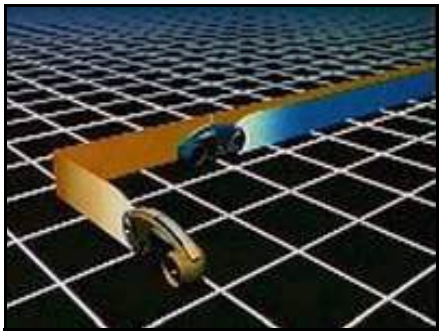
High Performance Computing using Portals over TNet



Diploma Thesis

Adrian Riedo
Summer 2000

Swiss Federal Institute of Technology Zurich, Switzerland
The University of New Mexico, Albuquerque USA

To my parents: with gratitude for your love and support.

Foreword

This report is the result of the diploma thesis on High Performance Computing at the Scalable Systems Lab at the University of New Mexico in summer 2000.

The design and implementation of a network protocol represents a challenging task. Knowledge on the hardware, the operating system, kernel programming and protocol design are required to achieve the goals.

With this work, the basic for further research is given. The preliminary implementation on the development system will serve for future experimentation and programming.

At this point I would like to thank Prof. Anton Gunzinger and Prof. Gerhard Troester of the Swiss Federal Institute of Technology Zurich and Prof. Arthur B. Maccabe of the University of New Mexico for their support. Many thanks also go to my technical advisors and helpers, namely Martin Heimlicher and Martin Frey of Supercomputing Systems Zurich, Rolf Riesen and Jim Otto of Sandia National Laboratories Albuquerque and Riley Wilson of the University of New Mexico.

Albuquerque, September 2000

Adrian Riedo

Statement of work

Statement of Work for Adrian Riedo

The goal of this project is to evaluate the possibility of developing a high-performance implementation of the Portals 3.0 API on TNet.

Background

The Portals 3.0 API was developed as a joint project between Sandia National Laboratories and the Scalable Systems Lab at the University of New Mexico. Like many other high-performance message passing APIs (e.g. Scheduled Transfer and Virtual Interface Architecture), the Portals API supports OS-Bypass. OS-Bypass is motivated by the high cost, in terms of time, associated with servicing interrupts during high speed communication. In OS-bypass, the relevant policies of the OS are implemented in a control program which is run on the Network Interface Card (NIC), thus eliminating the need to generate many of the interrupts associated with high speed communication. In addition to OS-Bypass, the Portals API also supports "application-bypass." Application-bypass is motivated by the need to minimize memory copies during communication. In application-bypass, the policies of the application regarding message placement are implemented on the NIC. Because the NIC is able to deliver messages to the correct location based on the contents of the message, the application is able to avoid a costly memory copy operation.

The company Supercomputing Systems in Zurich designed a custom network called TNet for the parallel computing project "Swiss-Tx" at the Swiss Federal Institute of Technology in Lausanne (EPFL). The message passing library MPI is installed and executed through the hardware interpreted Fast Communication Interface (FCI) that enables a direct store from one processor into the memory of another processor. Because the network interface card carries a large FPGA and 16 (or more) MB of memory, a flexible and fast implementation of any communication protocol can be done. By putting time-critical parts of the protocol into the hardware it is possible to optimize latency and throughput of high-performance networks.

Project Scope

The goal of this project is to design and develop an initial implementation of the Portals 3.0 API for TNet. We will start from the reference implementation of the Portals 3.0 API. The Portals 3.0 reference implementation uses a Network Abstraction Layer (NAL) to achieve independence of protection domains. That is, all of the calls to functions in the NAL are implemented as call-backs which may or may not cross protection domain boundaries. The three protection domains of interest are the application, the OS (kernel), and the domain defined by the control program on the NIC.

The primary goal of this project is to design an implementation of the Portals API that places as much of the functionality on the NIC as is feasible. This design would define the goal of a full implementation. A secondary goal is to develop a preliminary implementation of this design. In the preliminary implementation, much of the Portals functionality will remain in the application and OS domains and the NIC will have minimal functionality.

Prof. Barney Maccabe

The University of New Mexico
Albuquerque, USA

Prof. Gerhard Troester

Swiss Federal Institute of Technology
Zurich, Switzerland

Table of contents

CHAPTER 1 *Introduction*

Time is money! Even though we usually associate these words to business, it is also an important concept in computer science. Computers were built to solve problems faster than man could. During the evolution of the Computer, a main goal was (and will always be) to make computation faster and cheaper. The time factor becomes more and more important, when it comes to long calculations (e.g. scientific problems).

A couple years ago supercomputers were usually built from the bottom up in terms of hardware and software to achieve the lofty goals of High Performance Computing. With the fast growing technology of personal computers and the fact that those systems have become bulk ware, the idea to build supercomputers out of standard components is becoming common.

A major part of commodity based systems is the network and communications in general. In contrast to “one-box-supercomputers”, the liberty to interconnect the computational nodes (processors, memory) is restricted. The only feasible way to communicate is through the interfaces provided by the off-the-shelf computer (system bus). Depending on the final field of applications the network has to be chosen or designed wisely (NIC, switches, topology and of course the protocol).

Another very important factor is the portability of the applications among other (super)computers. It would be a waste of time if applications would have to be rewritten every time the system changes. The most common used standard that supports parallel applications and libraries nowadays is MPI (the Message Passing Interface) which defines the syntax and semantics of a core of library routines for programs in Fortran or C.

With a large number of computational nodes connected by a fast network on one side and a portable library for applications on the other, it is important that the protocol between these two worlds has to be designed the way to provide the maximum of the hardware resources to the applications.

It is exactly this part what the current report is all about. The related project is the evaluation and first implementation of the Portals API on TNet. Portals is the data movement layer that is used on one of world's fastest supercomputers at Sandia National Laboratories while TNet is a complete network developed by Supercomputing Systems Zurich, Switzerland.

After an analysis on both environments, different designs and its complexities are discussed. One concept is implemented finally on a development system.

CHAPTER 2

Analysis

This chapter will give a short overview of the Portals 3 application programming interface (API) and the TNet adapter specifications (with some background on the related software). A precise analysis provides the knowledge necessary to design a proper implementation.

Because the purpose of this project is to evaluate and implement a previously designed API on a different set of hardware, one could think that only Portals must be studied in depth. However, because of the different features of the hardware, it is also important to analyze the reference implementation of TNet.

After some basics on HPC an overview of both systems will be given that will help to design Portals over TNet.

2.1 Basics

The idea is simple: In a cluster-based supercomputer (supercluster), data should be transmitted from one computational node to another as fast as possible and still maintain scalability and portability. Fast means with the shortest latency and highest bandwidth possible. Adding more nodes to a cluster should increase the total computational power linearly - such a system is called scalable. To achieve these goals, several techniques are used.

2.1.1 Message Passing

In a parallel machine, data could be transferred in many different ways from one process to another, but not every solution is portable and, consequently, only useful for a specific hardware. A widely used paradigm for process intercommunication is **message passing**, which, as several systems around the world have shown, can be implemented efficiently. The Message Passing Interface **MPI** [2] is one of the most popular standards for distrib-

uted memory systems. This application programming interface lies just below the application and defines how data is communicated among processes in an application. Every underlying protocol (e.g. between MPI and hardware) should transfer messages efficiently.

A good way to learn the principles of MPI is to write some programs on mpich [3], which can be downloaded for free and runs on TCP/IP.

2.1.2 Data movement layer

The data movement layer is the protocol that provides the functionalities to MPI and the application to send and receive data over a specific type of network. This layer has to be designed to support the scalable properties of MPI while maintaining network independence. Furthermore, a good implementation will avoid memory copies because network bandwidth approaches memory bandwidth more and more on today's fast systems. Modern NICs even allow bypass of the OS and put arriving messages directly into the memory space of the application. **OS-Bypass** and **zero-copy** has become a state of the art requirement for modern message passing layers. Techniques that allow large message transfers, using remote puts and gets, without the need for any intervention on the part of the application or an application level thread are called **Application Bypass**. All these design rules are usually based on a connectionless communication model, meaning that no explicit connection is established before any transaction of data.

2.1.3 Network

To fulfill the requirements imposed by the data movement layer, configurable network hardware is needed. As a part of the protocol has to be done on the NIC, they have to be programmable. Network interface cards, such as Myricom's *Myrinet* use microprocessors while the *TNet* NICs of Supercomputing Systems are armed with large FPGAs, a reconfigurable hardware device. Both solutions can provide flow control on the card as well as implementation of pagetables and retransmission protocols. Of course, the rest of the network (e.g. switches) should not impose any scalability limitations.

2.2 Portals

Portals 3 is the data movement layer actually used by the CPlant cluster at Sandia National Labs. Its roots go back to the SUNMOS [12] and TeraFLOPS [12] projects and is designed to support commodity based clusters up to the order of ten thousand nodes. The Portals 3.0 API is well documented in [1]. This chapter will concentrate on the actual implementation on Myrinet and discuss the software architecture. Although Portals is well designed, it can be confusing to read the source code for newcomers as it is divided in many pieces. To understand the dataflow through the code, a step by step example will be given which will also help to design the network abstraction layer (NAL) for TNet.

2.2.1 CPlant environment

On the CPlant cluster, Portals 3 is one important component among others and is still under development. A closer look on the actual implementation (figure 2.1) shows the kernel modules (drivers) acting as mediators between application and network.

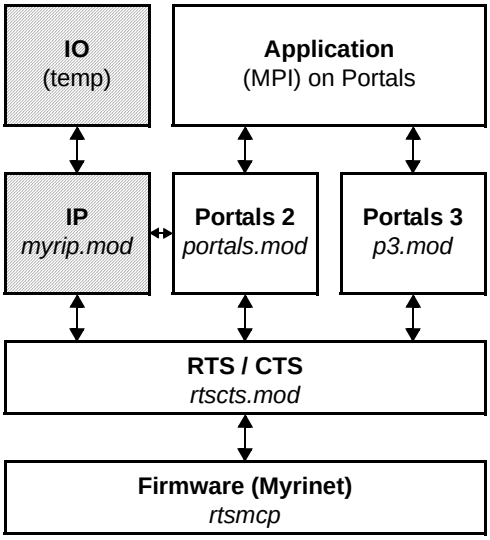


FIGURE 2.1. CPlant environment

The **application** builds the top level and includes the MPI library as well as the API side of Portals. A process can use the older Portals 2 or Portals 3 for communication. There are two reasons why Portals 2 is still included: compatibility to older applications and because some functions are not yet implemented in Portals 3.

An **IP module** is used as a temporary solution for **IO** (i.e. filesystem) over the high performance network as long as there is no version running on Portals. This is just an intermediate solution that does not scale.

The **RTS/CTS kernel module** and the corresponding **firmware** on the Myrinet NIC together form the low level send and receive functionalities. Every packet passes through the *ready to send, clear to send* unit and thus the kernel mode. The next chapter will show that Portals 3 is designed so that parts of it can be put into the NIC in the future and omit costly kernel calls.

Remark: The Computational Cluster contains more than 1000 nodes and is still growing. Each host (Alpha workstation) runs at 500MHz and contains 256 MB of main memory. Figure 2.2 shows the second phase of the CPlant evolution called “Siberia”.

FIGURE 2.2. CPlant “Siberia” at Sandia National Laboratories

2.2.2 Portal Addressing

This section will give a short introduction to the Portals 3 addressing mechanism. A Portal represents an opening in the address space of a process. A Portal *get* operation reads data from another process while a *put* performs a write operation.

The Portals addressing scheme for incoming data is an intricate hierarchy. The first identifier is a **match list**, a list of match entries each of which contains a set of *match bits*. These bits describe a specific pattern that the incoming data must match to use that match entry. Within each match entry is a list of **memory descriptors** that define a region in memory as well as the behavior associated with that region, like how many and what kind of operations can be done using it. Although the match entry contains a list of memory descriptors, only the first one is considered when matching incoming data. Each memory descriptor can contain an **event queue** that is updated when an operation is performed on the region to let the application know what has happened. The final piece of addressing information is an **offset** within the memory descriptor. Therefore, a remote memory address can be accessed through a match entry, a memory descriptor, and an offest in the region defined by the memory descriptor.

A more detailed explenation can be found in the Portals 3 API documentation [1].

2.2.3 Architecture

The implementation strategy of Portals 3 is to provide a highly platform and network independent API for message passing applications. The concept of a **network abstraction layer** (NAL) is used to make migration from one network environment to another easy. Portals 3 is divided into two parts: an application programming interface (API) and a library (LIB). Basically, the following graphic shows the software hierarchy, highlighting the relevant sourcefiles for the NAL.

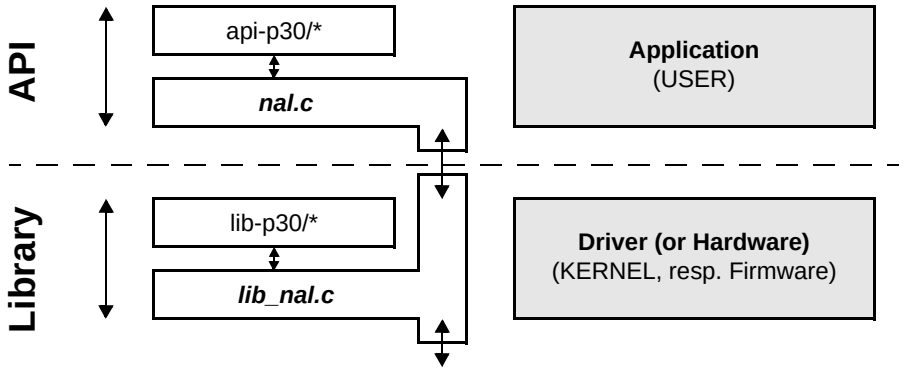


FIGURE 2.3. Portals 3 architecture

The current CPlant network is Myrinet and therefore the network abstraction layer is called MyrNAL. On the API-side the NAL is defined by **myr**nal.c and controls the communication out of the application. The main LIB-side NAL sourcefile is called lib_**myr**nal.c and accesses the library which is actually located in the kernel as a driver. These files separate hardware dependent calls from logical routines.

The function *forward* on the API-side is used to communicate to the library. For a library in form of a kernel module, *ioctl*, a widely used Unix programming function, is used to perform this operation. For the NAL on the library side, more functions are required as most of the Portals work is done here. Its main functionalities include *open*, *dispatch* and *close* of the library and communication to the next layer (*send*, *receive*).

The network abstraction layer makes migration to another network environment much easier as the programmer does not have to deal with the Portals internals in depth. This is especially true when the library still remains in kernel space in form of a driver.

A Portal ***put*** will serve as an example to have a closer look at the network abstraction layer and the dataflow in Portals 3. The following graphic shows the simplified situation of a point-to-point put from the initiator to the target.

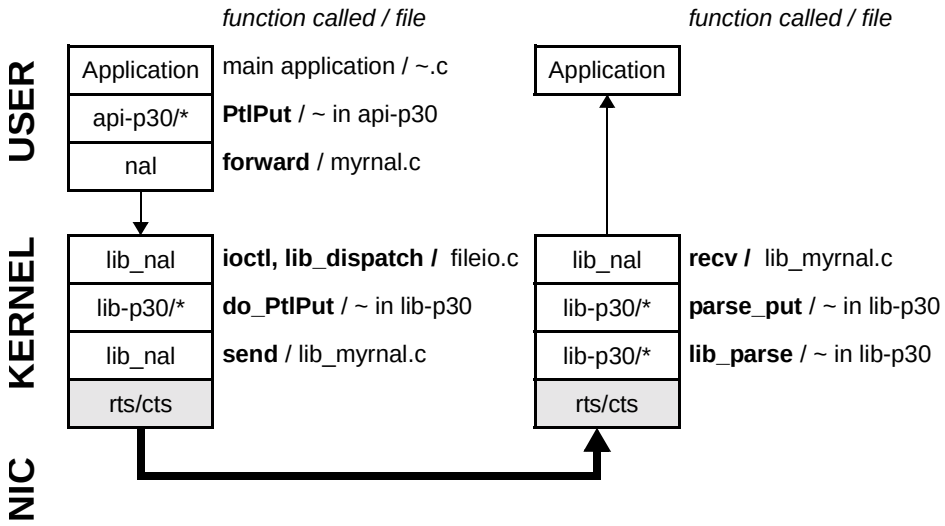


FIGURE 2.4. Simplified example of a Portal *put*

Figure 2.3 describes the simplified communication scheme that occurs during a call to *PtlPut*. After some initial processing by the API, control is passed down to the NAL. The NAL function *forward* calls *ioctl*, a Linux system call for communicating with a module. The library side of the NAL receives this call and immediately dispatches it to the Portals library, where all of the data necessary for communication with another node is assembled and handed back to the NAL again. The NAL then calls the routines that actually transmit the data over the NIC.

When the data arrives on the remote side, the incoming data handler passes the appropriate information to the Portals *lib_parse* routine and, consequently, to *parse_put*. This function finds the appropriate memory descriptor, if one has been setup, and calls the library side NAL to actually receive the data. The transaction concludes when *nal_recv* calls *lib_finalize* and the appropriate event queue is updated in user space with a *put* event.

2.2.4 Myrinet

Myrinet is a network product for High Performance Computing by Myricom Inc, Arcadia USA. The core of every 32 or 64 bit PCI Myrinet interface card is the so called LANai chip, a RISC based VLSI ASIC which controls host and packet interface as well as the onboard fast local memory. Currently, the NICs are shipped with LANai version 9 and can hold up to 8 MB of memory. The microprocessor in the LANai can be programmed for custom protocols to support OS Bypass for example. This program, that has to be down-

loaded to the NIC, is called Myrinet Control Program or simply MCP. The full-duplex links that are interconnecting the network devices can be electrical or optical. The network routing is source oriented, which is reflected in the low-priced “dumb” switches.

2.3 TNet

TNet is a complete network including hardware and software for High Performance Computing that has been developed by Supercomputing Systems (SCS) Zurich, Switzerland. The primary application of this product was the Swiss-Tx cluster located at the Swiss Federal Institute of Technology in Lausanne (EPFL) where its expected high bandwidth and low latency was proved in practice.

2.3.1 Background

The TNet hardware consists of 32bit as well as 64bit PCI network interface cards and intelligent, destination routed switches interconnected by Fiber channel technology. A NIC firmware, a TNet driver and the Fast Communication Interface (FCI) [8] together form the basic software to run message passing oriented applications. On the Swiss-Tx cluster, job management is done by Gridware’s *Codine* that is interacting with *Cosmos*, a process control service for TNet by SCS.

The concepts of TNet go back to SCS’s Remote Store Architecture technology, a technique that enables a direct store from one processor into the memory of another processor. It is the long research and experience of SCS on parallel computing that characterizes the high quality of this product.

2.3.2 Network Characteristics

This is a short overview of the TNet specification that will help the reader to understand the design and implementation later in this report. More specific information can be found in [4] and [5] while the focus in the following sections will be on the hardware.

2.3.2.1 Fundamentals

“It’s all about address mapping” could be a technical saying to describe the fundamentals of TNet. As the idea is to transfer (store) data from one node to another, the most direct way is to let the running process send the data down to the NIC (bypassing the OS) and out on to the network. The arriving packet on the remote NIC will then be copied into the appropriate memory area of the destination process.

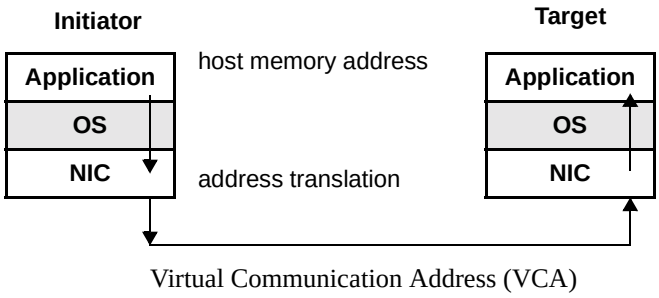


FIGURE 2.5. Remote Store Strategy

Address translation is done on multiple levels (PCI-Bus, Network, etc.) for a remote store. The important part is, that the network card can map the host memory to the network in form of Virtual Communication Addresses. The example above describes the situation after the target has sent a request to get a message to the initiator. This mechanism allows the target's NIC to handle the incoming data and place it at the right memory spot in the host.

TNet has been designed for large-scale clusters. Each NIC is addressed by a 16bit destination ID which gives a total of 65536 nodes possible. The actual network bandwidth is 1Gbit/s and will be multiplied on future TNet releases. Switches are remotely configurable and allow any network topology by preventing deadlocks. The 12 port full crossbar switches, which in contrast to Myrinet use destination routing, also include monitoring of the network performance. Variable size **micropackets** of maximal 128 bytes permit a fair access even on a highly loaded network. Reliability is improved by a hardware implemented **retransmission protocol**.

2.3.2.2 OSI layers

Comparing to the OSI reference model, TNet supports **unicast** and **multicast** on the network layer (broadcast is just a special case of multicast). Now imagine a huge cluster of computing nodes running multiple MPI applications and that one of the processes uses the MPI command to send a broadcast out to its "friends". On a unicast-only network this message would have to be sent n times while on a system with broadcast option, all of the nodes would receive the message, even those that are running different applications. So on TNet, a MPI broadcast will turn into a network multicast and reduce network traffic.

On the transport layer, TNet offers two communication types: **direct mapped** and **table mapped**. Direct mapped is a simple 2 point communication from one PCI adapter board to another in a direct addressing mode. Table mapped communication has been introduced for SMP boxes so that a message has only to be transmitted once over the network when it is addressed in an indirect mode to multiple processes on the destination node. In this case, the receiving NIC will locally copy the data to the corresponding processes.

2.3.2.3 Address Translation

The remote store strategy requires a network interface which can be programmed and updated during execution time. On TNet, two different OS bypass capable techniques were designed to translate addresses and store incoming packets into the host memory.

- A **contiguous memory block** (CMB) is preallocated at boot time and acts as a "transfer buffer" for sends and receives. This memory area is simply translated by an offset on the NIC to a virtual communication address on the network.
- A dynamic **pagetable** in the memory on the NIC allows mappings of memory regions of the host to the network.

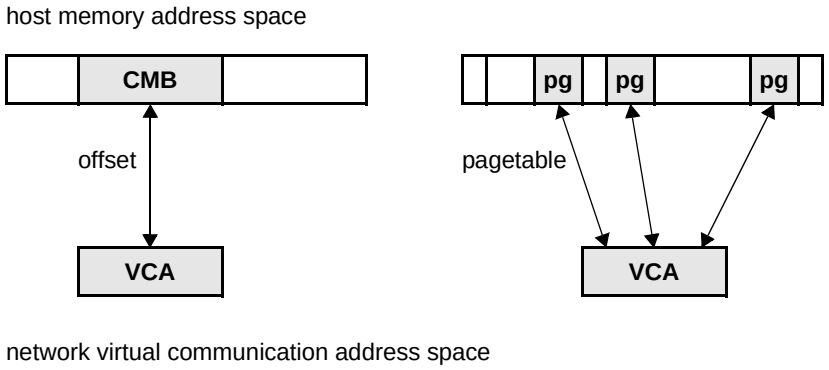


FIGURE 2.6. Address Translation on TNet

The CMB model will make programming easier as there is no modification of the NIC state to be done during execution. The drawback is obvious: incoming and outgoing data has to be copied at least once from or to the memory block.

2.3.2.4 Network Packet

A TNet network packet contains a small header with the Destination ID, remote interrupt ID, some identification flags and the Virtual Communication Address (resp. table index and offset for table mapped mode) followed by the payload (up to 56 words) and the network tail (CRC). The Virtual Communication Address and the Destination Address together identify a memory area of one specific node in the cluster. For the moment being, 32 out of the 64 bit wide VCA are used which leads to a total addressable memory of 4 GB Virtual Address Space (VAS) per network interface card and process. A pagetable with a pagesize of 4 kB would require 4 MB memory on the TNet Adapter to address 4 GB of hostmemory.

2.3.3 PCI Network Interface Card

At the time this project started, the 64bit TNet card was in production, so the available NIC was the 32bit version which will be described shortly. Again, more information is available in [4].

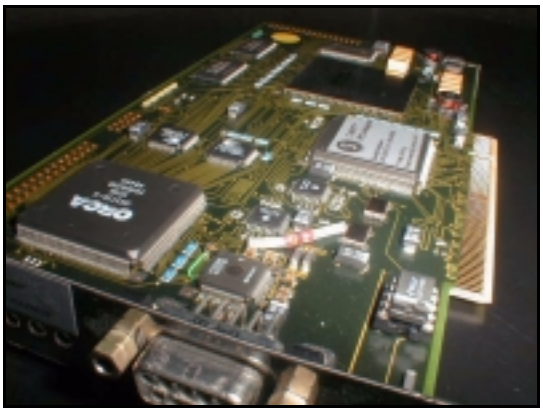


FIGURE 2.7. 32bit TNet PCI Adapter

In contrast to Myrinet, this adapter doesn't have a processor but instead two big Field Programmable Gate Arrays (FPGA) sharing the work. Even these devices are configurable there is not a program executed on the NIC. Once the FPGA's are configured they "behave" like hardware. Operations that have to be done on the network card are therefore designed as hard-wired parts. The advantage is obvious: time critical functions can be implemented very fast thanks to the high parallelism on the chip which runs approximately at the clockrate of the PCI bus.

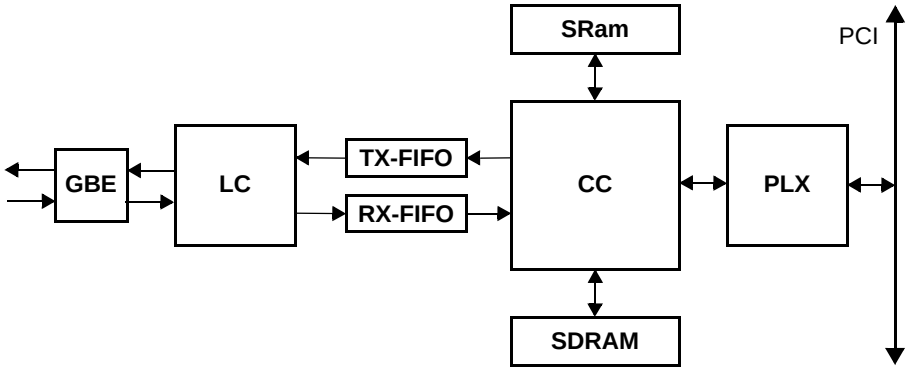


FIGURE 2.8. TNet PCI Adapter Block Diagram

The 32bit NIC consists of the following main parts:

- **CC**: The **Communication Controller** is a Lucent Orca 3T80-5 FPGA running at a clockrate of 31.25 Mhz (half the frequency of the LC).
  Its main functions are the Send (Tx) and Receive (Rx) units as well as checksum (CRC) generation and all the memory controllers (SRam, SDRam, FIFOs).
- **LC**: A Lucent Orca 3T30-6 FPGA is used as **Link Controller** and runs at 62.5 MHz. The handshaking and retransmission protocol are implemented here. Buffers for 3 packets OUT and 1 packet IN and the CRC check are also part of the LC.
- **PLX**: is the manufacturer of the **PCI9080 PCI bridge** which offers access to the back-end device, the CC.
- **SRAM**: The ID Validation Table is stored in the 128 x 18 Bit SRAM.
- **SDRAM**: TNet is shipped with **16 MB on board RAM** as a minimum and could also be assembled with bigger memory devices. The pagetable can be placed here and leaves enough space for the index to address translation table which is used for table mapped mode on SMP machines.
- **FIFO**: Send and receive Buffers
- **GBE**: The Gigabit Ethernet (GBE) controller VSC7211 from Vitesse Technologies finally builds the bridge between the network and the card.

The PCI-Adapter (registers and memory) is accessible from the host software (driver, application) through the PCI bridge controller (PLX) using a **global communication window** (gcw). When an application sets up a new communication it writes the Destination ID (or Multicast ID) to the *Sender Network Destination ID Register* (SNDIR) on the TNet adapter. Because any application can set this register, the Destination ID has to be cross-checked against the entries of a special table containing all valid destinations in the system. This ID validation is done in hardware on the TNet adapter (table in SRAM) as in software this could only be realized in kernel mode.

If a transmission error (CRC fail) occurs on a link on the network, the Link Controller (LC) on the TNet adapter automatically requests a retransmit from the sending network device. The buffers in the Link Controller are set to 3 outgoing packets and 1 incoming packet.

Besides, the TNet PCI Adapter supports **burst assembling**, which is especially useful in case of Programmed I/O (PIO), when data is transferred over the PCI-Bus to the NIC in small pieces (bursts). Throughput of the network can so be improved by assembling bursts into network packets. The size of the datablock to be transferred is first written to a register on the adapter to enable the assembling.

The TNet 32 bit PCI adapter specification was subject of the first Portals over TNet presentation on a weekly Scalable Systems Lab meeting at the University of New Mexico.

2.3.4 TNet Driver & FCI

The driver provides access to the TNet device for the operating system and is mainly used for initialisation (search device, download firmware, etc.), setting up new communication processes (*open*, *close*, *ioctl*), interrupt service routines (interrupt handler and kernel thread), monitoring, DMA handling service and memory mapping services. Fast Communication Interface (FCI) is the name of TNet's data movement layer and was developed by Supercomputing Systems and the Swiss Federal Institute of Technology in Zurich (ETHZ). Relevant parts of FCI, that are compiled into the driver, are used for communication to the application (e.g. remote interrupt calls).

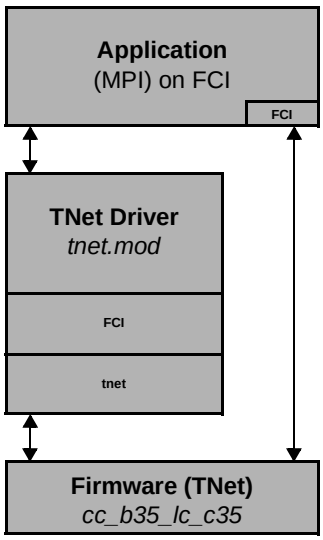


FIGURE 2.9. TNet environment

Access to the network interface card is managed by a lock on the card. Either the interrupt handler, the kernel thread or the application can communicate to the NIC after getting the lock. Write permission to the contiguous memory block is also handled by this lock to maintain data consitency.

MPI functionality can be implemented on top of FCI. In case of a **MPI unicast**, the mechanism used is similar to a Portal *get*. The receiver process sets up a transfer by requesting a message by the sender. While the request is sent, the pagetable on the network adapter is updated with the corresponding Virtual Communication Address so the expected message can be placed at arriving time directly at the right memory area of the host. A **MPI broadcast** (multicast on the TNet network) instead would consist of multiple receiver requests to one sender. Once all requests are transferred to the sender it sends out the broadcast using the specified Multicast Group ID. The TNet remote interrupt functionality is used to implement the MPI wakeup function in case of a non-blocking send.

2.4 Summary

The following points summarize the differences of the Portals over Myrinet and Fast Communication Interface over TNet implementations:

- **Network**: approx. the same throughput on the wire. NIC hardware concept completely different - microprocessor (Myrinet) vs. FPGA (TNet)
- **Complexity**: A Myrinet firmware can be programmed using a high level language as C - TNet firmware has to be designed in VHDL.
- **Performance**: The Myrinet LANai processor is actually too slow to operate a fast Portals 3 implementation - FPGA's on the TNet card can operate multiple requests in parallel and thus increase speed.
- **Address translation**: Portals uses flexible but relatively complex lists (memory descriptors) while FCI/TNet is based on a pagetable lookup on the NIC (or CMB).
- **OS Bypass**: Because Portals is in kernel, no OS Bypass is possible - FCI/TNet does OS Bypass.
- **Portability**: Portals can be easily implemented on different networks thanks to the network abstraction layer - the current FCI only runs on TNet.

By finishing this analysis, the necessary fundamentals are set to start the design of Portals over TNet.

CHAPTER 3

Design

This chapter contains the study on different implementation possibilities and the final design concept that has been chosen to build a first version of Portals 3 over TNet.

3.1 Case study

From the Portals point of view there are two evident solutions one could think of for an implementation on TNet:

- Hardware: Put the library side of Portals 3 entirely on the NIC
- Software: library still in kernel, change handling of dataflow in NAL so that TNet is used instead of Myrinet.

When looking at the problem from the TNet side there are also two major design concepts to make Portals 3 run on this network:

- FPGA redesign: a full implementation of the Portals library requires a redesign of the TNet firmware to support Portal Addressing
- CMB and/or Pagetable: make use of the current TNet firmware to communicate over the network

Both sights fully correspond on their first solution: Putting the library side of Portals 3 completely down to the NIC results in modifications on all levels (firmware to application). A software solution instead profits from a prewritten firmware & driver and can be implemented in different ways (CMB, Pagetable or even on top of FCI).

The impact in terms of complexity, performance and implementation time is discussed in the following sections.

3.1.1 Hardware solution

A hardware solution of the Portals 3 library represents one of the final goals of the Portals implementation strategy. In the case of TNet this would mean that the whole C code serves as a model to redesign the library in vhdl. While C is a high level programming language, vhdl is a description language for hardware and is usually used on a much lower level for performance reasons. Even vhdl supports some higher abstraction techniques it is still much different from common programming environments as C.

A “hardware library” has some big impacts on the concepts. First, the application will **send** its messages directly to the NIC. Besides the Portals API, functions to talk to the hardware must be compiled into the application so that *forward* can “jump” into the network card. Second, and this is the hard part, the matching in the Portals table and the search for the memory descriptor upon **receive** of a message is done entirely on the NIC. Finally there is still a **driver** (kernel module) needed to initialize the card, download the firmware and for some calls as remote interrupts.

3.1.1.1 Application

Besides the driver, the application sends and receives data directly to the network card. To avoid any conflict by simultaneous access from the driver, a control mechanism is needed. This can be done as actually on TNet where the NIC manages a *lock*. Either the driver or the application owns the *lock* and can send data. Another way is to deceive the driver and the application and let them think as that there are two independent network devices. The NIC then handles both requests an “merges” them.

3.1.1.2 Driver

Obviously, a driver is still needed even if the Portals library is implemented in hardware. Resource management, download of the firmware and initialization of the TNet card is handled by the driver as well as remote interrupt calls.

3.1.1.3 Firmware

While TNet address matching is based on a simple pagetable lookup, Portals 3 performs a search in the match entry list and then picks the first matching memory descriptor. The complexity of the firmware thus increases and requires more space in the FPGA. Searching in these lists result also in multiple memory accesses that can not be done in parallel.

The number of operations for a Portals address translation depends on the number of entries in the lists. The longer those lists get, the more time and memory accesses it takes to find a memory descriptor. Compared to the TNet pagetable technique, a faster and larger FPGA as well as faster memory (i.e. SRAM) would be needed to maintain the same lookup time as in TNet.

3.1.2 Software solution

A pure software solution takes advantage of the presence of the TNet firmware and driver. The Portals library is still compiled into a kernel module. A network abstraction layer for TNet (TNAL) replaces the one for Myrinet. One could also think of a Portals over FCI over TNet design to make programming easy but it is obvious that this “quick and dirty solution” doesn’t follow the proper implementation strategy of Portals.

3.1.2.1 Application

The API-side NAL function *forward* has to point to the TNet interface instead of the Myrinet interface.

3.1.2.2 Driver

As the firmware is kept, two techniques for the communication are possible: the contiguous memory block and the pagetable. For a first version, the CMB solution is a good choice. Later, an implementation using the TNet pagetable can be realized.

A possible solution would let the LIB-side NAL handle the packets to the contiguous memory block instead to the RTS/CTS module. Incoming messages carry a remote interrupt flag to wake up the driver and let him pass the information to the library and (upon match) copy to the application.

3.1.3 Conclusion

A hardware solution is a good project, but would definitely require much more time than is available for this diploma thesis because:

- Analysis of both environments and setup of the development system already take much time.
- Software modifications (driver / API-side NAL) alone become more complex than a pure software solution.
- FPGA is most likely too small to hold the complete library.
- SDRAM on NIC would slow down the benefits from the parallelism on the FPGA.
- VHDL design is time intensive. Usually one test per day can be run because of the *simulation* and the *place and route* steps.
- Debugging is harder.

A successful implementation of Portals over TNet for the time given is only feasible in the case of a software solution. Once a running Portals over TNet environment exists, it is then easier to design the hardware solution.

3.2 Design Concepts

As seen in the case study a full hardware implementation requires high knowledge of TNet and Portals which results in a long project time. Besides, a development system has to be set up first in order to start any experimentation. A former software solution would make a hardware implementation much more feasible as one could start from a working Portals over TNet environment instead implementing from scratch.

The idea of the whole project is to approach the final goal step by step while learning the concepts of both systems. For this reason, a software solution in the form of a Portals network abstraction layer for TNet is a good basis for further research. The design concept for the implementation of a TNAL is discussed in the following sections.

3.2.1 Portals & TNet modules

The following figure represents the situation of the Portals and TNet drivers, p3.mod and tnet.mod respectively. In TNet, the driver is used for initialization and interrupts (applications can talk directly to the NIC); the Portals 3 module is used for any call (by *forward* from the application and RTS/CTS from the network).

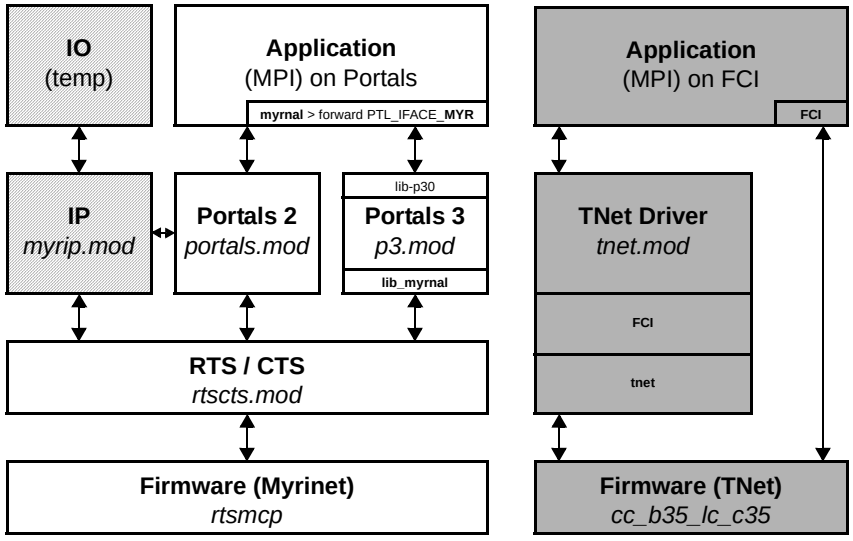


FIGURE 3.1. Comparing the CPlant and TNet environment

In both cases, communication between application and driver is done using the *ioctl* function that can perform a variety of control functions on devices. A filedescriptor that refers to the corresponding device (*/dev/portals*3 respectively */dev/tnet0*) is passed as first argument. The low level part of the TNet driver is built together with the relevant FCI functions into one module. On Cplant, a separate kernel module (RTS/CTS) for

communication to the device can also be addressed by the older Portals 2 and the IP module.

3.2.2 Hybrid module

The idea of the following design is to build a hybrid module that supports FCI and Portals 3 calls at the same time. This requires modification in the Portals 3 and the TNet code. A new network abstraction layer for TNet called TNAL will distinguish different forward "jumps" using the *PTL_IFACE* definition.

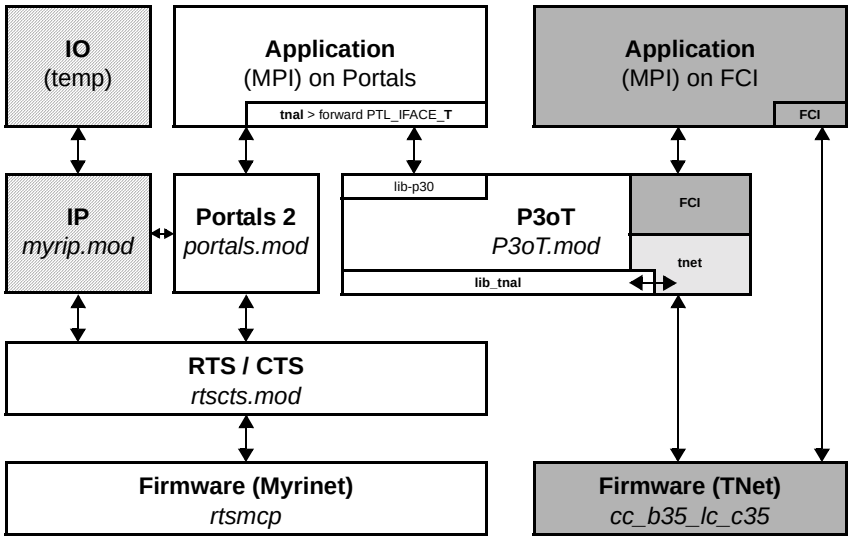


FIGURE 3.2. The P3oT hybrid module

The result is called "Portals 3 over TNet module" (P3oT.mod) which accepts Portals 3 calls using *PTL_IFACE_T* and communicates over TNet using the current firmware.

This concept has been presented in this form on a weekly meeting of the Scalable Systems Lab at the University of New Mexico. The implementation is based on this concept.

3.2.3 Communication

In the designed P3oT module, communication is handled in kernel mode. As a lot of time is already spent to jump into the kernel, the communication technique (CMB or Pagetable) becomes less significant in terms of latency. For this design, a contiguous memory block has been chosen to make a first implementation easier. Every incoming and outgoing transfer passes through the preallocated memory area.

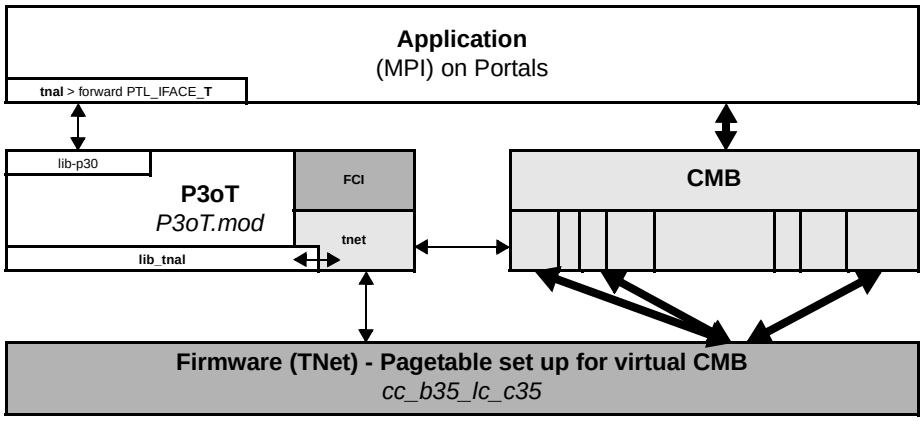


FIGURE 3.3. CMB communication model

The pagetable on the NIC is used to build a virtually contiguous memory block in the main memory of the host. Dataflow is represented by thick arrows on figure 3.3. The driver copies the messages from userspace to the CMB and vice versa. Transfers between NIC and host memory are done using DMA.

CHAPTER 4

Implementation

The content of this chapter represents the practical third of the Portals over TNet Project (besides analysis and design). After the setup of the development system, the two reference environments (Portals/Myrinet and FCI/TNet) are installed and serve as a basis for the project. The final product of this project, the P3oT module, is discussed in detail in the following sections.

4.1 Development System

Part of this project was to setup the environment for the development from bottom up. This includes the procurement of the hardware (computers, network cards) and the installation of the software (OS, applications, drivers). The computers that have been chosen for the development system are Compaq Alpha workstations (164LX) because of two reasons: first, at the begin of the project, TNet only ran under Tru64, an operating system by Compaq for their Alpha boxes - and second, Portals is actually used at Sandia National Laboratories on a Compaq Alpha cluster, the CPlant.

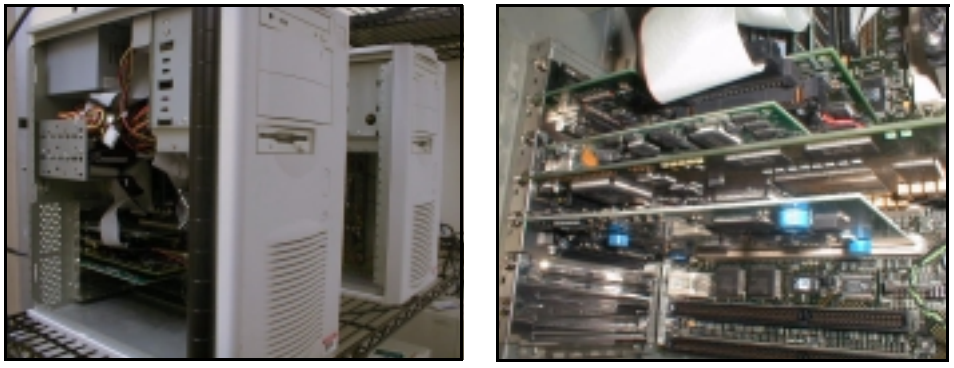


FIGURE 4.1. Development System at the Scalable Systems Lab, UNM

Each systems is equipped with a 21164 alpha processor, a 4.5 GB UW-SCSI hard disk, 320 MB memory and 100BaseT, Myrinet & TNet network interface cards. The graphic controllers have been removed as these computers are not used as workstations. To access the BIOS and the main console when no network connection is available (rebooting system, testing new kernels, system message outputs) the 164LX supports tty's on the first serial line. A PC based Linux box is used to remotely access the serial consoles by connecting the ports with a *null modem cable*. A telnet or SSH session to this Linux box followed by starting *cu* or *minicom* on the corresponding serial device will then redirect the main console prompt to the terminal emulation. This is commonly known as a *terminal server.*

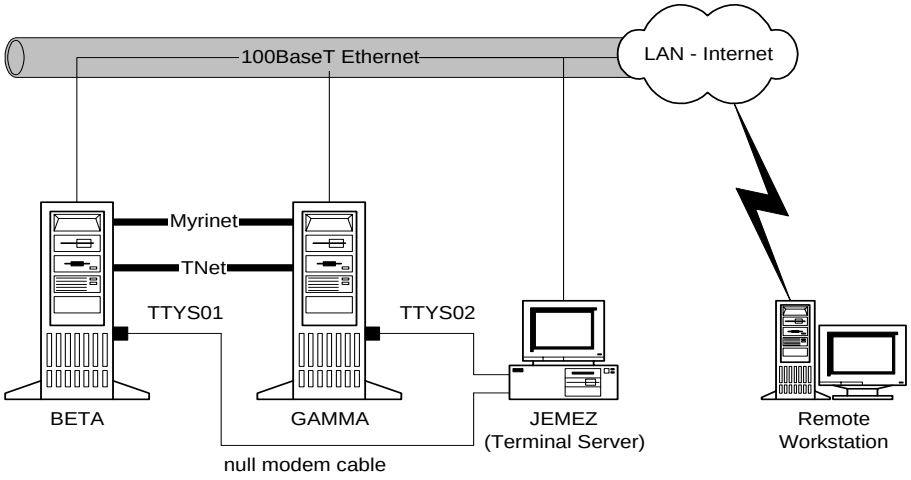


FIGURE 4.2. Development System network topology

In an early phase, both systems are capable of booting either Compaq TRU64 (v 5.0) or RedHat Linux (v 6.2, kernel 2.2.14) by selecting the flags on the *boot* command in the SRM BIOS console ("*boot*" starts TRU64, "*boot dva0 -fl 1*" launches Linux).
TRU64 has only been installed to get familiar with TNet, including FCI, MPI and Cosmos to install it later on Linux for the project. If TRU64 is not longer used, it is removed and the boot partition will hold the Linux kernel.

The software development is entirely done under RedHat Linux and GNU C. To start the programming from a working system, a mini CPlant using Myrinet and a TNet environment coexist under Linux. The tricky part is to apply the kernel patches that are needed for CPlant so that the TNet modules are still insertable and the kernel remains stable.
This two node development system now serves for any experimentation with Portals and TNet using a remote terminal.

4.2 Setup

The setup of the project environment involves the merging of two independent schemes, TNet and CPlant. This is accomplished by assimilating the TNet hierarchy into Cplant and developing the new module there. Following this scheme, a CPlant Makefile calls one of the original TNet Makefiles when needed, allowing the TNet object files to be compiled the same as before. These object files are then linked with the appropriate CPlant files to complete the module. The only change to the TNet environment needed is to let the Makefiles know where to find the new Portals header files.

The CPlant hierarchy is complicated, but only a few key regions are important for development. Compilation under the CPlant environment is made modular by allowing every subdirectory to have its own Makefile that is responsible for calling the Makefiles of its subdirectories recursively. Therefore, only the local Makefile is changed when part of the hierarchy is modified. Of course, the most important region for development is the location where the module specific files are located in *~/Cplant/top/compute/OS/portals/P3oT* (i.e. the library side NAL). A new Makefile is added here to compile and link the appropriate files from Portals as well as the TNet object files.

4.3 TNAL

TNAL, the network abstraction layer for Portals over TNet, consists of a API- and LIB-side. The implemented version is based on the software design in chapter 3.

4.3.1 API-side NAL

4.3.1.1 Architecture

The API-side implementation of the NAL is the shorter part. The *forward* function has to pass TNet specific calls instead of Myrinet calls.

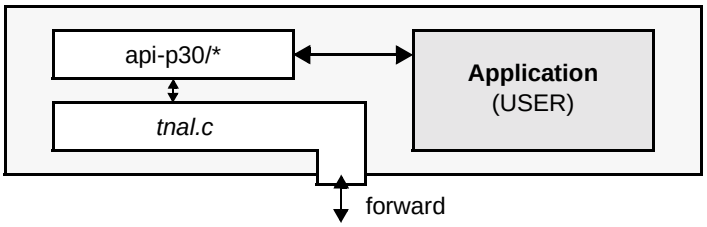


FIGURE 4.3. API-side NAL for TNet

The interface that is opened through *forward* is called *PTL_IFACE_T*. The code extracts in the following sections will explain the changes in *tnal.c*.

4.3.1.2 Code extracts

The three main functionalities in *tnal.c* are **open** & **close** of the device & library and **forward**, which is used for all transactions out of the user mode.

- *P3DEV* addresses the Portals 3 device, in this case it is defined as "*/dev/tnet0*" as the network interface is TNet. *PTL_IFACE_T* opens, after some checks, the device for read and write by using *p3fd* as filedescriptor. Before the Portals 3 library can be opened, some setup information on the **CMB** & **gcw** (see LIB-side NAL) is stored in *topen* and passed together with the *PTL_OPEN* instruction down to the driver using *ioctl*.

```
nal_t * PTL_IFACE_T(int interface,
                    ptl_pt_index_t ptl_size,
                    ptl_ac_index_t ac_size)

{..
p3fd = open(P3DEV, O_RDWR)
..
ioctl(p3fd, TNET_PTL_OPEN, &topen)
..}
```

- Data is transfered across the protection domain by *forward*, which after packing all arguments in *tforward* calls *ioctl*. The library will then use *PTL_DISPATCH* to handle the message to the Portals library.

```
static int forward(nal_t *nal,
                   int id,
                   void *args,
                   size_t args_len,
                   void *ret,
                   size_t ret_len)

{..
ioctl(p3fd, TNET_PTL_DISPATCH, &tforward)
..}
```

- Before the device can be closed, the Portals library has to be closed first. Both instructions are called in the *shutdown* function. Again, *ioctl* is used to pass *PTL_CLOSE* to the library.

```
static int shutdown(nal_t *nal, int interface)
{..
ioctl(p3fd, TNET_PTL_CLOSE, NULL)
..
close(p3fd)
..}
```

4.3.2 LIB-side NAL

As seen in chapter 3, the design of this preliminary implementation of Portals over TNet consists of a hybrid kernel module supporting Portals 3 and FCI. Of course, FCI could be removed to make the driver smaller but this would have no effect on performance and as this work is in a development state, it could be useful for testing or benchmarking.

4.3.2.1 Architecture

The design makes usage of the TNet driver (tnet.c) to replace the hardware dependent calls to Myrinet in the library-side NAL.

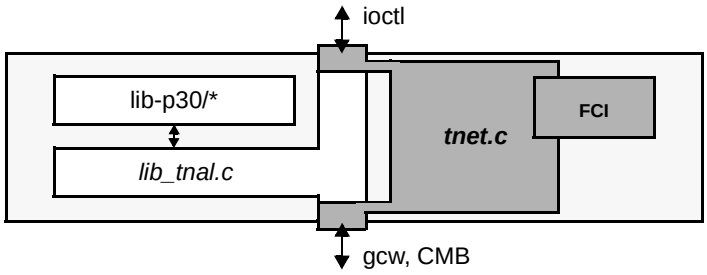


FIGURE 4.4. LIB-side NAL for TNet

The driver communicates to the application with *ioctl*. This call is much more functional in the TNet system (firmware download, configuration, card management, etc.). The three new cases *TNET_PTL_OPEN*, *TNET_PTL_CLOSE* and *TNET_PTL_DISPATCH* were therefore added in the file *tnet.c* to extend the functionalities for Portals 3.

Communication to the network card is done through the contiguous memory block. Data is written and read using the **global communication window** (gcw).

4.3.2.2 Dataflow

In a preliminary phase, data is always passed through the CMB. To make a first implementation simple, the contiguous memory block is divided by twice the number of remote nodes in the cluster (once for send and once for receive). The development system contains two nodes, thus one node is called remote and therefore the CMB is divided by two (one area to send to the other node and one for receiving messages).

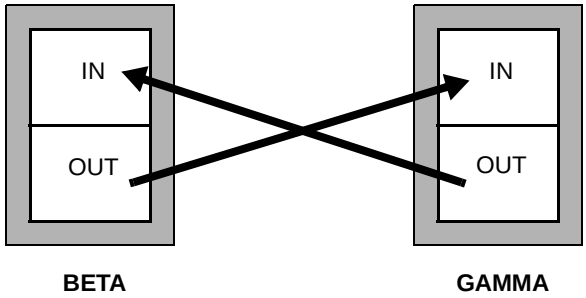


FIGURE 4.5. Contiguous Memory Block separation

This preliminary solution doesn't scale and should not be used in any further implementation.

4.3.2.3 Code extracts

Compared to the API-side NAL, modifications in the code for the LIB-side are much more complicated. The following extracts should help to understand the principle that was applied to build the P3oT kernel module.

- In *tnet.c*, three new cases to *TNET_ioctl* (performing the *ioctl* functionality) support now **open**, **dipatch** and **close** calls to the Potals 3 library.

```
case TNET_PTL_OPEN:
..
open_lib_tnal(..);
..
break;

case TNET_PTL_DISPATCH:
copy_from_user(..);
lib_dispatch(..);
copy_to_user(..);
..
break;

case TNET_PTL_CLOSE:
..
close_lib_tnal(..);
..
break;
```

- For an incoming call in form of a remote interrupt (at the moment this is the case for any transferred message), the header is copied out of the CMB and given as an argu-

ment to *lib_parse* (see *lib_tnal.c*). This has to be done only if *open_lib_tnal* has been called before. Otherwise it is not a Portals program doing this remote interrupt.

```
if(tnal_cb != NULL && tnal_data != NULL)
{..
memcpy(&hdr, tnal_data->cmb_kernel + (RECVBASE /
sizeof(TNET_UInt32)), sizeof(ptl_hdr_t));
lib_parse(tnal_cb, &hdr, NULL);
}
```

Once *lib_dispatch* is executed and arguments are passed, the library performs its operations. The good thing about the NAL is, that the design of a new network abstraction layer doesn't require deep knowledge on API or library. After the "magic" happened in the library, different calls to lib_tnal.c can occur. The following functions are implemented among others in *lib_tnal.c*. Most of the work was done in **open**, **close**, **send**, **recv** and **write**.

- **Opening** and **closing** the library through *ioctl:*

```
open_lib_tnal(struct inode *inode,
              struct file *file,
              void *gcw_user,
              void *cmb_user,
              TNET_UInt32 gcw_length,
              TNET_UInt32 cmb_length,
              void *gcw_kernel,
              void *cmb_kernel)

close_lib_tnal(void)
```

- **Send** simply copies header and data into the CMB and sends the content using DMA transfer mode. The *private* field in *tnal_send* is a NAL-specific value that will be passed to any callbacks produced as a result of this API call (in this implementation, **private* is *NULL*). The cookie is a pointer to a library private value that is passed to *lib_finalize* once the message has been completely sent. It should not be examined by the NAL for any meaning. The destination node is addressed with *dnid*. Process, group and rank id are, as this is also the case for Myrinet, not used. A pointer to the data in user space and its length are the last arguments in this functions.

```
static int tnal_send(nal_cb_t *nal,
                     void *private,
                     lib_msg_t *cookie,
                     ptl_hdr_t *hdr,
                     int dnid,
                     int pid_in,
                     int gid_in,
                     int rid_in,
                     user_ptr data,
                     size_t len)
```

As the Portals header information for the message to be sent is already in the kernel, it can be copied to the send portion of the allocated CMB with a simple **memcopy**. Because the pointer to the CMB is 32 bit aligned, the offset calculation must be divided by *sizeof(TNET_UInt32)*.

```
memcpy(nal_data->cmb_kernel + (SENDBASE / sizeof(TNET_UInt32)),
hdr, sizeof(ptl_hdr_t));
```

The message body instead is copied to the CMB from user space.

```
copy_from_user(nal_data->cmb_kernel + ((SENDBASE +
sizeof(ptl_hdr_t)) / sizeof(TNET_UInt32)), data, len);
```

The message is then sent out of the CMB in packets. The last packet carries the **remote interrupt** flag to wakeup the target, which will then copy the data out of its CMB into the application.

```
for(j=0;j<PACKETS;j++)
{
targetaddr = RECVBASE + j*PDIS;

nal_data->gcw_kernel[0]=
   /* destination id */
   (dnid & TNET_APP_SEND_DESCRIPTOR_IDMASK) |
   /* message type */
   TNET_APP_SEND_DESCRIPTOR_UNICAST |
   /* mapping type */
   TNET_B_APP_SEND_DESCRIPTOR_DIRECTMAPPED |
   /* descriptor pattern */
   TNET_B_APP_SEND_DESCRIPTOR_PATTERN |
   /* rti id */
   ((TNET_APP_REMOTEINTERRUPT_ID_FLAG3<<24) &
   TNET_B_APP_SEND_DESCRIPTOR_RTI_MASK);

nal_data->gcw_kernel[1]=(PLEN>>2);
nal_data->gcw_kernel[2]=(targetaddr>>2);
nal_data->gcw_kernel[3]=((targetaddr>>34) &
TNET_APP_SEND_ADDRESS_MSDW_MASK)
| TNET_APP_SEND_ADDRESS_MSDW_PROC(0x0);
mb();

/* write the data */
dma.target_gcs_base=0;

//TNET_DMA expects a user space address for the cmb
dma.source_virt_base=(void *)(nal_data->cmb_user + (SEND-
BASE+j*PDIS)/sizeof(TNET_UInt32));
dma.length=PLEN;
TNET_Ctl(nal_data->inode, nal_data->file, TNET_DMA, (void
```

```
*)(&dma))
}
```

After all packets are delivered, the library is informed with *lib_finalize* so Portals can update the event queue for the memory descriptor.

```
lib_finalize(nal, private, cookie);
```

- The function *write* is essentially a cross-protection domain memcopy using *copy_to_user* and is not discussed here. **Receive** instead is called in response to lib_parse, reads *mlen* bytes and deposits them into *data*.

```
tnal_recv(nal_cb_t *nal,
          void *private,
          lib_msg_t *cookie,
          user_ptr data,
          size_t mlen,
          size_t rlen)
```

Because the data is already arrived and sits in the CMB, *tnal_recv* just has to copy it to user space and call *lib_finalize* with the *lib_msg_t *cookie.*

```
copy_to_user(data, nal_data->cmb_kernel + ((RECVBASE +
sizeof(ptl_hdr_t)) / sizeof(TNET_UInt32)), mlen);

lib_finalize(nal, private, cookie);
```

- Since the NAL may be in a non-standard hosted environment it can not call *malloc*. This allows the library side NAL to implement the system specific *malloc*. In the current reference implementation the library only calls *nal->malloc* when the network interface is initialized and then calls free when it is brought down. The library maintains its own pool of objects for allocation so only one call to *malloc* is made per object type.

```
tnal_malloc(nal_cb_t *nal, size_t size)
{..
vmalloc(size)
..}

tnal_free(nal_cb_t *nal, void *ptr)
{
vfree(ptr);
}
```

4.4 Testing

Once the module is built, it must be loaded on both machines before it can be used. To accomplish this, a script was written that automatically copies the module to the */cplant/modules* directory as well as the corresponding directory on the other machine (test programs are also copied at this point), unloads the previous modules if they are still running and loads the new module along with several others. Because Portals 3 is still in the development stage, other modules are required to perform fundamental tasks. For example, the old Portals 2 module is sometimes used to initiate a job by synchronizing information on both nodes. This and other helper modules are loaded in the following order: *portals.mod* (Portals 2), *rtscts.mod*, *myrIP.mod* and, finally, the new *P3oT.mod* is loaded last.

With the help of the Portals 2 module, a test program can be run atomically from the Portals 3 testing directory (*~/Cplant/top/compute/OS/portals/p3tests*). These tests are compiled together into *p3test*, so they can be run one after the other and test all aspects of the Portals implementation. Using this program, preliminary results show a latency of around 80 microseconds for the P3oT module. It should be noted that this figure comes from an initial implementation with no optimizations and which still includes one memory copy to or from the CMB per node and an OS call upon sending. When these time consuming activities are circumvented, the latency will be much less.

CHAPTER 5

Conclusion

After a short recapitulation on the results of this project, some prospects about further work on Portals over TNet are presented. A final commentary is given about the overall project experience.

5.1 Results

The evaluation of the possibility to implement Portals 3 on TNet has been discussed in the design and shown in form of a preliminary software implementation. The experience that was obtained during this project is reflected in this report and will help for future research.

Even though the current implementation is in an alpha state, TNet offers some benefits compared to Myrinet. The CRC generation and checking is done in hardware. Together with the retransmit protocol, Portals 3 can be executed over a high reliable network. The P3oT module could in a next state be improved and eliminate costly interrupt calls. Implementing blocking calls would reduce latency highly.

A hardware implementation instead would result in a much longer project time as the Portals 3 library would have to be rewritten in vhdl. With the current solution, parts of portals can now be tested on TNet. The P3oT module serves as a running start point to go to the next implementation level.

5.2 Prospects

The following design idea uses a modified Portals 3 library that is compiled together with the API into the applications. This solution will make use of the current TNet firmware and use the pagetable and the ID validation table on the NIC as a different implementation of match lists and memory descriptors. Because pagetable entries do not correspond to the format of memory descriptors, the library would have to be updated.

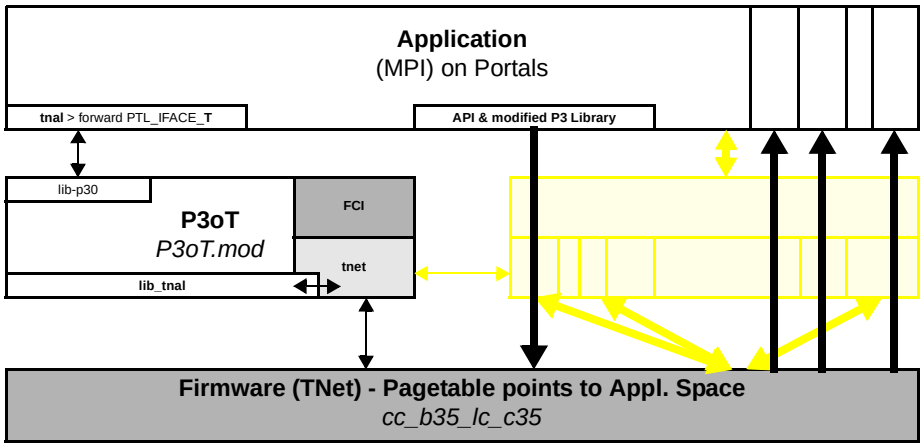


FIGURE 5.1. Design draft for next PoT implementation step

Remark: The design on figure 5.1 is only a draft. A deeper analysis is required in order to start the implementation.

5.3 Commentary

The field of High Performance Computing is very interesting but also challenging. Interesting, because of the variety of the work. A network designer deals with the hardware as well as with the software to make the “glue” that connects both worlds in an optimal way. The challenge could be described as “the art to make it run”. It requires a wide range of knowledge and perseverance to implement the ideas of a new design.

CHAPTER 6

References

[1] Ron Brightwell, Tramm Hudson, Rolf Riesen, Arthur B. Maccabe
The Portals 3.0 Message Passing Interface
Sandia National Laboratories & The University of New Mexico, 11/1999

[2] Message Passing Interface Forum
MPI: A Message-Passing Interface Standard
University of Tennessee, 1995

[3] William Gropp, Ewing Lusk
User's Guide for mpich, a Portable Implementation of MPI Version 1.2.0
Argonne National Laboratory, University of Chicago, 12/1999

[4] Martin Lienhard, Martin Heimlicher
TNet PCI Adapter Specification
Supercomputing Systems AG, 05/1999

[5] Josef Nemecek, Martin Frey
TNet Driver Specification
Supercomputing Systems AG, 10/1999

[6] Martin Frey
TNet CMM Library
Supercomputing Systems AG, 10/1998

[7] Stephan Brauss, Martin Frey
TNet Message Passing Specification
Supercomputing Systems AG, 10/1999

[8] *FCI, Fast Communication Interface*
Swiss Federal Institute of Technology Zurich & Supercomputing Systems, 2000

[9] Peter S. Pacheco
Parallel Programming with MPI
Morgan Kaufmann Publishers, ISBN 1-55860-339-5, 1997

[10] Alessandro Rubini
Linux Device Drivers
O'Reilly, ISBN 1-56592-292-1, 02/1998

[11] *The Official Red Hat Linux Alpha/SPARC Installation Guide*
Red Hat, Inc, 2000

[12] Adrian Riedo
The Portals over TNet website, important links to HPC sites
http://hpc.fribyte.ch, 2000

APPENDIX A

Technical Abbreviations

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input Output System
CC	Communication Controller
CMB	Contiguous Memory Block
CPlant	Computational Plant
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
FCI	Fast Communication Interface
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GBE	Gigabit Ethernet
GCW	Global Communication Window
HPC	High Performance Computing
IP	Internet Protocol

LAN	Local Area Network
LC	Link Controller
LIB	Library
MCP	Myrinet Control Program
MPI	Message Passing Interface
NAL	Network Abstraction Layer
NIC	Network Interface Card
OS	Operating System
P3oT	Portals 3 over TNet module
PCI	Peripheral Component Interconnect
PoT	Portals over TNet
RISC	Reduced Instruction Set Computer
RTS/CTS	Ready To Send / Clear To Send
SAN	System Area Network
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
SSH	Secure Shell
TCP	Transmission Control Protocol
VAS	Virtual Address Space
VCA	Virtual Communication Address
VHDL	Very high speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integration