Network Management Unit (NMU): A Network Interface Architecture for Job-Level Protection Domains

CVA MEMO 133

Nicholas McDonald and William J. Dally

Electrical Engineering, Stanford University, Stanford CA 94035
{nmcdonal,dally}@stanford.edu

October 17, 2013

Abstract

Today's distributed applications are built using processes and therefore are only able to rely on process-level protection domains. In this paper we argue that process-level protection domains are insufficient for distributed applications and propose job-level protection domains as a new high-level building block for distributed applications. We present a network interface architecture called the Network Management Unit (NMU) and show that it efficiently implements job-level protection domains and argue that this allows programmers to productively develop reliable high-performance distributed applications.

1 Introduction

After many decades of evolution, computer systems have converged on a process-based application model. Surrounding each process is a process protection domain which acts as a region of security to the process. Attributes provided by process protection domains include: low latency data access, physical hardware abstraction, data privacy, fault isolation, and fine-grained control of inter-process communication. The implementation of process protection domains relies heavily on the Memory Management Unit (MMU) as an efficient hardware device that provides real-time enforcement of the process-based policies set by the operating system. For applications that run on a single computer, the process-based application model is sufficient because applications exist within a single memory system.

Unlike applications that run on a single computer, distributed applications contain numerous processes working collectively on a network. Inherently, distributed applications span many memory systems and are network based rather than memory based. Application processes are often grouped by similarity into collections that comprise

a subsection of the application. We call these process collections *jobs*, although they could alternatively be called process groups or process clusters. Distributed applications are logically created from one or more jobs and jobs may even be shared between multiple applications. Even though modern distributed applications are logically created with jobs, they are physically comprised of processes. Process protection domains protect boundaries within memory systems, yet the bulk of distributed application data accesses are network based. As a result, distributed application processes are forced outside of their available protection domains for access to their own data and communication with other application modules.

By relying only on process protection domains for distributed application development, system designers are forced into making trade-offs between application performance, development productivity, and application reliability. To overcome this limitation, we present a methodology for network-based job-level protection domains as a new building block for distributed applications. We show that job protection domains can provide distributed application developers with low latency data access, physical hardware abstraction, data privacy, fault isolation, and fine-grained control of inter-job communication. Similar to process protection domains which are established by a node operating system and enforced with efficient memory-based hardware, job protection domains are established by a distributed operating system and enforced with efficient network-based hardware.

For implementation of job protection domains we propose a network interface architecture called the *Network Management Unit (NMU)*. The NMU is to jobs what the MMU is to processes. It enforces job-based policies defined by a distributed operating system in real-time. We show that NMUs provide strict enforcement of job isolation and argue that this allows programmers to productively develop reliable high-performance distributed applications.

NMU design applies to supercomputers, data centers, and cloud computing centers, or in general, to distributed systems that are physically or logically centralized. NMU design does not apply to distributed systems where the governing entity does not control every aspect of the machine (e.g. the Internet).

Contributions: This paper makes following contributions:

1. We present a methodology for network-based job protection domains as a higher level building block for distributed applications. This is the first work to present a network design that implements a job-oriented protection model.

2. We propose the Network Management Unit (NMU) as a network interface architecture that efficiently enforces job protection domains.

3. We provide a qualitative analysis of the efficacy of NMUs in terms of application performance, application reliability, and programmer productivity.

The remainder of the paper is organized as follows: In Section 2 we describe the protection domains that exist in current distributed applications. In Section 3 we propose a formal definition of jobs and job protection domains as a new high level building block for distributed applications. In Section 4 we present the Network Management Unit (NMU) and describe its architecture and operation. In Section 5 we analyze the

NMU's implementation efficacy for job protection domains. In Section 6 we discuss some of the issues related to NMU implementation. In Section 7 we present related work. In Section 8 we conclude the paper.

2 Process-based Distributed Applications

Distributed applications contain numerous processes often spanning many memory systems. For reasons of management and modularity, these processes are grouped by similarity in collections we call *jobs*. These jobs are often designed by different development teams within a company or between different companies. With varying levels of complexity, applications are comprised from one or more jobs, and jobs can even be shared by multiple applications.

In supercomputing platforms, applications often consist of a single job and are scheduled exclusively to a set of nodes. In contrast, modern data center applications consist of many jobs and jobs are often shared between applications. For example, a common architecture for web-based applications consists of three jobs: a web front-end, a cache, and back-end storage. The web front-end job is a collection of processes designed to handle HTTP requests and responses. The back-end storage job is a collection of processes used for persistent storage, typically on disk. The cache job is a collection of processes used for creating RAM-based or flash-based caching to overcome the relatively low throughput of the back-end storage job. The web front-end job uses the cache job and the back-end storage job for data storage and manages data transfer between itself and the two other jobs. Combined, these three jobs comprise one application that serves a user with an HTTP-based application. As companies create more applications, they can utilize the same back-end storage and/or cache jobs for multiple applications. They can also utilize multiple back-ends or caches that are customized to specific application workloads. This is the common area where jobs are shared between multiple applications. Companies like Facebook [4] [23] [5], Google [16] [8] [9], LinkedIn [10], Twitter [18] [19], and many more use this style of architecture. Many of the jobs running in today's data centers are instances of software packages such as relational databases (e.g. MySQL [12]), non-relational databases (e.g. HBase and BigTable [8]), in-memory key-value stores (e.g. Memcached [14]), distributed file systems (e.g. HDFS and GFS [16]), and parallel processing frameworks (e.g. MapReduce [9]).

With even more complexity than common data centers, cloud computing centers have all the complexity of standard data centers potentially multiplied by the number of clients using the facility. In efforts to reduce the overheads of hosting numerous computing clients, and in efforts to make the service more desirable to customers, many cloud computing providers have developed large scale jobs that are designed to aggregate the needs of many clients into one job or a set of jobs. Instances of this are found in Amazon's AWS [15], Microsoft's Azure [7], Google's Cloud Platform [22], and OpenStack [27]. These contain unified services for relational and non-relational databases, RAM-based storage, and distributed file systems. From an application's perspective, these are all job instances that are shared by many applications.

Jobs have become the logical building block of large scale distributed applications,

however, without job protection domains, distributed applications only have process protection domains to rely on. Distributed applications are unique in that inter-process communication (IPC) is extensively used for communication between application modules. Because process protection domains only protect memory system boundaries, the message passing mechanisms of IPC employed in distributed applications forces processes outside of their protection domain for access to nearly all of their data. Furthermore, asking an operating system for network access every time data needs to be accessed incurs excessive overhead. Our claim is that the lack of job-level protection domains is the culprit behind many of issues of distributed systems relating to poor performance, instability, and low development productivity. These effects have forced system designers and application developers into making trade-offs between application performance, application reliability, and programming productivity.

Instead of viewing protection domains only from a processing perspective, we must also view them from the perspective of the memory system. For single node applications, the memory system is quite simple. Given a *read* command and address, the corresponding data is returned from the memory address. Given a *write* command, address, and data, the corresponding data is written to the memory address. Because the MMU has pre-filtered all the invalid memory requests, the memory system needs not check credentials before fulfilling requests. For distributed applications, data storage systems must meticulously check requester identity authenticity and permissions. This increases program complexity while reducing application performance. On top of this, distributed applications must also guard themselves against denial-of-service attacks, which might be driven by faulty code or malicious attackers.

3 Job-based Distributed Applications

In this section, we present definitions for *jobs* and *job protection domains*. Our formal definition of a *job* is a collection of processes working together in the network for a common purpose. There are no stipulations on these collections that processes must originate from the same code. The purpose of defining a job is to group many sub-entities (processes) into a single higher level entity where privileges can be applied and adhered to as a whole. Combined, job and process protection domains provide a two-stage protection domain hierarchy. This is shown in Figure 1.

The contents of three example job definitions are shown in Table 1. For each job there is a unique job ID, job relative process IDs, physical network addresses, and a set of ports (described later). As shown by job A, multiple processes of the same job may be resident on the same node. Furthermore, shown by jobs B and C, processes from different jobs may be resident on the same node.

Table 1: Example Job Definitions

Job ID	**Process IDs**	**Addresses**	**Ports**
A	0,1,2,3,4,5	10,11,12,10,11,12	Ψ,Δ,Θ
B	0,1,2,3	20,21,22,23	Ψ,Λ,Π,Σ
C	0,1,2,3,4	20,22,24,26,28	Υ,Φ,Ω

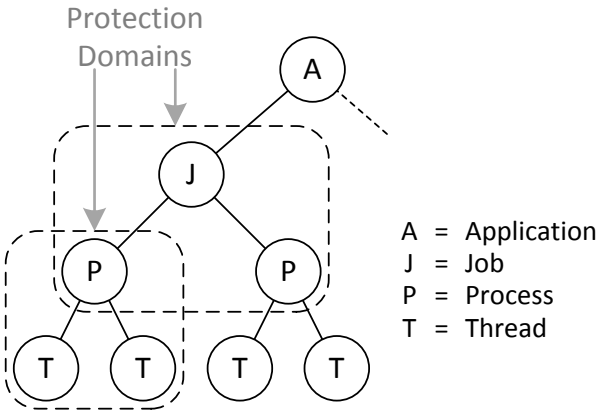

Figure 1: Multiple Protection Domains

Table 2: Job C's Protection Domain Definition

Job ID	Process IDs	Ports
B	2,3	Π,Σ
C	0,1,2,3,4	Υ

A job protection domain is a set of privileges that corresponds to a job. The first level of privileges in a job protection domain is a list of jobs that a job may communicate with. These are called *remote jobs* and the job list is called the *remote job list*. Table 2 shows an example job protection domain corresponding to job C from Table 1. The remote job list corresponds to the column labeled "Job ID". As shown, job C has been given permission to communicate with jobs B and C but not job A. Notice that it is a privilege for a job to communicate with the other processes within its own job. There are cases, such as web server front-ends, where processes within a job do not communicate with each other and therefore it is advantageous to keep these processes isolated.

The second level of privileges in a job protection domain describes which processes a job may communicate with. For each remote job contained in the remote job list, there is a corresponding list that contains the process IDs of all the processes that the job has been given access to. This list is called the *remote process list*. This is shown in Table 2 under the column labeled "Process IDs". This example shows that although job B is comprised of four processes, job C has been given access to only two of them.

The third level of privileges in a job protection domain describes the manner in which a job is allowed to access the jobs and processes it has been granted access to. This is implemented by a system called *ports*. Each job has complete control over the ports that it exposes. For each remote job in the remote job list, there is a list that contains the ports that the job has been given access to use for the corresponding remote job. This list is called the *remote ports list*. Table 1 shows that job A has chosen to expose three ports: Ψ, Δ, Θ. Because ports are just abstract identifiers, jobs can decide to bind to them in any way they wish. For example, even though jobs A

and B both declared a port with identifier Ψ, job A could bind that port to function call GetData() and job B could bind it to SelfDestruct(). There is no correspondence between port functionality across jobs and there is no stipulations dictating how a job may use a port. The only stipulation tied to the ports system is that all messages sent on the network have a specified port corresponding to the destination job. The column in Table 2 labeled "Ports" shows that job B has decided to only allow job C to access it on ports Π and Σ. Because job C's protection domain lists itself as a remote job, and the only port listed under the remote ports list is port identifier Υ, we can infer that port Υ is used for all intra-job communication and the rest of the ports are used for inter-job communication.

We have described the definitions of jobs and job protection domains. The responsibility of enforcing these protection domains lies in the system architecture on which the jobs execute, the network. Job protection domains have five essential attributes equivalent to the attributes of process protection domains, except they relate to jobs in a network rather than processes in memory. These attributes are: low latency data access, physical hardware abstraction, data privacy, fault isolation, and fine-grained control of inter-job communication. An implementation of job protection domains must guarantee that all four attributes exist without compromise.

4 Network Management Units

For an efficient implementation of job protection domains, as described in Section 3, we present a network interface architecture called the Network Management Unit (NMU). Just as processes are created and defined by a node's operating system, jobs are created and defined by a distributed operating system. Both of these operating systems are responsible for the protection domains of the entities they've created, thus a node's operating system is responsible for process protection domains and the distributed operating system is responsible for job protection domains. While each node's operating system manages only a single node's memory system, the distributed operating system manages the network fabric on which all node's communicate. We will provide insights to the functionality necessary in a distributed operating system relevant to job protection domains and NMUs, however, the details of distributed operating system implementation is beyond the scope of this paper.

The MMU's responsibility is to enforce the node's operating system defined process-based policies, and the NMU's responsibility is to enforce the distributed operating system defined job-based policies. Both of these devices are designed such that after they are configured they are able to enforce policies in real-time without assistance from their corresponding operating systems. Like all other network interface architectures, the NMU provides a physical interface that connects a processor node to an interconnection network. This section describes the architectural data structures contained within the NMU that enable job protection domains.

4.1 Implementation

An NMU exists on each node as a client of the virtual memory system and as such has access to data using virtual memory addresses (IOMMU functionality). For storage of its own data structures, the NMU is allowed to allocate its own virtually mapped physical memory pages. The NMU is able to access the page tables that correspond to all local processes that are part of job protection domains (further referred to as *resident processes*). With page table information, and IOMMU functionality, the NMU has access to the memory regions of the resident processes and can perform memory operations in their behalf. This is the first architectural feature towards supporting low latency network access.

The NMU implements an indirect memory-mapped register set that gives each resident process its own unique register file. This is accomplished with help from the node's operating system. The NMU responds to a wide range of physical memory addresses and the operating system maps each resident process to the NMU using a unique memory mapped address. The NMU keeps a table that maps each resident process's physical memory access address to its job ID and process ID. This table is called the *access address table*. When a resident process accesses the NMU via its unique memory mapping, the NMU uses the access address table to determine the process's identity, which consists of a job ID and process ID pair. This system removes the ability for a resident process to falsify its identity to the NMU. Each process's unique register file contains status and control registers for communicating with the NMU (described in Sections 4.3 and 4.4).

For each resident process, the NMU contains a table containing the privileges that have been granted to it as part of its job. This table is called a *privilege table*. It contains all the information describing which remote jobs may be accessed, the remote processes within those jobs, and the ports that may be used. This is the information that was discussed in Section 3 and found in Table 2. When a resident process attempts to access the network, it specifies the destination by specifying its job ID and process ID. This is known as a *virtual network address*. The process also specifies the destination port. The NMU checks the process's privilege table to determine if it has been given access to the destination job, process, and port. If the proper privilege exists, the NMU allows the message to enter the network. If the proper privilege does not exist, the NMU blocks the message from entering the network. Resident processes are not allowed to specify the physical network address of the destination. The NMU performs virtual-to-physical network address translation before sending validated messages.

No privilege checking is needed for receive operations because all checking is performed in the sender's NMU. This is possible because all NMUs are governed by a single trusted entity, the distributed operating system. As the NMU delivers the message to the corresponding process, it informs the process of the virtual network address of the sending process. Inherent identity authenticity is established and receiving processes are guaranteed on a per-port basis that only those with granted privileges can access them.

For implementation of virtual-to-physical network address translation, the NMU contains a data structure called the *job mapping table*. The procedure of virtual-to-physical network address translation requires information about virtual addresses (job

IDs and process IDs) and physical addresses. Instead of holding physical network addresses in the privilege tables, the NMU's job mapping table contains the mapping between process IDs and physical network addresses. This information is held for all remote jobs that all resident jobs have access to. This allows the NMU's privilege tables to be smaller, densely encoded, and accessed in parallel with the job mapping table. A lookup into the job mapping table is indexed by the destination job ID and process ID and the output is the physical network address.

The division between the process's memory space, the NMU's process accessible memory space, and the NMU's private memory space is shown in Figure 2.

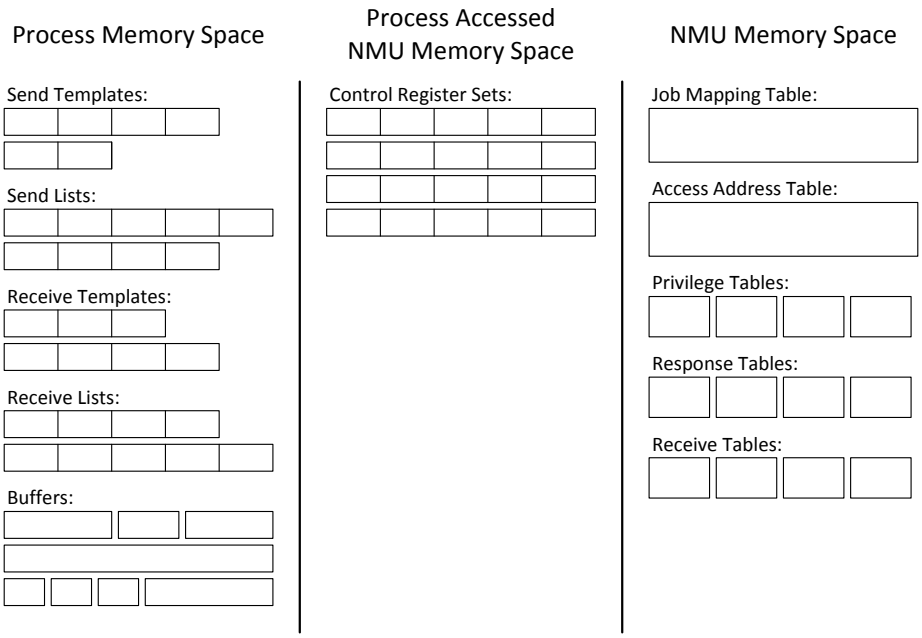

Figure 2: Process and NMU Memory Regions

4.2 Request-Response Protocols

As described so far, the implementation of job protection domains forces both parties in a two-way transaction to have proper privileges of communicating with each other. For most situations, this is sufficient, however, in many modern data centers there are jobs that exist solely to provide responses to client requests. The protocols used in these transactions are therefore called *request-response protocols (RRPs)*. When using RRPs, there are also situations where the requester desires the corresponding response to be forwarded to an alternate recipient. Essentially, the requester is posting a request to the responder in behalf of the designated response recipient.

To support RRPs, the NMU contains an extra data structure for each resident process called the *response table*. The response table is a data structure that is a temporary

holding area for one-time-use communication privileges. The response table supports two-way and three-way RRPs. Each entry in the response table is indexed via a unique identifier called a *handle*. Each entry contains: the requester's virtual network address, the recipient's virtual network address, the recipient's physical network address, and the recipient's port. Information about RRPs are contained in optional message headers created by the NMU.

For an example, let's examine a data center with three jobs: A, B, C. We will encode jobs, processes, and ports as:

`<job>:<process>:<port>`

Figure 3 will be used to show the progression of a 3-way request-response transaction between jobs A, B, and C. The right side of the figure shows the privileges obtained by each job as time progresses. As shown at the top, job A has been given the privilege to communicate with B:2:Φ and C:3:Ω. Jobs B and C have not been given any communication privileges. A:1 creates an RRP request message and informs its NMU of the desire to send the request to B:2:Φ and that the response should be sent to C:3:Ω. The NMU checks to see if job A has been given access to B:2:Φ, and if so, it generates the physical network address of B:2. It performs the same check for C:3:Ω and generates its physical network address. The NMU then sends the RRP request to B:2:Φ tagged with an RRP request header that contains information specifying the desired recipient by virtual and physical network address and port. This procedure occurs at Figure 3's timestamp 1.

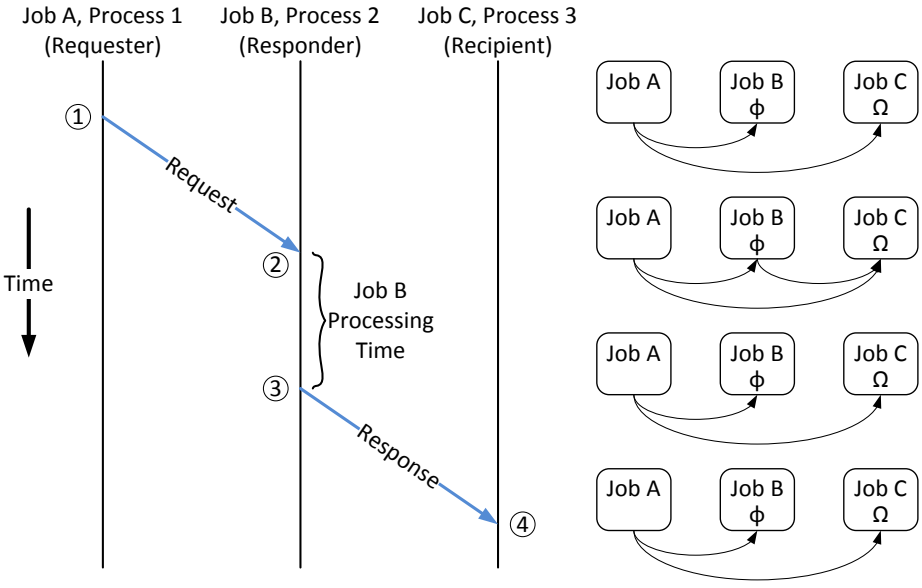

Figure 3: 3-Way Request-Response Transaction

When the NMU of B:2 receives the request it stores an entry in its response table identifying the requester's virtual network address, the recipient's virtual and physical

network address, and the recipient's port. This is Figure 3's timestamp 2, and as shown in the diagram, process B:2 now has the privilege to communicate with C:3:Ω. As the request is passed from the NMU to the corresponding process (further called the *responder*), the handle of the response table entry is also given to the responder.

After the responder has processed the request and generated the response message, it informs its NMU of the desire to send the message. Instead of specifying the typical destination virtual address and port, the responder specifies the handle of the response table entry it was given. The NMU retrieves the requester and recipient information from the response table and the response is then sent to the specified recipient, which is C:3:Ω. The response is tagged with an RRP response header that specifies the requester's virtual address. After the response table entry is used, it is deleted. No virtual-to-physical network address translation is needed because the recipient's physical network address is contained within the response table entry. This procedure occurs at Figure 3's timestamp 3. Notice that B:2's privilege to communicate with C:3:Ω has been removed.

When the recipient's NMU receives the response message, seeing that it is tagged with an RRP response message, gives the message to the recipient along with the requester's virtual network address. This procedure occurs at Figure 3's timestamp 4 and, as shown, doesn't alter the privileges of any of the jobs.

This example illustrates a secured three-way RRP transaction where job B indirectly acts on behalf of job A to give job C the information requested. Notice here that jobs B and C both have no privileges of network communication but because others can access them, they can act in behalf of other jobs. Also notice that upon receiving messages, processes inherently know the identity of the sender and falsification of identity is impossible. Two-way RRP transactions are a simplified version of the RRP system where the requester specifies itself as the recipient.

This system for RRP transactions is very useful for large jobs that service many clients, such as Amazon's S3 [15] or Google's BigTable [8]. These jobs don't actively send network messages to other jobs, but instead exist to fulfill requests made by their clients. Holding static mappings for all client jobs would require a very large job mapping table and many privilege tables which would result in high latency access times and wasted memory. The response table provides a mechanism that removes the need for static job mappings and only requires enough storage to handle the number of outstanding requests. Furthermore, along with all types of NMU transactions, the RRP system provides inherent sender identity authenticity. This greatly reduces the code complexity of large jobs that interact with many clients. Having inherent knowledge of identity removes the need for identity credentials between jobs and processes.

4.3 Send Templates

The NMU provides applications with end-to-end zero-copy network access using a system called *templates*. The templates system has many similarities with gather-scatter lists found in common DMA engines, except that it works in connection with job protection domains and are specified on a per-port basis. The template system also contains a type field which allows the use of immediate values as well as buffer references.

To send a message the process first creates a *send template* that contains information about the layout of a message, but does not contain any message data. The template's first element is a count that specifies the number of remaining elements. All remaining elements describe a type of data, of which there are three types: immediate, fixed size buffer, variable sized buffer. Immediate and fixed size buffer entries also specify a length while variable sized buffer entries do not. The process then creates a *send list* that contains the message's information. The list's first element is a state indicator that is used to flag the process of send completion and must be marked *active* before using. All remaining elements are immediate values, fixed size buffer pointers, and variable sized buffer pointers with their corresponding lengths. Before sending a message, the process writes the location of the send template and send list into its NMU send control register. To send the message, the process writes the desired destination job, process, and port into the NMU send control register, then triggers a write-only "send" register. After checking permissions, the NMU transfers the data directly from the process's memory space to the network. After the message has been received at the endpoint, the NMU informs the process of successful delivery via marking the send list as *deactive* and optionally via an interrupt.

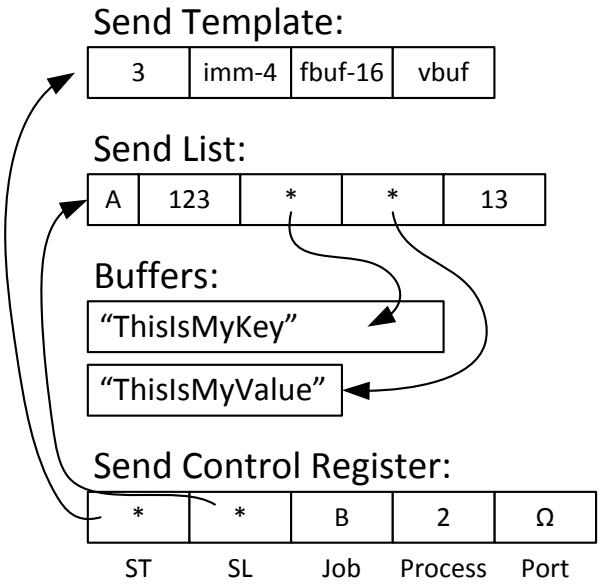

Figure 4: Data Structures in Send Procedure

To illustrate the operation of send templates, let's assume job A, process 1 (A:1) is sending a write request to a key-value store. The key-value store is located in job B, process 2 (B:2) and the port specified for write requests is port Ω. The store contains tables of key-value maps identified by 32-bit identifiers. Keys are fixed sized 16 byte character arrays and values are variable sized character arrays with a maximum size of 32 bytes. A:1 desires to map key "ThisIsMyKey" to value "ThisIsMyValue" in table 123. The process creates a send template with 3 type elements: 4-byte immediate, 16-

byte fixed sized buffer, variable sized buffer. The process then creates an active send list that contains: 4-byte table identifier, memory address of the 16-byte key buffer, memory address the 13-byte value buffer, length of value buffer. The process writes the memory location of the send template and the send list into the send control register as well as the desired destination job, process, and port. These data structures are shown in Figure 4. The message is assembled and sent after the process triggers the send action by writing to the "send" register. The assembled message is shown in Figure 5. After the message has been received at the destination, the NMU marks the send list as deactive.

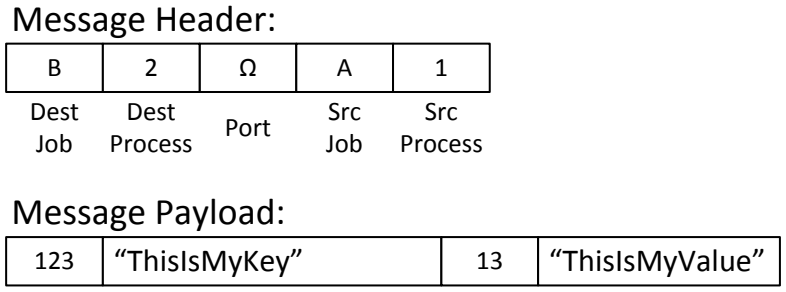

Figure 5: Assembled Message

4.4 Receive Templates

The receiving procedure of the NMU utilizes *receive templates* and *receive lists* which are similar to the send templates and send lists described in Section 4.3. The only structural difference is that receive templates contain a maximum size for variable sized buffers, whereas send templates do not. This is necessary because the process must allocate the buffers before they are used. Receive templates are specified on a per port basis and are held within a data structure called the *receive table* which is held for each resident process. The receive table maps a single port to a single receive template, however, for each receive template there is a fixed depth queue wherein references to receive lists are stored. The receive list queue size is allocated at setup time, but the entries are added and removed during runtime.

Before receiving messages, a process must register a receive template for each job port and one or more receive lists for each receive template. All receive lists are marked *ready* as they are linked with the receive list queue. When a message is received, the NMU inspects the message header for the destination job, process, and port then it performs a lookup into the receive table for the receive template and next available receive list. Using the information in the receive template and receive list, the NMU places the data directly into receive list and buffers specified by the receive list. After having placed the message, the NMU marks the receive list as *used*, unlinks it from the receive list queue, and optionally interrupts the process. When another message is received on the same port, the next available receive list will be used. It is the responsibility of the process to guarantee that enough receive lists are available for

each receive template.

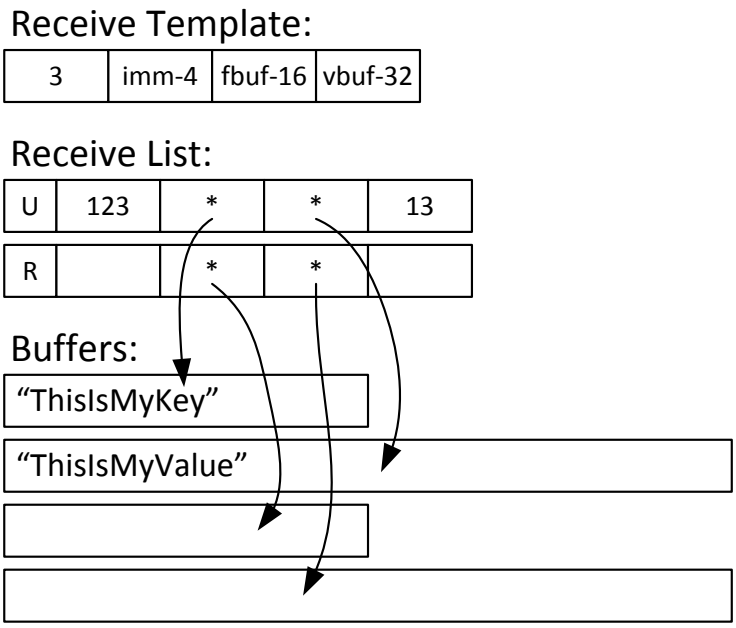

Figure 6: Data Structures in Receive Procedure

To illustrate the operation of receive templates, let's continue from the example in Section 4.3 and view the perspective of the key-value store as it receives the message previously sent by A:1. In preparation to receiving messages on port Ω, B:2 creates a receive template similar to the send template used by A:1 and registers it with port Ω in the NMU. B:2 also creates two receive lists by creating two sets of buffers and registers the two receive lists with the NMU. When the message arrives at B:2's NMU, the NMU inspects the message header (shown in Figure 5) and determines that the destination is B:2:Ω. A lookup into B:2's receive table results in the designated receive template and the next available receive list. Using the receive template and receive list, the NMU places the message data in the receive list and corresponding buffers. The NMU then marks the receive list as used, removes it from the receive list queue, and optionally interrupts the process. Figure 6 shows the state of the receive data structures after having received the message sent by A:1.

The NMUs port-oriented template system provides a complete end-to-end zero-copy solution that places message data right where the user needs it and frees the processor from performing wasteful copying.

4.5 Management

As a network interface architecture, the NMU is an entity of the network in the distributed system. As such, control over its functionality lies within the responsibilities of the distributed operating system. Even though the NMU stores data structures within

the system memory of its corresponding node, it is the distributed operating system that fills these data structures with the policies that regulate job protection domains. The distributed operating system uses special control messages to communicate with the NMUs in the network. These messages control the construction, evolution, and deconstruction of job protection domains.

5 Analysis

To illustrate the effectiveness of the Network Management Unit (NMU) we compare it to the effectiveness of the Memory Management Unit (MMU) at enabling process protection domains. The content in Sections 2 and 3 argue that a true protection domain only exists if the following attributes are present: low latency data access, physical hardware abstraction, data privacy, fault isolation, and fine-grained control of inter-domain communication. This section analyzes the NMU's effectiveness in providing these attributes.

Low latency data access: The NMU provides low latency data access by allowing processes to interact with it directly and utilizing an efficient port-oriented template system that supports an end-to-end zero-copy solution. Without excessive concern of how and where data is stored, programmers are able focus on the application under development and spend less time overcoming data access related performance obstacles.

Physical hardware abstraction: The NMU implements an efficient system for *virtual network addressing* that allows programmers to ignore the complexities of physical network addressing and management. This job-oriented virtual network abstraction supports high productivity without compromising performance.

Data privacy and fault isolation: The NMU provides strict isolation between jobs on a network as jobs are guaranteed that their processes can't be accessed by unprivileged jobs. The NMU stops all invalid network accesses before they enter the network. Isolation allows job implementation to be greatly simplified because jobs are assured that received messages could have only been generated by authorized entities. Because sender identity authenticity is inherently built into NMU-enabled job protection domains, the use of identity credentials is not needed. Furthermore, sender-based job isolation makes it impossible to post a denial-of-service attack from one job to another where communication privileges were not previously established. This is true regardless of whether the attack is malicious or accidental.

Fine-grained control of inter-domain communication: On top of enforcing policies that control which processes get to communicate, the NMU enforces policies that control in which ways they are allowed communicate. This is contained in the NMU's *ports* system. This is a place where the NMU excels over the MMU. For shared memory inter-process communication, the MMU regulates who has access and which permissions they have. In this regard, the NMU works just like the MMU, except the permissions are application specific through the user programmable ports system. Ports allow an application to divvy out permissions on a fine-grained basis so that application functionality and sender identity authenticity are always maintained and easy to use. The NMU's mechanism for one-time-use privileges in request-response protocols allows jobs to limit their interaction with other jobs on a time-based level.

6 Discussion

The NMU-enabled network we've presented provides an efficient platform for distributed application development. There are few additional implementation aspects that can affect the efficacy of the job protection domains the NMU provides.

6.1 Integration with the CPU

The placement of the NMU within the network can have a significant effect on its performance. The network access latency of a network interface located on a common peripheral bus (e.g. PCI Express [2]) is bounded by the latency of the bus. Moving the network interface closer to the processor by locating it on a coherent processor interconnect (e.g. Intel QuickPath Interconnect [21]) allows the network interface to be accessed faster and have quicker access to system memory. Locating the network interface on same die as the processor (e.g. IBM PowerEN [6]) would yield even lower access latencies as the network interface can be accessed as fast as other on-chip devices. Some designers have taken the opposite approach and have moved the network interface into the network by integrating it with a network switch (e.g. Cray Gemini [1] and Cascade [13]). While this doesn't produce low network interface access latencies, the end-to-end latency is reduced because two hops have been removed. These trade-offs must considered when determining the optimal placement of the NMU.

6.2 Trusting the Node Operating System

The optimal location of the NMU data structures depends on the trust model the system has for the node operating systems. For establishing job protection domains, collaboration between the distributed operating system and the node operating systems is needed. The node operating system is expected to provide two functions. First, it must not allow the NMU's allocated memory to accessed by any entity other than the NMU. Second, it must ensure each resident process has a unique physical page mapping to the NMU and that the NMU contains the proper mapping between access address ranges and job ID and process ID pairs.

For situations where the node operating system can't be trusted with access to the NMU's data structures, we propose two solutions. The first option is the NMU stores all its data structures in the system memory and the NMU computes and stores digital signatures for each entry of the structures. This solution can't block tampering but it can detect it and take appropriate actions. This solution incurs an overhead every time an entry is altered, which for structures like the response table or receive table could be very often. The second option is the NMU stores all its data structures in a dedicated memory that is only accessible by the NMU. This solution doesn't incur the overhead associated with the first solution, however, it increases the system cost by having a separate memory for the NMU.

If the node operating system doesn't properly maintain the integrity of the unique physical page mapping each resident process has with the NMU, the job protection domains may become corrupt, however, the severity of corruption is only within the jobs

the node has access to. Protection domain corruption due to faulty node operating systems can't span wider than the domains the operating system knows about. This is true because each node only contains the job mappings of the jobs the resident processes are members of and the remote jobs they have access to.

6.3 Job Mapping and Node Allocation

The methods in which jobs are created, nodes are allocated, and network addresses are computed can have a significant effect on performance. It is important that the virtual-to-physical network address translation be efficient as all messages, besides RRP responses, sent by the NMU require it. It is also important that the job mappings be small as they are held within the job mapping table of the NMU.

Allocating nodes for a job in a contiguous physical network address range has the benefit of having very dense job mappings and fast translation computation. Unfortunately, contiguous allocation is inflexible and leads to poor system utilization. At the other extreme, one-by-one node selection results in mapping tables that can flexibly utilize the entire system. Due to the large mapping table, the virtual-to-physical translation may consume many cycles. A hybrid approach of the these two methods is to allow a job to be allocated to multiple contiguous sets of nodes. This allows the mapping to be dense and flexible.

7 Related Work

Data center stacks such as Hadoop [28], Amazon Web Services [15], Windows Azure [7], and OpenStack [27] provide job-level scheduling, resource management, node allocation, and communication abstraction, however, they aren't able to provide strict job-level isolation due to the limitations of the underlying network infrastructure.

VLANs offer a very coarse-grained hardware isolation solution, however, the arrival of new jobs is several orders of magnitude more frequent than the rate at which VLANs can be updated and switches reconfigured [17] [24].

In efforts to provide isolation, some data center operators use switches that perform 5-tuple filtering that filters traffic based on rules that match on protocol, source address, destination address, source port, and destination port [20]. NIC-based offload engines have been proposed [26] [11] that are able to perform 5-tuple filtering in the receiver's NIC. Network switches and routers are available that perform this function close to the sender in the top-of-rack switch. True sender based filtering could be achieved if the NIC-based offload engines were configured to filter outgoing traffic. Even in this scenario, the rules these systems comply to relate to physical endpoints and abstract port numbers and none of them are aware of jobs or processes.

The InfiniBand architecture provides a mechanism called protection domains that allow a consumer to control which set of its Memory Regions and Memory Windows can be accessed by which set of its Queue Pairs [3]. Because these domains are placed upon queue pairs created by processes, job-level protection domains could be achieved if all processes within a job shared the same domain. Furthermore, inter-job communication could be achieved by sharing protection domains between jobs. The downside

to InfiniBand protection domains is that they don't provide isolation because they are enforced at the receiver. They also don't provide a fine-grained communication model on which specific privileges can be placed.

MPI provides a high level view of jobs and abstracts the communication details from the programmer [25]. This provides high productivity, however, it reduces performance as it is implemented in software. It also provides no isolation as filtering is performed at the receiver.

If performance was not a concern, the NMU implementation could be implemented within the operating system's network stack. This would provide the reliability of a hardware-based NMU system, but would suffer from very low performance which would in turn reduce productivity due to high data access latencies.

8 Conclusion

In this paper we have introduced the network management unit (NMU) that provides low-latency inter-process communication while enforcing a job-based isolation policy. Low-latency communication is achieved by providing protected user-level memory-mapped access to the network and by using send and receive *templates* that allow the arguments of complex messages to be marshalled and unmarshalled by hardware without copying. A job-based protection policy that matches the structure of many distributed applications is implemented in a scalable manner by associating lists of privileges (accessible jobs and ports) with each job. Request-response protocols are facilitated by a mechanism that allows single-use privileges to be passed as arguments. Message authorization is checked at the sender to prevent unauthorized messages from congesting the network. Processes within each job are addressed via network virtual addresses that are translated to physical nodes in the NMU.

The NMU and the job-based protection abstraction it supports facilitates the construction of efficient distributed applications. A typical application is constructed from a number of jobs (web tier, key-value store, database, etc.) that interact in a controlled manner as governed by the set of privileges. Job-based protection with sender-side authentication could be implemented by patching together existing mechanisms such as InfiniBand remote keys and 5-tuple filtering in a top-of-rack switch. However the NMU provides an integrated solution that reduces overhead and implementation complexity compared to a patchwork implementation.

The NMU provides efficient run-time mechanisms to support a distributed protection model much in the way an MMU provides run-time mechanism to support memory protection within a single node. The NMU allows a distributed operating system to configure a set of jobs and privileges much in the way an MMU allows a node operating system to configure a set of processes and memory segments. The efficient run-time mechanisms of the NMU allow distributed applications to be written with strong security without incurring the application complexity or run-time overhead associated with application-level software implementations.

References

[1] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87. IEEE, 2010.

[2] Don Anderson, Tom Shanley, and Ravi Budruk. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[3] InfiniBand Trade Association. Infiniband architecture specification, vols 1 & 2, release 1.2, october 2004, 2004.

[4] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebooks distributed data store for the social graph. USENIX Association, 2013.

[6] Jeffrey D Brown, Sandra Woodward, Brian M Bass, and Charles L Johnson. Ibm power edge of network processor: A wire-speed system on a chip. *Micro, IEEE*, 31(2):76–85, 2011.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Nick Dellamaggiore and Eishay Smith. Linkedin: A professional social network built with java technologies and agile practices, 2008.

[11] Luca Deri, Joseph Gasparakis, Peter Waskiewicz Jr, and Francesco Fusco. Wire-speed hardware-assisted traffic filtering with mainstream network adapters. In *Advances in Network-Embedded Management and Applications*, pages 71–86. Springer, 2011.

[12] Paul DuBois. *MySQL*. Pearson Education, 2008.

[13] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray cascade: a scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 103. IEEE Computer Society Press, 2012.

[14] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[15] Simson L Garfinkel. An evaluation of amazons grid computing services: Ec2, s3, and sqs. In *Center for*. Citeseer, 2007.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[17] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1):68–73, 2008.

[18] Todd Hoff. Scaling twitter: Making twitter 10000 percent faster, 2009.

[19] Todd Hoff. The architecture twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds, 2013.

[20] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 51–62. ACM, 2008.

[21] Nasser Kurd, Jonathan Douglas, Praveen Mosalikanti, and Rajesh Kumar. Next generation intel® micro-architecture (nehalem) clocking architecture. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 62–63. IEEE, 2008.

[22] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[23] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Strafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013.

[24] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: performance isolation for cloud datacenter networks. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 1–1. USENIX Association, 2010.

[25] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[26] Yaron Weinsberg, Elan Pavlov, Yossi Amir, Gilad Gat, and Sharon Wulff. Putting it on the nic: A case study on application offloading to a network interface card (nic). In *Consumer Communications and Networking Conference, IEEE CCNC*, 2006.

[27] Xiaolong Wen, Genqiang Gu, Qingchun Li, Yun Gao, and Xuejie Zhang. Comparison of open-source cloud management platforms: Openstack and opennebula. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pages 2457–2461. IEEE, 2012.

[28] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.