Time Division Multiplexing of Network Access by Security Groups in High Performance Computing Environments

by

Joshua Ferguson

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2012 by the
Graduate Supervisory Committee:

Sandeep Gupta, Chair
George Ball
Georgios Varsamopoulos

ARIZONA STATE UNIVERSITY

May 2012

ABSTRACT

It is commonly known that High Performance Computing (HPC) systems are most frequently used by multiple users for batch job, parallel computations. Less well known, however, are the numerous HPC systems servicing such data so sensitive that administrators enforce either *a*) sequential job processing - only one job at a time on the entire system, or *b*) physical separation - devoting an entire HPC system to a single project until recommissioned. The driving forces behind this type of security are numerous but share the common origin of data so sensitive that measures above and beyond industry standard are used to ensure information security. This paper presents a network security solution that provides information security above and beyond industry standard, yet still enabling multi-user computations on the system. This paper's main contribution is a mechanism designed to enforce high level time division multiplexing of network access according to security groups. By dividing network access into time windows, interactions between applications over the network can be prevented in an easily verifiable way.

ACKNOWLEDGEMENTS

I would like to acknowledge . . .

To my wife for her unwavering support

Contents

Page

List of Figures

Chapter 1

Introduction

High Performance Computing (HPC) systems are comprised of numerous individual computing systems networked and administrated together such that they can be used as a single system. Popular examples of these systems include some custom made such as the Cray I and the modern IBM Sequoia [11], though more common are simple Computer Clusters in which Commercial Off The Shelf (COTS) equipment is utilized [5]. Security within these systems is often enforced through traditional Operating System (OS) mechanisms of memory protection and file access privileges [5].

Application developers for this domain of systems spans a broad spectrum, ranging from undergraduate students learning concurrent programming to defence contractors executing classified simulations. Moving towards the most demanding end of the spectrum, security concerns among application-side stakeholders increase and additional methods are employed to enforce information security. At some point along this spectrum stakeholders decide on physical separation to satisfy security concerns. The reasons behind this can be numerous, but likely stem from two major goals: simplicity of implementation and verification; and risk aversion/management. It is undeniable that physical separation provides a level of information security that is hard to replicate through the use of software, however the financial costs are significant - devoting entire HPC systems to running jobs sequentially.

This paper presents Time Division Multiplexing (TDM) of network access as a viable alternative to physical separation. By modulating network access between application security groups we can provide an intuitive security mechanism capable of being verified in real-time. Furthermore by implementing this mechanism at the operating system level it becomes transparent to user applications, meaning no modification to existing application code is necessary.

Chapter 2
Related Work

The problem statement and proposed solution represent the intersection of two somewhat disparate fields - Time Division Multiplexing and High Performance Computing Security. Related works are therefore divided between the two.

High Performance Computing Security

The size and cost of HPC environments dictates that each system is somewhat unique. The security solutions implemented within each are similarly unique. Sandholm et al. [15] make an attempt at rectifying this larger problem by creating a framework that uses automates user access permissions and resource allocation using ”XACML (eXtensible Access Control Markup Language)”. They further extend their solution by tying it in to existing job submission tools (Globus Toolkit [13] and NorduGrid [14]).

Time Division Multiplexing

Chapter 3

Problem Definition

We begin by defining an abstract HPC environment through which the general case of our security challenge is shown. In this section we provide brief descriptions of the major resources common to most HPC systems. Furthermore, to design our mechanism, certain assumptions must be made on how each resources is operated.

Environmental Assumptions

There are four basic resources in most HPC systems represented in Figure 3.1 as *a*) compute nodes, *b*) persistent storage, *c*) administrative nodes, *d*) network infrastructure. Worth consideration also is the process of job allocation and the execution of jobs.

Compute Nodes

Compute nodes are independent computing devices designated to run user submitted applications. These devices are capable of storing temporary data locally. They send and receive data across network infrastructure for three main purposes:*a*) storing or accessing data on the persistent storage devices; *b*) relaying data between

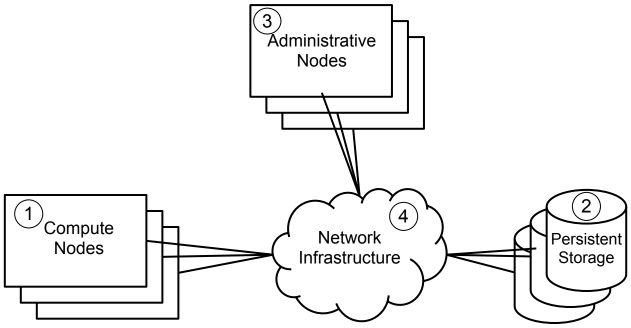

Figure 3.1: An abstract HPC environment.

Figure 3.2: The $KG-200$ Inline Media Encryptor, certified by the NSA for use in securing persistent storage [1].

other compute nodes working in tandem on the same user application; *c*) and sending or receiving commands (or reports, as the case may be) from the administrative nodes, through which users interact.

It is assumed that these compute nodes do not co-locate user applications (e.g., two different user applications are running in the same system memory) and that user applications are not given administrative access at this level. No assumption is made about the use of virtual machines on compute nodes.

Persistent Storage

Persistent storage devices are capable of storing large quantities of user application data, and are usually of much higher capacity than the compute nodes. User data is commonly co-located across disks using RAID (redundant array of inexpensive disks) technology [9] for higher storage efficiency and redundancy.

It is assumed that Inline Media Encryptors (IMEs) and Posix permissions are used to enforce data access rules within persistent storage [6]. IMEs have been certified for use in classified networks by the U.S. National Security Administration since 2006 [1].

Administrative Nodes

Administrative nodes are computing devices where *a*) both administrators and users interact with the system, common tasks of which include issuing job or system commands, accessing reports and results, and performing maintenance; *b*) resource management software is centrally located and executed [10], common examples include IBM's Tivoli Workload Scheduler and the MOAB Cluster Suite by Adaptive Computing [2][8].

It is assumed that the scheduler located here is capable of providing access to the list of current running applications and the hardware resources devoted to them.

Network Infrastructure

Network infrastructure devices facilitate the transmission of data between nodes within the HPC system. Mediums vary widely and include copper, optical, and wireless. The most common technologies used in HPC environments are Ethernet and InfiniBand [4][12].

It is assumed that the network infrastructure uses Internet Protocol to communicate among nodes.

Job Execution

Best practices for developing jobs run on HPC systems dictates the minimization of I/O, both to disk and over the network [17]. This I/O minimization is due to the dramatic increase in access time as data moves further away from the CPU and main memory. It's over 50 times more costly to access 1MB of data from the network than it is from main memory [7]. This overhead increases to almost a factor of 100 if that data is initially read from disk then sent over the network [7].

In the effort to minimize the cost of I/O transactions, previous researchers have shown that batching I/O into larger transactions can reduce overhead [16]. The difference between sequentially reading 1K files from network disks and reading 256MB from network disks shows a factor of 1700 improved performance by reading in larger batches [17]. The batching of I/O, especially the most costly forms

Action	Time to Complete
L1 cache reference	0.5 ns
L2 cache reference	7 ns
Main memory reference	100 ns
Read 1 MB sequentially from memory	250,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns

Table 3.1: Access time examples showing the magnitude of difference between data over I/O and data locally stored [7].

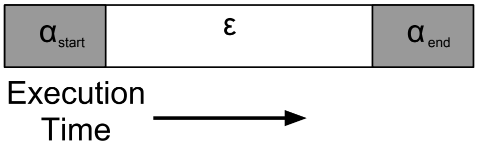

Figure 3.3: The assumed model of application execution in an HPC environment. α_{start} and α_{end} are periods where execution is I/O bound, and ε is the prominent period where execution is CPU bound. This structure adheres to research showing batched I/O minimizes the I/O cost in terms of time.

(disk and network) is therefore considered best practice when possible [3]. It is therefore assumed that job developers will attempt to maximize I/O batching, the optimal case of which would have an I/O transaction history similar to that shown in Figure 3.3.

Security Challenge

The security fear of customers with sensitive data is that a different customer could, through chance or intention, acquire or manipulate their data. There exist three main mediums across which data may be exposed.

		Solutions		
Hardware Location	**Security Challenge**	Board Separation	IME	Posix Permissions
Compute Nodes	Local Data Storage	✓		✓
	Local Data Processing	✓		✓
Persistent Storage	Co-location of user data		✓	✓
Administrative Node	Accept and Schedule User Jobs			✓
Network Infrastructure	Transmit intra-application data between compute nodes			
	Transmit data to/from compute nodes and persistent storage			
	Transmit commands from scheduler to compute nodes			

Table 3.2: Security challenges and technology used to solve them

Chapter 4
Design Goals

Here we describe the major goals in the design and implementation of the TDM security mechanism.

Mechanism and Policy Separation

Envisioning the tool as enabling a more secure form of HPCaaS, the type, number, and scale of jobs assigned to any of the systems can be widely varied. To handle this, at the least, the tool must be capable of receiving and modifying policy at the start of each new job. Furthermore, to provide more fine-grained control an online policy modification scheme is desirable for managing performance tradeoffs between security groups. We define the mechanism and policy split of our tool to be an ability to receive and actuate policy changes at the finest granularity in which the tool operates. [Mechanism Policy split is really along the difference between the formal notion that nodes must be transparently separated from the medium and how it is accomplished (with a queue, at the TCP layer, on ethernet, etc.)]

Network Fabric Agnostic

Two major technologies are used to network HPC systems: Ethernet and InfiniBand. Any tool for improving security across the broad spectrum of HPC systems must be capable of operating in each. Further, numerous network topologies exist within these technologies; switched fabric and tree structures are the most common for InfiniBand and Ethernet, respectively. For our purposes we define this tool to be network fabric agnostic if the tool is conceptually capable of being implemented in either Ethernet or InfiniBand networks.

User Application Transparent

A fundamental requirement of the tool is that it be transparent to user applications. Applications written for HPC environments are often quite complex and it is likely that customers would be reluctant to make even minor modifications, especially to programs written in the past that are under re-use. For our purposes we define user application transparency as the ability to run an application without modification to successful end on an HPC system using our tool, given that it can also do so on a system not using our tool.

Chapter 5
Formal Definition

Suppose S is a finite set of security groups, s.t. each security group $s \in S$ is made up of a number of compute nodes. Given S, the language our mechanism operates on can be generated by the following grammar. Because the language is dependent on the security groups S, this grammar must be generated based on it. This is done in two steps:

First, we define the base grammar:

$G^1 = (V^1, \Sigma^1, R^1, \mathscr{A})$, where

$V^1 = \{\mathscr{A}, W\}$ non-terminal symbols

$\Sigma^1 = \{\emptyset\}$ terminal symbols

$R^1 = \{ \quad \mathscr{A} \rightarrow \varepsilon,$ rules of production

$\quad \mathscr{A} \rightarrow W\mathscr{A}|W\}$

This base grammar, through the non-terminal symbols and production rules, establishes a means of generating the base language form of unordered windows ($W \in V^1$) in an arbitrary length such as WW or $WWWWW$.

Next, we generate the S specific definitions. To do so it is first necessary to define notation for two special terminal symbols and three special sets:

$o_{s,i}$ - an open command issued to node i within security group s,

$a_{s,i}$ - an acknowledgement received from node i within security group s,

θ_s - the set of all $o_{s,i}$ terminals for security group s,

α_s - the set of all $a_{s,i}$ terminals for security group s, and

$\pi(A)$ - the set of all permutations of the set A.

These definitions allow us to define a final, special set:

$$\Lambda_s = \pi(\theta_s) \times \pi(\alpha_s)$$

Intuitively, Λ_s is a set of ordered sets expressing each permutation of θ_s matched with each permutation of α_s. For example, given a security group s made up of two elements s.t. $s = \{1,2\}$, Λ_s is defined:

$$\begin{aligned}\Lambda_s = \{&(o_{s,1}, o_{s,2}, a_{s,1}, a_{s,2}), \quad (o_{s,2}, o_{s,1}, a_{s,1}, a_{s,2}),\\ &(o_{s,1}, o_{s,2}, a_{s,2}, a_{s,1}), \quad (o_{s,2}, o_{s,1}, a_{s,2}, a_{s,1})\}\end{aligned}$$

The sets within Λ_s represent all legitimate command sequences within a window (W) for security group s. The key property of the sets within Λ_s is that each node within the security group is issued an open command, in any order, followed by acknowledgements from each node within the security group, once again in any order.

With these definitions established we can now formally define an S specific grammar:

$G^2 = (V^2, \Sigma^2, R^2, \emptyset)$, where

$V^2 = \{W\}$

$\Sigma^2 = \{[o_{s,i}, a_{s,i}] : \forall i \in \forall s \in S\}$

$R^2 = \{[W \rightarrow \lambda] : \forall \lambda \in \Lambda_s : \forall s \in S\}$

These definitions add new terminal symbols and the necessary production rules to generate them. Note the use of Λ_s in the production rules. These rules provide every possible command sequence possible for any window W s.t. every node issued an open command is required to report back with an acknowledgement before continuation onto another window.

Finally, the language our mechanism accepts for security group S can be formed using the union of the previous two grammars:

$$G = (V, \Sigma, R, \mathscr{A}), \text{ where}$$
$$V = V^1 \cup V^2$$
$$\Sigma = \Sigma^1 \cup \Sigma^2$$
$$R = R^1 \cup R^2$$

Chapter 6

Implementation

As a proof of concept we have implemented a version of the tool for the Linux operating system using C++11. In this section we will describe the tool's architecture, operation, and how it adheres to the design goals from the previous section.

Overview

Algorithm 1 Window Controller opening and closing network access windows.

```
function Open_Windows(Scheduler)
    Scheduler.initialize();
    while End_Command_Not_Received do
        Security_Group ← Scheduler.get_next_group();
        Security_Group.state ← STATE.OPEN;
        for each node ∈ Security_Group do
            send(node.address,
                    Security_Group.crypto_sign(COMMAND.OPEN));
            node.state ← STATE.OPEN;
        while Security_Group.state == STATE.OPEN do
            node_response ← block_on_receive_message();
            if node_response.state == STATE.CLOSED then
                node.state ← STATE.CLOSED;
            else
                throw ERROR.UNCLOSED_NODE;
            Security_Group.state ← STATE.CLOSED;
            for each node ∈ Security_Group do
                if node.state == STATE.OPEN then
                    Security_Group.state ← STATE.OPEN;
```

The tool is composed of four major components: the window scheduler, ingress controller, egress controller, and the state controller. The window scheduler can be located on any administrative node within the system, preferably co-located with the system job scheduler. The remaining controllers are located throughout the

Algorithm 2 Node Control Mechanism opening and closing access to the network.

```
function Node_Control_Mechanism
    Queue ← initialize_queue;
    while exit_command_not_received do
        state ← Close_Network_Access(Queue);
        while open_command_not_received do
            message ← block_on_receive_message();
        state ← Open_Network_Access(Queue);
        sleep(message.time);
        state ← Close_Network_Access(Queue);
        send_acknowledgement(state);

function Close_Network_Access(Queue)
    state.egress ← Network_Egress.enqueue(Queue);
    state.ingress ← Network_Ingress.drop_packets();
    return state

function Open_Network_Access(Queue)
    state.ingress ← Network_Ingress.accept_packets();
    state.egress ← Queue.process_packets_to_network();
    return state
```

HPC environment, with a copy on each compute node that is designated to execute user applications.

The system provides enhanced security by time dividing access to the network according to security groups. As jobs are scheduled on the system, the window scheduler must be informed of the intended location and their assigned security group. As jobs are run on compute nodes throughout the system the window scheduler communicates with the state controller on each node to designate time windows. During any individual time window only one security group has authorization to access the network. For any given time window in which a security group does not have access to the network, outgoing network packets are stored in a local queue while incoming packets are just ignored and deleted. The window scheduler is tasked with alternating time window authorization between security groups.

The following subsections describe each components operations in further detail.

State Controller

The state controller has three major tasks:

1. Securely send and receive communication with the window scheduler for the system
2. Transit both the ingress and egress controllers between states of network access and denial
3. Collect and store performance data on the egress queue's memory usage

Ingress Controller

The ingress controller is a firewall of incoming network packets and has two states:

1. Open access of network packets to applications on the node
2. Closed to network packets except for those from explicitly allowed sources (a whitelist style of firewall)

During the open state incoming packets are processed normally. During the ingress closure, incoming packets, except those allowed by the whitelist, are dropped. The whitelist is designated to allow only necessary infrastructure communications such as Network Time Protocol (NTP), performance measurements, and especially packets from the window scheduler.

Egress Controller

The egress controller, similar to the ingress controller, has two major states:

1. Open flow of packets onto the network
2. Diversion of outgoing packets into a blocked queue

Window Scheduler

The window scheduler has three major tasks:

Determine system network access states for the next time window Validate the closure of the previous time window Communicate the next time window states to compute nodes To determine the network access states for the next time window the scheduler must run a scheduling algorithm on a few historical inputs. The base case scheduling algorithm is round robin (i.e., equal window size for each security group). To improve performance, a number of heuristics have been considered for the creation of a dynamic priority scheduling algorithm: the egress controller memory usage of compute nodes, number of TCP timeouts, and externally imposed priorities.

Chapter 7

Performance

Expected Values

Leading systems in high performance computing can serve as indicators of where commercial HPC systems will be in the upcoming years. The Titan system at Oak Ridge National Laboratories (ORNL) is one such system, the technical details of which are displayed in Table 7.1.

Here we attempt to quantify how the mechanism affects memory usage at the compute node level and effective bandwidth.

$T_{on,n}$ Time, within a window, where compute node n has access to the network.

T_{window} Length in time of a single window.

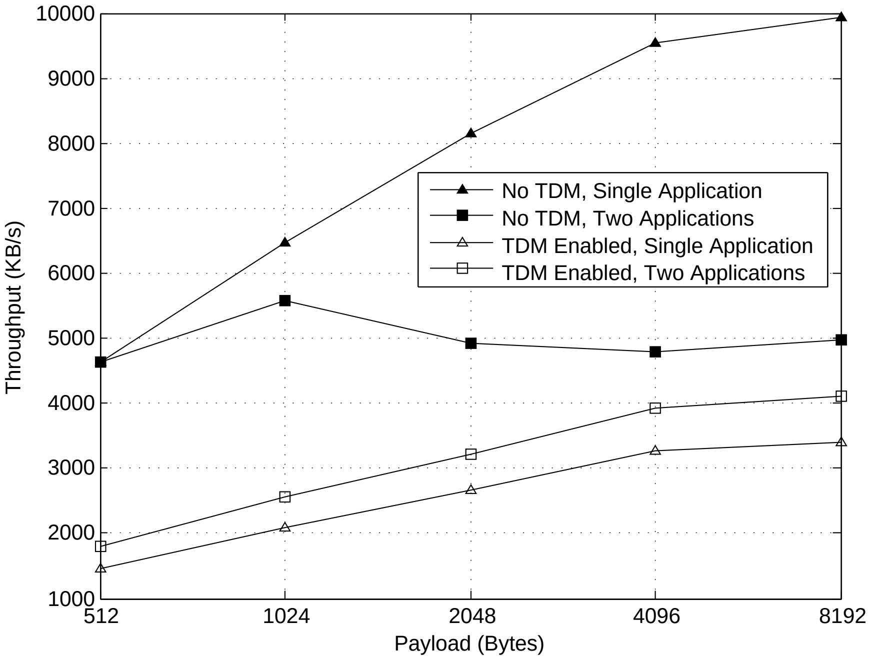

Figure 7.1: TDM effect on TCP application performance.

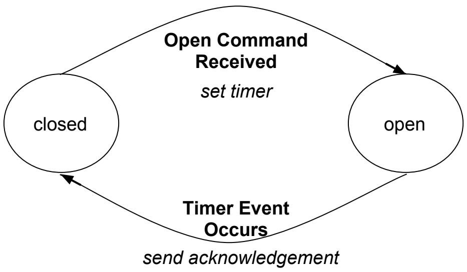

Figure 7.2: State diagram of a compute node.

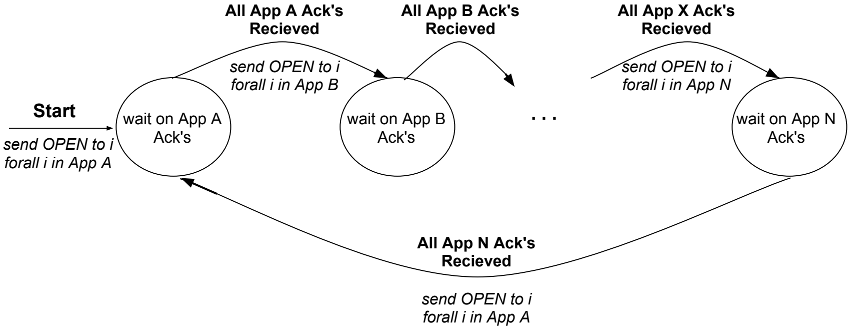

Figure 7.3: State diagram of an example window controller.

Technical Specifications	
CPUs	16 cores @ 2.2GHz
Main Memory	32GB @ DDR3

Table 7.1: Technical specifications of compute nodes within the Titan at ORNL.

$\Pi_{app,n}$ Speed at which the application on compute node n generates network traffic.

$\Pi_{NIC,n}$ Speed at which the NIC on compute node n can transmit traffic onto the network.

Case Study

To verify and test the mechanism a test bed was created out of five Dell 1955 servers seen in Figure 7.5. These servers run Ubuntu server version 12.04 and had their network interfaces configured and connected according to Figure 7.4.

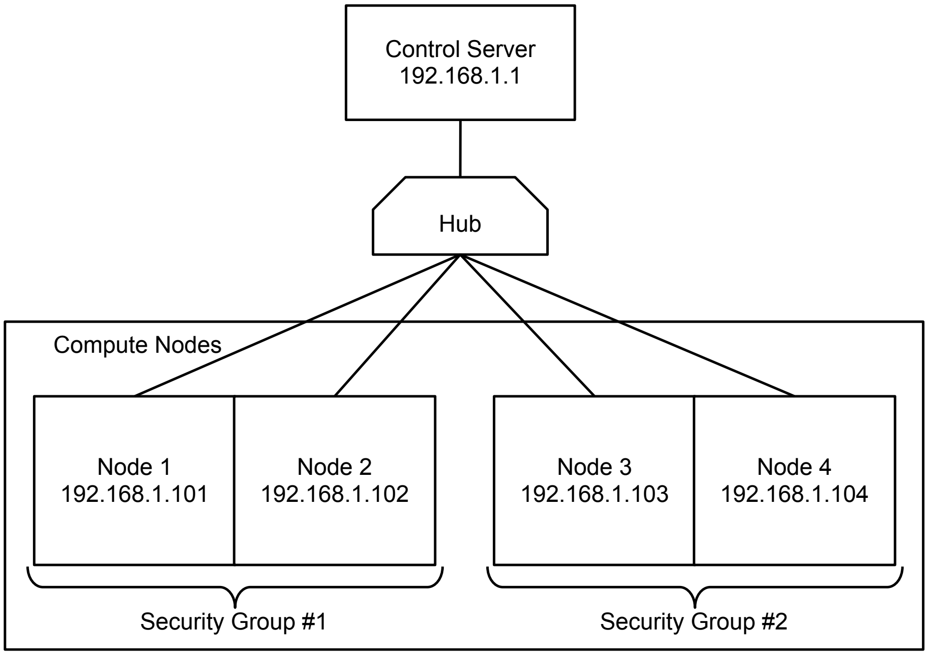

Figure 7.4: The network architecture of our demonstration test bed.

To simulate job execution each node ran a program that sent ICMP (ping) commands at stochastic intervals and speeds to the other member other member of the security group. The mechanism, started and controlled at the control server (see Figure 7.5) alternated between network access for each security group at a time interval of one second of network access per group. A Wireshark (REF PACKET SNIFFING) trace was ran at the control server and the results of which can be seen in Figure 7.6 showing a history of packets sent through the network. The black entries represent communications originating from the control server, while the red and blue entries represent packets originating from nodes within the security group. MAKE IMAGE OF RANDOMIZED PINGING AND COMPARE THE BEFORE AFTER - MENTION THIS IS EGRESS INFORMATION, FIND WAY TO SHOW INGRESS.

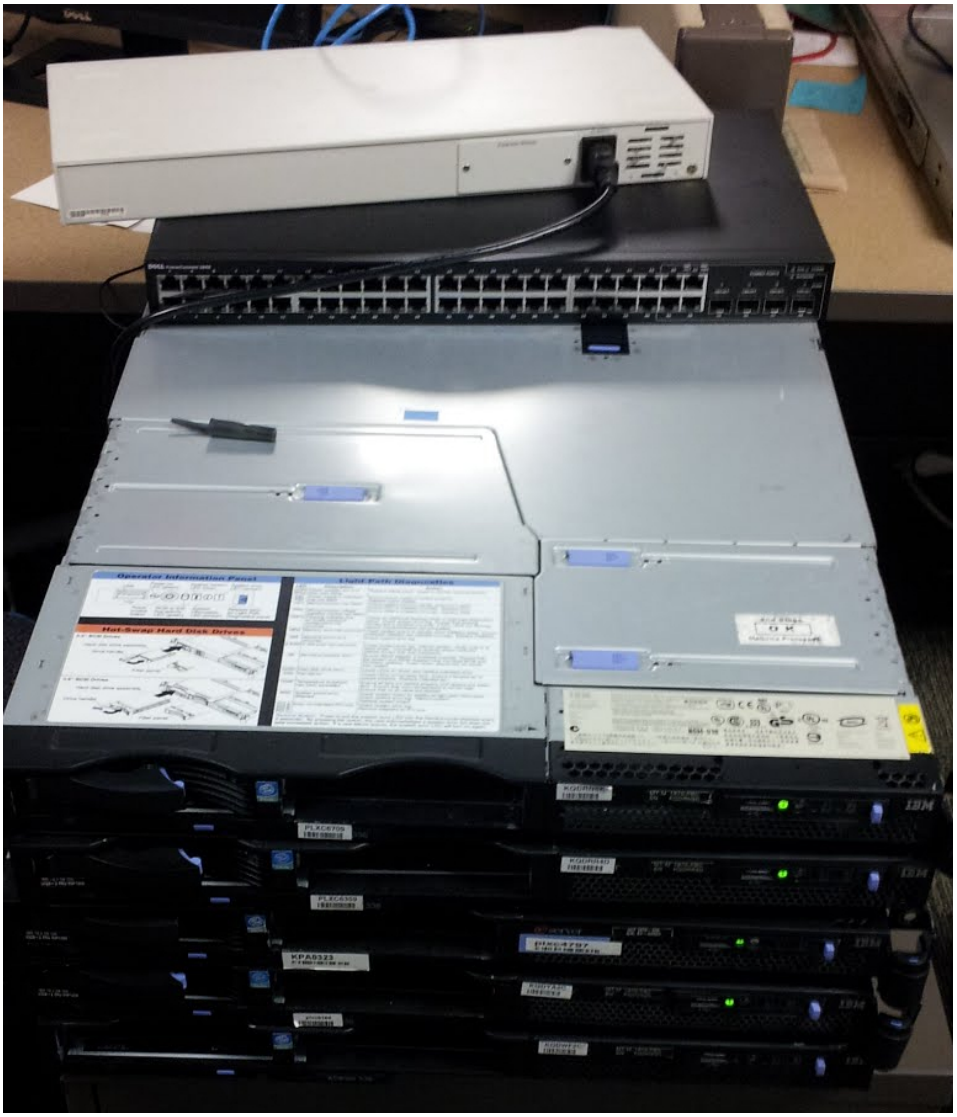

Figure 7.5: The TDM test bed located in Impact Lab at Arizona State University.

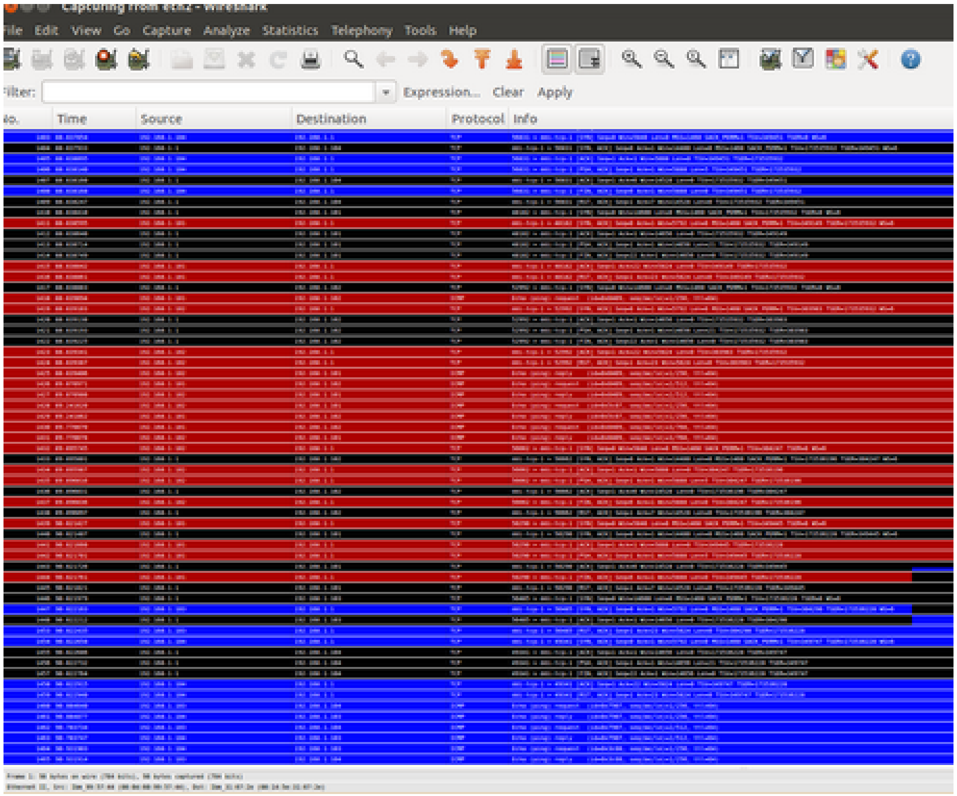
Capturing from eth2 - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply
No.
Time
Source
Destination
Protocol
Info

Figure 7.6: An example trace of the mechanism's time division property captured using the packet capturing application Wireshark. The colored records represent traffic based from compute nodes of two separate security groups - red and blue.

Security Validation

Chapter 8

Conclusion

Further Work

Implement dynamic scheduling algorithm on the window scheduler using memory usage statistics from compute nodes.

Bibliography

[1] United States National Security Administration. Inline media encryptor. `http://www.nsa.gov/ia/programs/inline_media_encryptor/index.shtml`, January 2009.

[2] V.S. Arackal, B. Arunachalam, MB Bijoy, BB Prahlada Rao, B. Kalasagar, R. Sridharan, and S. Chattopadhyay. An access mechanism for grid garuda. In *Internet Multimedia Services Architecture and Applications (IMSAA), 2009 IEEE International Conference on*, pages 1–6. IEEE, 2009.

[3] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. Investigation of leading hpc i/o performance using a scientific-application derived benchmark. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[4] M. Bozzo-Rey, M. Jeanson, M.N. Nguyen, C. Gauthier, M. Barrette, P. Vachon, K. Gaven-Venet, H.Z. Lu, S. Allen, and A. Veilleux. Design, deployment and bench of a large infiniband hpc cluster. In *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, pages 8–8. IEEE, 2006.

[5] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, Upper SaddleRiver, NJ, USA, 1999.

[6] G.N. Cohen, B. Kamenel, and C.M. Kubic. Security for integrated ip-atm/tactical-strategic networks. In *Military Communications Conference,*

1996. MILCOM '96, Conference Proceedings, IEEE, volume 2, pages 456 –460 vol.2, oct 1996.

[7] J. Dean. Designs, lessons and advice from building large distributed systems. Presented at Large-Scale Distributed Systems and Middleware (LADIS), 2009.

[8] D.B. Jackson. On-demand access to compute resources, April 7 2006. US Patent App. 11/279,007.

[9] R.H. Katz, G.A. Gibson, and D.A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842 –1858, 1989.

[10] A. Keller and A. Reinefeld. Anatomy of a resource management system for hpc clusters. *Annual Review of Scalable Computing*, 3(1):1–31, 2001.

[11] N. Leavitt. Big iron moves toward exascale computing. *Computer*, 45(11):14–17, 2012.

[12] B. Madai and R. Al-Shaikh. Performance modeling and mpi evaluation using westmere-based infiniband hpc cluster. In *Computer Modeling and Simulation (EMS), 2010 Fourth UKSim European Symposium on*, pages 363–368. IEEE, 2010.

[13] University of Chicago. Globus toolkit homepage. `http://www.globus.org/toolkit/`, 2013.

[14] University of Oslo. Nordugrid homepage. `http://www.nordugrid.org/`, 2013.

[15] Thomas Sandholm, Peter Gardfjäll, Erik Elmroth, Lennart Johnsson, and Olle Mulmo. An ogsa-based accounting system for allocation enforcement across hpc centers. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 279–288. ACM, 2004.

[16] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 42. IEEE Press, 2008.

[17] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. In *Cray Users Group Meeting (CUG)*, pages 7–10, 2007.