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 2013 by the
Graduate Supervisory Committee:

Sandeep Gupta, Chair
George Ball
Georgios Varsamopoulos

ARIZONA STATE UNIVERSITY

May 2013

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 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 (Time Division Multiple Access, or TDMA) 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 and thank Dr. Gupta for taking a chance and inviting me to work in Impact Lab, Dr. Varsamopoulos for his immense technical guidance (especially regarding Linux), and Dr. Ball for his earnest support of this thesis and its goals. I am intellectually and personally indebted to the members of Impact Lab for their help with the myriad of tasks that arose during my time with the lab. Special mention must go to Dr. Tridib Mukherjee, Dr. Ayan Bannerjee, and (soon to be Dr.) Zahra Abbasi for helping me with their seemingly boundless knowledge of the research we did, as well as Robin Gilbert for being a true friend and great colleague. I thank Raytheon for their capital support of our research. Finally, I thank my parents and brother for their love and support.

To my wife, Sara, for her unwavering support

Contents

CHAPTER Page

CHAPTER Page

List of Figures

Figure Page

Chapter 1

Introduction

High Performance Computing (HPC) systems consist of numerous individual computing systems networked and administrated together such that they can be used as a single system. Examples of these systems from popular culture include custom made models such as the Cray I (historically one of the first systems deemed HPC) and the modern IBM Blue Gene [24]. More common examples are simple Computer Clusters such as Beowulf clusters in which Commercial Off The Shelf (COTS) equipment is utilized [9]. These latter systems are simple enough that they are frequently implemented by single users within hobbyists' homes [8]. Figures 1.1 , 1.2, and 1.3 show examples of these systems.

1.1 Security Concerns

Application developers for these systems span a broad spectrum, ranging from undergraduate students learning concurrent programming to defense contractors executing classified simulations. Key characteristics of this spectrum are shown in Figure 1.4. Moving towards the most demanding end of the spectrum, security concerns among application-side stakeholders increase substantially and additional

Figure 1.1: A beowulf cluster. [35]

Figure 1.2: The Cray I. [32]

Figure 1.3: IBM's Blue Gene. [23]

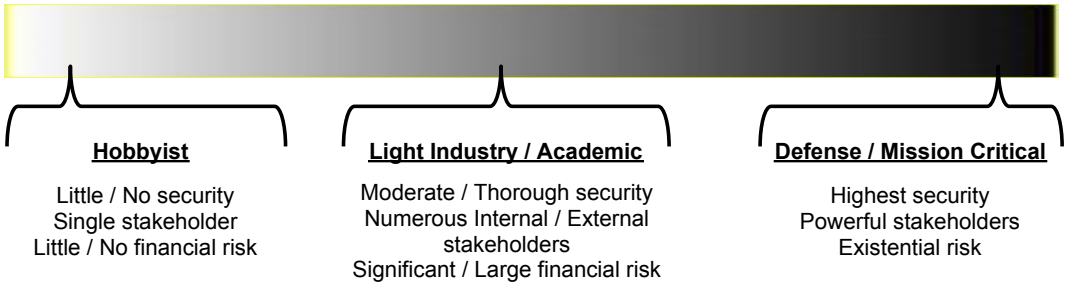

Figure 1.4: The spectrum of environmental security requirements based on uses and stakeholders.

methods are employed to enforce information security. At some point along this spectrum stakeholders demand physical separation of the system from other users during operation to satisfy security concerns. The reasons behind this can be numerous, but stem from two major goals: simplicity of implementation and verification; and risk aversion/management. In the defense industry particularly, information security breaches can threaten the existence of entire programs due to certification revocation from agencies such as the Department of Defense (DoD) [29], the DoD's Defense Security Service (DSS) agency [30], and the National Institute of Standards and Technology (NIST). Such risk reasonably implies physical separation of systems under operation from other users as well as numerous other physical security requirements.

It is undeniable that physical separation provides a level of information security that is difficult to replicate through the use of software, however the financial costs are significant - devoting entire HPC systems to a single project, or running jobs sequentially with downtime for data cleansing between [30].

1.2 Time Division Multiple Access Scheme

This paper presents a Time Division Multiple Access (TDMA) scheme of network access as a viable alternative to physical separation. By modulating network access

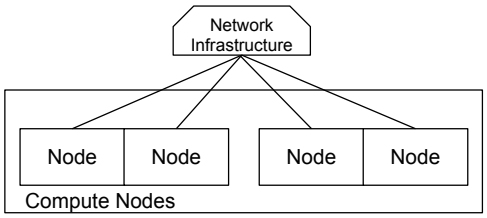

Figure 1.5: Unmodified computing nodes.

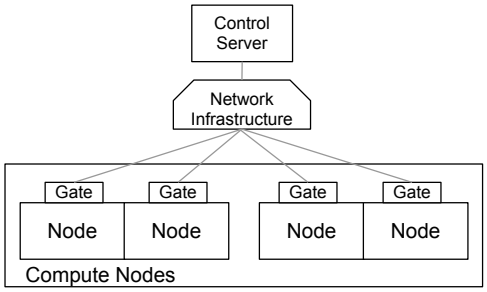

Figure 1.6: TDMA overlaid onto Figure 1.5

between application security groups we can provide an intuitive security mechanism, verifiable in real-time, capable of mimicking aspects of the security provided by physical separation. 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. Providing a mechanism for operating multiple user applications on a single HPC system securely with the support of stakeholders can provide substantial monetary savings and efficiency gains over physical separation.

The scheme works by inserting gates to network access at each computing node (computing devices devoted to executing user applications) within the system. These gates are modulated open and closed by a central administrative program, denoted as the *control server*, using knowledge of users and the data they own, denoted as *security groups*. The key operation of the scheme is the control server modulating network access of individual computing nodes such that systems executing application(s) from one security group never have access at the same time as systems containing data from a different security group.

Chapter 6 provides details on the implementation of this scheme, and Chapter 5: Section 5.3 formally defines the scheme's operations.

Chapter 2

Related Work

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

2.1 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. [34] make an attempt at rectifying this larger problem by creating a framework that 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 [28] and NorduGrid [31]).

Allcock et al. [3] developed a high-speed data transport protocol, GridFTP, as well as a corresponding administrative service providing for the creation, registration, and secure transportation of scientific computing datasets. For efficient execution, HPC applications must carefully consider characteristics of the data set under operation such as file size statistics, data creation/consumption rates, and logical distribution [10]. GridFTP implements management of these characteristics while maintaining customizable security using the authentication mechanisms defined in RFC 2228 "FTP Security Extensions" [15]. This solution, while useful in most scientific computing setting, still allows for application data, albeit encrypted, to be visible over the network to other user applications. This visibility renders it insufficient for the requirements of customers with the most stringent data security

needs.

2.2 Time Division Multiple Access

Mages and Feng [27] patented a similar control scheme of computing resources via a centralized controller over the network. Their scheme, however, specifies only local media resources of the node as under the control of the central administrative node. Furthermore, their patent is intended for a much wider distributed use as digital rights management and security in consumer media devices, rather than our work on security in HPC environments.

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.

3.1 Assumptions on the Computing Environment

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

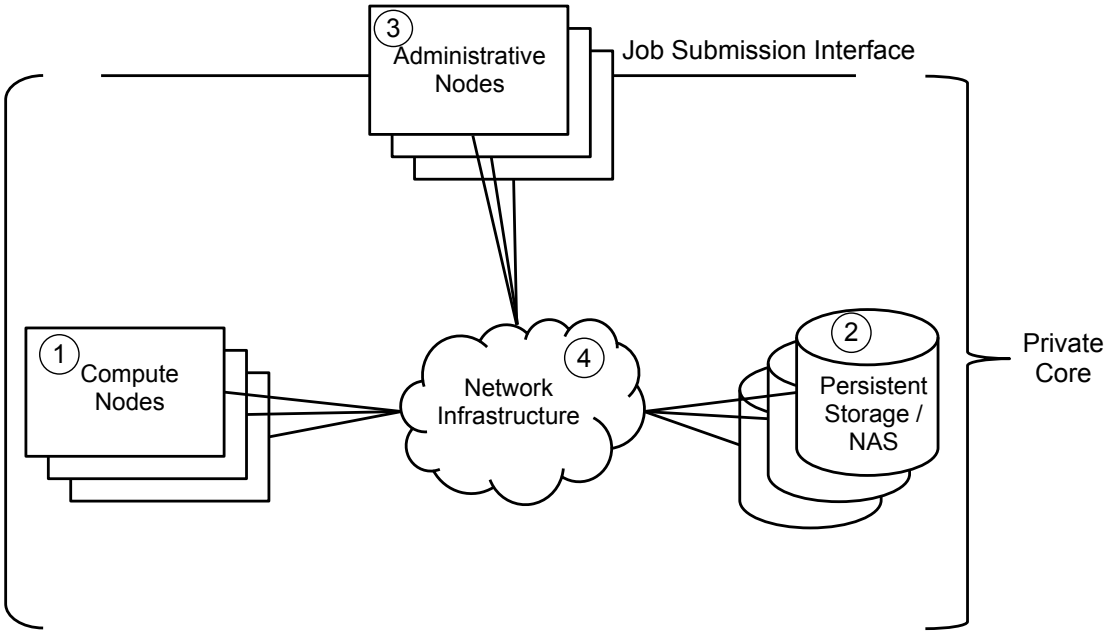

Figure 3.1: An abstract HPC environment.

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

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 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 as Network-Attached Storage (NAS) devices are capable of storing large quantities of user application data, and are usually of much higher capacity than the compute nodes. These devices commonly use RAID (redundant array of inexpensive disks) technology [18] 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 [11]. 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 [19], common examples include IBM's Tivoli Workload Scheduler and the MOAB Cluster Suite by Adaptive Computing [4][16].

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 [7][26].

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 [37]. 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

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 [12].

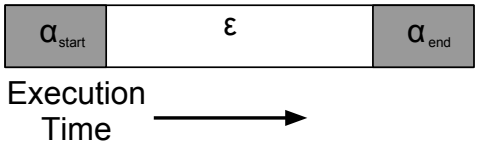

Figure 3.3: The assumed model of application execution in an HPC environment. α_{start} and α_{end} are periods where execution is I/O bound, and ε is the prominent period where execution is CPU bound. This structure adheres to research showing batched I/O minimizes the I/O cost in terms of time.

than it is from main memory [12]. This overhead increases to almost a factor of 100 if that data is initially read from disk then sent over the network [12].

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 [36]. 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 [37]. The batching of I/O, especially the most costly forms (disk and network) is therefore considered best practice when possible [6]. 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.

		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

3.2 Security Challenge

The security fear of users with extremely sensitive data is that a different user could, through chance or intention, acquire or manipulate their data. The four basic resources described above represent the resources across which data may be exposed. Table 3.2 organizes the aforementioned ways in which these resources are secured [9].

So far we have described ways in which three of the four shared resources are secured by technology that is either certified by national defense agencies (as in the case of persistent storage) or intuitive and easily verifiable (as in the case of compute node board separation and administrative POSIX permissions). This leaves the shared networking resources as a point of data sharing.

3.3 INSUFFICIENT SOLUTIONS

There exist current solutions for the strict problem of preventing plaintext data sharing, but these solutions lack certain qualities that make them sufficient for the narrow solution we're seeking when trying to assure users assuming significant risk. These existing solutions fail through their lack of thoroughness, lack of intuitiveness, inability to be simply verified, and even through established security flaws.

Encryption

Socket to socket encryption is a common solution to preventing data theft over a network, though somewhat out of place in an HPC environment due to their overhead. This solution falls short in its inability to prevent data sharing. Though the data is encrypted, the value of the plaintext (unencrypted data) is commonly so high that copying, storing and later decrypting the network traffic a risk to be prevented.

Virtual Local Area Networks (VLANs)

VLANs, standardized in IEEE 802.1Q [25], are a common solution to preventing data sharing specifically within large computing infrastructures such as data centers and mainframes, though sometimes used in HPC environments. VLANs work by tagging traffic on network switches and routers according to configurable tables of LAN membership matched with physical interface, with the intention of mimicking the configurability and security of LANs. Within the VLAN specification there lie two inherent flaw with VLANs to verifiability that meets our envisioned users' needs, as well as a few unresolved security flaws.

VLAN solutions are difficult to verify in two ways: *a*) the logic of VLAN technology is hidden within firmware which is expensive to analyze. As VLAN technology improves and becomes more complicated, this problem will only increase in future versions of the systems [14][21]. *b*) VLAN hardware is commonly man-

ufactured by international firms that may have pressure from outside governments to include secret backdoors in the firmware, further exacerbating the previous verification difficulty. Examples of this uncertainty can be seen in a special report by the U.S. House of Representatives from October 8th, 2012 [33].

Furthermore, due to backwards compatibility standards outlined in IEEE 802.1Q, the tagging mechanism of VLANs can be abused via a ”double-encapsulated 802.1Q / Nested VLAN” attack which works by placing two VLAN tags on a packet. By doing so, during certain situations it is possible for packets to ”escape” their VLAN designation and convey messages to hosts outside their configured VLAN [38].

Chapter 4

Design Goals

Chapter 3's discussion on security flaws within HPC environments shows that there exists an untouched niche for a software solution that replicates the security of physical user separation. Here we describe the requirements and goals in designing our solution to this problem.

4.1 A More Thorough and Intuitive Network Security

The gap between the security mechanisms discussed in Chapter 3: Section 3.3 and the current approach of physical user separation is large. From manufacturing and political policy problems [33] to simple security flaws [38], current solutions are insufficient. For users in this domain to accept a software approach to network security in HPC environments, the proposed solution must be thorough, simple, intuitive and easily verifiable. The thoroughness of physical user separation is inherent in that it operates at the very lowest level of network operations, the physical layer. The closer our mechanism approaches this layer, the more thorough it will be considered.

4.2 Dynamic Control

Within HPC environments the type, number, and scale of jobs assigned to any of the systems can be widely varied. To handle this, the solution must be capable of receiving and modifying policy at the start of each new job. Furthermore, to manage performance tradeoffs a fine-grained control of the mechanism at higher resolution than job submission rate is desired.

4.3 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 solution to be network fabric agnostic if it is conceptually capable of being implemented in either Ethernet or InfiniBand networks.

4.4 User Application Transparent

A fundamental requirement of the solution 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 solution, given that it can also do so on a system not using our solution.

Chapter 5

Time Division Multiple Access of Network Access

The TDMA scheme introduced in Chapter 1: Section 1.2 mimics the security of physical separation by enforcing temporal separation of network access (and denial) according to *security groups*. This network access denial must be enforced at the operating system level on each compute node and controlled by the *control server* which separates them into time windows. By denying disparate *security groups* the ability to send or receive on the network during the same time window, both passive (packet sniffing) and active (packet insertion) data sharing are prevented.

Figure 5.2 shows an intuitive graphical example of network access switching between *security groups* as the system passes through time windows.

5.1 Centralized Scheduling

The size and order of time windows are controlled by a scheduling policy within the *control server*. Figure 5.1 give a simple example of a round-robin policy on two *security groups*, although more complex scheduling can be created to optimize for throughput or enforce prioritization. Because the *control server* modulates the opening and closing of network access at each time window, very fine grained control of scheduling policy can be achieved. It is possible to switch from round-robin

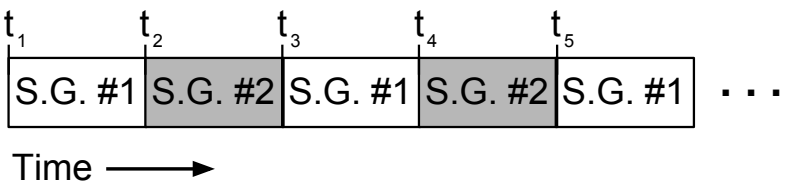

Figure 5.1: A simple round-robin time window policy for two *security groups* (S.G.#1 and #2).

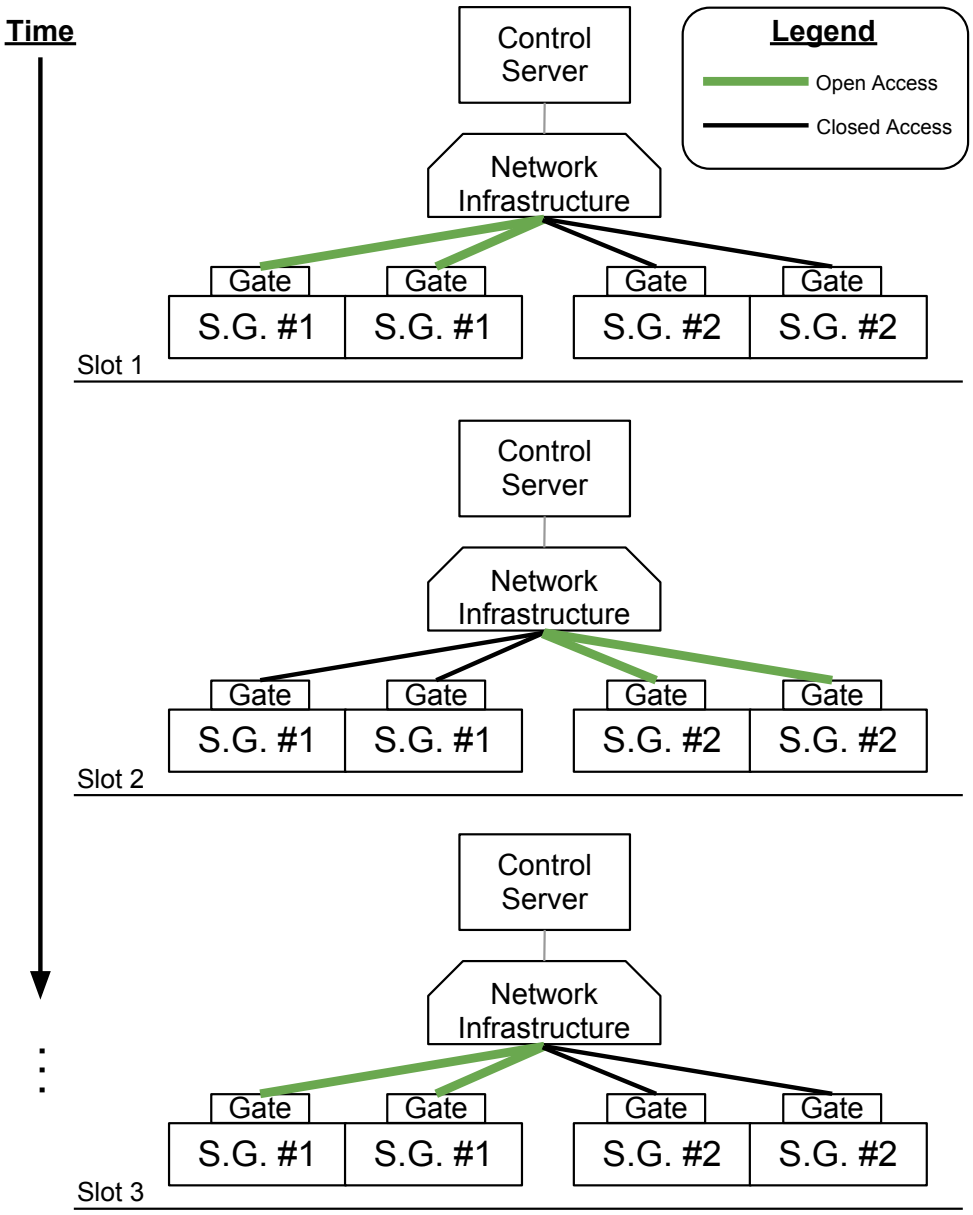

Figure 5.2: A simple example of network access switching between two *security groups* (S.G.#1 and #2).

scheduling during times of low network traffic to modulated window size during heavy usage by particular users, or according to whatever quality of service scheme best fits the applications. A number of heuristics are useful to consider for the creation of a dynamic priority scheduling algorithm: queue memory usage of compute nodes, number of TCP timeouts (see Section 5.2), and externally imposed priorities.

To perform network access modulation the *control server* must have knowledge of resource assignment within the system. This can be managed by a shared data structure between the TDMA *control server* and the system resource manager, such as a one-to-one mapping from IP address to user application (i.e., *security group*) or by providing the *control server* with API access to the resource manager.

Dynamic Job Scheduling

As jobs are added and finish according to the resource manager, TDMA manages this dynamism by:

1. preparing nodes to run user application (performed by resource manager) and registering these nodes and their network information with control server, and

2. the control server recognizing and adding/removing this security group to TDMA access/denial schedule.

Job Failure

Additional considerations must be made for the case when a scheduled job fails in some way. There are two major ways in which jobs can fail on the system, and each corresponds to a specific response from TDMA:

1. the job fails *gracefully*. In this case, some aspect of the user application fails but does so in a way that does not halt the *compute node* it is assigned. Under

this scenario, TDMA operates normally and the resource manager and job scheduler are left to reallocate those resources and reschedule the job.

2. the job fails and halts an assigned *compute node*. In this case of a *compute node* halting error, TDMA on that node would be unable to function. This causes the TDMA system to enter an insecure state. Resumption from this insecure state can only happen by cancelling all jobs on the system and restarting again from initialization data. This is how TDMA operates in a fail safe manner. In the case that TDMA enters an insecure state (deviation from the operations defined in 5.3) it must be assumed that the erroneous *security group* could have either received network traffic from an outside *security group* or transmitted network traffic to an outside *security group*.

5.2 Constraints on TDMA

For TDMA to function in an HPC environment, a number of practical constraints must be enforced. These constraints revolve around ensuring that user applications have ample access to network resources. This implies that they do not experience disruptively long periods of network denial.

TCP Timeout

The main consideration is of TCP timeouts, since TCP is the predominant data transfer protocol in HPC [2]. Although application programmers can specify TCP timeout values [13], a practical goal is to prevent delaying communication beyond the default TCP timeout. Default values for TCP timeout vary across operating systems with some Linux distributions being as high as 20 seconds, and Windows operating systems being as low as 5 seconds.

Given a TCP timeout value for the operating systems in use within an HPC environment:

1. Time Window Size - any individual time window cannot be longer than the default TCP timeout value. If any individual time window is that large, then all other security groups are guaranteed to have their TCP connections reset waiting for their time window.
2. "Fair" Window Scheduling - time windows must be scheduled in such a manner that no individual security group experiences a period of network denial that lasts longer than the default TCP timeout. This constraint implies a periodicity to time window scheduling by the control server.

Window Size

Given a TCP timeout value, $timeout$, for the operating systems in use within an HPC environment, we define a relationship between window size, window transition overhead (the time it takes to transition from one window to another), and the timeout value.

Suppose SG is a set of *security group*s, $sg \in SG$, on the system, W_{sg} is the size of an individual time window allotted to sg, and O_{sg} is the time it takes to enter and exit that time window (overhead). For any individual time window W assigned to a *security group* $current\_sg$, then the following must be true:

$$\{W_{sg} + O_{sg} : \forall sg \in (SG/current\_sg)\} < timeout$$

5.3 Formal Definition

Before discussing our implementation of the solution, we first define an abstract definition that describes how the mechanism works beyond any specific implementation and in a more formal way than the beginning of this chapter. The best way to describe this mechanism is through a language that represents the mechanism in operation. This language is defined formally by stating the grammar that generates it.

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:

$$
\begin{aligned}
&G^1 = (V^1, \Sigma^1, R^1, \mathscr{A}), \text{ where} \\
&V^1 = \{\mathscr{A}, W\} && \text{non-terminal symbols} \\
&\Sigma^1 = \{\emptyset\} && \text{terminal symbols} \\
&R^1 = \{ \quad \mathscr{A} \rightarrow \varepsilon, && \text{rules of production} \\
&\qquad\quad \mathscr{A} \rightarrow W\mathscr{A} | W\}
\end{aligned}
$$

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:

$$G^2 = (V^2, \Sigma^2, R^2, \emptyset), \text{ where}$$
$$V^2 = \{W\}$$
$$\Sigma^2 = \{[o_{s,i}, a_{s,i}] : \forall i \in \forall s \in S\}$$
$$R^2 = \{[W \rightarrow \lambda] : \forall \lambda \in \Lambda_s : \forall s \in S\}$$

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:

$G = (V, \Sigma, R, \mathscr{A})$, 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 Chapter 4.

6.1 Overview

The tool is composed of four major components: the control server, ingress controller, egress controller, and the state controller. The control server 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 HPC environment, with a copy on each compute node that is designated to execute user applications.

As jobs are scheduled on the system, the control server must be informed of the Internet Protocol, or IP, addresses assigned to that application and the assigned security group. As jobs are run on compute nodes throughout the system the control server communicates with the state controller on each node to designate time windows. This communication is sent via a standard TCP communications socket. 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 not sourced according to a "whitelist" firewall are ignored and deleted. The control server is tasked with alternating time window authorization between security groups.

The following subsections describe each components operations in further detail.

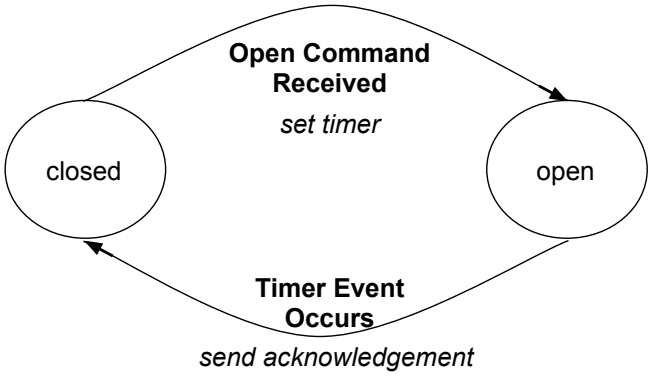

Figure 6.1: State diagram of a compute node running the state controller.

6.2 State Controller

The state controller has four major tasks:

1. Securely send and receive communication with the control server including receiving open network access commands and sending acknowledgements that the ingress and egress controllers have been transitioned to a state of network denial (closed).
2. Transit both the ingress and egress controllers between states of network access and denial.
3. Track time for which it must issue state transitions and communications with the control server.
4. Collect and store performance data on the egress queue's memory usage.

A state machine of the state controller is shown in Figure 6.1. For a detailed algorithm of the state controller see Algorithm 2 in Appendix A.

6.3 Ingress and Egress Controllers

The ingress and egress controllers are firewall rules created and manipulated by the state controller. The rules are implemented using *iptables* and *NetFilter*, the packet filtering firewall technology that has come standard with the Linux kernel since version 2.4 [39]. These controllers occur in two states, network access and network denial. When in the network access state, they are to allow network packets to flow freely to and from local applications. When in the network denial state each controller has a specific task:

1. The ingress controller must deny and drop all network packets received except those specifically allowed according to a whitelist. This whitelist is specifically issued by the control server at the beginning of execution, and dictates the IP and MAC address of administrative services within the environment from which communication must not be broken. Example of such services include Network Time Protocol, resource scheduling communication, and the control server itself must have an entry in this whitelist.

2. The egress controller must enqueue outgoing packets exempting those specifically allowed through according to the aforementioned whitelist. These packets must be stored in main memory in preparation for transmission once network access is again granted.

When these controllers transition out of the network denial state, a certain protocol must be followed in order to avoid the system dropping packets. All nodes within the security group to which a state of network access is now being issued must transition their ingress controllers before any single node within the group transitions their egress controllers. The reason simply being that upon transitioning

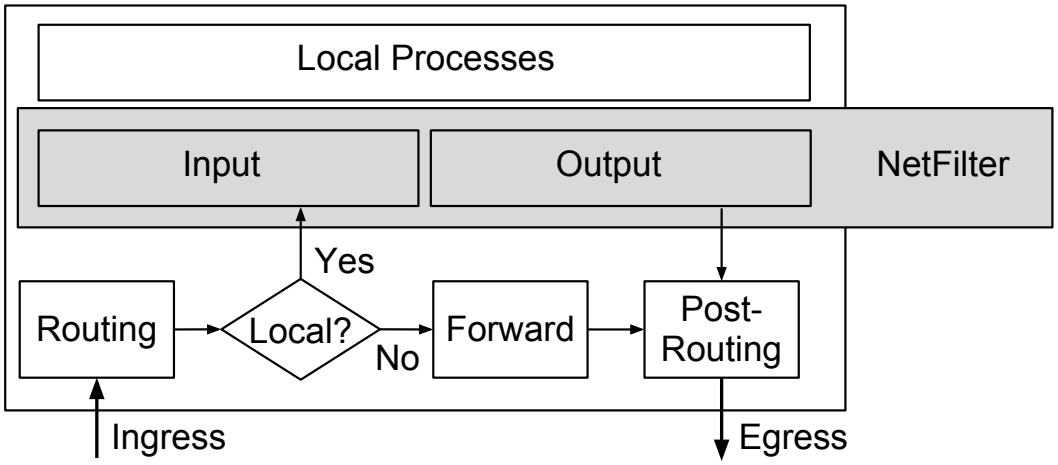

Figure 6.2: Data flow architecture of *iptables*, the packet filtering firewall with *NetFilter* located within the Linux kernel. The input and output ”chains” within *NetFilter* provide an interface for administrators to control and filter packets sent into user space.

egress controllers into the network access state, they will begin transmitting (in a first-come first-served order) the packets sitting in their queue. If nodes within that security group have yet to transition their ingress controller, packet arrivals sourced from within their own security group would be dropped as if they were sourced outside the security group.

iptables and NetFilter

Figure 6.2 provides a concise overview of the components within *iptables*. With *iptables* enabled, all packets sent and received from the operating system are passed through a series of checks. At each check, *iptables* is capable of transforming the data within that packet according two edicts *a*) routing tables and *b*) *NetFilter* chains before data is delivered from the network to local processes and before local processes can deliver packets to the network medium.

It is within *NetFilter* that the logic of the ingress and (most of the) egress controllers are located. Within *NetFilter*'s input and output processes administrators

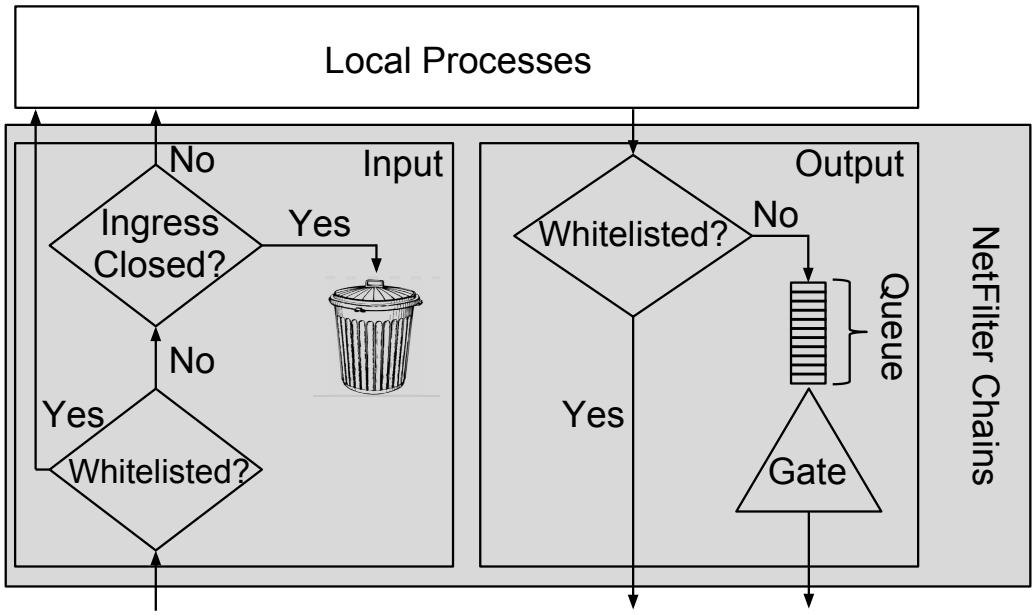

Figure 6.3: A detailed look at the logic within the *NetFilter* chains that makeup the Ingress and Egress controllers on compute nodes.

can insert "chains" of logic to redirect the flow or, or manipulate the contents of packets. Figure 6.3 provides a visual summary of the ingress and egress controller logic located within *NetFilter* as chains.

The queue structure in Figure 6.3's output section is controlled by the egress controller registering a special subroutine to process the packets when in the network access state. This subroutine is registered using *NetFilter*'s $libnetfilter\_queue$ library [40].

6.4 Control Server

The control server has three major tasks:

1. Determine system network access states for the next time window.
2. Validate the closure of the previous time window.
3. Communicate the next time window states to compute nodes.

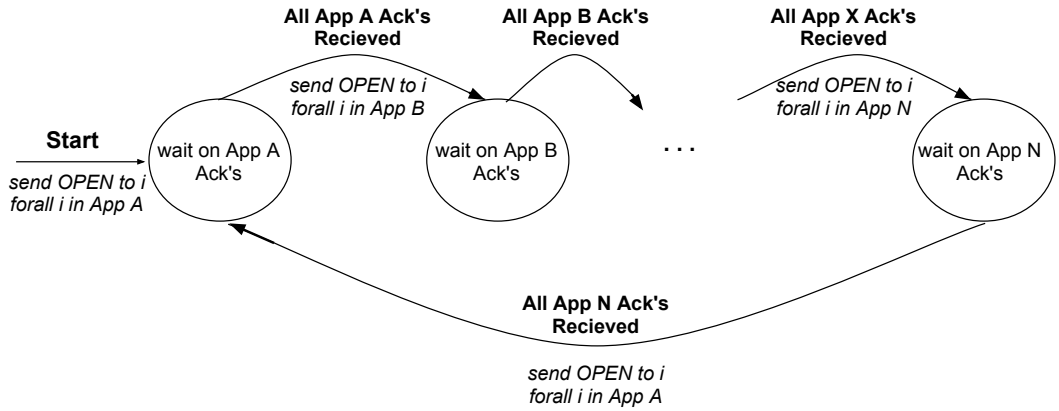

Figure 6.4: State diagram of an example window controller.

To determine the network access states for the next time window the scheduler must run a scheduling algorithm on a few historical inputs. A state machine of the control server is shown in Figure 6.4. The control server transitions between states of open network access for individual security groups. During these states the control server waits to receive acknowledgement from the state controllers of that security group showing network access denial has been enforced for all nodes of the group.

For a detailed algorithm of the control server see Algorithm 1 in Appendix A.

Chapter 7

Performance

To analyze the performance of TDMA a series of network performance tests were run using Netperf, an open source network performance tool that is installed by default on most Linux distributions [17] and the common UNIX *ping* command. These tests were run on a TDMA testbed run in Impact Lab at Arizona State University. The results showed a small, but significant overhead in terms of reduced throughput when using TDMA to separate network traffic, as well as a significant impact to round trip time (RTT). The results also clearly show the temporal division of network traffic between security groups.

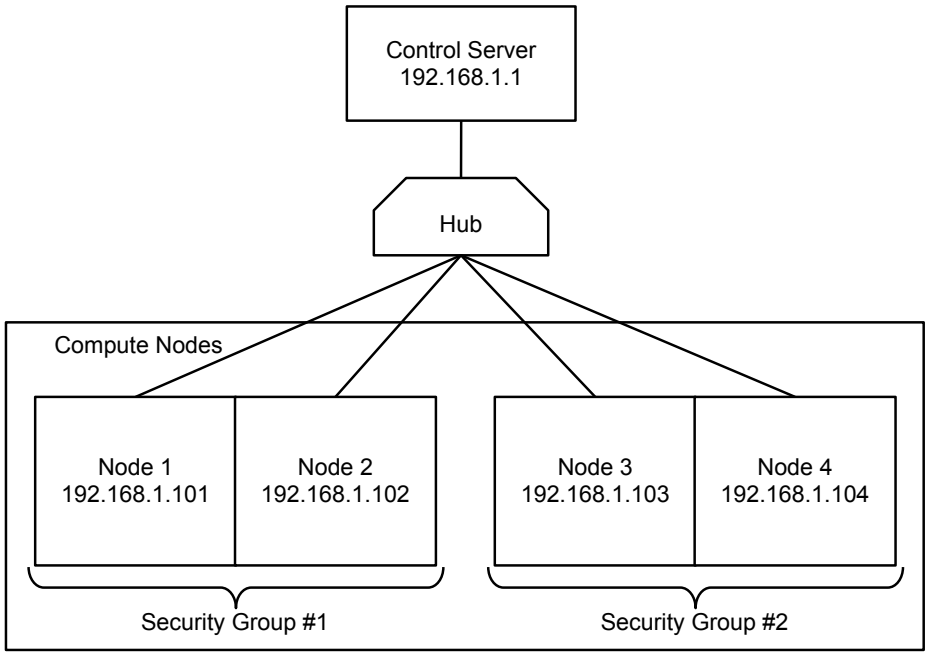

Figure 7.1: The network architecture of TDMA testbed.

7.1 TDMA TESTBED

The testbed consisted of five servers (four compute nodes and one control server) each connected to a single hub according to the layout shown in Figure 7.1. For reproduction purposes, the TDMA testbed is described in more detail in Appendix B.

7.2 PERFORMANCE TESTS

Using the TDMA testbed we ran three series of performance tests, one to display the temporal division of security group packets, another to measure TDMA's overhead on system throughput, and a final test of impact on RTT.

Netperf

Netperf is the de facto standard for network throughput measurement [20][5][22][41] from which we utilized two major TCP communication test types:

1. TCP Stream Performance (TCP_STRM) - a simple test of bulk data transfer in a client to server manner. This tests the combined throughput of both the network and the networking interface of the client and server.

2. TCP Request Response (TCP_RR) - a test of TCP request and response rates between client and server. In addition to measuring the network's throughput, this test measures the throughput of TCP transaction handling rather than just buffered data transfer, the main goal of the TCP Stream Performance test.

Both of these tests are customizable by the packet payload size, send buffer, and receive buffer.

Temporal Division

The main goal of this test was to display the temporal division of security group packets. As such, variations on payload size and send/receive buffers makes little

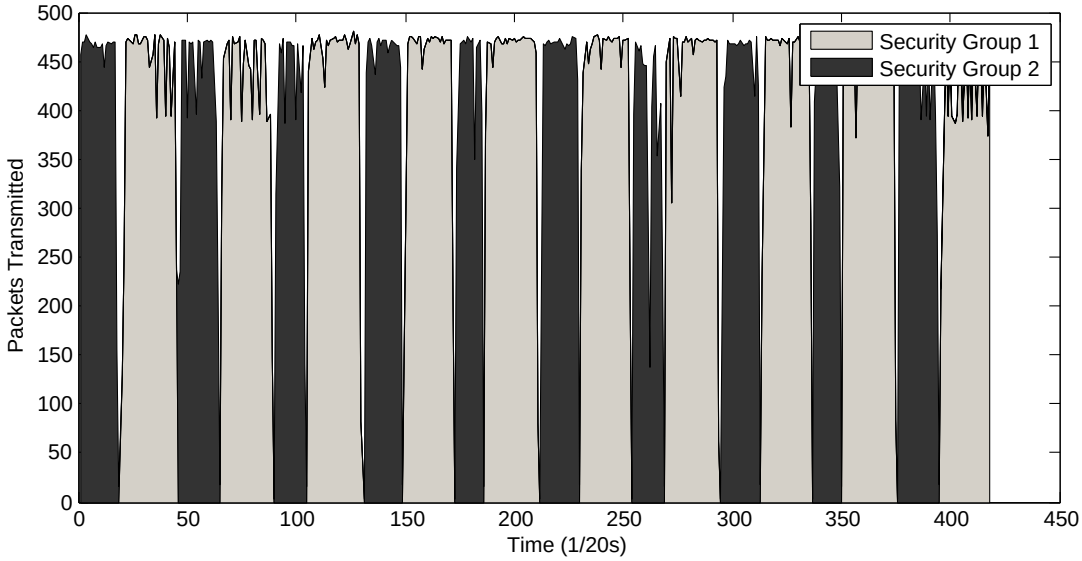

Figure 7.2: A trace of network traffic under performance testing while TDMA controls access.

difference. Figure 7.2 shows an historical trace of packets transferred by two security groups, both running TCP_ STRM tests under TDMA. TDMA was configured to have time windows of 1 second, the divisions of which are clearly visible.

Overhead

This series of tests was used to measure TDMA's overhead when compared to the same tests run without TDMA. In it, both TCP_ RR and TCP_ STRM tests were ran on variations of payload size for 5 minutes each, with two security groups. Payload sizes (in bytes) were varied by powers of two: 1024, 2048, 4096, 8192, and 16384. The resulting throughputs were averaged through payload sizes to create throughput values for each test with and without TDMA enforcement. Figure 7.3 shows the results of this test series. It is visible that, although significant, the overhead of TDMA is not dramatic. The overhead on TCP_ RR was an averaged 14% reduction in throughput, while TCP_ STRM tests showed a smaller 11% reduction in throughput.

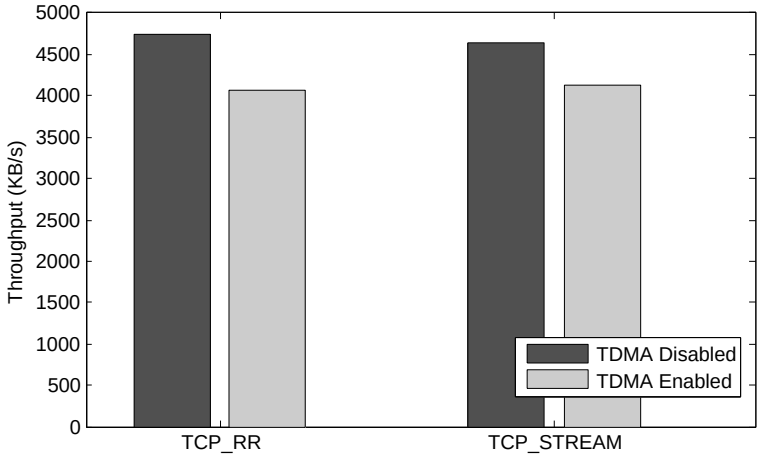

Figure 7.3: The impact of TDMA on TCP performance under two different 'netperf' tests.

ping

ping is the de facto network connectivity test. Using Internet Control Message Protocol (ICMP) messages, *ping* sends a message from a local network node to a remote node, and the remote node should respond back if it receives this message. If a response is received from a remote node then network connections are possible. Reasons for *ping* to fail are either a lack of connectivity or firewall rules on either machine that deny ICMP communication (not uncommon in production settings for security reasons). For testing purposes, ping also includes the RTT of the message, i.e. the time it took for the local node to receive a response.

TDMA's effect on RTT

To test TDMA's effect on RTT, two tests were performed: the test bed run with *ping*s issued every .2 seconds for 5 minutes without TDMA , and another test using *ping*s issued every .2 seconds for 5 minutes with TDMA enabled. Without TDMA, these *ping*s have a mean RTT of under $1ms$. The RTT of *ping*s with TDMA enabled are shown in Figure 7.4.

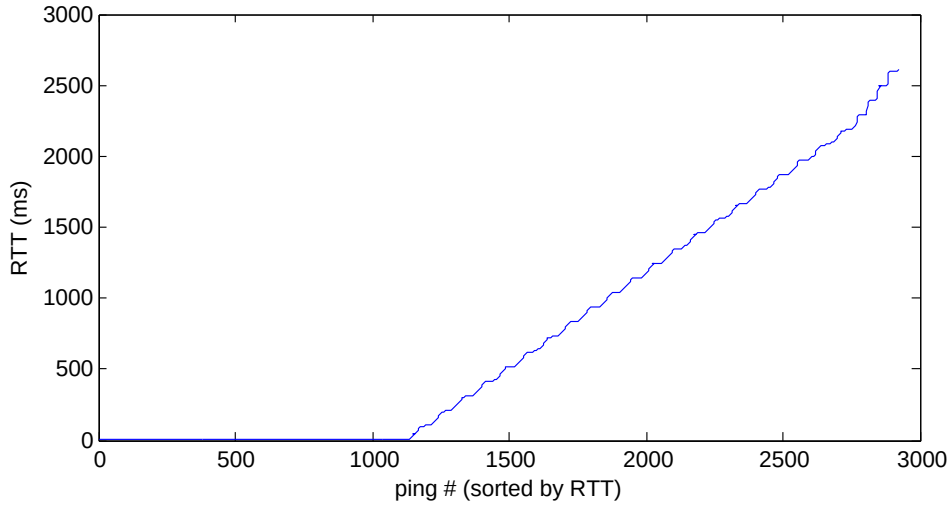

Figure 7.4: RTT of *ping* under TDMA.

Figure 7.4 shows the RTT of *ping*s under TDMA sorted according to RTT. The results can be divided into two major sections, *ping*s with RTT comparable to operation without TDMA (the low RTT on the left portion of the graph) and RTT affected by TDMA (the right side of the graph).

The linear growth (see Figure 7.5) of RTT under TDMA can be understood by recognizing how the *ping* messages interact with TDMA. As the *ping* messages are issued at a constant interval, during periods of network denial these messages are being queued in order. Messages enqueued at the beginning of a network denial period wait longer in the queue to reach the network than messages enqueued just before network access is enabled. Due to the steady period of message generation (every .2 seconds), there is a steady linear increase in RTT for these messages.

Furthermore, when, during a period of network access, these messages are received by the remote node, there is a chance that the remote node will not have processed the message to respond in time before another period of network denial is enforced. This can drive the RTT up to the time of two full time windows, in this case 2 seconds (1 second per time window).

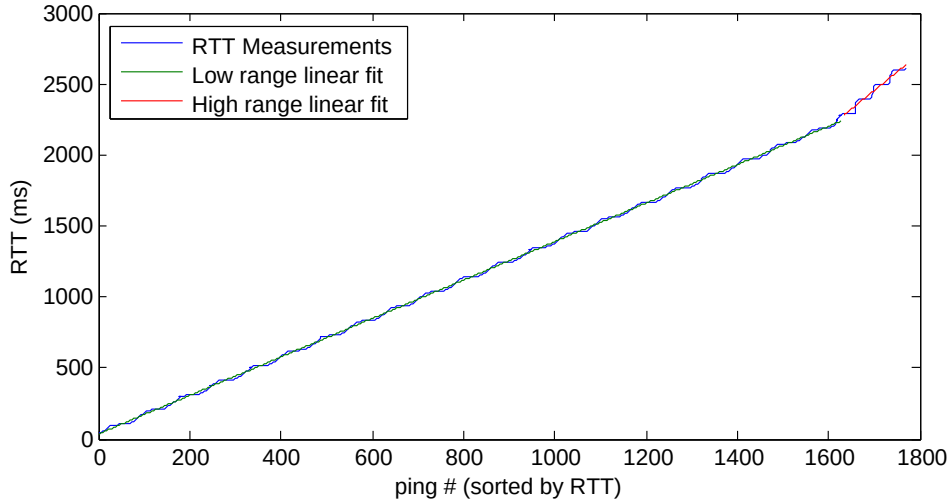

Figure 7.5: Linear regression fits of two high RTT sections within RTT results under TDMA.

Chapter 8

Conclusion

The problem of preventing data sharing among users in an HPC environment is a difficult one. Further, creating a solution that is intuitive and secure enough to convince highly sceptical stakeholders with great levels of risk of its security narrows available options. Solutions that exist in similar domains are presented in Section 3.3, and shown to be insufficient for security reasons or difficult to verify.

This thesis presents a method of solving this niche problem that is thorough, intuitive, and verifiable in real time using simple packet sniffing. TDMA was inspired by the desire to mimic the physical separation of users that is so convincing to the stakeholder we are trying to persuade. This inspiration manifested in a solution that performs temporal separation of users in lieu of physical separation.

TDMA's operations are then formalized in Section 5.3 as a grammar, providing an unambiguous description of operations. Out implementation of TDMA on an Ethernet network is then described and analyzed in Chapters 6 and 7. TDMA's effects are shown to be apparent in Figure 7.2 and its overhead shown to be within reason considering the massive performance and financial benefits that come from running workloads concurrently.

8.1 Further Work

While TDMA has been implemented for a small Ethernet testbed, more work remains to be done. To gather more verbose performance measurements, TDMA needs to be enforced on standard HPC workloads, rather than simple network performance tests. While round-robin scheduling is sufficient for small testbeds, more intelligent scheduling based on application bandwidth need should be implemented

and tested. Finally, to thoroughly convince skeptical stakeholders, formal verification of implementation code should be performed to ensure the formal design in Section 5.3 matched the code that implements it.

APPENDIX A

Implemented TDMA Algorithms

Algorithm 1 Control Server opening and closing network access windows.

```
1:  function Open_Windows(Scheduler)
2:      Scheduler.initialize();
3:      while End_Command_Not_Received do
4:          Security_Group ← Scheduler.get_next_group();
5:          Security_Group.state ← STATE.OPEN;
6:          for each node ∈ Security_Group do
7:              send(node.address,
                        Security_Group.crypto_sign(COMMAND.OPEN));
8:              node.state ← STATE.OPEN;
9:          while Security_Group.state == STATE.OPEN do
10:             node_response ← block_on_receive_message();
11:             if node_response.state == STATE.CLOSED then
12:                 node.state ← STATE.CLOSED;
13:             else
14:                 throw ERROR.UNCLOSED_NODE;
15:             Security_Group.state ← STATE.CLOSED;
16:             for each node ∈ Security_Group do
17:                 if node.state == STATE.OPEN then
18:                     Security_Group.state ← STATE.OPEN;
```

Algorithm 2 State Controller Mechanism opening and closing network access to nodes by security group.

```
1:  function Node_Control_Mechanism
2:      Queue ← initialize_queue;
3:      while exit_command_not_received do
4:          state ← Close_Network_Access(Queue);
5:          while open_command_not_received do
6:              message ← block_on_receive_message();
7:          state ← Open_Network_Access(Queue);
8:          sleep(message.time);
9:          state ← Close_Network_Access(Queue);
10:         send_acknowledgement(state);

11: function Close_Network_Access(Queue)
12:     state.egress ← Network_Egress.enqueue(Queue);
13:     state.ingress ← Network_Ingress.drop_packets();
14:     return state

15: function Open_Network_Access(Queue)
16:     state.ingress ← Network_Ingress.accept_packets();
17:     state.egress ← Queue.process_packets_to_network();
18:     return state
```

APPENDIX B

TDMA Testbed Details

Figure B.1: The TDMA test bed located in Impact Lab at Arizona State University.

To verify and test the mechanism a test bed was created out of five Dell 1955 servers seen in Figure B.1. These servers run Ubuntu server version 12.04 and had their network interfaces configured and connected according to Figure 7.1.

Bibliography

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

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann L Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[3] Bill Allcock, Joe Bester, John Bresnahan, Ann L Chervenak, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, Steven Tuecke, and Ian Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Mass Storage Systems and Technologies, 2001. MSS'01. Eighteenth IEEE Symposium on*, pages 13–13. IEEE, 2001.

[4] 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.

[5] Richard Blum. *Network Performance Open Source Toolkit: Using Netperf, tcptrace, NISTnet, and SSFNet*. Wiley, 2003.

[6] Julian Borrill, Leonid Oliker, John Shalf, and Hongzhang Shan. Investigation of leading hpc i/o performance using a scientific-application derived bench-

mark. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[7] 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.

[8] Robert G Brown. Engineering a beowulf-style compute cluster. *Duke University Physics Department*, 2004.

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

[10] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.

[11] 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.

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

2009.

[13] Lars Eggert and Fernando Gont. Tcp user timeout option. 2009.

[14] Jean-Yvesc Emery, Cédricc Vandenweghe, Brunoc Thery, et al. Security management process of at least one vlan of an ethernet network, April 13 2011. EP Patent 2,073,455.

[15] Marc Horowitz and Steve Lunt. Ftp security extensions. Technical report, RFC 2228, October, 1997.

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

[17] Rick Jones, Karen Choy, and David et al. Shield. netperf - a network performance benchmark. `http://www.netperf.org/netperf/`. Version 2.5.0.

[18] 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.

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

[20] J King. Parallel ftp performance in a high-bandwidth, high-latency wan. *SC2000, November*, 2000.

[21] Raymond Kloth. Derived vlan mapping technique, March 27 2001. US Patent 6,208,649.

[22] Samad S Kolahi, Shaneel Narayan, Du DT Nguyen, Yonathan Sunarto, and Paul Mani. The impact of wireless lan security on performance of different windows operating systems. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, pages 260–264. IEEE, 2008.

[23] Argonne National Laboratory. The ibm blue gene/p supercomputer installation at the argonne leadership computing facility. `http://en.wikipedia.org/wiki/File:IBM_Blue_Gene_P_supercomputer.jpg`, December 2007.

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

[25] IEEE Local, Metropolitan Area Network Standards Committee, et al. Virtual bridged local area networks, 1998.

[26] 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.

[27] Kenneth G Mages and Jie Feng. Method of secure server control of local media via a trigger through a network for instant local access of encrypted data on local media, April 6 1999. US Patent 5,892,825.

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

[29] Department of Defense. Dod manual 5200.01-v3 dod information security program: Protection of classified information, February 2012.

[30] Department of Defense Defense Security Service. Clearing and sanitization matrix as of: June 28, 2007, 2007.

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

[32] Clemens Pfeiffer. Ein cray-1, aufgenommen im deutschen museum, mnchen. http://en.wikipedia.org/wiki/File:Cray-1-deutsches-museum.jpg, November 2006.

[33] Mike Rogers and Dutch Ruppersberger. Investigative report on the u.s. national security issues posed by chinese telecommunications companies huawei and zte, October 2012.

[34] 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.

[35] Alex Schenck. Picture of a beowulf cluster. http://en.wikipedia.org/wiki/File:Beowulf.jpg, January 2008.

[36] 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.

[37] 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.

[38] Cisco Systems. Vlan security white paper - cisco catalyst 6500 series switches. `http://www.cisco.com/en/US/products/hw/switches/ps708/products_white_paper09186a008013159f.shtml`, 2002.

[39] The Netfilter webmaster. The netfilter.org project. `http://www.netfilter.org/`, 2010.

[40] Harald Welte. The netfilter.org libnetfilter_queue project. `http://www.netfilter.org/projects/libnetfilter_queue/index.html`, 2010.

[41] Binbin Zhang, Xiaolin Wang, Rongfeng Lai, Liang Yang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm). *Network and Parallel Computing*, pages 220–231, 2010.