diagram of NS simulation process](image)

Figure 4.1: View of NS

NS is written in C++ with OTcl as front-end. NS supports class hierarchy both in C++ and OTcl. For each task, every user must want your program to run fast and configure conveniently. No programming language meets the two requirements very well. C++ is fast to run but slower to change, whereas OTcl run slower but can be changed very quickly and conveniently. So, NS uses C++ for detailed protocol implementation and OTcl for configuration. Meanwhile, NS provides a linkage technique to link the two languages. The compiled C++ objects are made accessible to the OTcl interpreter through an OTcl linkage. In this case OTcl can access and control the C++ objects. It is also possible to declare a member function in C++,
and define this function in OTcl. The user can invoke an OTcl command in C++ to call the function defined in OTcl. Figure 4.2 shows an object hierarchy example in C++ and OTcl.

![Object Hierarchy Example](image)

**Figure 4.2: C++ and OTcl: The Duality**

Figure 4.3 shows the general architecture of NS. OTcl is an extension of Tcl. Tccl is used for linkage between C++ and OTcl. The whole thing together makes NS. NAM (Network Animator) is a very nice graphical simulation display tool, which can graphically present information such as run-time status of the nodes and links, even the result of computation at any time during the execution of the program.

Figure 4.4 you can see a screenshot of a NAM window where the most important functions are being explained. Once a trace file has been loaded into NAM, an
animation window will appear. We can save the current network layout to a file or print the current network layout.

Features of NS

In this section, we present a very simple simulation example and introduce two important components: Event Scheduler and the Packet. Our goal of this section is to show some features of NS and basic rules to design OTcl code in NS. Please understand that we cannot give you all features of ns here, or a full Tcl tutorial. I'd like to mention, if you do not want to add any new functionality to ns, you do not need any C++ programming language. In this thesis, we will add the complicated distributed algorithms into NS, so C++ is essential.
1. A simple simulation example

Here we create a file called "sample.tcl". See comments in Figure 4.5 for the explanation. This script file creates three nodes topology and implements ftp application. $ns at time "event" invokes the discrete event scheduler. The scheduler schedules the execution at a specified time.

2. Event Scheduler

In this section we talk about the discrete event schedulers of NS. As described in Section 4.1, the main users of an event scheduler are network components that simulate packet-handling delay or that need timers. In Figure 4.5 we can see the event scheduler schedules simulation events, such as when to start an FTP application, when to finish a simulation. The scheduler runs by selecting the next earliest event, executing it to completion, then selecting the next. The simulator is single-threaded, so it is unnecessary to worry about the deadlock, fairness or starvation. Figure 4.6 shows how the event scheduler works.

3. Packet

A NS Packet is composed of a stack of headers (see Figure 4.7). When a NS simulator is created, the headers are all registered. In actual network, the packet should carry some data, but NS developers think it is nonsense to carry data around the simulated network. Even though NS allows the packet to carry data, but only few applications and agents support it. Fortunately, we have an approach to attach data to packet by adding one more header and writing the data into this header.
#create a simulator object instance and assign it to variable ns.
set ns [new Simulator]

#open a trace file is given for the output of ns,
#which can be use for analysis later.
set f [open out.tr w]
$ns trace-all $f

# open a nam-trace file, which will be used as the input file of NAM
set nf [open out.nam w]
$ns namtrace-all $nf

#define the colors for the flows for different kind of messages in NAM
$ns color 1 black
$ns color 2 red

#create some nodes.
set n0 [new Node]
set n1 [new Node]
set n2 [new Node]

#create links, set bandwidth, delay and type of the packet queue
$ns duplex-link $n0 $n1 2mb $20ms DropTail
$ns duplex-link $n1 $n2 1mb $10ms RED

#setup TCP agent and attach it to n0
set tcp [new Agent/TCP]
$ns attach-agent $n0 $tcp

#set the color of ftp flow to 1 (black)
$tcp set fd_ 1

#setup TCPSink agent and attach it to n2
set sink [new Agent/TCPSink]
$ns attach-agent $n2 $sink

#connect the two agents
$ns connect $tcp $sink

#setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp

#define the procedure called "finish". proc finish {} { global ns nf
$ns flush-trace
close $nf
exec nam out.nam &
exit 0
}

#Schedule events for FTP
$ns at 1.0 "$ftp start"
$ns at 2.0 "$ftp stop"
#call the procedure "finish"
$ns at 1.5 "finish"
$ns run

Figure 4.5: A Simple Simulation Example

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Data Path

Event Scheduler

Packet

header
data (optional)

Packet

header

data (optional)

cmn header

ip header
tcp header
rtp header
trace header

uid : unique id
ptype : pkt type
size  : simulated pkt size
ts  : time stamp

Figure 4.6: Scheduler

Figure 4.7: Structure of Packet
CHAPTER 5

BFS IN NS

The Breadth First Search (BFS) tree provides a very efficient structure to use as a basis for broadcast communication. In this chapter, we will first introduce the definition of BFS. Then we will introduce a purely synchronous algorithm to build BFS tree and then we combine the synchronous algorithm with $\alpha$ and $\beta$ synchronizers. We will also present an asynchronous algorithm and then evaluate the communication complexity. We will give an example in detail for implementation of the synchronous BFS algorithm combined with $\alpha$ synchronizer in NS.

Definition 5.0.4 (Breadth First Search tree) The breadth first search (BFS) tree of a connected network $G$ is a spanning tree that given a root $r_0$, for any node $p$, the path leading from the root to $p$ in the tree is minimum-hop path in $G$.

Synchronous BFS Algorithm

In a synchronous BFS algorithm, each node will send messages to each of its neighbors in exactly one round. The root starts to form level 0, and sends level messages to all its neighbors in round 1. If node $p$ receives one or more message in round $i$, $p$ knows it is on level $i$, and then sends level messages to its neighbors in round $i+1$. Node $p$ will take the node from which it received the first message in round $i$ as its parent.

The choices of the parent imply that the resulting tree is a breadth-first-search tree. The communication complexity is $2|E|$, since each node sends messages exactly
once. And the time complexity is \( O(N) \), because the tree may also be a chain, in this case, after round N-1 each node has determined its parent.

**Combination with Alpha Synchronizer**

The synchronous BFS algorithm mentioned in the previous section cannot be directly implemented in a fully asynchronous network. But with synchronizer methodology, we can simulate it. See Section 3.2.1 for reference to the synchronizer. We set level 0 to root, and the levels of the other nodes as undefined. The combined algorithm is based on the comparison algorithm, such that at each round every node computes the levels received from all neighbors and updates \( min\_level_{id} \) (minimum level of itself). Then it sends \( min\_level_{id} + 1 \) to each neighbor. At the last round (diameter round) the minimum level is the final result (we assume each node knows the diameter). For optimized algorithm, node \( p \) will send \( min\_level_p + 1 \) message to neighbor \( q \) at round \( i+1 \), only if \( min\_level_q \) (where node \( p \) received \( min\_level_q + 1 \) message from \( q \) in round \( i \)) is greater than \( min\_level_p + 1 \). Otherwise node \( p \) just sends safe message to neighbor \( q \) at round \( i+1 \).

In this section we will present the steps in detail to implement the combination of synchronous algorithm with a synchronizer in NS. Our program was designed on the Agent level of NS.

**I. The header file**

First we create a new header file ‘alphabfs.h’ (see figure 5.1) and declare a new data structure for the new Alphabfs packet header to carry the relevant data.

‘packet_type’ will be set in any of the following conditions:

- It is set to ‘explore’ if node \( p \) at level \( i \) is going to explore level \( i+1 \)
- It is set to ‘ack’ if node \( p \) is going to send an acknowledgement to node \( q \) who sent an ‘explore’ message to \( p \)
- It is set to ‘safe’ to mean that it is a ‘safe’ message of the synchronizer
we need to declare the class ‘AlphabfsAgent’ as a subclass of the class ‘Agent’. If node $p$ needs to communicate with each neighbor, say five neighbors, we should attach five ‘AlphabfsAgent’s to node $p$, and each agent connects to one of the agents of each neighbor respectively.

II. The C++ file

Figure 5.2 shows the linkage between Tcl and C++. We need to give the packet a default value of the size; otherwise lots of warning will be generated. This code also binds the variable ‘size’ which can be accessed in Tcl and C++.

In figure 5.3 first we define a function ‘command’ for the sending actions. The function ‘command’ is called when a Tcl command for the class ‘Alphabfs’ is executed, such as ‘$alphabfsAgent($link) explore $level’ (note: alphabfsAgent($link) is an instance of the Agent/Alphabfs, ‘explore’ is a Tcl command). NS will parse the command in ‘command( )’ function, if no match is found, the command will call the ‘command( )’ function for the base class ‘Agent’. Here we set different packet size to

```c
struct hdr_alphabfs {
    int level;
    int pulse;
    char* packet_type;
    // header access methods
    static int offset_; // required by PacketHeaderManager
    inline static int& offset() { return offset_; }
    inline static hdr_alphabfs* access(const Packet* p) {
        return (hdr_alphabfs*) p->access(offset_);
    }
}

class AlphabfsAgent : public Agent {
    public:
    AlphabfsAgent();
    virtual int command(int argc, const char* const* argv);
    virtual void recv(Packet*, Handler*);
};
```

Figure 5.1: Alphabfs Header File : alphabfs.h
different type of packet in order to distinguish the types of messages in the output trace file.

In figure 5.3 we also need to define a function ‘recv’ for the receiving actions. ‘tcl.eval( )’ plays an important role here. For example, we declare a procedure ‘recvexplore’ in the function ‘recv’, ‘tcl.eval(out)’ will call the Tcl interpreter to execute the procedure ‘Agent/Alphabfs instproc recvexplore { }’ defined in the Tcl file ‘alphabfs.tcl’. At last, the node frees the received packet and waits for the next message.

III. Modify the ns source files

Now we copy ‘alphabfs.cc’ and ‘alphabfs.h’ into ‘ns-2’ directory, add ‘alphabfs.o’ to ‘Makefile’, and then edit the file ‘packet.h’ to register the new header of the packet. See Figure 5.4.

To make the two new C++ files compiled, we must do ‘make depend’ before we do ‘make’. Then we also need to add a line to the file ‘ns/tcl/lib/ns-packet.tcl’ to register the new header for OTcl. See Figure 5.5.
int AlphabfsAgent::command(int argc, const char* const* argv) {
    if (strcmp(argv[1], "explore") == 0) {
        size_ = 256; // packet size
        fid_ = 1; // flow color defined in OTcI
        // Create a new packet
        Packet* pkt = allocpkt();
        // Access the Asyncbfs header for the new packet:
        hdr_.alphabfs* hdr = hdr.alphabfs::access(pkt);
        // Set the temp next level for the children
        hdr->level = atoi(argv[2]);
        // Store the pulse in the 'pulse' field
        hdr->pulse = atoi(argv[3]);
        // Type of packet
        hdr->packet_type = "explore";
        // Send the packet
        send(pkt, 0);
        // return TCL_OK, so the calling function knows that the
        // command has been processed
        return (TCL.OK);
    }
    // If the command hasn't been processed by SyncAgent::command,
    // call the command() function for the base class
    return (Agent::command(argc, argv));
}

void AlphabfsAgent::recv(Packet* pkt, Handler*) {
    char out[30];
    // obtain a reference to the instance of NS to access other methods.
    Tcl& tcl = Tcl::instance();
    // Access the IP header for the received packet
    hdr_ip* hdr = hdr.ip::access(pkt);
    // Access the Alphabfs header for the received packet
    hdr.alphabfs* hdr = hdr.alphabfs::access(pkt);
    if (hdr->packet_type == "explore") {
        // create a new acknowledge packet to send
        Packet* pckt = allocpkt();
        hdr.alphabfs* hdrack = hdr.alphabfs::access(pckt);
        hdrack->packet_type = "ack";
        hdrack->pulse = hdr->pulse;
        send(pckt, 0);
        // call procedure 'recvexplore' defined in file 'alphabfs.tcl'
        sprintf(out, "%s recvexplore %d %d %d", name(),
                hdr->level, hdr->pulse,
                hdr->pulse >> Address::instance().NodeShift[1];
        tcl.eval(out);
    }
    // Discard the packet
    Packet::free(pkt);
}

Figure 5.3: Function 'command' and 'recv'
enum packet_t {  
    PT_TCP,  
    PT_UDP,  
    ...  
    // insert new packet types here  
    PT_ALPHABFS, // packet protocol ID for our alphabfsagent  
    PT_NTYPE // This MUST be the LAST one  
};

class p_info {  
public:  
    p_info() {  
        name_[PT_TCP] = "tcp";  
        ...  
        // 'Alphabfs' will be shown in the trace file  
        name_[PT_ALPHABFS] = "Alphabfs";  
        name_[PT_NTYPE] = "undefined";  
    }  
    ...  
}

Figure 5.4: packet.h

Foreach prot {  
    AODV  
    ...  
    ALPHABFS  
} {  
    add-packet-header $prot  
}

Figure 5.5: ns_packet.tcl

We also need to modify 'ns/tcl/lib/ns-defult.tcl' to set a default packet size.

IV. The Tcl file

We are not going to present the full Tcl code here, but just show how to define 'count_safe' procedure in figure 5.6, which is called in 'alphabfs.cc'. We declare some global variables here, but each node can only access its own variables. For example, we store all values of the levels to array 'temp', but each node can only access the items of array 'temp' which the index is equal to its own id.

When we run this script Tcl file, we can start Nam (NS Animator). Figure 5.7 shows a screenshot of some run-time information of BFS in Nam. On each node, its
Agent/Alphabfs instproc count_safe { step from} { 
    global ns ack_missing safe_missing ...
    # get its own ID
    $self instvar nodeID
    set nodeID [$$node__id]

    # count the no. of 'safe' msgs received for the current and next rounds
    if { $step == $round($nodeID) } {
        set safe_missing($nodeID) [expr $safe_missing($nodeID) - 1]
    } else {
        set next_safe_missing($nodeID) [expr $next_safe_missing($nodeID) - 1]
    }

    if { $safe_missing($nodeID) == 0 } {
        #compute and update the minimum level of the node here
    }

    if { $round($nodeID) < $diameter } {
        count the number of 'ack's the node should receive for the next round
        set round($nodeID) [expr $round($nodeID) + 1]
        set safe_missing($nodeID) $next_safe_missing($nodeID)
        set next_safe_missing($nodeID) $link_num($nodeID)
        for each neighbor:
            send 'safe' or 'explore' message
    } else {
        #last round
        #output result.
    }
}

Figure 5.6: Procedure 'count_safe' in alphabfs.tcl

temporarily computed level at a specified round can be shown in NAM. The type of flows along the links can be distinguished by different colors, which are defined at the beginning of the program. By checking the output information in NAM, we can find whether our code is efficient or where it might need improvement.

We observed that any two nodes could be at different rounds at a specific time. The round difference between directly connected nodes cannot be larger than 1, but other nodes can be have round differences larger than 1. This is because α synchronizer here
is "local", in the sense that the nodes only involve synchronization among neighbors rather than among arbitrary nodes.

Our original goal is to create a synchronizer as a platform (front-end) where some applications, such as BFS can be directly implemented on top of the synchronizer. Because of the structure and functionality limitations of NS we had to mix the synchronous algorithms and the synchronizer.

Combination with Beta Synchronizer

From the Tcl code in Section 5.2, we can see that each node must send at least one safe message to every neighbor in each round. The Beta synchronizer can improve the previous algorithm. We will only present some of the differences in implementation between the $\alpha$ and $\beta$ synchronizers.

First we create a regular spanning tree $T$ as a sub-topology of the original one for the safe message communications. If node $p$ receives an acknowledgement for each
message of current round, we say node $p$ is safe. If node $p$ receives a safe message from each child in its sub-tree, and node $p$ is safe itself, then node $p$ sends a safe message to its parent and tells him that node $p$ and all nodes in $p$'s sub-tree are safe. If node $p$ is the root of $T$, then it broadcasts a new.round message down to the spanning tree to tell every node to start the next round. If node $q$ receives new.round message, it forwards this message to its children via the edges of its sub-tree. Then it performs the computations, gets the minimum level of itself and then starts a new round.

For $\beta$ synchronizer, each node only maintains knowledge of the number of messages for the current round. It is different from Simple and $\alpha$ synchronizers in which each node can also receive the messages of round $i+1$ during the current round $i$. Since the node sends safe message via sub-topology, it reduces the communication complexity.

Asynchronous BFS Algorithm

In this section, we present an asynchronous BFS algorithm where the communication complexity is lower than the synchronous algorithm combined with $\alpha$ synchronizer. We should not conclude that the synchronizer is useless. In fact, the synchronizer algorithm is implicitly used in the asynchronous algorithm in this section. Comparing the synchronous and asynchronous algorithm, we will see that asynchronous algorithm is much more complicated.

We will create the BFS tree level by level here. Assume the construction of level $l$ is complete; we are going to create level $l+1$. The root (initiator) will broadcast 'forward $l+1$' message down to the tree until it reaches level $l$. If node $p$ on level $l$ receives a 'forward $l+1$' message, then sends 'explore $l+1$' message to those 'unexplored' neighbors (Node $p$ already knows the neighbors on level $l-1$).

Node $q$ only takes the neighbor from which $q$ received first 'explore $l+1$' message as $q$'s parent, and then sends 'accept $l+1$' message to $q$'s parent and 'reject $l+1$' message to other senders of 'explore $l+1$' message.
34

Agent/Asyncbfs instproc recv_forward {next_level from} {
    # my level was just created, it is time to explore the next level
    if { $mylevel($nodeID) == [expr $next_level - 1] } {
        foreach link [array names asyncbfsAgent] {
            if { [lindex $link 0] == $nodeID } {
                if {! [info exists "uplevel_nbr($nodeID [lindex $link 1])"]} {
                    asyncbfsAgent($link) explore $next_level
                }
            }
        }
    } else { ;#forward "forward" down to the tree
        foreach link [array names mychildlink] {
            if { [lindex $link 0] == $nodeID } {
                if {! [info exists "no_forward($nodeID [lindex $link 1])"]} {
                    asyncbfsAgent($link) forward $next_level
                }
            }
        }
    }
}

Figure 5.8: Procedure 'recv_forward' in asyncbfs.tcl

If node p receives at least one 'accept l+1' received, node p will forward this message to p’s parent, otherwise forward 'reject l+1' message. If the node p receives a reject l+1 message from each child, node p knows that its sub tree is complete. For optimization, one should not send 'forward level' message to the child who previously sent a 'reject' message. Figure 5.8 shows the actions when a node receives a 'forward level' message.

Complexity Analysis

We summarize the complexities of different algorithms in this chapter with Table 5.1. From this table we notice that the β synchronizer has a better communication complexity than a synchronizer. The result of the testing in NS also shows the efficiency of β synchronizer. See Table 5.2.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Synch + $\alpha$</td>
<td>$O(Diam \cdot</td>
<td>E</td>
</tr>
<tr>
<td>Synch + $\beta$</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Asynch</td>
<td>$O(</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 5.1: Complexity of BFS Algorithm

<table>
<thead>
<tr>
<th>Network Topology</th>
<th>No. of Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synch + $\alpha$</td>
</tr>
<tr>
<td>No. of Node</td>
<td>No. of Link</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>22</td>
<td>30</td>
</tr>
<tr>
<td>22</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 5.2: Number of Messages of BFS Algorithm

We did not consider the number of messages for $\beta$ synchronizer during the construction of the spanning tree $T$ for safe and pulse message communications. We assumed the spanning tree $T$ has already been created. The number of messages of $\beta$ synchronizer is less than $\alpha$ synchronizers, significantly in a dense network. In Table 5.1 we can see the communication complexities of 'Synch + $\beta$' and 'Asynch' algorithms are identical in big-O-notation, but in Table 5.2 the difference shown is in the number of the messages on our experimental result. One of the reasons for this difference is that we add 'ack', 'safe' and 'pulse' messages in $\beta$ synchronizer. But these messages do not increase the message complexity. The other reason for fewer messages in the asynchronous algorithm is because the resulting BFS tree is shorter.
CHAPTER 6

LEADER ELECTION IN NS

For all Leader Election algorithms in this chapter, we assume that every node contains a unique ID. The node with minimum ID will be selected as the leader. We first introduce a purely synchronous leader election, and then we combine the synchronous algorithm with \( \alpha \) and \( \beta \) synchronizers. We will also present an asynchronous algorithm and then evaluate the communication complexity.

Synchronous Leader Election Algorithm

Now we present a very simple synchronous algorithm called FloodMin algorithm. Every node maintains a record of the minimum ID (its own ID initially). At each round, each node sends the minimum ID to all neighbors, upon receipt of a minimum ID from each neighbor, it performs the computation and gets a new minimum ID and starts next round. After diameter rounds, if the minimum ID is the node's own ID, the node knows it is the leader; otherwise, it is not a leader. The synchronous algorithm requires that every node knows the diameter of the network. It is easy to see the time complexity is \( O(Diameter) \) and the number of messages is \( Diameter \times |E| \), where \(|E|\) is the number of links.

Note that the synchronous algorithm also works correctly if the node knows the upper bound diameter rather the actual value of the diameter. But the complexity measures increase because they are evaluated depending on the upper bound diameter rather than the diameter.
Combination with Synchronizer

Since we have already made $\alpha$ and $\beta$ synchronizer work correctly with synchronous BFS algorithm, we can just simply modify the code to make the synchronizers work with synchronous Leader Election algorithm. This wonderful feature of the synchronizer makes the programmer design the code more efficiently.

The difference between BFS and Leader Election algorithms combined with $\alpha$ or $\beta$ synchronizer is that in the Leader Election algorithm each node maintains a record of the minimum ID and propagates the minimum ID throughout the network; whereas in BFS algorithm each node records the minimum level of itself, sends the level messages to the neighbors and records its parent and children. Since we already know the difference of the two applications of the synchronizer, it is easy to modify the code presented in the previous chapter to make the synchronous Leader election algorithm combine with $\alpha$ or $\beta$ synchronizer.

Asynchronous Leader Election Algorithm

Here, we introduce an asynchronous Leader Election algorithm called 'wave' algorithm [15]. Each node can initiate a separate wave. Each message of the wave must be tagged with the ID of the initiator in order to distinguish them from the messages of different waves. No matter how many waves are generated, only the wave of the smallest initiator will run to a decision.

Each node will at most be involved in one wave, called $caw$ (currently active wave) at a time. Initially we set $caw(ID)$ to its own ID for each node. If node $p$ receives a 'token $\$initiator$' message of wave from a neighbor $q$, and if $caw(p) < \$initiator$, just drop the message, effectively causing this wave to fail. But if $caw(p) > \$initiator$, node $p$ joins this wave, take $q$ as its parent and resets $caw(p) = \$initiator$, then forward the wave message to other neighbors. Eventually each node must join the wave of the smallest initiator. If node $p$ receives a 'token $\$initiator$' message with the
same initiator from each neighbor, and if \( p = \$initiator \) then send \('leader \$initiator'\) to each neighbor to inform the leader is \$initiator, if \( p \neq \$initiator \) then send \('token \$initiator'\) to its parent. The decision always takes place in the node that is the smallest initiator. The decider broadcasts the decision to all other nodes. Each node will know the leader upon the receipt of \('leader \$initiator'\) message. See algorithm 6.0.2 for detail.

We summarize the complexities of different algorithms in this chapter in Table 6.1. The complexity of the synchronous Leader Election algorithm combined with \( \alpha \) or \( \beta \) synchronizer is the same as the one of the synchronous BFS algorithm combined with \( \alpha \) or \( \beta \) synchronizer. From Table 6.1 we can see \( \beta \) synchronizer is more efficient than \( \alpha \) synchronizer.

The result of our testing in NS also shows the efficiency of \( \beta \) synchronizer. See table 6.2. We did not consider the number of messages for \( \beta \) synchronizer during the construction of the spanning tree \( T \) for \textit{safe} and \textit{pulse} message communications. We assumed the spanning tree \( T \) has already been created. The number of messages of \( \beta \) synchronizer is less than the one of \( \alpha \) synchronizer, significantly in dense network.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>(O(Diam \times</td>
<td>E</td>
</tr>
<tr>
<td>Synch + ( \alpha )</td>
<td>(O(Diam \times</td>
<td>E</td>
</tr>
<tr>
<td>Synch + ( b )</td>
<td>(O(</td>
<td>E</td>
</tr>
<tr>
<td>Asynch</td>
<td>(O(N \times</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 6.1: Complexity of Leader Election Algorithm

<table>
<thead>
<tr>
<th>Network Topology</th>
<th>No. of Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Synch + ( \alpha )</td>
</tr>
<tr>
<td>11 12 5</td>
<td>234</td>
</tr>
<tr>
<td>11 14 5</td>
<td>278</td>
</tr>
<tr>
<td>22 30 6</td>
<td>700</td>
</tr>
<tr>
<td>22 50 6</td>
<td>1210</td>
</tr>
</tbody>
</table>

Table 6.2: Number of Messages of Leader Election Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In Table 6.1 the communication complexities of the algorithm of 'Synch + β' is better than the one of 'Asynch' in big-O-notation. But in Table 6.2 we can also see the big difference between them in the number of the messages on our experimental result. One reason is that we add 'ack', 'safe' and 'pulse' messages in β synchronizer. But those messages do not increase the message complexity. Another reason is that in asynchronous algorithm the initiator with smallest identity starts its wave very early, in that case, other nodes' waves will be terminated very soon and some nodes even do not have chance to generate their own waves. The density of the network is also a fact in determining the number of messages. The complexity of asynchronous algorithm in Table 6.1 is in a general case.
Algorithm 6.0.2 AsynchLeaderElectionAlgorithm

# initially for all nodes
set caw($nodeID) $nodeID
set rec($nodeID) 0 ;# number of 'token' message received
set lrec($nodeID) 0 ;# number of 'leader' message received

if { node p is an initiator} {
    send 'token p' to each neighbor
}

while { $lrec(p) > no. of neighbor } {
    # receive msg from q
    if { $msg == 'leader $initiator'} {
        set lrec(p) [expr $lrec($p) + 1]
        set leader(p) $initiator
    } elseif { $msg == 'token $initiator'} {
        if { $initiator < $caw(p) } ;# reinitialize algorithm
            set caw(p) $initiator
            set rec(p) 0
            set parent(p) q
            send 'token $initiator' to all neighbors except q
        } elseif { $initiator == $caw(p) } { ;# reinitialize algorithm
            set rec(p) [expr $rec($p) + 1]
            if { $rec(p) == no. of neighbor } {
                if { $caw(p) == p } {
                    send 'leader $initiator' to each neighbor
                } else {
                    send 'token $initiator' to p's parent
                }
            } else {
                send 'token $initiator' to p's parent
            }
        } else { ;# $initiator > caw(p)
            # drop the message
        }
    }
}

if { leader(p) == p } {
    # p knows he is the leader
} else {
    # p knows leader is $leader(p)
CHAPTER 7

USER MANUAL

Our goal in this chapter is not to give a comprehensive user manual of NS, but make it easier for the readers to run our program on their own machine with little knowledge of NS. We will talk about something, which is not in the official NS MANUL but the users may experience when writing code in NS. Before we finished our work, ns-2.1b9a was released. But there is no big difference from the previous version ns-2.1b8a, in which our programs were designed. We will give a simple manual of ns-2.1b8a under Redhat Linux 7.3 then we will focus on implementing the distributed algorithm in NS.

NS is developed on several kinds of Unix (FreeBSD, Linux, SunOS, Solaris). Ns also can be built under Windows, but there are limitations to the features of NS under Windows. On Windows, NS needs other software such as Microsoft Visual C++ to make NS work. We recommend choosing Linux as the operating system, because it is free software with g++/gcc compiler built-in, furthermore Linux can be installed on x86 machine.

We can download the NS software for free at NS website [5]. NS requires glibc version 2.1 or higher and it works well under gcc/g++ version 2.9.6. If your operating system is Redhat 7.x, you do not need to change the environment to build NS. Please do not try to upgrade your gcc/g++ to version 3.0.x, because NS cannot be built directly under this new version of gcc/g++. If you insist to use it, you need to modify hundreds of C++ files of NS to meet the requirement of gcc/g++ 3.0.x. Decompress the NS software to any directory. You will get a sub directory "ns-allinone-2.1b8a", and then change the directory to "ns-allinone-2.1b8a", install it by following the
Now, we should get NS built on the machine with no errors and we can implement our algorithms in NS. We have already given a simple sample in Chapter 4 and more detail about designing C++ and OTcl code for the distributed algorithm in Chapters 5 and 6. To implement the BFS algorithm combined with Alpha synchronizer, we present a checklist of what should be done before we recompile NS. For other algorithms, please refer to this checklist.

- Copy “alphabfs.h” and “alphabfs.cc” to “ns-allinone-2.1.b8a/ns-2.1.b8a” directory.

- Make sure you registered the new agent header by modifying “packet.h” and “ns-2.1.b8a/tcl/lib/ns-packet.tcl”. See Chapter 5.

- Set default value for the new configurable parameters, such as ‘packet_size’ in “ns-2.1.b8a/tcl/lib/ns-default.tcl”. See Chapter 5.

After the three steps, modify your “Makefile” under the directory “ns-2.1.b8a” by adding ‘alphabfs.o’ in the object file list and recompile NS. We must do “make depend” before we do “make” if we make some changes in “Makefile”. Now we can copy the Tcl script files “alphabfs.tcl” and “alphabfstopo.tcl” to any directory and execute command “ns alphabfs.tcl” to run the program. We will get a display window (NAM) to view the real-time execution and trace files. We can also output information to the terminal.

With these steps you should be able to run BFS algorithm in your own computer. To run other algorithms just follow the same steps as above.
CHAPTER 8

CONCLUSIONS

In this thesis, we experimented two distributed synchronizers, α and β synchronizers. There are many efficient synchronous algorithms, but those algorithms cannot be implemented in the practical network (asynchronous network) directly. The synchronizer allows the user to design a synchronous algorithm and run it in the asynchronous network.

To see the usefulness of the synchronizer, we implemented them by considering two applications: Leader Election and the construction of BFS tree in NS. According to our theoretical analysis and testing result of our experiments in NS, we conclude that the most important advantage of the synchronizer is to make the programmers design the code efficiently. Theoretically the communication complexity of synchronous algorithm combined with β synchronizer is identical or similar to asynchronous algorithm, but from our testing result we think β synchronizer cannot always offer a better solution concerning about the number of messages than some asynchronous algorithms. Our testing result demonstrates that β synchronizer is always much more efficient than α synchronizer.

Even though we found NS is unpolished and incomplete software, we were eventually able to implement the distributed algorithms in NS successfully. Some of our difficulties experienced in our work were the fact that users must understand the internal details of NS in order to implement any algorithms. At least NS offered us a platform to implement our distributed algorithm. In the future, NS developers will change the API of NS so that the users can specify a cluster of nodes or links that behave in lock-step dynamic synchrony [5]. With the effort of the NS developers and
users, NS will become a much better simulation program.
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
VITA

Graduate College
University of Nevada, Las Vegas

Wei Huang

Local Address:
1676 Jupiter Ct. #F
Las Vegas, NV 89119

Degrees:
Bachelor of Science, Electrical Engineering, 1992
Fuxin Institute of Mining and Technology, P.R.China

Thesis Title:
Distributed Synchronizer in Network simulator (NS) software

Thesis Examination Committee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Committee Member, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. Sebastien Tixeul, Ph.D
Graduate Faculty Representative, Dr. John Wang, Ph.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.