University of Nevada, Reno

Software Distributed Shared Memory System with Fault Detection Support

A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

by

Jiangtao Zeng

Dr. Angkul Kongmunvattana / Advisor

March 10, 2002
Abstract

Distributed shared memory (DSM) system provides simplicity of shared-memory programming on cluster of computers. Compared to hardware approach, software DSM is a relatively low-cost, low-risk entry into the concurrent computing arena. However, fault detection becomes essential when software DSM is used to execute long-running applications. When one of the remote processor crashes or the process running on it fails, the system should provide the application with the ability to be aware of the failure and to take appropriate actions, such as cleaning up, terminating the program, and migrating the job.

TreadMarks is a state-of-the-art software DSM implementation developed on standard UNIX systems. However, it does not provide fault detection support due to its usage of simple, connectionless UDP protocol. In this project, the UDP protocol used by TreadMarks was changed to TCP protocol. TCP is connection-oriented, reliable, and of byte stream type. A fault detection mechanism using the error codes returned by the TCP socket layer was implemented. By replacing UDP with TCP, the complexity of TreadMarks implementation can be reduced by removing redundant reliability services previously used in TreadMarks. By exploiting error codes for fault detection, we avoid any potential communication overhead because those error codes are automatically generated by TCP protocol.

Our experiments show that the impact of changing UDP to TCP on the system performance depends on the scale of the application and the synchronization technique adopted by the application. Shorter-running applications and applications relying on locks for synchronization would suffer more than longer-running ones and those relying on barrier for synchronization. Our experiments also show that the fault detection using error codes returned by TCP does not incur any overhead.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.2</td>
<td>Software</td>
<td>31</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Benchmark applications</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Experimental results &amp; discussion</td>
<td>33</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Sequential execution time</td>
<td>33</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Parallel execution time</td>
<td>34</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Comparison of parallel execution times under UDP and TCP</td>
<td>36</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Stalling time</td>
<td>37</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Lock idle and abnormal performance of Water</td>
<td>39</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Performance with fault detection</td>
<td>39</td>
</tr>
</tbody>
</table>

6 Conclusion 40
List of Figures

2.1 Data Structures Used in TreadMarks ........................................... 7
2.2 Communication Flow During Start-up Stage ................................. 9
2.3 Communication Flow In Data Distribution ................................. 9
2.4 Communication Flow In the Call of Tmk_barrier ..................... 10
2.5 Communication Flow In the Call of Tmk_lock .......................... 11
2.6 Communication Flow In the Call of Tmk_segv_sigio_handler ........ 11
2.7 UDP client-server exchange ..................................................... 12
3.1 TCP client-server exchange ....................................................... 14
4.1 TCP three-way handshake ....................................................... 20
4.2 TCP connection termination ................................................... 20
4.3 Fault: Termination of remote process ..................................... 21
4.4 Fault: Crashing of remote host ............................................... 22
4.5 Fault: Crashing and rebooting of remote host ......................... 24
4.6 Fault: Crashing and rebooting of remote host ......................... 25
4.7 Fault: Shutdown of remote host ............................................. 26
4.8 Fault: Restart of remote host ............................................... 28
4.9 Fault: Restart of remote host ............................................... 29
4.10 Fault: Restart of remote host ............................................. 30
5.1 Sequential Execution Times Under UDP and TCP ..................... 34
5.2 Parallel and Sequential Execution Times Under UDP and TCP ........ 35
5.3 Comparison of Parallel Execution Times Under UDP and TCP .......... 37
5.4 Stalling Times Under UDP and TCP ....................................... 38
List of Tables

5.1 Benchmarks and Problem Sizes ......................... 31
5.2 Sequential Execution Times Under UDP and TCP 34
5.3 Comparison of Parallel Execution Times Under UDP and TCP 36
5.4 Performance with Fault Detection 39
Chapter 1

Introduction

1.1 Software distributed shared memory system (DSM)

In the traditional computation model, events happen sequentially, one after another. But in some cases, a pair of events can happen in arbitrary order, or even concurrently, which makes concurrent programming possible. Today, powerful multiprocessor computers are built by assembling many microprocessors for running the parallel applications.

There are two practical models of concurrent programming, based on two different multiple processor computer designs. In shared-memory multiprocessors, each processor can read, or write, any element in a common memory. Each processor has a large cache to reduce the amount of traffic between processors and common memory. In distributed-memory multiprocessors, each processor has its own memory, and data in the memory of one processor is inaccessible to other processors. Where processes must share data, they explicitly send and receive messages that contain data.

In distributed-memory concurrent programming, situation is complicated by two aspects: data movement and synchronization. In distributed-memory computing, data sought by one processor, but available only on another, must be explicitly sent by the one that has it to the one that needs it, and explicitly received by another. With several instructions in progress simultaneously on different processors, synchronization is necessary when the result of one computation is explicitly required before another computation on another processor can begin.

Since there is only one common memory, shared-memory programming does not need to move data to make it accessible to another processor, so it only requires the consideration of synchronization, which makes shared-memory programming easier than distributed memory programming. However, shared-memory multiprocessor computers have disadvantages since they must be carefully designed and assembled, and are therefore expensive.

Distributed-memory multiprocessors can be assembled from ordinary computers on a high-speed network. Software interfaces like PVM and MPI make it possible to send and receive messages along the network, from one processor to another, and thus to consider a set of engineering workstations to be a distributed-memory multiprocessor computer. For this reason, distributed-memory
multiprocessing has become popular, in spite of its complexity.

With the message-passing paradigm of distributed-memory system, the application program must have full awareness of the system organization, making it relatively difficult to program. So distributed shared-memory system (DSM) is proposed to build the shared-memory paradigm on top of the distributed-memory machines, by providing a shared memory abstraction such that the users have a shared-memory environment with message passing taken care of by the DSM layer.

As the DSM system has little knowledge about application programs, it may conduct unnecessary message exchanges to maintain data coherency. The issue of how to make fast and efficient data movement is thus important for the design of DSM. Hardware solutions were proposed to enforce data coherency, but the cost involved in special-purpose hardware may pose a major concern. Software DSMs, on the other hand, put an emphasis on efficient data movement through reducing the number of messages exchanged among processors, since sending messages under software approach is more expensive than hardware approach. In comparison to hardware approach, software DSM is a relatively low-cost, low-risk entry into the concurrent computing arena.

1.2 TreadMarks DSM

TreadMarks is one of the software DSM systems developed on standard Unix systems such as SunOS and Ultrix, which allows shared-memory concurrent programming on a distributed memory multiprocessor computer, or on a network of workstations [12]. When “shared” memory is accessed on one processor, TreadMarks determines whether the data is present at that processor, and if necessary transmits the data to that processor without programmer intervention. When “shared” memory is modified on one processor, TreadMarks ensures that other processors will be notified of the change, so that they will not use obsolete data values.

By employing recently developed techniques such as Lazy Release Consistency (LRC) and Multiple-writer protocols, TreadMarks efficiently reduces the number of messages exchanged among processors and the impact of false sharing.

TreadMarks is a state-of-the-art software DSM implementation, which was originally developed at Rice University. Various researchers around the world developed their projects on the basis of TreadMarks, that is, to propose modifications to TreadMarks to improve the system performance ([21] [15] [5]).

1.3 UDP versus TCP

TreadMarks uses User Datagram Protocol (UDP) for the communications among processors. UDP is a simple transport-layer protocol. It is connectionless, unreliable, and of datagram type. The application writes a datagram to a UDP socket, which is then sent to its destination. But there is no guarantee that a UDP datagram ever reaches its final destination. So a lot of work have to be
pushed into the applications, such as acknowledgments from the other end, time-outs, retransmis-
sions, duplicate detection, sequencing of packets, and the like.

On the other hand, Transmission Control Protocol (TCP) is connection-oriented, reliable, and of
byte stream type. When TCP sends data to the other end, it requires an acknowledgment in return.
If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer
amount of time. TCP also sequences the data by associating a sequence number with every byte
that it sends to deal with the segmentation and duplication.

Usually UDP is faster than TCP, since UDP requires only two packets to exchange a request and
a reply, while TCP requires about 10 packets (see description in section 4.1.2), assuming that a new
TCP connection is established for each request-reply exchange. However, if a TCP connection used
for multiple request-reply exchanges, then the cost of the connection establishment and teardown
is amortized across all the requests and replies. Considering the effort of maintaining a reliable
UDP connection by the application, it is reasonable to believe that replacing UDP currently used
by TreadMarks with TCP protocol would not degrade the performance of the system significantly,
because the application is relieved of acknowledgments, time-outs and retransmission, etc. thanks
to less system calls and interrupt handling.

1.4 Fault detection

One of the important components of the distributed system is fault detection and fault recovery.
When one of the remote processor crashes or the process running on it fails, the system should
provide the application with the ability of being aware of the abnormal and take appropriate action
to clean up and terminate the program.

In current TreadMarks implementation, there is not too many fault detections owing to the feature
of the communication protocol it uses. UDP is connectionless, i.e. there is no connection between
a UDP client and a UDP server. UDP has no connection setup or teardown. So the only fault UDP
protocol can detect is when server is not running. But with TCP’s stream socket, different error
codes will be returned upon different error scenarios. By correctly understanding the error codes
returned by the stream sockets, we can use them to locate process failures in a timely manner. Be-
cause these error codes are generated automatically when there is communication with the failed
process, the fault detection mechanism does not incur any overheads.

1.5 Related work

Distributed shared memory (DSM) system has been drawing continuous attention in recent years,
most of them focus on memory consistency models ([13] [17] [15] [8] [2]) and performance speedup
([21] [5] [4] [7] [1]).
Some of the research efforts address the fault tolerance and recoverability of DSM. James et al. [9] studied the achievable fault tolerances of shared memory consistency conditions in the form of $t$-resilience, the ability to withstand up to $t$ nodes failures in the context of totally asynchronous message passing system, where message delays are not known a priori. Kermarrec [14][16] proposed a recoverable DSM based on Backward Error Recovery (BER) mechanism, where a recovery point is periodically saved on stable storage and restored if a failure occurs. In addition, the replication required for a BER mechanism to ensure the permanence of recovery data is exploited during failure-free executions, thus allows to increase the efficiency of DSM. Janssens et al. [10] also presented the design of an independent checkpointing method for DSM recovery that takes advantage of DSM’s specific properties to reduce rollback overhead. Fleisch et al. [3] investigated the trade-offs between the degree of fault tolerance, operational costs, and configurability for various DSM coherence protocols.

Fault tolerance and recovery deals with issues after a node failure has been successfully detected, but it does not address how the failure is detected. As a matter of fact, fault detection in software DSM has not been a fruitful topic. Through literature reading, few papers are found to be about fault detection in a software DSM environment. Hosseini et al. [6] proposed a distributed processor level fault diagnosis algorithm where each processor test some peer processors that are assigned to it dynamically from time to time. Tierney et al. [22] designed a methodology to monitor the behavior of all distributed application elements and some of operating system elements by generating precision event logs. Stelling et al. [19] presented a fault detection service based on periodic ”i-am-alive” messages, i.e. a sort of polling technique. Either above-mentioned polling technique or watch-dog technique involves certain overhead and has to make a tradeoff between system performance and speed of fault detection.

Duarte Jr. et al. [11] proposed a network fault management based on SNMP (Simple Network Management Protocol), where information of nodes are replicated and broadcasted to SNMP agent groups. This method reduces communication overhead, but still involves overhead of replication node information termed Management Information Base (MIB) objects. Neves et al. [18] implemented a fault detection mechanism using message responses from socket layer on a REnoverable NEtwork of Workstations (RENEW) which offers the standard message passing interface (MPI). This method causes the least overhead to the system performance, and in fact was the inspiration to our project.

1.6 Research goals

In this project, we first replace UDP with TCP in the TreadMarks system to investigate how the transport layer protocol will affect the DSM system performance. Then we implement the fault detection mechanism in TreadMarks DSM using the hints from the TCP socket layer.

In the detection mechanism, the failed processes are only found if the surviving processes try to communicate with them. However, as long as the limitations of this technique are understood, it can be used to discover certain types of faults rapidly.
This paper is organized as follows: Chapter 2 briefly introduces the UDP implementation in TreadMarks DSM. Chapter 3 illustrates the features of TCP protocol, compares it with UDP used by TreadMarks, and summarizes ways of how to replace UDP with TCP. Chapter 4 describes in detail how TCP responds to various communication errors, thus provides hints of node failures. Some of the implementation issues are discussed at the end of the chapter. Experimental platform is presented in Chapter 5, as well as the discussions over the results. Chapter 6 concludes the paper.
Chapter 2

UDP Implementation in TreadMarks

This chapter briefly introduces the UDP implementation in TreadMarks DSM, including the principal data structures employed and the communication flows between processors. This will provide an outline of how processors cooperate with each other in the DSM environment, and how message passing mechanism is utilized to build a shared-memory layer on top of the distributed network.

2.1 Data structures

Figure 2.1 gives an overview of the data structures used by TreadMarks. The principal data structures[12] are:

- **page_array** with one entry for each shared page. Each entry in the page_array contains:
  - The current state: no access, read-only access, or read-write access.
  - An approximate copyset specifying the set of processors that are believed to currently cache this page.
  - For each page, an array indexed by processor of head and tail pointers to a linked list of write_notice records corresponding to write notices received from that processor for this page. If the diff corresponding to the write notice has been received, then a pointer to this diff is present in the write notice records. This list is maintained in order to decrease interval indices.

- **proc_array** with one entry for each processor. Each entry in proc_array contains a pointer to the head and the tail of a doubly linked list of interval records, representing the intervals of the processor that the local processor knows about. This list is also maintained in order to decrease interval indices. Each of these interval records contains a pointer to a list of write_notice_range records for that interval, within each range are pages that have been modified by that processor in that interval. Also each write notice record contains a pointer to its interval records.

- A set of interval records, containing mainly the vector timestamp for that interval. Each time a process executes a release or an acquire of the shared memory, a new interval begins
Figure 2.1. Data Structures Used in TreadMarks
and the interval index is incremented. Intervals of different processes are partially ordered by the *vector timestamp*.

- **vector timestamp**, contains an entry for each processor. The entry for processor $p$ in the vector timestamp of interval $i$ of processor $p$ is equal to $i$. The entry for processor $q \neq p$ denotes that most recent interval of processor $q$ that precedes the current interval of processor $p$.

- A set of *write notice* records. Each *write notice* is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. The actual modifications are carried by *diff*.

- A *diff pool*. A *diff* is a runlength encoded record of the modifications to the page, created when a processor requests the modifications to that page or a write notice from another processor arrives for that page.

### 2.2 Communication flows

In addition to the establishment of connection during the start-up stage, communications among processors occur when there are needs for synchronization, data distribution, or maintenance of data coherency. The synchronization procedure can be further divided into two categories: lock and barrier. The maintenance of data coherency is needed when there is a page fault, triggering a ‘segmentation violation’ interrupt, which is handled by the corresponding interrupt handler.

- **Start up**.

  Figure 2.2 illustrates the message flows during the start-up stage. Processor #0 first starts the process itself, then remotely starts the process on processor #1, passing its configuration of connection, such as its port numbers available, to processor #1. Then processor #0 performs first the passive connection to accept processor #1’s active connection, followed by the active connection to processor #1 who is performing the passive connection. The same sequence of events take place when processor #0 establishes connections with other processors. After each processor is activated by processor #0, it performs active+passive connections with processors started before itself, and passive+active connections with processors started after itself.

- **Distribution**

  Function $\text{Tmk}_{-}\text{distribute}$ in TreadMarks is used to explicitly share private memory between processes. When invoked by a single process, which is usually processor #0, it causes values in private memory on that processor to be replicated in corresponding memory locations in all other processors. The processor who owns the data sends a request for distribution to each of its peers, piggybacking the data to be distributed into the request. Upon the receipt of the request and the data, the peer sends back a confirmation, as shown in Figure 2.3.

- **Barrier**
Figure 2.2. Communication Flow During Start-up Stage

Figure 2.3. Communication Flow In Data Distribution
A barrier is a synchronization device that requires all processes to wait for the last of them to “catch up”. The function `Tmk_barrier` causes all processors to pause until all of them have invoked `Tmk_barrier` with the same barrier index.

Barriers have a centralized manager. At barrier arrival, each client sends signal to the barrier manager, piggybacking its vector timestamp and local information about latest intervals. When the manager arrives at the barrier, it incorporates these intervals into its data structures. When all barrier arrival signals have been received, the manager then sends message to all clients to allow them proceed, piggybacking global information about latest intervals. The client incorporates these intervals and continues the process execution. Figure 2.4 illustrates the communication flows in response to the call of `Tmk_barrier`.

- **Lock**

A lock is a synchronization device that enforces one-process-at-a-time access. The function `Tmk_lock_acquire(L)` in TreadMarks causes a process to pause while another process owns lock `L`, and to take ownership of the lock and continue when the lock becomes available.

All locks have a statically assigned manager. The lock manager records which processor has most recently requested the lock. The lock acquirer sends request to the manager, who then forwards it to the processor that last requested the lock if itself is not the previous holder. As a result, the lock request arrives at the processor that either holds the lock or did the last release on it.

When the lock is released, the releaser informs the acquirer of the releaser’s vector timestamp, and information about the latest intervals it is aware of. After receiving this message, the acquirer incorporates those information into its data structures. The communication flows are illustrated in Figure 2.5.

- **segv handler**

Figure 2.4. Communication Flow In the Call of `Tmk_barrier`

n = Tmk_nprocs
# m = barrier manager

\[
\begin{align*}
\text{signal when} & \quad \text{arrive at barrier point} \\
\# m & \quad \# 1 \\
\text{allow to proceed when} & \quad \text{signals are received} \\
\text{from all processors} & \quad \# m-1 \\
\# m+1 & \quad \# m-1
\end{align*}
\]
When there is a page fault, if the faulting processor does not have a copy of the page, it requests a copy from a member of the page’s approximate copyset. The approximate copyset for each page is initialized to contain processor #0. If write notices are present for the page, the faulting processor determines the set of necessary diffs and the set of processors to query, then sends out requests for the diffs in parallel. The processors that received the requests create diffs locally, and send them back to the faulting processor. The communication flows are illustrated in Figure 2.6.

2.3 UDP client-server exchange

Figure 2.7 shows the function calls for a typical UDP client-server. The UDP client does not establish a connection with the server. Instead, the client just sends a datagram to the server using
the `sendto` function, which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom` function, which waits until data arrives from some client. `recvfrom` returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

In TreadMarks, the function `connect` is called after the socket is created (function `socket`) at both client and server sides. Calling `connect` does not result in a real connection between the client and the server. Instead, the kernel just records the IP address and port number of the peer, which are contained in the socket address structure passed to `connect`, and returns immediately to the calling process[20]. So with a connected UDP socket, the destination IP address and port is not needed to be specified any longer for an output operation. Thus, `write`, `send` or `sendmsg` can be used instead of `sendto`. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by the `connect`. Similarly for the server side, `read`, `recv` or `recvmsg` are used instead of `recvfrom`. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in the `connect`. Datagrams arriving from other protocol addresses are not passed to the connected socket.
Chapter 3

TCP Implementation

This chapter first illustrates the communication procedures between a server and a client under TCP transport layer protocols. Then the technique used by TreadMarks to provide reliability to the UDP sockets is compared with the reliability services provided by TCP functions. Finally, ways to change from UDP sockets to TCP ones are summarized.

3.1 TCP Overview

The service provided by TCP to an application is different from the service provided by UDP. First, unlike UDP is connectionless, TCP provides actual connections between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.

TCP protocol also provides reliability. When TCP sends data to the other end, it requires an acknowledgment in return. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time. After some number of retransmissions, TCP will give up.

TCP also sequences the data by associating a sequence number with every byte that it sends. If there are several segments arriving a TCP socket out of order, the receiving TCP will reorder the segments based on their sequence numbers before passing the data to the receiving application. If TCP receives duplicate data from its peer, it can detect the duplicate from the sequence numbers, and discard the duplicate data.

Finally, TCP provides flow control. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer.

3.2 Procedures of TCP client-server exchange

Figure 3.1 shows a timeline of the typical scenario that takes place between a TCP client and server.
Here function `socket` creates a socket for the type of communication protocol desired. `connect` establishes a connection for the client with the server. Note that the client here does not have to call `bind` before calling `connect`, the kernel will choose both an ephemeral port and the source IP address for it if necessary. But a TCP client can `bind` a specific IP address and port to its socket to assign the source IP address and the port that will be used for IP datagrams sent on the socket. Of course the IP address must belong to an interface on the host. `connect` initiates TCP’s three-way handshake to establish a connection. The function returns only when the connection is established or an error occurs.

Function `listen` is only called by the server, and is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function. When a socket is created by `socket`, it is assumed to be an active socket, that is, a client socket that will issue a `connect`. The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
Function `accept` is called by a TCP server to return a completed connection. If `accept` is successful, its return value is a brand new descriptor that was automatically created by the kernel. This returned *connected* socket is different from the *listening* socket created by `socket`. For a concurrent server, it normally creates one listening socket, which then exists for the lifetime of the server, while the kernel then creates one connected socket for each client connection that is accepted.

### 3.3 UDP versus TCP

Since UDP is a connectionless, unreliable, datagram protocol, applications using UDP have to add reliabilities themselves. Following is the typical code TreadMarks uses to send a request and receive a reply:

```
rexmit:
  if (0 > sendmsg(fd, &req_hdr, 0))
    Tmk_perrexit();
  Tmk_tout_flag = 0;
  setitimer(ITIMER_REAL, &Tmk_tout, NULL);
retry:
  if (0 > recv(fd, (char *)&rep, sizeof(rep), 0))
    if (Tmk_tout_flag) {
      Tmk_err();
      goto rexmit;
    } else if (errno == EINTR)
      goto retry;
    else
      Tmk_perrexit();
  if (req_typ.seqno != rep.seqno) {
    Tmk_err();
    goto retry;
  }
```

Above code segment can be divided into two parts: the first one is to send the request, the second one is to receive the confirming reply. After the request is sent out, TreadMarks set a timer of 30 seconds, and use a global variable `Tmk_tout_flag` to indicate the timer expiration. When it is failed to receive a reply, it first checks the time-out flag. If the receiving failure is resulted from the timer expiration, the request will be retransmitted. Otherwise, if the system call `recv` is interrupted by a signal, TreadMarks resumes the system call `recv` under the same timer. In the case of successful receipt of the reply, the sequential number of the reply is compared with the the one of
the request just sent out. If they don’t match, TreadMarks returns to the point of receiving and give 
`recv` another try.

Also TreadMarks provides two handlers for I/O signals of each kind of request: one for the 
first request, and the other for the duplicate one. For example, for the request of `lock`, there are 
`Tmk_lock_sigio_handler` and `Tmk_lock_sigio_duplicate_handler`; for the request of `distribute`, there are 
`Tmk_distribute_sigio_handler` and `Tmk_distribute_sigio_duplicate_handler`.

From above analysis, it can be seen that TreadMarks adds many reliability services to the UDP 
sockets, such as acknowledgments, time-outs, retransmissions, duplicate detection, sequencing of 
packets, etc.. However, as described in section 3.1, all of these reliability services are built-in fea-
tures of TCP protocol. This inspired us that we can use TCP sockets instead of UDP ones, which not 
only will simplify the codes significantly, but also might improve the system performance because 
of less system calls and interrupt handling.

### 3.4 Implementation issues

Modifications to replace UDP protocol with TCP in TreadMarks includes three levels: changing 
socket type, removing redundant security, and fixing up deficiency.

Changing socket type is achieved by replacing UDP socket functions with TCP ones, includ-
ing `socket`, `bind`, `listen`, `accept`, `close` etc. as stated in section 3.2. Then we can 
remove statements that deal with sequencing, acknowledgments, time-outs, and retransmissions. 
TCP can do these for us. But since interrupted system calls happen in both UDP and TCP, in the 
code segment presented earlier in section 3.3, we have to keep the part of checking if `errno == 
EINTR` when we remove redundant reliability services added by TreadMarks to UDP sockets.

The biggest challenge is to fix the deficiency of TCP compared with UDP. Since TCP is connection-
oriented, of byte-stream type, it does not provide any record delimiter. Thus it is not uncommon 
that one message is sent out in several messages, and several messages are received as one single 
message. The former happens when the message size is too large, and the latter happens when the 
message size is too small.

Take the lock acquisition as an example. One node may send several lock acquire requests to 
another node in a very short period. One of the requests might be for itself; the others might be for-
warded lock requests for other nodes. In this case, the receiver will probably receive several requests 
as one single one, and call lock request handler for only once. The receiver is not aware that there 
are other requests have been sent to it, while other nodes who request those locks might have to wait 
forever for the expected replies. This will cause a deadlock, and furthermore trigger process failures.

UDP does not have such worries. Because it is connectionless and of datagram type, it is guaran-
teed that each datagram is sent as one single message and received as one single message. So there
is nothing dealing with record boundaries in the original TreadMarks implementation. When we replace UDP with TCP, we have to make sure that data are retrieved along the right boundaries.

Usually there are two ways to add record boundaries to a byte stream[20]: adding delimiters and specifying size at the beginning of the message. Adding delimiters further involves two concerns: to distinguish the same delimiter characters appearing in message body from those functioning as delimiters, and to search through the received data to find delimiters and segment data. Specifying size is rather straightforward. The receiver counts the number of bytes when it is receiving the message, and then checks it against the number of size stated at the beginning of the message. This way we can easily know if the data is received completely, incompletely, or over-received. In our project, the method of specifying message size was chosen, because it is simple, and takes less effort to modify the codes.
Chapter 4

Fault Detection Mechanism

In this section, we’ll first summarize the fault types that can be detected by the fault detection mechanism employing hints from socket layer. This mechanism is tightly related to the principles of establishment and termination of a TCP connection. So next we’ll take a close look at the procedure of normal establishment and termination of TCP connection. Then we’ll examine how error codes are returned by the socket layer upon each type of error. Finally some implementation issues are discussed.

4.1 Methodology

4.1.1 Fault types

In this project, we mainly examine five distinct types of faults, each resulting in the termination of a process, but with different behaviors observed at the TCP socket interface. The five types of faults that are considered are:

Termination of remote process  The fault terminates the process, but does not affect the rest of the system. Examples of this type of fault are:

- the process is aborted because of an illegal instruction;
- the process is killed by the owner of the machine;
- the process exits because one of the assertions of the program is violated.

Crashing of remote host  The remote host where the process is running crashes permanently or stays down for a long period of time. Examples of this type of fault include:

- a permanent failure in one of the machine’s components;
- an unattended machine crashes but no one is available to restart it.

Crashing&rebooting of remote host  The remote host where the process is running crashes, and then reboots. Possible causes of such faults include:

- power failures;
• incorrect usage by users.

**Shutdown of remote host** The remote host where the process is running is shut down by an operator.

**Restart of remote host** The remote host where the process is running shuts down, and then reboots. A machine might be restarted because:

- a new software version requires rebooting in order to be installed;
- the machine is not performing as expected.

Before we talk about the error codes returned from these abnormal connection scenarios, let’s take a close look at the normal TCP connection establishment and termination. An increased knowledge of the underlying protocols will help us a lot with network programming.

### 4.1.2 TCP normal startup - three-way handshake

There are minimum three packets required to exchange for establishing a TCP connection, hence this procedure is called TCP’s *three-way handshake*. Following is the detailed scenario occurs in this three-way handshake:

1. The server is prepared to accept an incoming connection by calling `socket`, `bind`, and `listen`.

2. The client calls `connect`, which causes the client TCP to send a `SYN` (synchronization) segment to the server, telling the server the client’s initial sequence number for the data that the client will send on the connection. Normally there is no data sent with the `SYN`.

3. The server acknowledges the client’s `SYN` and the server sends its own `SYN` containing the initial sequence number for the data that the server will send on the connection. The server sends its `SYN` and the `ACK` (acknowledgment) of the client’s `SYN` in a single segment.

4. The client acknowledges the server’s `SYN`.

Figure 4.1 shows the procedure of TCP’s three-way handshake. The client’s initial sequence number is shown as $J$ and the server’s initial sequence number as $K$. The acknowledgment number in an `ACK` is the next expected sequence number for the end sending the `ACK`. Since a `SYN` occupies 1 byte of the sequence number space, the acknowledgment number in the `ACK` is the number in corresponding `SYN` plus 1.
4.1.3 TCP normal termination

TCP takes four segments to terminate a connection, which is depicted as following:

1. Either the server or the client calls `close` first, which causes this end’s TCP to send a `FIN` (finish) segment.

2. The other end receives the `FIN`, and acknowledged it by TCP. The receipt of the `FIN` is also passed to the application as an end-of-file, since the receipt of the `FIN` means the application will never receive any additional data on the connection.

3. Sometime later the application that received the end-of-file will `close` its socket. This causes its TCP to send a `FIN`.

4. The TCP on the system that receives this final `FIN` (the end that did the active close) acknowledges the `FIN`.

Figure 4.2 illustrates the above procedure of TCP connection termination. Note that a `FIN` occupies 1 byte of sequence number space just like a `SYN`, therefore the `ACK` of each `FIN` is the sequence
number of the FIN plus one.

The sending of each FIN occurs when the socket is closed. In addition to the calls of close by an application, the termination of a Unix process can also make this happen. When a process terminates, either voluntarily (calling exit or having the main function return) or involuntarily (receiving signal that terminates the process), all open descriptors are closed, which will cause a FIN to be sent on any TCP connection that is still open.

4.1.4 Fault: Termination of remote process

Figure 4.3 illustrates the timeline and state transition diagram. We analyze chronologically the events happened after the fault occurs, the state transition, and possible error codes that could be generated in each state.

![Figure 4.3. Fault: Termination of remote process](image)

1. **state A**
   
   When a process terminates, as part of process termination all open descriptors are closed. This causes a FIN to be sent by the remote TCP to the local TCP, and the local TCP responds with an ACK. Local process enters state A, in which:

   - a read operation will receive end-of-file because of the FIN sent by the remote TCP. The local process can deal with end-of-file in any way it desires.
• a write operation will cause the remote TCP to respond with an RST (reset) since the process that had that socket open has terminated. This has local process enter into state B.

2. state B
Upon local TCP receiving the RST caused by the write operation in state A, local process enters state B, in which:

• The first read operation will return an error of ECONNRESET (Connection reset by peer). All following read will return 0. Local process enters state F.
• A write operation will trigger the signal of SIGPIPE (signal of broken pipe) to the process. The default action of this signal is to terminate the process. If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns error EPIPE (Broken pipe). Local process enters state F.

3. state F
In state F, the fault is detected. Local process terminates, all open sockets are closed. Local TCP sends FIN to the remote TCP to accomplish the second half of the connection termination.

4.1.5 Fault: Crashing of remote host

Figure 4.4 illustrates the timeline and state transition diagram.

Figure 4.4. Fault: Crashing of remote host
1. state A

When the remote host crashes, all the processes that are running on the machine terminate their execution, but nothing is sent out on the existing network connections. Local process enters state A, in which:

- a read operation will block forever. So we need to place a timeout on the call to read to avoid this.
- a write operation will return to the process quickly, but local TCP will continually transmit the data segment, trying to receive an ACK from the remote host. Local process enters state B.

2. state B

In state B, when the local TCP is continually transmitting the data, both read and write operations are working well. For Berkeley-derived implementations, TCP retransmits the data segment 12 times, waiting for around 9 minutes before giving up. When the local TCP finally gives up (assuming the remote host has not been rebooted during this time), an error is returned to the calling process. Local process enters state C.

3. state C

Upon the moment the local TCP finally gives up and an error is returned, local process enters state C, in which:

- The first read operation will return an error of ETIMEDOUT (Time out). All following read return 0. Local process enters state F.
- a write operation will trigger the signal of SIGPIPE (signal of broken pipe) to the process. The default action of this signal is to terminate the process. If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns error EPIPE (Broken pipe). Local process enters state F.

4. state F

In state F, the fault is detected. Local process terminates, all open sockets are closed. Local TCP sends FIN to the remote TCP to accomplish the second half of the connection termination.

4.1.6 Fault: Crashing and rebooting of remote host

With this type of fault, the first two states are the same as that with the fault of crashing of remote host. No FIN is sent out, if the local process is not actively sending data to the remote host when the remote host crashes, the local one is not aware that the remote one has crashed. When a write operation is performed, local TCP will continually send the data till time out.

At the remote side, during the booting of a machine, there is an initial period where incoming messages are not acknowledged, then in a second phase, communication is restarted. But after the rebooting, its TCP loses all information about connections that existed before the crash. Therefore the remote TCP responds to the received data segment from the local TCP with a RST.
There are two scenario here in which the local process receives time-out error and RST in different sequence. Each one results in different error codes returned, which are described as the following:

1. state \textit{A}
   When the remote host crashes, all the processes that are running on the machine terminate their execution, but nothing is sent out on the existing network connections. Local process enters state \textit{A}, in which:
   - a \textit{read} operation will block forever. So we need to place a timeout on the call to \textit{read} to avoid this.
   - a \textit{write} operation will return to the process quickly, but local TCP will continually transmit the data segment, trying to receive an \textit{ACK} from the remote host. Local process enters state \textit{B}.

2. state \textit{B}
   In state \textit{B}, when the local TCP is continually transmitting the data, both \textit{read} and \textit{write} operations are working well. For Berkeley-derived implementations, TCP retransmits the data segment 12 times, waiting for around 9 minutes before giving up.
   
   (a) state \textit{C}
   If the remote host reboots during this time, i.e. before time-out occurs, as shown in Figure 4.5, the remote process responds to incoming data segment with a \textit{RST}. Upon receiving the \textit{RST}, local process enters state \textit{C}, in which:

![Figure 4.5. Fault: Crashing and rebooting of remote host](image)

*Figure 4.5. Fault: Crashing and rebooting of remote host*

case 1: rebooting before time-out
- The first read operation will return an error of ECONNRESET (Connection reset by peer). All following read will return 0. Local process enters state F.
- A write operation will trigger the signal of SIGPIPE (signal of broken pipe) to the process. The default action of this signal is to terminate the process. If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns error EPIPE (Broken pipe). Local process enters state F.

(b) state D

If the booting of remote host takes longer than time-out period, that is, the local TCP finally gives up, an error is returned to the calling process. Local process enters state D, as shown in Figure 4.6. In this state:

![Diagram](image_url)

**Figure 4.6. Fault: Crashing and rebooting of remote host**

- The first read operation before the receipt of RST will return an error of ETIMED-OUT (Time out). All following read will return 0. Local process enters state F.
- The first write operation before the receipt of RST will trigger the signal of SIGPIPE (signal of broken pipe) to the process. The default action of this signal is to terminate the process. If the process either catches the signal and returns from
the signal handler, or ignores the signal, the write operation returns error **EPIPE** (Broken pipe). Local process enters state $F$.

- **state $E$**
  
  If there is no **read** or **write** is performed before the receipt of the **RST** from remote host, then upon the receipt of **RST**, local process enters state $E$, in which:
  
  - The first **read** operation will return an error of **ECONNRESET** (Connection reset by peer). All following **read** will return 0. Local process enters state $F$.
  
  - a **write** operation will trigger the signal of **SIGPIPE** (signal of broken pipe) to the process. The default action of this signal is to terminate the process. If the process either catches the signal and returns from the signal handler, or ignores the signal, the write operation returns error **EPIPE** (Broken pipe). Local process enters state $F$.

3. **state $F$**

   In state $F$, the fault is detected. Local process terminates, all open sockets are closed. Local TCP sends **FIN** to the remote TCP to accomplish the second half of the connection termination.

4.1.7 **Fault: Shutdown of remote host**

   Now we consider what happens if the remote host is shut down by an operator while our parallel program is running on that host. The timeline and state transition are shown in Figure 4.7.

---

**Figure 4.7. Fault: Shutdown of remote host**

---

26
1. When a Unix system is shut down, the init process normally sends the SIGTERM signal to all processes. This signal, indicated by an ellipse in Figure 4.7, can be caught to do all necessary cleanup jobs, or send warnings to inform all its peers that its host machine is shutting down.

2. init process waits for some fixed amount of time (often between 5 and 20 seconds), and then sends the SIGKILL signal to any process still running. This signal cannot be caught.

3. If the remote process does not catch the signal SIGTERM and terminate, it will be terminated by SIGKILL. When the process terminates, all open descriptors are closed, FIN is sent out, with the same procedures (state A to state F) discussed in section 4.1.4 (Termination of remote process) initiated.

4.1.8 Fault: Restart of remote host

When a machine restarts, it executes shut down first, followed by a booting. The booting procedures can be further divided into two phases. In the initial phase of booting procedure, no messages are sent by the machine, even to respond to incoming requests. During the second phase of booting, messages start to be transmitted and received as usual, however, all the knowledge about the previous connections is lost. The TCP responds with RST to all incoming messages. The timelines and state transitions are shown in Figure 4.8 to Figure 4.10, according to different scenarios in which write operation occurs in different phases of shutdown and booting.

1. state A
   In the phase of machine’s shutdown, init process sends SIGTERM and SIGKILL to terminate all processes that are running. As a result, all open descriptors are closed, and FIN is sent out. Local process enters state A, in which:
   - a read operation will receive end-of-file because of the FIN sent by the remote TCP. The local process can deal with end-of-file in any way it desires.

2. state A-B to F (case 1)
   If the first write operation occurs before booting initiates, or after booting finishes, as shown in Figure 4.8, it will cause the remote TCP to respond with an RST (reset), just as what happens in shutdown of remote host and termination of remote process. This makes local process go through state B to state F (same as step 2 in section 4.1.4: Termination of remote process).

3. state A-C
   If the first write operation occurs between booting initiates and finishes, the remote TCP can not respond to the incoming messages. So the local TCP will continually transmit the data segment, trying to receive an ACK from the remote host. Local process enters state C.
   
   (a) state D-F (case 2)
   If booting finishes before local TCP gives up retransmission, i.e. before time-out occurs, everything will then follow the same sequence of steps as discussed in step 2a in section 4.1.6: Crashing and rebooting of remote host. The timeline and state transition is shown in Figure 4.9.
Figure 4.8. Fault: Restart of remote host

case 1: write before booting initiates or after booting finishes

(b) state $E-F$ and state $E-G-F$ (case 3)
If booting takes longer than time-out period, i.e. booting finishes after local TCP gives up, everything will then follow the same sequence of steps as discussed in step 2b in section 4.1.6: Crashing and rebooting of remote host. The timeline and state transition is shown in Figure 4.10.

4.2 Implementation issues

From the discussion in section 4.1, we can see that all faults will result in no more than three kinds of error code: EPIPE, ECONNRESET or ETIMEDOUT. So if we could capture these three error codes, we are able to be aware of communication failure in time.

Based on the above analysis, statements are added after each recv-like and send-like statements to check if the size of received data is 0, or the error number returned by TCP is one of EPIPE, ECONNRESET or ETIMEDOUT. If yes, a failed process has be detected. The node who received the error number should then notify all other alive peers, and then terminate itself. Here,
great care was taken to have other nodes be notified as quickly and efficiently as possible.

We can not have the node who detected process failure broadcast the information to all other alive nodes. There are two main drawbacks with this solution. First, it is quite likely that more than one nodes detected the process failure at the same time. Then if each of them broadcasts to other nodes, there will be a message flooding which will cause unnecessary network traffics, and make the abortion inefficient. Secondly, if more than one nodes detected the failure, it is quite possible that one node sends out notices to other nodes and terminates itself, therefore is not able to receive the notices from other nodes who also detected the failure at that time. This could make the situation much more confusing and slow down responses to the abortion.

In our implementation, a new request type named REQ_ABORT is defined, and a special protocol is designed for abortion requests. For all those nodes other than processor #0, if it detects a process failure, it should report it to processor #0 by sending a REQ_ABORT request, which includes its own id, the id of the processor that is down and the error type, and then terminate. For the #0 node, upon detecting a process failure, it should send REQ_ABORT request to all other nodes except the one that is down, and then terminate. These two steps are encapsulated in a function named Tmk_abort.

In Tmk_abort_sigio_handler, the nodes other than #0 just simply terminates. The node #0
will send \texttt{REQ\_ABORT} request to all other nodes except the one that sent the request and the one that is down, and then terminate. Node #0 also keeps an \texttt{abort\_flag} to track if an abortion request has been received before when there are more than one processors reporting the process failure before it sends all notices out.

By using this fault detection mechanism, we successfully avoids the message flooding and confusions. The response to the abortion is quick, and the aborting procedure is efficient.
Chapter 5

Experiments

5.1 Experimental setup

5.1.1 Hardware platform

The experiments are run with TreadMarks on a cluster of eight Sun Blade 1000 workstat. Each machine contains dual 750MHz UltraSPARC-III processor modules and 512M of physical memory. The workstations are connected by a single LinkSys EtherFast 10/100 switch using category 5 UTP cables.

5.1.2 Software

The operating system running on the platform is SunOS 5.8.

5.1.3 Benchmark applications

Seven applications were used in the experiments: 3D-FFT, SOR, BARNES, CG, GAUSSIAN, RAYTRACE and WATER. All of them are taken from the TreadMarks distribution. The problem sizes for each benchmark are shown in Table 5.1.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>problem size</th>
<th>memory consumption</th>
<th>communication pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT</td>
<td>5000 steps on 128 × 64 × 128 array</td>
<td>48M</td>
<td>blocked, all-to-all</td>
</tr>
<tr>
<td>SOR</td>
<td>1000 steps on 6000 × 4000 grid</td>
<td>91.6M</td>
<td>irregular, all-to-all</td>
</tr>
<tr>
<td>Barnes</td>
<td>45414 particles</td>
<td>7.3M</td>
<td>irregular, hierarchical</td>
</tr>
<tr>
<td>CG</td>
<td>LARGE</td>
<td>125K</td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>2048 steps on 4096 × 4096</td>
<td>64M</td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>teapot</td>
<td>32M</td>
<td></td>
</tr>
<tr>
<td>Water</td>
<td>1500 steps on 1728 molecules</td>
<td>1M</td>
<td>structured, many-to-many</td>
</tr>
</tbody>
</table>
**FFT** numerically solves a partial differential equation using three dimensional forward and inverse FFT’s. It is assumed that the input array is $n_1 \times n_2 \times n_3$, organized in row-major order. The 3-D FFT first performs a $n_3$-point 1-D FFT on each of the $n_1 \times n_2$ complex vectors. Then it performs a $n_2$-point 1-D FFT on each of the $n_1 \times n_3$ vectors. Next, the resulting array is transposed into an $n_2 \times n_3 \times n_1$ complex array and an $n_1$-point 1-D FFT is applied to each of the $n_2 \times n_3$ complex vectors. The computation on the array elements along the first dimension of $A$ is distributed so that for any $i$, all elements $A_{ijk}$ $0 \leq j < n_2$, $0 \leq k < n_3$ are assigned to a single processor. No communication is needed in the first two phases, because each of the $n_3$-point FFTs or the $n_2$-point FFTs is computed by a single processor. The processors communicate with each other at the transpose, because each processor accesses a different set of elements afterwards.

**SOR** is a method of solving partial differential equations, it performs red-black Successive Over-Relaxation on a grid. The program divides the red and the black array into roughly equal size bands of rows assigning each band to a different processor. The execution is divided into two phases separated by a barrier. Within each phase, a processor reads the boundary elements written by the neighboring processor in the previous phase.

**BARNES** is an N-body simulation using the hierarchical Barnes-Hut method. A tree-structured hierarchical representation of physical space is used. Each leaf of the tree represents a body, and each internal node of the tree represents a "cell”, a collection of bodies in close physical proximity. The major data structures are two arrays, one representing the bodies and the other representing the cells.

There are four major phases in each time step.

1. **MakeTree**: Construct the Barnes-Hut tree.
2. **Get my bodies**: Partition the bodies among the processors.
3. **Force Computation**: Compute the forces on my own bodies.
4. **Update**: Update the positions and the velocities of my bodies.

Phase 1 is executed sequentially, because running in parallel slows down the execution. In phase 2, dynamic load balance is achieved by using the cost-zone method, in which each processor walks down the Barnes-Hut tree and collects a set of logically consecutive leaves. Most of the computation is spent in phase 3.

**CG** The Conjugate Gradient method is an effective method for symmetric positive definite systems. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors need to be kept in memory. In every iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.
**GAUSSIAN** performs Gaussian elimination.

**RAYTRACE** renders a 3-D scene onto a 2-D image plane using optimized ray tracing. A uniform hierarchical grid is used to represent the scene. A ray is traced through each pixel in the image plane. Multiple rays are generated through reflection or refraction, creating ray trees. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used for task stealing.

**WATER** uses an $O(n^2)$ algorithm to simulate the forces and potentials among Water molecules in liquid states. The main data structure in Water is a 1-D array of records, where each record represents a molecule. The record stores the molecule’s center of mass, and for each of the atoms, the computed forces, the displacements and their first six derivatives. During each time stamp, both intra- and inter-molecular potentials are computed. To avoid computing all $n^2/2$ pairwise interactions among molecules, a spherical cutoff range is applied.

The algorithm statically divides the array of molecules into equal contiguous chunks, assigning each chunk to a processor. The bulk of the interprocessor communication happens during the force computation phase. Each processor computes and updates the intermolecular force between each of its molecules and each of $n/2$ molecules following it in the array in wrap-around fashion.

### 5.2 Experimental results & discussion

This section discusses the experimental results from a variety of view points, including comparisons of UDP and TCP implementations under the same execution conditions, sequential and parallel execution times under the same protocols, with or without Fault Detection.

In all the figures presented in this section, the execution time is normalized for the sake of comparison. Each time bar is broken down into 4 following categories. The “busy” time is the time spent during useful computation. It also includes interrupt times associated with software DSM, which we cannot isolate. The “DSM overhead” represents the computation that TreadMarks performs on consistency information, including the time spent on garbage collection. The “synchronization idle” is the time the system stalls for synchronization. The “memory idle” represents the time the memory subsystem stalls for remote memory requests to be fulfilled.

#### 5.2.1 Sequential execution time

First we look at the sequential execution time under different transfer protocol implementation. Figure 5.1 shows the time bars, and Table 5.2 shows the results.

We can see that there is no difference between UDP and TCP implementations when the applications are executed sequentially. This makes sense because there is no communication overhead at all when all the program is running on a single node. So it is obvious that the implementation of
DSM won’t affect the sequential running time of the applications.

5.2.2 Parallel execution time

Figure 5.2a and Figure 5.2b show the parallel execution time compared with the sequential execution time under UDP and TCP, respectively. From the figures we can see that most of the benchmarks run much faster under both UDP and TCP because of the computation load has been divided to 8 processors that work at the same time. There are exceptions, however, with Raytrace under both UDP and TCP, and Water under TCP. With 8 processors working on it simultaneously, the

Table 5.2. Sequential Execution Times Under UDP and TCP

<table>
<thead>
<tr>
<th>benchmark</th>
<th>udp (s)</th>
<th>tcp (s)</th>
<th>increase percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT</td>
<td>23606.93</td>
<td>23666.59</td>
<td>0.25%</td>
</tr>
<tr>
<td>SOR</td>
<td>9029.50</td>
<td>9042.93</td>
<td>0.14%</td>
</tr>
<tr>
<td>Barnes</td>
<td>32.47</td>
<td>32.24</td>
<td>-0.07%</td>
</tr>
<tr>
<td>CG</td>
<td>53.33</td>
<td>53.35</td>
<td>0.04%</td>
</tr>
<tr>
<td>Gaussian</td>
<td>649.14</td>
<td>667.97</td>
<td>0.03%</td>
</tr>
<tr>
<td>Raytrace</td>
<td>0.83</td>
<td>0.83</td>
<td>0%</td>
</tr>
<tr>
<td>Water</td>
<td>34313.41</td>
<td>34057.12</td>
<td>-0.75%</td>
</tr>
</tbody>
</table>
Figure 5.2. Parallel and Sequential Execution Times Under UDP and TCP
execution times didn’t drop, but rose significantly instead.

Examining the time bars of Raytrace, we can see that the busy time when \( n = 8 \) is much smaller than that when \( n = 1 \), i.e. the parallel running of the application does help reducing computation time a lot. The increase of total execution time is resulted mainly from the synchronization idle and memory miss idle, the communication overhead between processors. This is due to the characterization of Raytrace. Raytrace is a comparatively small problem at the sense of parallelism, whose sequential running time is even less than 1 second. In such case, the cost of message passing between nodes will be dominant and it will increase the total time dramatically.

Water is a different story. Its performance under UDP is as good as other applications, with the parallel running time less than 20% of the sequential one. But under TCP, the total execution time is much larger than its sequential counterpart. From the time bars we can see that this abnormality is due to the huge increase of synchronization idle time. We’ll examine the problem of Water in depth in later sections.

5.2.3 Comparison of parallel execution times under UDP and TCP

Figure 5.3 and Table 5.3 summarize the comparison of parallel execution times under UDP and TCP. The results are the averages of the results of 3 times run.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>udp (s)</th>
<th>tcp (s)</th>
<th>increase percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-FFT</td>
<td>5939.25</td>
<td>6535.74</td>
<td>10.04%</td>
</tr>
<tr>
<td>SOR</td>
<td>2039.00</td>
<td>2046.04</td>
<td>0.35%</td>
</tr>
<tr>
<td>Barnes</td>
<td>10.10</td>
<td>14.48</td>
<td>43.37%</td>
</tr>
<tr>
<td>CG</td>
<td>19.93</td>
<td>29.35</td>
<td>47.27%</td>
</tr>
<tr>
<td>Gaussian</td>
<td>86.03</td>
<td>89.04</td>
<td>3.50%</td>
</tr>
<tr>
<td>Raytrace</td>
<td>1.21</td>
<td>2.42</td>
<td>100%</td>
</tr>
<tr>
<td>Water</td>
<td>5806.86</td>
<td>127187.93</td>
<td>21.90 times</td>
</tr>
</tbody>
</table>

From Figure 5.3 we can see that there is almost no cost increase for one of the applications (SOR), only a slightly cost increase, which is no more than 10% for two of them (FFT and Gaussian), a little bit large cost increase for two of them (Barnes and CG), a large increase of cost, almost doubled, for Raytrace, and a extremely huge one for Water, which is 22 times of the run time under UDP. Thus, among all 7 applications, 3 of them have the acceptable performance when UDP protocol is replaced by TCP.

Excluding Water, let’s focus on the six applications without or with reasonable cost increases. The tendency is shown in Table 5.3: the increase of execution time is less for longer running applications than for shorter ones. This is because that although TCP requires more packets than UDP does for exchanging a request and a reply (see description in section 4.1.2). However, if a TCP
Figure 5.3. Comparison of Parallel Execution Times Under UDP and TCP

connection is used for multiple request-reply exchanges, then the cost of the connection establishment and teardown is amortized across all the requests and replies. Thus the more messages passed between nodes, the less the total cost increase is. This is another reason to the large cost increase of the extremely short running program like Raytrace.

Different applications respond differently to the protocol change. SOR and Gaussian have the same message passing overhead under UDP and TCP; 3D-FFT stalls longer for memory miss under TCP than UDP, but not too much; most of the applications (Barnes, CG, Raytrace and Water) has more synchronization idle time under TCP than UDP. So we can conclude that changing from UDP to TCP affects the synchronization procedures most, which in turn affects the overall performance of the system.

Again, the abnormal performance of Water will be discussed in separate section.

5.2.4 Stalling time

To explore the details of the system performance, we break down the stall times in above figures into several categories as shown in Figure 5.4. For the memory miss idle time, we divide it into the stall times for cold misses (“page idle”) and coherence misses (“diff idle”). For the synchronization idle time, we divide it into the stall times for acquiring locks (“lock idle”) and at barrier arrivals (“barrier idle”).
From Figure 5.4 we can see that changing from UDP to TCP affects the page idle time least, then the diff idle time. Almost all of the applications have the same page and diff idle times under UDP and TCP. 3D-FFT has a slightly larger diff idle time under TCP than UDP, but the percentage is less than 20%. This is because large amount of data consistencies are involved in the transpose stage of 3D-FFT.

For some of the applications that only or mainly exploit barrier for synchronization, the stalling time at barrier arrivals is the main reason of the system performance degradation. The barrier idle time in Barnes increases 60% from UDP to TCP, CG increases 200%. But as we explained earlier, these two applications are so-called short running programs with sequential time less than 60 seconds, and parallel time less than 20 seconds. There are not enough requests and replies to amortize the cost of TCP connection establishment and teardown. It is the same case as in the increase of barrier idle time in Raytrace.

For those applications that exploit both lock and barrier for synchronization, there are both increases in barrier idle time and lock idle time. But the rise of lock idle time usually dominates the cost change. In Raytrace, the lock idle time increases almost 700%, and in Water, the lock idle time under TCP is high up to 85 times of that under UDP.
5.2.5 Lock idle and abnormal performance of Water

At this point we can conclude that the abnormal huge performance degradation of Water under TCP results from the significant increase of the cost for acquiring locks.

As we’ve known in section 3.4, during the procedure of acquiring locks with TCP, we have to deal with the case in which multiple messages are received as one message due to the lack of record boundaries in TCP protocol. This handling procedure involves lots of if-else statements and mapping of pointers, which obviously incurs more overhead to the lock acquisition. The number of lock acquisitions in Water is high up to 13027494, so even if the incurred overhead is rather small for one single lock acquisition, the accumulated cost would be a big number.

5.2.6 Performance with fault detection

To study the performance with fault detection, we run 3D-FFT for 5 times on both DSM without Fault Detection and with Fault Detection, respectively. The parameters for 3D-FFT is 200 iterations on $128 \times 64 \times 128$ array. The results are summarized in Table 5.4.

<table>
<thead>
<tr>
<th>run #</th>
<th>total execution time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>with FD (s)</td>
<td>without FD (s)</td>
</tr>
<tr>
<td>1</td>
<td>194.18</td>
<td>194.98</td>
</tr>
<tr>
<td>2</td>
<td>195.16</td>
<td>194.60</td>
</tr>
<tr>
<td>3</td>
<td>193.98</td>
<td>196.93</td>
</tr>
<tr>
<td>4</td>
<td>194.40</td>
<td>194.78</td>
</tr>
<tr>
<td>5</td>
<td>195.02</td>
<td>197.98</td>
</tr>
<tr>
<td>average</td>
<td>194.55</td>
<td>195.85</td>
</tr>
</tbody>
</table>

From Table 5.4 we can see that there is almost no overhead for the Fault Detection. Because our fault detection technique exploits the error codes returned by the TCP protocol, which are generated automatically when there is communication with the failed process, therefore there is no extra cost for the fault detection mechanism.
Chapter 6

Conclusion

In this project, the protocol for inter-processor communication used in TreadMarks DSM was modified from UDP to TCP to investigate the impacts of underline protocols on the performance of DSM. Then a Fault Detection mechanism exploiting error codes returned by TCP was implemented to detect process failures.

Our experiments show that changing from UDP to TCP doesn’t affect the maintenance of memory consistence very much, including stalling time for both cold misses (page idle) and coherence misses (diff idle). However, the change of protocol does degrade the system performance by increase stalling time for synchronization, especially for lock acquisition. As a consequence, the performances of those applications that depend on locks for synchronization are degraded a lot, such as Water and Raytrace. And the performances of those who mainly depend on barriers for synchronization are not affected much, such as SOR and Gaussian.

The performances of applications that run longer time are degraded much less than those run shorter time, such as Barnes, CG and Raytrace. Because the cost of TCP connection establishment and teardown is amortized among all requests and replies passing through it, the more communications a connection undertakes, the less the overhead contributes to the overall program performance.

Finally, the Fault Detection using error codes returned by TCP does not incur any overhead, because the error codes are generated automatically by TCP when a failed process is involved in communication.
Bibliography


