CDP: A MULTITHREADED IMPLEMENTATION OF A NETWORK COMMUNICATION PROTOCOL ON THE CYCLOPS-64 MULTITHREADED ARCHITECTURE

by

Ge Gan

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical and Computer Engineering

Winter 2006

CDP: A MULTITHREADED IMPLEMENTATION OF A NETWORK COMMUNICATION PROTOCOL ON THE CYCLOPS-64 MULTITHREADED ARCHITECTURE

by

Ge Gan

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Gonzalo R. Arce, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: _____

Daniel Rich, Ph.D.
Provost

ACKNOWLEDGMENTS

I would like to thank Prof. Gao for giving me the opportunity to study in CAPSL. Without his help, I would not have been able to come here and start my Ph.D. program.

I thank Dr. Hu. He is the person that gave me the opportunity to join the Cyclops-64 team and take part in such an interesting project. He shared his system software development experience with me and helped me a great deal in writing this thesis.

I thank Juan del Cuvillo, who gave me many invaluable suggestions when I was developing the CDP module.

I thank Joseph Manzano and Geoffrey Gerfin. Joseph's feedback on my CDP design was very helpful. I also thank Geoffrey Gerfin, who helped me to collect the performance data.

I thank Andrew Russo, who helped me to review this paper and gave me many useful suggestions on writing.

Lastly, I thank everybody in CAPSL who helped me in the last two years. They are very smart people. It has been a wonderful experience to work together with them.

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

A trend of emerging large-scale multi-core chip design is to employ multi-threaded architectures - such as the IBM Cyclops-64 (C64) chip that integrates large number of hardware thread units, main memory banks and communication hardwares on a single chip. A cellular supercomputer is being developed based on a 3D connection of the C64 chips. This paper introduces our design, implementation, and evaluation of the Cyclops Datagram Protocol (CDP) for the IBM C64 multi-threaded architecture and the C64 supercomputer system. CDP is inspired by the TCP/IP protocol and its design is very simple and compact. The implementation of CDP leverages the abundant hardware thread-level parallelism provided by the C64 multithreaded architecture.

The main contributions of this paper are: **(1)** We have completed a design and implementation of CDP that is used as the fundamental communication infrastructure for the C64 supercomputer system. It connects the C64 back-end to the front-end and forms a global uniform namespace for all nodes in the heterogeneous C64 system; **(2)** On a multithreaded architecture like C64, the CDP design and implementation effectively exploit the massive thread-level parallelism provided on the C64 hardware, achieving good performance scalability; **(3)** CDP is quite efficient. Its Pthread version can achieve around 90% channel capacity on the Gigabit Ethernet, even it is running at the user-level on a single processor machine; **(4)** Extensive application test cases are passed and no reliability problems have been reported.

Chapter 1

INTRODUCTION

Cyclops-64 (C64) is a multithreaded architecture developed at the IBM T.J. Watson research center [1]. It is the latest version of the Cyclops cellular architecture that employs a unique multiprocessor-on-a-chip design [2] and it integrates a large number of thread execution units, main memory banks, and communication hardware on a single chip. See Figure 1.1. The C64 chip plus the host control logics

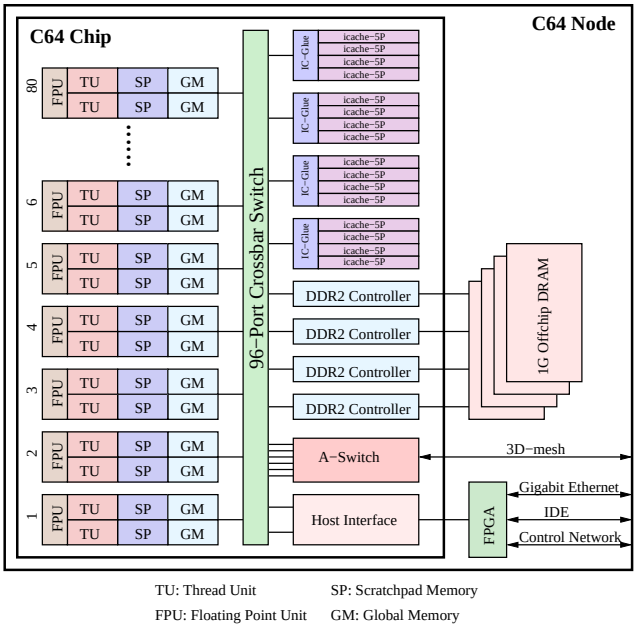


Figure 1.1: Cyclops-64 Node

and the off-chip memory becomes the building block (i.e. the C64 node) of the C64 supercomputer system. See Figure 1.2. The C64 supercomputer system consists of

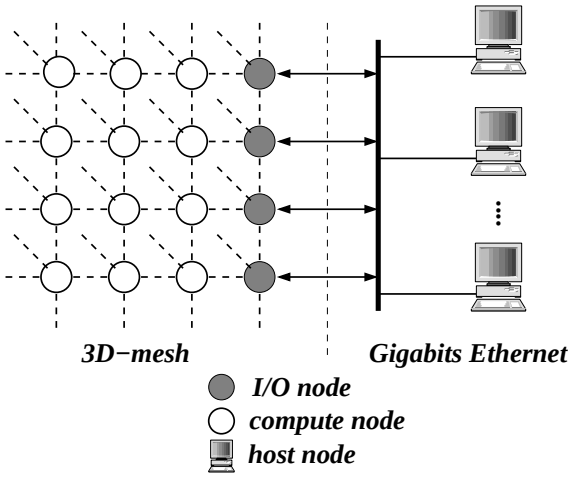


Figure 1.2: Cyclops-64 Supercomputer

tens of thousands of C64 nodes that are connected by the 3D-mesh network and the Gigabit Ethernet and can provide petaflop computing power.

To interconnect the two different subnetworks in the C64 supercomputer, we designed the Cyclops Datagram Protocol (CDP). CDP is a projection of the conventional network communication protocol (TCP/IP) to the modern C64 multithreaded architecture. It is a datagram-based, connection-oriented communication protocol and supports reliable data transfer and provides a full-duplex service to the application layer. We have implemented the very popular BSD socket API in CDP. This provides a user-friendly programming environment for the C64 system/application programmers.

We have implemented the whole CDP protocol on the C64 thread virtual machine (the C64 TVM) [3]. The C64 thread virtual machine is a lightweight runtime system that resides inside the C64 chip. It provides a mechanism to directly map the software threads onto the C64 hardware thread units. It also provides a familiar and efficient programming interface for the C64 system programmers. Currently the C64 hardware is still under development, so the C64 thread virtual machine is running on the C64 FAST simulator.

We have explored a multithreaded methodology in the development of the CDP protocol. A fine-grain thread library called TiNy Thread (TNT) library [3] is used to implement the CDP protocol. The TNT thread library is part of the C64 Thread Virtual Machine and implements the C64 fine-grain thread model [3].

We have evaluated the performance of CDP through micro-benchmarking. From the experimental results, we have two observations: **(1)** The multithreaded methodology used in the implementation of CDP is very successful. It effectively exploits the massive thread-level parallelism provided on the C64 hardware and achieves good performance scalability. The speedup of a CDP benchmark that uses 128 receiving threads is 82.55. **(2)** As a communication protocol, CDP is efficient. A Pthread version of CDP achieves around 90% channel capacity on Gigabit Ethernet, even it is running at the user-level on a single processor Linux machine.

In the next section, we will give a problem formulation and briefly introduce the our solution.

Chapter 2

BACKGROUND

As shown in Figure 1.1, a C64 chip has integrated 80 "processors", which are connected to a 96-port crossbar network. Each processor has two thread units, one floating point unit, and two SRAM memory banks, each 32KB. A thread unit is a 64-bit, single issue, in-order RISC processor core operating at clock rate of 500MHz. The execution on the thread unit is not preemptable. A 32KB instruction cache is shared among five processors. There is no data cache on the chip. Instead, a portion of each SRAM memory bank can be configured as scratchpad memory (SP), which is a fast temporary storage that can be used to exploit locality under software control. All of the remaining part of the SRAM form the global memory (GM) and is uniformly addressable from all thread units. There is no virtual memory subsystem on the C64 chip.

The A-switch interface in the chip connects the C64 node to its six neighbors in the 3D-mesh network. In every CPU cycle, A-switch can transfer one double word (8 bytes) in one direction. The 3D-mesh may scale up to several ten thousands of nodes, which form the powerful parallel compute engine of the C64 supercomputer. The C64 compute engine is attached to the host system through Gigabit Ethernet. See Figure 1.2. The whole C64 system is designed to provide petaflop computer performance. It is targeted at applications that are highly parallelizable and require enormous amount of computing power.

Given the C64 multithreaded architecture and the C64 supercomputer system, we are interested in two questions regarding the implementation of CDP:

- Is it possible to implement CDP in a way such that it effectively utilizes the massive thread-level parallelism provided on the C64 hardware and achieve good performance scalability?

- Is the communication protocol we developed for the C64 architecture an efficient one?

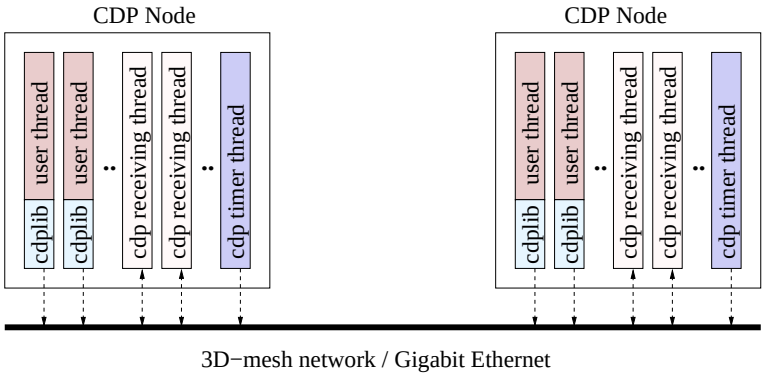


Figure 2.1: CDP Multithreaded Implementation

In order to answer these questions, we came up a multithreaded solution, shown in Figure 2.1. A CDP program at runtime consists of a set of TNT threads [3]: the CDP receiving threads, the CDP timer thread, and the CDP user threads. The receiving threads are responsible for handling the asynchronous events specified in the CDP protocol implementation, while the timer thread handles the synchronous events. The CDP code called by the user threads implements the semantics defined by the BSD socket API. These threads cooperate with each other to implement the full semantics and functions of the CDP protocol. A find-grain lock algorithm is proposed to ensure that operations on different CDP connection can be done in parallel. Chapter4 will give a detailed description.

The rest of the paper is organized as follows. Section 3 briefly introduces the CDP communication protocol. Section 4 discusses the multithreaded implementation of CDP. Section 5 presents the experimental results and analysis. Section 6

introduces some related works. Section 7 is our conclusion. We will talk a little about our future work in section 8.

Chapter 3

CDP PROTOCOL

CDP is inspired by TCP/IP, yet simpler and more compact. See Figure 3.2. Such a design is based on the consideration that both the C64 architecture and the network topology of the C64 supercomputer are simple. For the C64 chip, each thread unit is a single-issue RISC core. Its execution is not preemptable and there is no virtual memory subsystem. These features indicate that the C64 chip is not good at running complicate control intensive programs. Meanwhile, there are only two subnets in the C64 supercomputer system (Figure 1.2). One of them is reliable (the 3D-mesh), the other one is very stable (Gigabit Ethernet bit-error-rate is smaller than 10^{-10}). Given these properties, we are able to make some customizations to make the protocol simpler. This helps us to focus on studying our multithreaded implementation method.

It is not our intention to discuss the cutting-edge techniques for network protocol design in this paper. So, we will only briefly introduce the CDP protocol here and focus on protocol implementation in the next section.

3.1 Overview

Figure 3.1 shows the position of CDP in the protocol stack. According to the OSI reference model, CDP corresponds to the *Transport* layer plus the *Network* layer. This implies that CDP should implement the main functions (or at least some) of these two layers that are specified in the OSI reference model. Below are the main features of CDP protocol:

- CDP is a datagram-based, connection-oriented communication protocol.
- CDP is a reliable communication protocol. It supports timeout retransmission on the CDP connection.
- CDP provides simple packet routing function.
- CDP uses sliding-window based flow control mechanism to avoid network traffic congestion.
- CDP provides a full-duplex service to the application layer.

The CDP library has implemented the very familiar BSD Socket programming interfaces for the CDP program developers.

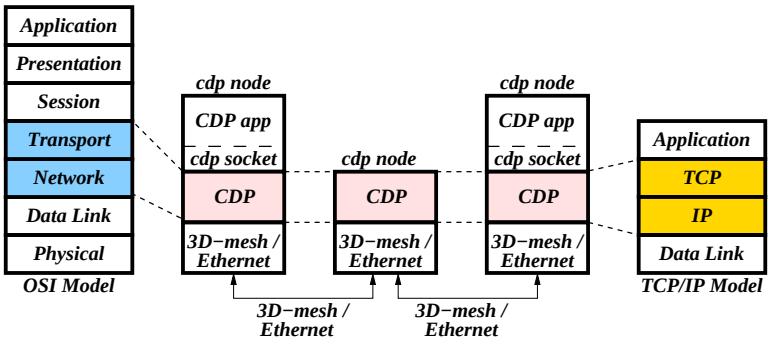


Figure 3.1: OSI Reference Model, TCP/IP Reference Model, and CDP Protocol Stack

3.2 Protocol Design

Figure 3.2 shows the CDP header format. The CDP header can be viewed as a merging of the IP header into the TCP header (or reverse) with some customizations being applied.

In Figure 3.2, the *destination node* is used for addressing and routing. The 4-tuple <*destination node, destination port, source node, source port*> is used to identify a unique CDP connection, while the *"sequence number"* field identifies an

individual CDP packet on a specific connection. CDP does not support selective or negative acknowledgments. So the receiver uses the *"acknowledgment number* field to tell the other side that it has successfully received up through but not including the datagram specified by the *"acknowledgment number*. The *"flags"* field contains some control flags similar to TCP header.

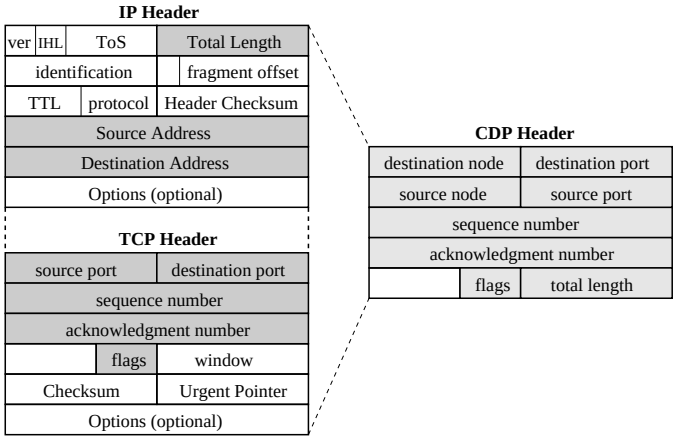


Figure 3.2: CDP Packet Header Format

We do not allow *fragment* and *defragment* in CDP. Furthermore, also do not calculate checksum for the CDP datagram. This is because both underlying subnets are error-free.

As for the CDP connection, the finite state automata used to direct the connection state transition is shown in Figure 3.3. This finite state automata is similar to the one used in TCP/IP. The difference is that, instead of using a 4-way handshake protocol [4] to terminate a connection, CDP uses a simplified 2-way handshake protocol. This is because we do not want to support a "half-close" CDP connection. This makes sense to most applications. When one end of the communication closes its connection, it always means that it does not have any data to send out and that it does not want to receive any data from the other side. So, there is no problem to close the whole connection. With this simplification, four

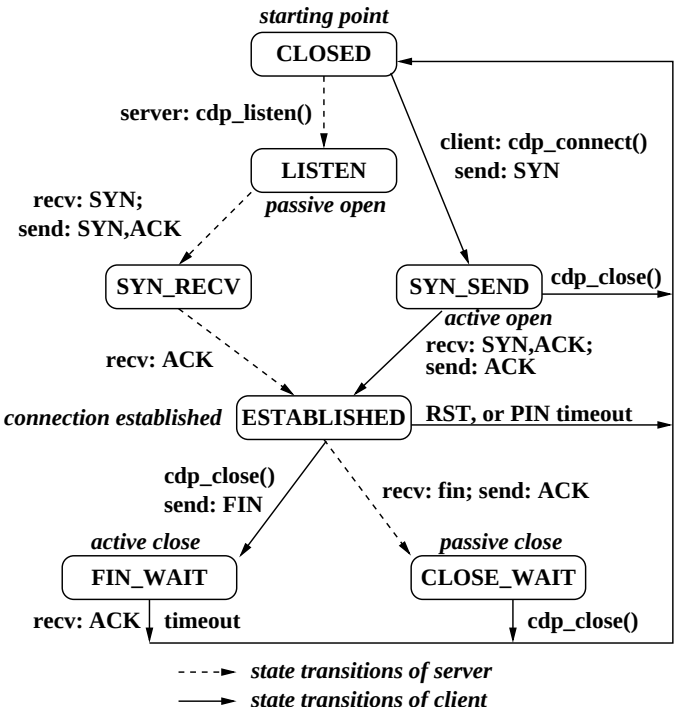


Figure 3.3: CDP State Transition Diagram

states (*FIN_WAIT_2, CLOSING, TIME_WAIT, LAST_ACK*) are are removed from the CDP state transition automata.

Chapter 4

CDP PROTOCOL IMPLEMENTATION

In this section, we will introduce the multithreaded methodology we used to implement the CDP communication protocol. (If not otherwise specified, the word "CDP" always refers to the implementation of CDP.) The internals of CDP can be viewed as a collection of data objects (e.g. socket) and a set of TNT threads that operate on those data objects in parallel. Figure 4.1 shows the global picture of a CDP program at runtime.

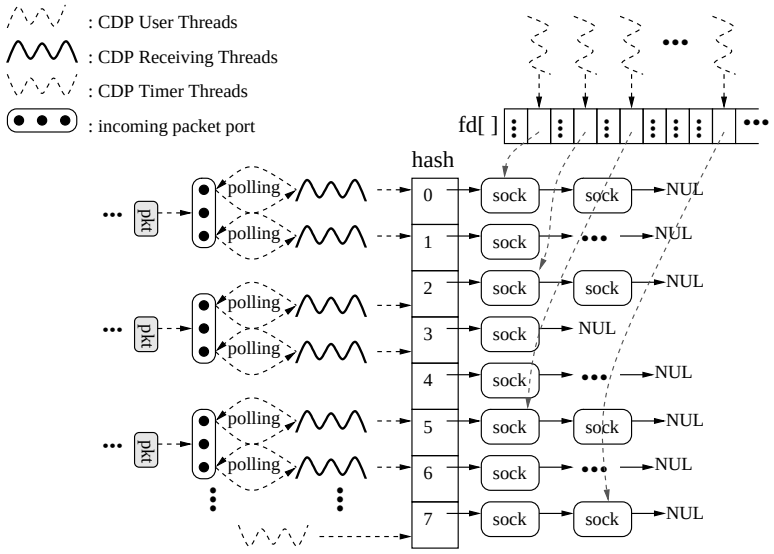


Figure 4.1: CDP Threads and Data Objects

The implementation of the CDP protocol has defined some important data structures and program constructs. Here I will give a brief introduction to each of them. In the latter section, I will present a detailed description.

- CDP Socket: A data structure used to implement the CDP connection. All resources and information that are required by a CDP connection are stored here.

- Socket Buffer: A data structure used to hold a CDP packet. It also maintain the status information of the CDP packet.

- Send Buffer: The buffer for outgoing CDP packets. It holds all packets that have been sent out but not acknowledged.

- Receive Buffer: The buffer for incoming CDP packets. It holds all packets that have been received.

- Slide Window: A logical data structure used to implement CDP packet retransmission and flow control.

- CDP Receiver: A daemon thread. It listens on the Service Access Point (SAP) of the underlying layer protocol for the incoming CDP packets.

- CDP Timekeeper: A daemon thread. As its name suggests, it is the clock in the universe of the CDP program. Its main job is to trigger synchronous events of the CDP program at runtime. One of most important synchronous events is the retransmission of the outgoing CDP packets in the sending buffer of the CDP socket.

4.1 CDP Socket

The most important data structure in CDP is the CDP socket (or socket for short). Socket has two functions. First, it acts as the interface (through the file descriptor: *fd[]*) to the user; second, it represents the CDP connection endpoint. All the important information about a CDP connection, like *address, state, receiving buffer, sliding-window*, etc, are maintained in the socket. When the user wants to

open a connection, a socket is created first. All the operations on the connection are actually performed on the corresponding socket. The sockets are linked into hash lists to improve the efficiency of socket searching. See Figure 4.1. The hash key is a function of the 3-tuple <*destination port, source node, source port*>. The hash function ensures that each hash list is evenly populated. Locks are attached to the socket and the hash list to guarantee mutually exclusive access.

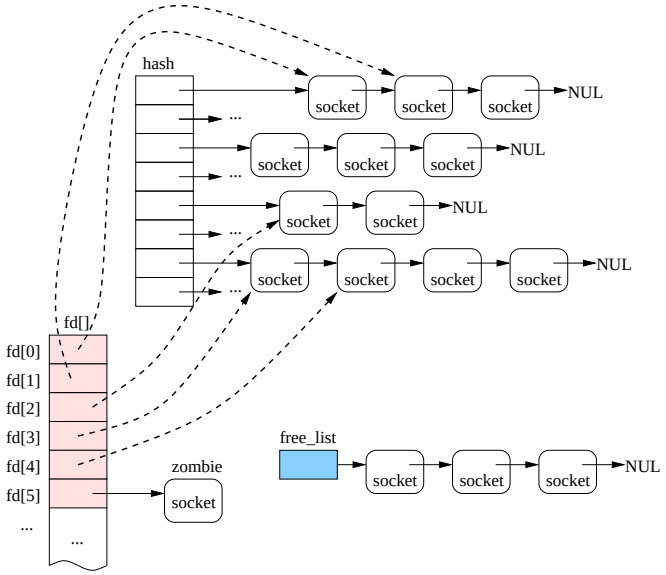


Figure 4.2: CDP Socket Hash List, Free List, and File Descriptor Array

4.1.1 Socket: Hash List and File Descriptor

A CDP node in the system may create multiple connections. Therefore, it needs to maintain a lot of CDP sockets. These sockets are organized in a hash list, as shown in Figure 4.1. The active sockets are hashed by the tuple <*node, port*>, which is the CDP address of the node at the remote side of the connection. This mechanism will make the CDP sockets evenly distributed in the hash list and make the searching for a specific CDP socket fast.

The internal CDP threads access CDP sockets through the hash list, while the user threads access CDP sockets through a small integer, which, by convention,

is called the *File Descriptor*. It is an index into the file descriptor array. This is shown in Figure 4.1. This design is derived from the UNIX convention, in which the sockets and the files are viewed as the same data objects and are accessed through the same programming interfaces provided by the file system. The core data object used in the programming interfaces of the UNIX file system is the *file descriptor*. Each element of the file descriptor array stores a single pointer points to the corresponding socket data object in the hash list.

4.1.2 Socket: Establish a CDP connection

The CDP socket is created when the user thread calls function *cdp_socket*. The function first allocates an empty CDP socket from the CDP socket free list. If the free list is running out of empty sockets, it will pre-allocate a few from the system memory allocator, insert them into the free list, and, at the same time, return one of them to the user thread. Then, it tries to obtain a free file descriptor from the file descriptor array. If the file descriptor array does not have an empty slot, it will extend the array exponentially and return a free file descriptor from it. The newly created CDP socket is associated with newly allocated file descriptor instantly. The socket is still in the *closed* state. It will not be moved to the hash list until the user thread calls function *cdp_connect* or *cdp_listen* on the corresponding file descriptor.

cdp_connect is usually called by the CDP client. It sends out, on behalf of the socket specified in the argument list, a *syn* packet to the remote side to request to establish a CDP connection. It then puts the socket in *syn_sent* state and move it to the CDP socket hash list. At the same time, a timer is setup on the socket for the completion of the creation of the connection. The *syn_sent* socket stays in the hash list, waiting for the replying *syn-ack* packet from the server side. If it receives the expected *syn-ack* packet before the timer expires, it will be transformed to *established* state, which means the CDP connection has been established. Otherwise, it will be

put into *closed* state and be moved out from the hash list by the CDP timekeeper thread. Users can release the closed socket by calling function *cdp_close*.

On the server side, the socket will be put into *listen* state and moved to hash list by calling function *cdp_listen* on the corresponding file descriptor. The socket will stay in the hash list, waiting for a *syn* packet from the client side that requests to establish a CDP connection. After the server receives a *syn* packet, it will reply with a *syn-ack* packet accordingly. Instead of creating a new socket which is in *syn-recv* state, the server just inserts the *syn-ack* packet into the sending buffer of the listening socket and then sets up a timer on it for the completion of the creation of the connection. If the server receives the *ack* packet for this *syn-ack* packet before the timer expires, it means the 3-way handshake protocol is finished and the connection has been established. At this moment, a new socket is allocated for this connection. It is set directly to the *established* state. Instead of moving the new socket to hash list, it is inserted into the *.established* socket list of the listening socket. The user thread will call function *cdp_accept* to obtain this socket, or connection. If the server does not receive the expected *ack* packet before the timer on the corresponding *syn-ack* packet expires, the *syn-ack* packet will be dropt, and no other action is taken, as if the connection is terminated silently.

As we may notice, there is no socket that is in the *syn_recv* state in the 3-way handshake protocol of the CDP connection creation process. This is a little bit different from the CDP design that has been discussed in section 2.3 and Figure 3.3. In our implementation, on the server side, the time to create a new CDP socket is delayed to the moment when the connection creation is success. The CDP packet *syn-ack* serves as a placeholder for the "conceptual" *syn_recv* socket which exists in the design. To be more specific, the *syn_recv* socket only exists in the design, not in the implementation. We made this change based on two observations:

- First, according to the protocol design, *syn_recv* is an ephemeral state for

a socket in the 3-way handshake process. There are not many operations performed on it during that period. Its only purpose is to wait for the acknowledgment to the *syn-ack* packet that was received just now.

- Second, delaying the allocation of a new CDP socket avoids the redundant alloc/free operations in case when the CDP connection establishment fails.

According to these observations, not introducing a *syn_recv* socket in the 3-way handshake process simplifies the protocol implementation without sacrificing any semantics of the CDP protocol.

4.1.3 Socket: Terminate a CDP connection

If the user thread terminates one of its CDP connections, the corresponding socket will be closed and be moved from the hash list to the free list. The closed socket will stay there until it is reused by another CDP connection session. In some cases, the CDP connection is terminated by the remote side. When this happens, the local socket associated with this connection will be closed and moved out from the hash list. But it will not be enqueued into the free socket list. It is still pointed to by the file descriptor. It waits for the local user thread to copy the user data left in its receiving buffer. After that, the user can safely close it. This scenario is clearly shown by the file descriptor *"fd[]"* in Figure 4.1. We call this kind of socket as *zombie* socket. Because the CDP connection represented by this socket has already been terminated, but there are still some data and resources need to be reclaimed by the user threads.

4.2 CDP Socket Buffer

CDP Socket buffer is the data structure used to hold a CDP packet. Logically, CDP socket buffer is composed of two parts: the *data block* and the *control block*, as shown in figure 4.3. The *data block* part holds the real protocol packet, while the

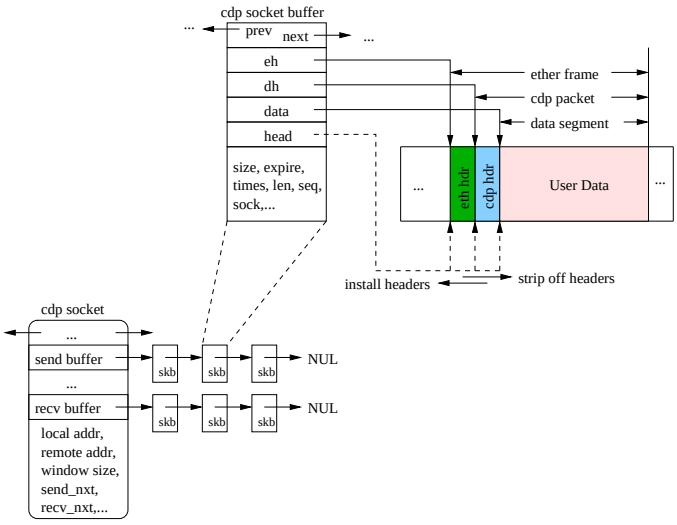


Figure 4.3: CDP Socket and Socket Buffer

control block part contains some meta-data that are used to manipulate the protocol packet. These include a set of pointers that point to different headers of the packet, the sequence number, the number of times that this packet has been transmitted, the number of seconds that the current timer on this packet will expire, the length of the current Protocol Data Unit, i.e. PDU, etc.

A new CDP socket buffer is allocated when the user thread calls the function *cdp_send* on a local CDP socket to send some data to the remote side. The user data will be packed into a protocol packet. At each layer of the protocol stack, the protocol header will be concatenated with SDU (Service Data Unit) to form the PDU (Protocol Data Unit) of this layer. The PDU will be handed over to the lower layer protocol for further processing until it is transmitted. On the receiver side, a new CDP socket buffer is allocated to hold an incoming packet. The packet will be processed at each layer of the protocol stack where the protocol header will be stripped off from the packet. The remaining part of the packet will be passed to the upper layer until the user get the data being transferred. Figure 4.4 shows the details of these processes.

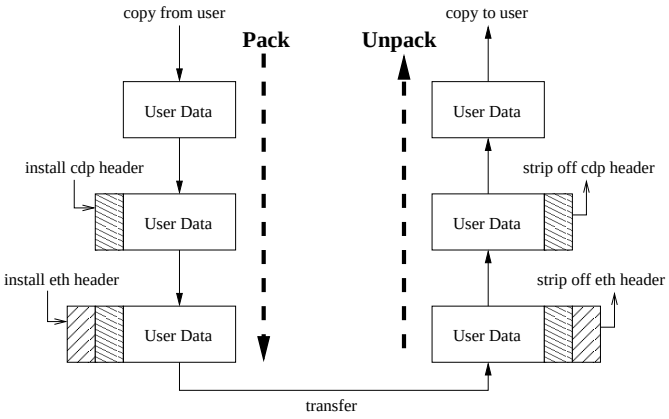


Figure 4.4: Pack and Unpack CDP Packet

Notice that in Figure 4.4, at the boundary of adjacent protocol layers in the protocol stack, the protocol data unit of the current layer needs to be copied to the next layer (upper layer or lower layer). To avoid unnecessary memory copies in the CDP program, we invented the "push" and "pull" operations, by which the memory copies are reduced to a couple of pointer movements. Look at figure 4.3 and the code below:

```
unsigned char *
skbuf_push(struct cdp_skbuf *skb, int len)
{
    skb->head -= len;
    skb->len += len;
    return skb->head;
}

unsigned char *
skbuf_pull(struct cdp_skbuf *skb, int len)
{
    skb->head += len;
    skb->len -= len;
    return skb->head;
}
```

The CDP socket buffer maintains a tuple <*head, len*> in its control block. *head* is a pointer points to the first byte of the protocol data unit of the current protocol

layer. *len* is the value that records the length of the current protocol data unit. When a new socket buffer is allocated, enough headroom is reserved in its *data block*. Therefore, when a CDP packet is crossing the boundary of the protocol layers, we can call function *skbuf_push* to "push" enough space into the packet to install the current protocol header. Or, we can all function *skbuf_pull* to strip off a protocol header from the packet. No memory copy is involve in this process. Only a small number of memory locations, i.e. the tuple - <*head, len*>, are changed. This optimization on memory operations can greatly improve the performance of CDP program. Another optimization on memory operations that needs to be mentioned here is the socket buffer pool. The CDP socket buffer is used intensively in the CDP program. It is allocated for every incoming and outgoing packet. It is also need to be freed when a CDP packet has been acknowledged or dropt. In order to avoid intensive malloc/free function calls, we create a CDP socket buffer pool at the initialization stage of the CDP program. The pool contains a certain number of CDP socket buffers with different sizes. The number of socket buffers in the pool is kept above a value, which is the watermark of the pool. When the socket buffers in the pool are exhausted or the number of socket buffers in the pool is below the watermark, a big chunk of memory will be requested from the system and a certain amount of socket buffers will be made out of it. When the number of socket buffers in the pool is greater than an upper bound, say double watermark, some socket buffers will be freed to the system to keep the number of socket buffers in the pool no more than 1.5*watermark. In this way, we reduce the number of malloc/free function calls to less than 0.1% of which they are supposed to be called. This makes the memory management in CDP more efficient.

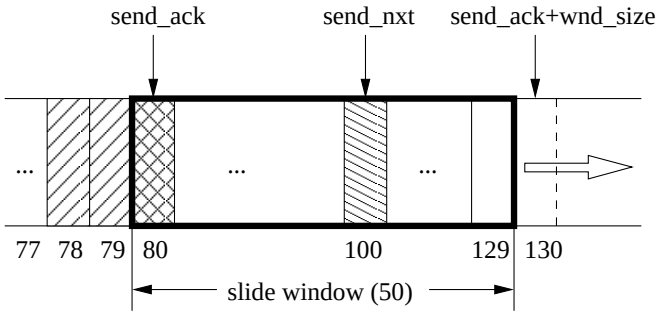
4.3 Sending Buffer, Receiving Buffer, and Sliding Window

To enable transmit flow control, we implement sliding window in CDP. Sliding window allows the CDP sender send out a specified number of CDP packets before

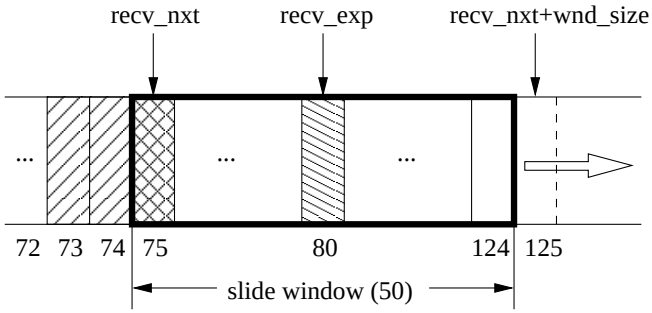
the first packet is acknowledged. On the receiver side, the CDP receiver will drop any CDP packet that is outside of the sliding window. This will avoid network congestion and improve the throughput of the link. Lets use the example shown in figure 4.5 to illustrate the flow control mechanism. In this example, we assume the sliding window size is 50.

In the CDP socket of the sender side, the program maintains a pair of numbers *(send_ack, send_nxt)* that specify the current status of the sliding window. *send_nxt* is the number that will be assigned, as the sequence number, to the next outgoing CDP packet when the user thread calls *cdp_send* and *send_ack* is what tells that all outgoing packets with "smaller" sequence number have already been acknowledged by the receiver on the remote side. Here, the quotes surrounding the word *smaller* is to deliver the information that the case that the sequence number wraps around at $2^{64} - 1$ has been taken into account. Look at the example in figure 4.5 (a). The value of *send_ack* is 80. Therefore, packets 79,78,77,... has already been sent out and been acknowledged by the receiver. The sender now expects to receive the acknowledgment to packet 80, the one that is still on the flight. We can still send out new packets to the receiver, as long as it is inside of the sliding window. The next one is packet 100, specified by variable *send_nxt*. We can send all packets with sequence number from 100 to 129 and stop at 130. We can not send out packet 130 until we get the acknowledgment to packet 80, when the sliding window is stepping forward and have packet 130 inside of it.

On the receiver side (see figure 4.5 (b)), the value of *recv_nxt* is 75, which means that the user has already gotten packets 74,73,72,..., and the next packet it will get from the receiving buffer is packet 75 if it calls *cdp_recv*. The packet that CDP receiver wants most to receive is packet 80, specified by the variable *recv_exp*. These values indicate that packets 79,78,77,76, and 75 have already been received by the CDP receiving thread and they are now in the receiving buffer, waiting for a



(a) Sliding Window of Sending Side



(b) Sliding Window of Recving Side

Figure 4.5: Snapshot of Sliding Window: Send & Recv

chance to be copied by the user thread. The difference between the packet specified by *recv_nxt* and *recv_exp* is that the former one might have been received by the CDP receiver thread, but the user didn't get a chance to copy it from the receiving buffer; the latter one has not yet been received by the CDP receiving thread and it is the packet that the CDP receiving thread want most to receive. Any packet that has sequence number outside of the sliding window will be dropt by the CDP receiving thread, like packet 125.

From the example in Figure 4.5 (a) and (b), we can conclude that packet 80 was lost, because the sender has sent it out (and is waiting for the acknowledgment) while the receiver has not gotten it (still waiting for packet 80). Since the sender has not gotten the acknowledgment to packet 80, it will continue to retransmit it until the receiver receives it and acknowledges it. Otherwise, the sender will give up

if the retransmission exceeds a specified number of times.

4.4 CDP Threads

Figure 4.1 shows that there are three kinds of threads in a CDP program: the user thread, the receiving thread, and the timer thread. These threads cooperate with each other to realize the full semantics and functions of the CDP protocol.

The user threads are created by the user. There can be a lot of them. They are not part of the CDP implementation. But the user threads may call CDP API (*send(), recv(), bind(), listen()*, etc.) to access the internal CDP data objects, especially the socket. The user thread may establish a lot of CDP connections at runtime. However, in any transaction with the CDP module, it can only work on one connection, i.e. one socket. The user thread obtains the socket through a *file descriptor*, following the Unix convention. The user threads never travel the hash lists to search for a socket. Sometimes, the user threads may insert a new socket into the hash list (by calling **accept()** or **connect()**), but they never delete sockets from the hash list.

The receiving threads are created by the C64 runtime system as TNT threads [3] and therefore, the execution of receiving thread is not preemptable and can not be interrupted. They always poll on the *"incoming packet port"* for new packets. See Figure 4.1. These *"incoming packet ports"* are the places where the underlying protocol handler will put the incoming packets. All receiving threads are doing the same type of work: polling on a specific *port* for incoming packets; fetching a packet from the port if there is one available; searching the socket hash list and looking for the socket that needs to take this packet; processing the packet and the socket according to the operations specified by the CDP protocol; the packet is dropped or queued into the receiving buffer of the socket according to the result of the processing. The receiving threads neither insert sockets in the hash list, nor delete sockets from the hash list.

There is only one timer thread in the C64 runtime system. The timer thread is responsible for processing the synchronous events in the CDP program. A large number of these synchronous events are the timeout retransmission of CDP packets. Every one second, the timer thread is woke up from sleep by the hardware timer. It then traverses every hash list and visits every socket to handle the timeout events. If the timer thread founds that the current socket being visited is in *closed* state, or needs to be closed, it will remove the socket from the hash list. Timer thread will go to sleep again after it finishes visiting all the sockets in the program. Timer thread is the only thread that can remove socket from the hash list, but it never inserts new socket into it.

4.5 Parallelism

Generally, the performance of a network protocol is largely decided by the efficiency of the receiving side. This is easy to understand because it does not make much sense to send out more data if the receiver can not receive it. Therefore, the performance of CDP can be stated as: *"the number of CDP packets that can be processed per time unit by the receiving side"*. This can be characterized by the equation below:

$$P = \bar{N} \times t \times \rho(t) \tag{4.1}$$

In equation 4.1, $\bar{N}$ is the average number of packets can be processed by a single receiving thread per time unit, assuming that the underlying network link has infinite bandwidth. Its value is inverse proportional to the number of operations that need to be performed when processing one CDP packet. Actually, this is largely decided by the protocol design. t is the number of receiving threads used in the system. It is treated as a configurable system parameter. $\rho(t)$ is a factor that measures the parallelism in the program. $\rho(t)$ is a function of t. If t increases, $\rho(t)$ will decrease because the overhead of resource contention increases. The maximum value of $\rho(t)$

equals to 1 when t is 1. Generally, $\rho(t)$ is decided by the resource contention among the CDP threads. Higher contentions causes lower parallelism, which means smaller value of $\rho(t)$. So, $\rho(t)$ is inverse proportional to the resource contention.

According to Figure 4.1 and the discussion above, there are two kinds of resources that may limit the parallelism of a CDP program: the *incoming packet port* and the *socket hash list*.

The *incoming packet port* can be implemented as a container data structure (list, queue, etc.). A lock is associated with each port to guarantee mutually exclusive access. The *incoming packet ports* are the interface between the CDP receiving threads and the underlying device drivers. Since the cost of copying a new CDP packet from the port is almost a constant, the only thing that may have great impact on the performance scalability of CDP is the number of ports used in the C64 runtime system. The experimental results show that one *incoming packet port* can support 16 receiving threads without harming the performance scalability too much. Chapter5 has a very detailed discussion on this. Here we will focus on the *socket hash list*.

The access efficiency of the *socket hash list* has great impact to the performance scalability of CDP. This is because all threads need to access the *socket hash list* before they can do any operation on a socket. The only exception is that the user thread may access the socket directly through the *file descriptor*. There are three kinds of operations performed on the hash list: socket insertion, socket deletion, and socket searching. The socket insertion operation happens when a connection is established and is only performed by the user threads. The socket deletion operation happens when a connection is closed and is only performed by the timer thread. These two kinds of operations are not as frequent as the socket searching operation, which happens every time when an incoming packet is received by a receiving thread. All these operations on the hash list need to be performed exclusively if

they may cause data conflict.

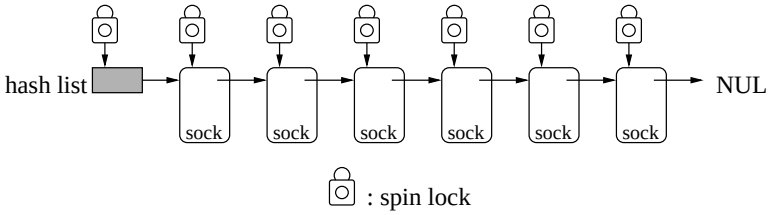


Figure 4.6: Hash List: Coarse-Grain Lock

A coarse-grain lock solution is shown in Figure 4.6. A spin lock is attached to every socket on the hash list. No matter what operation (insertion, deletion, and searching) is performed on the hash list, the thread first tries to obtain the spin lock associated with the list head. If the thread can not get the lock, it will busy wait; if the thread gets the lock, it will hold it until the operation is finished. The spin lock on the socket is to make sure that no two threads operate on the same connection at the same time. So, after a thread finds the expected socket on the hash list, it will also try to obtain the spin lock on the target socket before it releases the lock on head node of the the hash list.

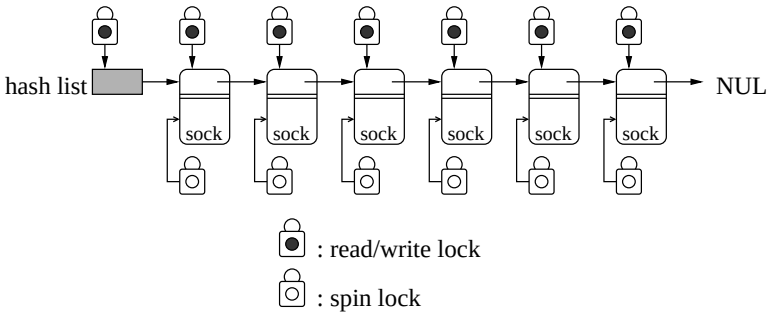


Figure 4.7: Hash List: Fine-Grain Lock

It can be easily figured out from the description that multiple receiving threads are forced to traverse the hash list sequentially, even though they may search for different sockets. A more efficient solution is shown in Figure 4.7. The original spin lock is splitted into two: one is the read/write lock that is only used

to protect the list pointer recorded in the socket; the other is the spin lock used to protect the connection related data fields. The spin lock on the list head is also replaced with a read/write lock. This fine-grain lock scheme allows multiple receiving threads to traverse the hash list in parallel. When the receiving thread wants to search a socket on the hash list, it does not need to lock the whole list. It only needs to *read_lock* the read/write locks attached to the current node being visited. When the user threads or timer thread wants to perform insert/delete operations on the hash list, they need to *write_lock* the read/write lock on the list node to make sure mutual exclusion is enforced. For the socket insertion operation, the new socket is always inserted on the list head. Therefore, only the read/write lock on the list head needs to be *write_lock'ed*. For the socket deletion operation, two consecutive list nodes need to be *write_lock'ed*. They are the node to be deleted and the node previous to it. Since only the timer thread can do socket deletion, it is the only thread that tries to grab two locks at the same time. Therefore, deadlock will never happen. Although a socket insertion or socket deletion operation will force other threads that access the same socket to wait, it does not lock the whole linked-list. The threads that work on other segment of the list can still proceed.

We did not consider using lock-free algorithms [5] [6] to implement the socket hash list. [5] uses extra *auxiliary nodes* in the linked-list to help implementing lock-free operations. This algorithm consumes more memory and makes the linked-list structure and operations more complicated than our algorithm. [6] depends on the hardware double-compare-and-swap atomic primitive which is not supported on the C64 architecture.

The fine-grain lock solution leverages the different linked-list access patterns of different CDP threads. The philosophy of this scheme is to make the common cases fast and keep the the whole design simple. In Chapter5 we compare the coarse-grain lock and the fine-grain lock scheme by experiments. The experimental

results demonstrate that the fine-grain lock solution has much better performance scalability than the coarse-grain lock solution.

4.6 Programming Interfaces

We have implemented the exact BSD socket API in CDP. They are **socket()**, **bind()**, **listen()**, **connect()**, **accept()**, **send()**, **recv()**, and **close()**. Except some minor differences in the argument list, the semantics of these functions are the same as in the BSD socket API [7]. Therefore, CDP supports the client/server programming model. Users can only access the *socket* data object through the *file descriptor*, which follows the Unix convention. See the *"fd[]"* array in Figure 4.1. Below is a detailed description of each interface:

- `int cdp_init(int node_num, const char *ifv[], int ifi, int forward);`

  cdp_init is the only function that is required on the host side but not on the Cyclops-64 side. It is used to do some important configuration for the cdp library: libcdp-host.so on the host machine and must be invoked before any cdp_XXX function can be called. The function accepts two parameters. The first one is the node number of the host machine(or cyclops node) where the cdp application is running. This node number is a global name for each host machine(or cyclops node) in the CDP communication domain of the Cyclops-64 computer system. The second parameter is the name of the Ethernet interface used by the cdp library. Usually, it is eth0, eth1, or eth2 etc. The exact value depends on the configuration of the host machine. The function will return a non-zero value if it meets any problem during execution. Otherwise, it will return zero to indicate the success.

- `int cdp_socket(void);`

  cdp_socket creates a communication endpoint for the application that uses cdp. This communication endpoint is called cdp socket, just like its sibling in BSD

socket library. -1 is returned if an error occurs; otherwise the return value is a descriptor referencing the socket. The only difference is that, in BSD socket library, the socket descriptor is in the same name space as file descriptor, but in cdp library, they are in different name space. The cdp socket created by this procedure will be assigned the local address: the node-port pair. node is the node number passed as the argument of cdp init (on host) or obtained from my pe (on cyclops). port is assigned by the system, which is unique on the current node. Most of other cdp * functions need this socket descriptor as its argument to reference the corresponding socket. Upon return of this function, the socket is in open state and is ready for further configuration, such as bind and connect. But it is not ready for send or recv data. This function accepts no parameters.

- `int cdp_close(int s);`

  Calling this function will close the socket pointed to by the socket descriptor sockfd. After this function call, the cdp socket referenced by sockfd is reclaimed by cdp library and can no longer be used to send or recv data. On success, it returns zero. Otherwise, a non-zero value is returned.

- `int cdp_bind(int s, struct cdp_sockaddr *local_addr);`

  This function is used to set the local address for the socket referenced by sockfd . The local address is specified by the formal argument *local_addr*, which is a pointer that points to a data structure declared like this: struct cdp_sockaddr int node; int port; ; The (node,port) pair is similar to the (ip,port) pair in TCP/IP protocol. It identifies a unique communication end point in cdp communication domain of Cyclops-64 computer system. The four-tuple (source node, source port, destination node, destination port) identifies a unique cdp connection. On this connection, the data transfer is datagram based and is reliable. Every cdp packet send out from this socket is attached

with this local address as its source address. On success, it returns zero. Otherwise, a nonzero value is returned.

- `int cdp_listen(int s, int backlog);`

  This function will put a socket to listen state. Socket in listen state always serves as the master socket of a server application. It listens on a well known port and ready to receive connection request from the client side. After the client and server cooperate to finish the three- way handshake protocol (the same as the connection creation protocol in TCP), the server will create a socket in established state and let the master socket, or listening socket, to take care of it. Once the user call cdp accept on this listening socket, it will return the user with a connection accomplished socket.

- `int cdp_accept(int s, struct cdp_sockaddr *addr, int nonblocking);`

  This function is usually called on a master socket (in listen state) of a server application. If there are connection established sockets in the completed connection queue (see Unix Network Programming, Vol1, Page104), one of the connected socket will be removed from the queue and returned to the user. If there are many, which one is removed is uncertain. If there are no such kind of socket in that queue, the behavior of the calling thread depends on the value of nonblocking. If nonblocking equals to zero, the calling thread will sleep on this socket until there is a connection established. If the value of nonblocking is not zero, this is a nonblocking call. The calling thread will return instantly. Upon return, the descriptor of the newly connected socket is returned. If there is any error, a -1 will be returned.

- `int cdp_connect(int s, const struct cdp_sockaddr *remote_addr);`

  This function will initiate the three-way handshake transaction with the server. Usually, the server application is listening on a socket with well known port number. The client send out the connection creation request to that socket.

The client and server will cooperate to finish the three-way handshake protocol. After that, a CDP connection will be established. For detailed information about three-way handshake, please refer to any books on TCP/IP protocol. The calling thread may sleep, waiting for the acknowledgment from the server side. On success, a zero will be returned, otherwise, a non-zero value will be returned.

- `int cdp_send(int s, const void *buf, size_t len);`

  This function is used to send a certain amount of data to the remote side of the cdp connection. The identifier of the cdp connection is recorded in the socket referenced by sockfd in the argument list. It is a four-tuple (source node, source port, destination node, destination port), in which the source address is set by calling function cdp bind and the destination address is set by calling function cdp connect. The data chunk to be send is pointed to by buf and its size is len . The size of the data chunk being sent should not exceed 1480 bytes. This is because the cdp protocol can not do fragmentation and de-fragmentation, and the MTU size of Ethernet is 1500. After subtracting the size of the cdp header, we get this number. On success, it will return the number of bytes being send successfully. Otherwise, a negative value is returned.

- `int cdp_recv(int s, void *buf, size_t len, int nonblocking);`

  This function is used to receive a cdp datagram from the socket specified by sockfd . The data being received will be put in the bu er pointed to by buf and its size is given by len . The function will block the thread that make the call if nonblocking equals to zero and there is no CDP packets in the receiving queue. The thread making the call will sleep on the socket until it receives a CDP packet. If nonblocking is not zero, the function will not block on the socket. It will get the next CDP packet (if there is one) and copy its data

segment to buf. Otherwise, it will return immediately. On success, it will return the number of bytes being received successfully. On failure, a negative value is returned.

Chapter 5

PERFORMANCE EVALUATION

We have designed two experiments to evaluate our CDP implementation. The first experiment is to investigate the performance scalability of CDP; the second experiment is to assess the CDP throughput performance on the real hardware and compare it with that of the TCP/IP implementation in the Linux kernel.

5.1 Performance Scalability

One of the unique feature of CDP is its threaded implementation. The CDP program can spawn a specified number of CDP receiving threads (dynamically or statically) to handle the same number of CDP connections simultaneously. For this reason, the CDP program obtains good performance scalability. The experimental results support our arguments.

5.1.1 Simulation

The experiment is performed on the C64 FAST simulator. FAST is an execution-driven, binary-compatible simulator for the C64 multithreaded architecture. It can accurately model not only the functional behavior of each hardware component in a single C64 chip, but also the functions of multiple C64 chips that are connected in 3D-mesh. Although FAST is not cycle-accurate, it still estimates the hardware performance by modeling the instruction latencies and resource contentions at all levels of the C64 system.

In this paper, our performance evaluation is restricted on a single C64 chip. This allows us only focus on a variety of design alternatives that may affect the performance of CDP protocol. The purpose of the simulation is not to analyze the microscopic runtime behaviors of CDP protocol. The goal is to evaluate the performance scalability of the threaded implementation of CDP on the C64 multithreaded architecture. According to this, we measured the number of cycles that a CDP program need to take to process a specified number of CDP packets and the speedup that obtained when using different number of CDP receiving threads.

The test case used in the experiment is a microbenchmark. At the beginning of the program, 128 connections are created. The number of connections is not fixed and fluctuates around 128 at runtime. This is done to model the connection creation/termination events in the real world. Later, the specified number (1-128) of CDP receiving threads are spawned. As we mentioned in the last section, these threads are programmed as TNT thread in the C64 virtual machine. So, the execution of receiving threads is not preemptable and may not be interrupted. Once started running, the receiving threads poll on the *incoming packet ports* for new packets. In addition to the receiving threads, an extra TNT thread is created to dynamically generate random CDP packets and feed them into the *incoming packet ports*.

To highlight the quality of CDP protocol, we assume infinite bandwidth of the underlying network links. Therefore, every time when a CDP receiving thread reads a port, there is always a packet ready to be copied. There is no latency in between.

5.1.2 Experimental Results

Figure 5.1 shows the performance scalability of the threaded implementation of CDP protocol. We fed 256,000 packets into the program and run different number (1-128) of receiving threads to handle them. All packets have the same amount of

payload: 1472 bytes. The figure shows the time (cycle number) that the program used to process all the packets and the speedup obtained when using different number of receiving threads. We have presented in Figure 5.1 the experimental results

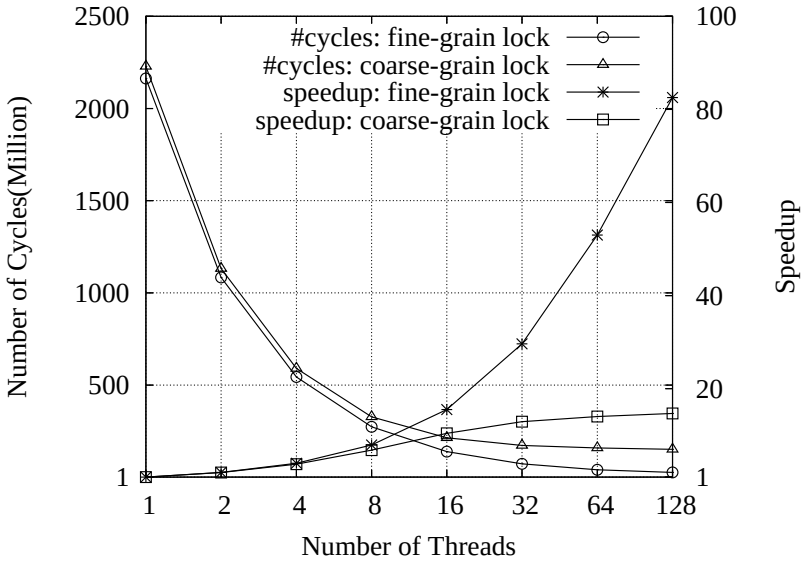


Figure 5.1: CDP Performance Scalability: Fine-Grain Lock vs. Coarse-Grain Lock

of two different design alternatives: the fine-grain lock and the coarse-grain lock, which are described in detail in Chapter4. The figure tells that the performance of both versions scale up well when the number of CDP receiving threads increase from 1 to 128. However, the scalability of the fine-grain lock version is better than the coarse-grain lock version.

For the test program using fine-grain lock, the number of cycles taken to process 256,000 packets decreases from 2162.8M to 26.2M when the number of receiving threads increases from 1 to 128. Thus, the speedup is 82.55. With the test program using coarse-grain lock, the number of cycles it takes to process the same number of packets decreases from 2228.3M to 151.8M when the number of receiving threads increases from 1 to 128. The speedup is only 14.46, much less than the program using fine-grain lock. This indicates that there are higher resources contention in

the program using coarse-grain lock than using fine-grain lock, as we have argued in Chapter4.

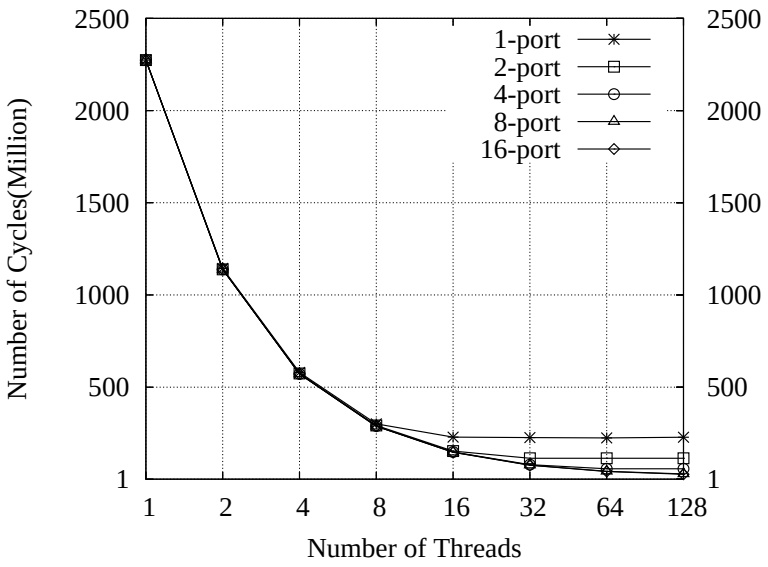


Figure 5.2: CDP Performance Scalability: Using Different Number of Ports (Time)

Figure 5.2 and Figure 5.3 show how CDP performance scalability is affected by the number of *incoming packet ports* used in CDP implementation. This is because multiple receiving threads may poll on a single port for incoming packets. Their accesses to this port are serialized through a lock associated with it. If the incoming packets flow into the C64 system in a speed that is faster than the receiving threads can tolerate, according to Little's law [8], the internal buffer of the underlying network devices will be exhausted soon. This may cause more packets to be dropped and thus hurt the the performance of CDP. The solution to this problem is to increase the number of *incoming packet ports* that can be used by the underlying network device drivers and the CDP protocol. The network device driver may put a new packet into a different and vacant port instead of waiting for an occupied one. Meanwhile, the receiving threads can also fetch packets from multiple ports in parallel. Therefore, in the same time unit, more CDP packets can be processed.

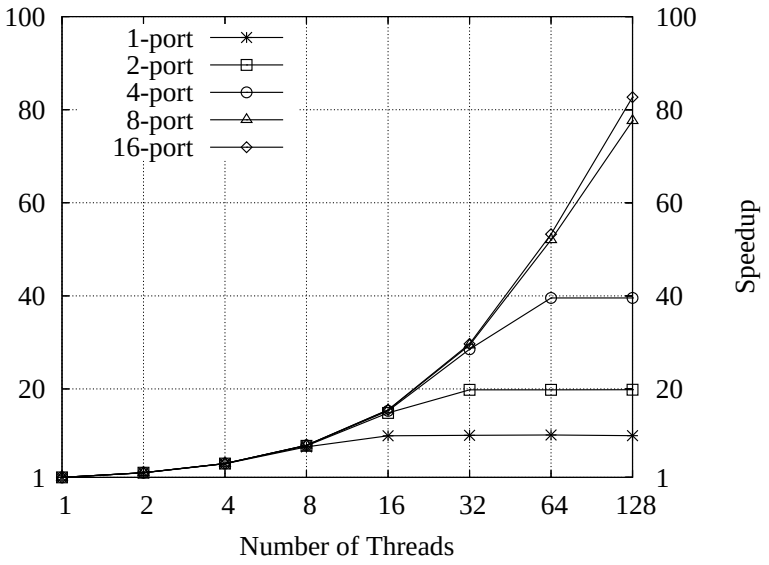


Figure 5.3: CDP Performance Scalability: Using Different Number of Ports (Speedup)

We measured the running time (Figure 5.2) and speedup (Figure 5.3) of the CDP test programs that processes 256,000 packets with 1 to 128 receiving threads under different configurations of *incoming packet ports*. The results show that, when the number of ports is less than 8, the speedup will stop scaling before the number of receiving threads reaches 128. If the number of ports equals to 8, the speedup scales very well in the range of 1 to 128 receiving threads. To continue increasing the port number will not help improving the speedup. See the curves denoted as 16-port and 8-port in Figure 5.3.

We have not done the same experiment for test cases with receiving threads more than 128. The reason is that there are only 160 thread units on a single C64 chip, and some of them need to be reserved for other user/system tasks. So, practically, we believe 128 is a reasonable upper bound for the number of receiving threads that we use in a CDP program.

5.2 Throughput

In order to evaluate the efficiency of CDP, we have designed an experiment which measures the throughput performance of a single-thread CDP version on real machine. The throughput metric is important because CDP is supposed to be used in an environment where bulk data transfer is the majority network traffic. We have performed the same experiment for TCP and UDP. We compared the experimental results of the three protocols and found that the performance of CDP is comparable with TCP and UDP, even CDP runs as a user-mode program but TCP and UDP run in the Linux kernel.

5.2.1 Experimental Platform

Because the C64 hardware is still under development, we do not have the real silicon C64 machine to run the CDP program. Furthermore, the C64 simulator does not support full-system simulation [9] [10] [11]. It only simulate the architectural behavior of the C64 chip. Therefore, it is not possible to generate the exact CDP performance number on the C64 simulator.

In this condition, we adapted the TNT thread implementation of CDP to POSIX thread and ran the CDP communication protocol as a user-level program on Linux. We use the *packet socket* [7] interface (supported by all kinds of Linux platforms) in the CDP library to access the Ethernet device directly. Through this interface, we can encapsulate our proprietary CDP packet in the Ethernet frame and broadcast it to the Ethernet. Although the performance number obtained in this way is not 100% accurate, it still gives us enough insight into the CDP performance character.

The experiment was conducted on two compute nodes of a Penguin Performance Cluster, which is made by the Penguin Computing Inc.. Both of the nodes have the same hardware configurations. To be more specific, an AMD Opteron 200 processor, dual Broadcom BCM5721 10/100/1000 Gigabit Ethernet cards, 4GB of

ECC DDR SDRAM, with Linux kernel 2.4 installed. We developed the microbenchmarks to measure the throughput performance of CDP, TCP, and UDP. The CDP version of the test case is linked to the CDP communication protocol library. The other two versions are linked to *libc* to use the corresponding protocols. The microbenchmarks are client/server programs. The client tries its best to send fixed size datagrams to the server side (through different protocols) and see how many bytes can be transferred during a fixed amount of time.

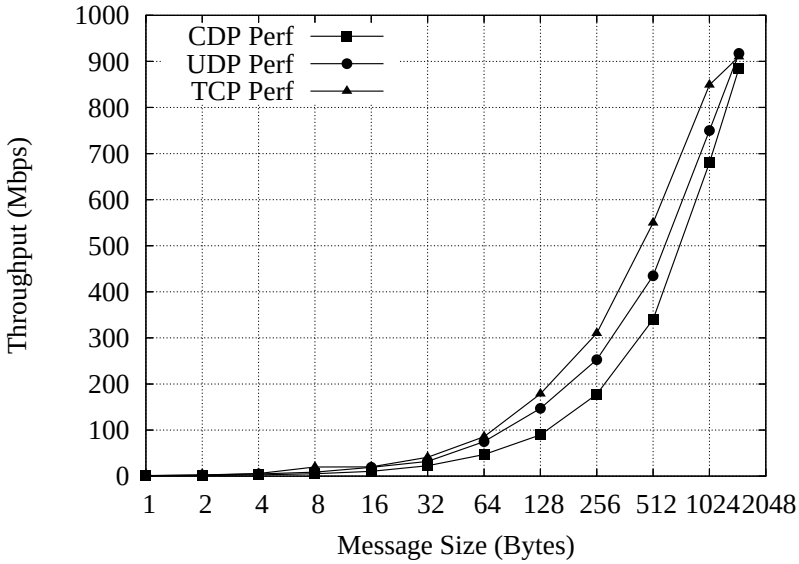


Figure 5.4: Throughput of CDP, UDP, and TCP under different message sizes: 1-1472 bytes

5.2.2 Experimental Results

Figure 5.4 is the result of the experiment. The curves show the throughput of each protocol at different message sizes. The peak performance of CDP throughput is 884Mbps, which implements 88.4% channel capacity of the underlying Gigabit Ethernet. For the other two protocols, the maximum throughput of UDP is 920Mbps, and 927Mbps for TCP. All of these three protocols reach their peak performance number at message size 1472 bytes. We stopped at 1472 bytes

because, under the limitation of Ethernet MTU (1514 bytes), this is the biggest user datagram that CDP can send. As we can tell from these numbers, the peak throughput of CDP is a little bit smaller than TCP and UDP. This doesn't mean, however, that the design and implementation of CDP are poor. There are three reasons that can explain this result: **(1)** CDP has more interprocess context switches. The CDP test case needs at least 3 pthreads at runtime, while the test cases using TCP or UDP use only one Linux native process. Since the POSIX thread is implemented as a native process on Linux platform, there are more processes competing for processors in the CDP test case than in the TCP or UDP test case. **(3)** CDP has more memory-memory copies. In the CDP test case, user data need first be copied into the internal buffer of the CDP library, then be copied into Linux kernel space for further transfer. In the test cases using TCP or UDP, user data is directly copied into the kernel space. Compared with TCP or UDP, CDP test case needs two more memory copies in a send/receive session. **(3)** CDP has more kernel-mode/user-mode switches. In the CDP test cases, CDP library is running at user-level, while TCP and UDP are running at kernel-level. There are more kernel-mode/user-mode switches in the CDP test case than in the TCP or UDP test cases.

All of these are adverse factors that cause performance degradation in the CDP test case. However, these negative factors do not exist in the real C64 hardware platform. On the real C64 hardware, CDP protocol runs at kernel-level (as TNT threads) in the C64 thread virtual machine. There is no kernel-mode/user-mode switches, and no extra memory to memory copies either. Moreover, the TNT threads run on separate C64 hardware thread units and the execution of TNT threads are NOT preemptable. So, there is no competition for processors and no interprocess context switches. With these advantages from the real C64 platform, the performance of CDP will increase and may outperform TCP/IP. (currently, the peak performance of CDP is within the range of 95.4% of the TCP peak performance and

96.1% of the UDP peak performance).

In order to make an accurate comparison between CDP and TCP/IP & UDP/IP, we need to offset the negative effects caused by extra kernel-mode/user-mode switches and inter-process context switches in the CDP test case. We can achieve this in two ways: either add some "counterbalance code" in the TCP/UDP test cases, or directly implement CDP in Linux kernel. However, both methods are not quantitatively accurate. So, we do not have the motivation to make such a kind of comparison. After all, it is not our intention to design a new protocol to beat TCP/IP and replace it. Our goal is to develop a simple and compact communication protocol for a special multithreaded hardware and explore a multithreaded methodology that can effectively exploit the massive thread-level parallelism on the hardware and achieve good performance scalability.

Chapter 6

RELATED WORK

There is some literature about the implementation of TCP/IP in a variety of computer systems. [12] introduces the experience in running TCP/IP protocol stack on wireless sensor network. [13] is about the work on implementing TCP/IP on small embedded devices. These devices are usually 8-bit or 16-bit microcontrollors. [4] contains a thorough explanation of how TCP/IP protocols are implemented in the 4.4BSD operating system. [14] is a similar book that gives a comprehensive introduction to the TCP/IP implementation in the Linux kernel. However, all of these works are focused on implementing the functions and features of TCP/IP protocol that are documented in RFC. They seldom discuss the implementation methodology. This paper explores the multithreaded method to implement a network communication protocol.

Here we make a brief comparison between the CDP implementation in the C64 thread virtual machine (or C64 TVM) and the TCP/IP implementation in Linux kernel. In the Linux kernel, interrupt is used to notify the arrival of a new packet [15], while in the C64 TVM, the CDP receiving threads poll on the *incoming CDP packet port* for new packets. Thus the CDP receiving thread responds instantly to the incoming packets. In the Linux kernel, the processing of an IP packet is split into two halves: the top-half is the urgent and fast interrupt handler [15] and the bottom-half is the slow and deferrable protocol handler [14]. But in the C64 TVM, the CDP receiving thread process a new packet without any stop until it is accepted

or dropped. In the Linux kernel, the protocol handler is treated as a *softirq* [16] and is executed in the *ksoftirqd* kernel thread. The new packets need to wait until the *ksoftirqd* thread is scheduled to a CPU. Therefore, there is a longer latency between the arrival of a packet and its processing in the Linux kernel than in C64 TVM. Actually, the protocol handler is just one of the many tasks that need to be executed by the *ksoftirqd* kernel thread. In the C64 TVM, the CDP receiving thread is bound to a physical thread unit (no thread scheduling overhead) and is dedicated to processing incoming CDP packets. In addition, the number of CDP receiving threads is system parameter that can be changed. The C64 TVM can increase the number of receiving threads if the network traffic is heavy. This is not possible for the Linux kernel, in which the number of *ksoftirqd* kernel threads is a constant.

Chapter 7

CONCLUSION AND FUTURE WORK

In the previous sections, we have reported our design, implementation and evaluation of CDP, a simple network communication protocol for the C64 multithreaded architecture, which provides however all the necessary functionalities for real applications. We have also discussed our multithreaded methodology used to implement the CDP protocol. According to the analysis on the experimental results, we have these conclusions:

- Given a multithreaded architecture like C64, which has integrated a huge number of hardware thread units, we can develop a lightweight communication protocol for it such that the implementation of the protocol effectively leverages the massive thread-level parallelism provided by the hardware and thus obtains very good performance scalability.

- The communication protocol we developed for the C64 multithreaded architecture is efficient. The performance of the single-thread version of CDP implemented by using pthread can achieve about 90% of the channel capacity on Gigabit Ethernet, even it is running at the user-level on a Linux machine.

Our future work will focus on trying to improve the memory access efficiency of CDP by utilizing the non-uniform memory space feature of the C64 architecture. In addition, we also try to investigate how the resource contention for the crossbar switch influence the performance scalability of CDP. We will investigate methods for

optimizing the data layout of the critical data objects in the CDP implementation, and therefore decrease the contention for crossbar switch.

BIBLIOGRAPHY

[1] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Towards a Software Infrastructure for Cyclops-64 Cellular Architecture," in *HPCS 2006*, Labroda, Canada, June 2005.

[2] G. Almasi, C. Cascaval, J. Castanos, M. Denneau, D. Lieber, J. Moreira, and H. Jr, "Dissecting Cyclops: A detailed analysis of a multithreaded architecture," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 1. ACM SIGARCH, March 2003, pp. 26–38.

[3] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Tiny threads: A thread virtual machine for the cyclops64 cellular architecture," in *IPDPS'05 - Workshop 14*, Washington, DC, USA, 2005.

[4] R. Stevens, *TCP/IP Illustrated, Volume 1&2*. Addison Wesley Press, 1994.

[5] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Symposium on Principles of Distributed Computing*, 1995, pp. 214–222. [Online]. Available: citeseer.ist.psu.edu/valois95lockfree.html

[6] M. Greenwald, "Non-blocking synchronization and system design, phd thesis, stanford university technical report stan-cs-tr-99-1624, palo alto, ca, 8 1999." 1999. [Online]. Available: citeseer.ist.psu.edu/greenwald99nonblocking.html

[7] R. Stevens, B. Fenner, and A. Rudoff, *Unix Network Programming: The Sockets Networking API, Volume 1*. Addison Wesley Press, 2004.

[8] D. P. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall; 2nd edition, 1991.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78–103, 1997. [Online]. Available: citeseer.ist.psu.edu/rosenblum97using.html

[11] P. S. Magnusson, "A design for efficient simulation of a multiprocessor," in *MASCOTS '93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*. Society for Computer Simulation, 1993, pp. 69–78.

[12] A. Dunkels, "Full TCP/IP for 8 Bit Architectures," in *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003. [Online]. Available: http://www.sics.se/ adam/mobisys2003.pdf

[13] A. Dunkels, T. Voigt, and J. Alonso, "Making TCP/IP Viable for Wireless Sensor Networks," in *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, Jan. 2004. [Online]. Available: http://www.sics.se/ adam/ewsn2004.pdf

[14] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler, *Linux Network Architecture*. Prentice Hall, 2004.

[15] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers (3 edition)*. O'Reilly Media, 2005.

[16] D. P. Bovet and M. Cesati, *Understanding Linux Kernel (3 edition)*. O'Reilly Media, 2005.

Appendix

IMPORTANT DATA STRUCTURE USED IN CDP

A.1 The definition of CDP Socket and Socket Buffer

```
struct cdp_socket {
    struct cdp_socket *next;
    struct cdp_socket *prev;
    int             state;      /* socket state */
    unsigned int    ref;        /* reference count */
    __MUTEX_T__     ref_mtx;    /* only protect .ref */
    __MUTEX_T__     glb_mtx;    /* protect all other fields */
    unsigned int local_node;    /* local node number */
    unsigned int local_port;    /* local thread number */
    unsigned int remote_node;   /* remote node number */
    unsigned int remote_port;   /* remote thread number */
    unsigned int wnd_size;      /* slide window size for sending */
    unsigned int send_nxt;      /* the seq number for the next data
                                   packet sent out */
    unsigned int send_ack;      /* Data packet with seq number
                                   _smaller_ than this number (not
                                   include)has already been recv'ed.
                                   Data packet with seq number
                                   equal or bigger than this number
                                   has not yet been acknoledged */
    unsigned int buf_size;      /* size of recv'ing buffer */
    unsigned int recv_nxt;      /* the next packet that will recv'ed
                                   by the user through cdp_recv() */
    unsigned int recv_exp;      /* the packet that we expect to
                                   receive but NOT yet receive. All
                                   packets _smaller_ than this
                                   number has already been recv'ed. */
    unsigned int    flags;      /* flags */
```

```
    unsigned int    ping;
    unsigned int    age;
    struct cdp_skbuf_head   recv_queue;
    struct cdp_skbuf_head   send_queue;
#ifdef  __CYCLOPS_HOST__
    pthread_cond_t          recv_cond;
    pthread_cond_t          send_cond;
#endif
    struct cdp_socket_head  established;
    unsigned int            backlog;
};

struct cdp_skbuf {
    struct cdp_skbuf    *next;
    struct cdp_skbuf    *prev;
    unsigned char       *head;  /* first byte of the current layer */
    struct ethhdr       *eh;    /* eth header */
    struct cdphdr       *dh;    /* cdp header */
    unsigned char       *data;  /* user data */
    struct cdp_socket   *sock;
    unsigned long        size;   /* size of the skb object */
    unsigned long        seq;
    unsigned long        expire;
    unsigned long        times;
    unsigned long        len;    /* current packet size, this value
                                    is changing with the growing and
                                    shrinking of the packet */
    unsigned char       pkt[0];
};
```

A.2 Inter Connection Families in TOP 500

Interconnect	Count	Share%	ΣRmax(GF)	ΣRpeak(GF)	#P
Myrinet	87	17.40%	369286	560601	100546
Quadrics	14	2.80%	131249	173218	39108
Gigabit Ethernet	256	51.20%	795582	1511012	249726
Infiniband	36	7.20%	221074	316047	53068
Crossbar	12	2.40%	114397	149382	16674
Mixed	5	1.00%	71924	91853	15396
NUMAlink	9	1.80%	55509	61028	9728
SP Switch	42	8.40%	281098	406406	66036
Proprietary	26	5.20%	629527	806058	291296
Fireplane	1	0.20%	2054	3053	672
Cray Interconnect	9	1.80%	109965	133803	28988
RapidArray	3	0.60%	8388	10374	2357
Totals	500	100%	2790054.02	4222834.82	873595