Sikker: A Distributed System Architecture for Secure High Performance Computing

Nicholas McDonald
nmcdonal@stanford.edu

William J. Dally
dally@stanford.edu

Stanford University
450 Serra Mall
Stanford, CA 94305

Abstract

After decades of evolution, the network requirements of data centers, supercomputers, and cloud computing facilities are beginning to converge requiring high performance network access while supporting a secure computing environment for numerous concurrently running applications with complex interaction policies. Unfortunately, current network technologies are unable to simultaneously provide high performance network access and robust application isolation and security. As a result, system designers and application developers are forced into making trade-offs between these requirements.

We propose Sikker[1], a new network architecture for distributed systems under a single administrative domain. Sikker includes a novel service-oriented security and isolation model with a corresponding network interface controller, called a Network Management Unit (NMU), that enforces this model while providing high performance network access.

We show that Sikker's security model satisfies the complex interaction policies of modern large-scale distributed applications. Our experimentation results show that even when implemented on very large clusters under worst case access patterns, the message latency incurred by Sikker is 52ns on average and 66ns at the 99th percentile, a negligible increase. Smaller clusters and/or more realistic access patterns bring these overheads down in the 35-45ns range. Sikker's service-oriented security and isolation mechanism removes the need for high overhead software-based implementations imposed by current systems. Sikker allows distributed applications to operate in a secure environment while experiencing network performance on par with modern supercomputers.

1. Introduction

The number and variety of applications and services running in modern data centers, cloud computing facilities, and supercomputers has driven the need for a secure computing platform with an intricate network isolation and security policy. Traditionally, supercomputers focused on performance at the expense of internal network security while data centers and cloud computing facilities focused on cost efficiency, flexibility, and Internet compatibility all at the expense of performance. In contrast to their historical differences, the requirements of these computing domains are beginning to converge. As data center and cloud computing applications increase in complexity, size, and quantity, they require higher network bandwidth and lower predictable latency. As supercomputers become more cost sensitive and are simultaneously utilized by many clients, they require a higher level of application isolation and security. The advent of cloud-based supercomputing brings these domains even closer by merging them onto the same network fabric.

The vast majority of modern data centers utilize network technology that has evolved from networking equipment designed for wide area networks where numerous untrusted domains communicate. For distributed systems under a single administrative domain, the network can be vastly improved by removing unnecessary overheads and adding extra network functionality supporting higher performance, security, and productivity. This is the basis of design for all high-performance interconnection networks (e.g. Cray Cascade [7], IBM Blue Gene/Q [5], Mellanox InfiniBand [23], etc.) in addition to software-based network virtualization techniques that assume a trusted network where security checks can be performed in a hypervisor before entering the virtualized network (e.g. OpenStack Neutron [8], VMware NSX [29], etc.).

The unfortunate truth is that modern network technologies have not provided distributed systems that are capable of supercomputer-like network performance while simultaneously providing robust application security and isolation. As a result, system designers and application developers are forced to make trade-offs between performance, security, and isolation leaving deficiencies in their system and creating higher development and runtime overheads for application developers.

In this paper, we present a new distributed system architecture, called Sikker, that includes an explicit security and isolation policy. The goal of this system is to provide the highest level of network performance, equivalent to a supercomputer without any security mechanisms, while enforcing the highest level of application security and isolation required by the complex interactions of modern large-scale applications. Sikker formally defines a distributed application as a collection of distributed services with well-defined interaction policies. Sikker utilizes specially architected network interface controllers, called Network Management Units (NMUs), to enforce application security and isolation policies while providing efficient network access. These NMUs operate directly

[1] Sikker is a danish translation for "safe"

under the control of a system-wide trusted network operating system, and as such, are not vulnerable to compromises of individual host operating systems. This makes Sikker a good candidate architecture for systems requiring bare-metal computing, virtual machine computing, and hybrid computing schemes.

This paper makes the following contributions:

- We present a new distributed system network security and isolation model that fits directly to modern large-scale applications. This is the first work to present a network architecture that implements a service-oriented security model and process-oriented authentication.
- We show how modern large-scale applications fit into this model and how they can be modified to make use of the model under our design.
- We present the Network Management Unit (NMU), a high performance network interface controller that, under the direction of a network operating system, enforces the security and isolation policies of Sikker.
- We provide a qualitative analysis of the security and isolation provided by Sikker.
- We provide a quantitative analysis of the performance of the Network Management Unit.

2. Motivation

2.1. Service-Oriented Applications

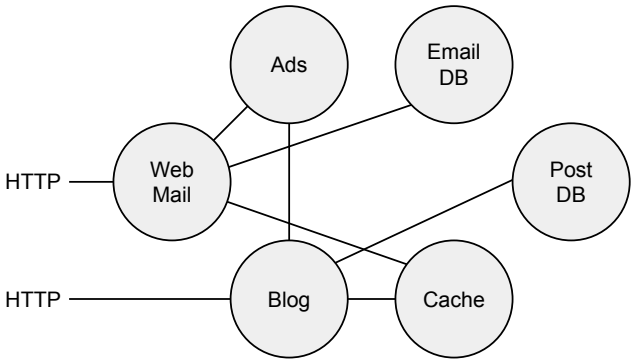


Figure 1: High level service connectivity

Modern large-scale distributed applications are commonly comprised of thousands of processes. For reasons of management, separation of development, modularity, and fault tolerance, these processes are grouped by similarity into collections called *services*. A service is a collection of processes developed and executed for the purpose of implementing a subset of an application's functionality. Applications can be comprised of one or more services, often tens or hundreds, and services are often shared between many applications. Figure 1 shows a simplified diagram of six services interacting to fulfill the functionality of two applications. Each service has a defined application programming interface (API) that it exposes to provide functionality to other services or entities. Even though a modern data center might contain thousands of services, each service generally communicates with a relatively small subset of the total services in order to fulfill its designed functionality. Furthermore, it is common for a service to use only a portion of another service's API.

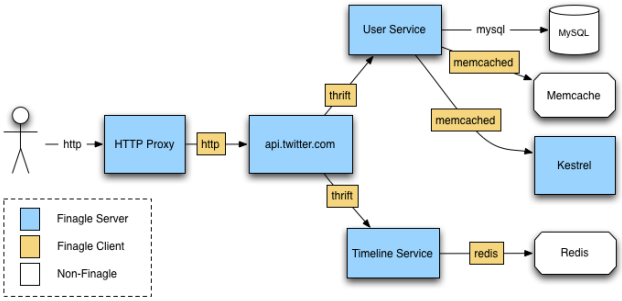


Figure 2: Twitter's 'Finagle' RPC system [27]

Figure 2 is a diagram created by Twitter to illustrate the operation of their protocol-agnostic communication system. Similarly, figure 3 is a diagram created by Netflix illustrating their architecture on Amazon's cloud computing platform. For both of these designs, there exists several services custom written for the application, as well as several services written by third-parties. Both of these diagrams show that when designing an application at a high level, application developers divide the application's functionality into services with well-defined APIs to achieve modularity.

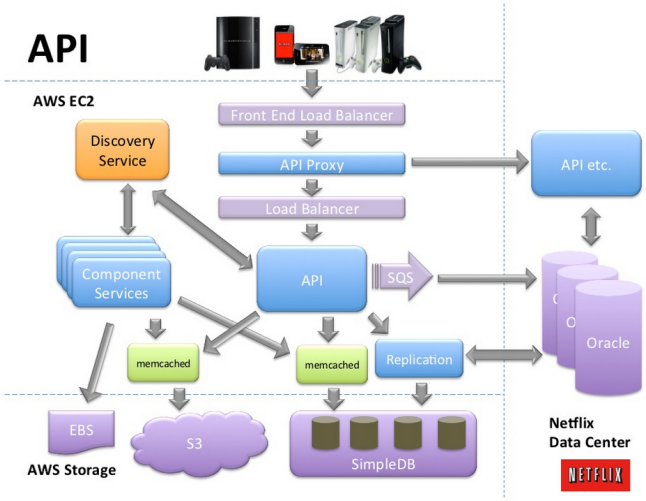


Figure 3: Netflix's architecture on Amazon's AWS [16]

An inspection of the code of any given service would reveal the implicit interaction privileges it desires with other services. In most cases, the code expressing the desired interactions does not contain IP addresses or TCP port numbers, but instead contains service names, process identifiers, and API functions. For example, we might see a service called *NewsFeed* desiring

to communicate with a service called *UserAccounts* using its process #6 and using an API function called *UserLogIn*.

2.2. Network Performance

The highest level of network performance available today is found in supercomputing interconnection networks such as Cray Cascade [7], IBM Blue Gene/Q [5], and Mellanox InfiniBand [24]. In order to achieve our goal of high network performance while providing application security and isolation, we define our metrics for *high performance* relative to the performance of supercomputer network interconnects. In order for a system to claim to be high performance, we argue that it must be able to provide approximately the same level of network performance as a supercomputer.

Modern supercomputers achieve high bandwidth and low, predictable latency. For example, InfiniBand networks manufactured by Mellanox Technologies achieve round-trip times on the order of 2µs and bandwidths as high as 56 Gbps [24]. The Cray Cascade system achieves unidirectional latencies as low as 500ns[2] and provides 93.6 Gbps[3] of global bandwidth per node [7].

One of the major strategies that supercomputers use to achieve high performance is allowing applications to bypass the operating system and interact with the network interface directly. All major high performance computing fabrics (e.g. Cray, IBM, Mellanox, Myricom) have taken this approach. This technique is commonly referred to as *operating system bypass* or *OS-bypass*. Along with providing lower network latency, OS-bypass can also provide lower CPU overhead as applications can offload transport protocol handling on the network interface. OS-bypass has one major ramification, namely, bypassing the kernel (or hypervisor) removes the kernel's ability to impede outgoing network traffic in an effort to provide sender-side security and isolation features. Later in this paper, we show that this is a desirable feature and not a drawback.

2.3. Current Technologies

Supercomputer: To achieve the highest level of network performance, modern supercomputers employ operating system bypass mechanisms, protocol off-loading, and low overhead remote memory semantics. Additionally, for the sake of performance, these systems employ minimal security and isolation mechanisms. For isolation, some fabrics use coarse-grained network partitioning schemes. While these schemes are efficient at completely isolating applications from each other, they don't provide a mechanism for controlled interaction between applications. This becomes problematic if the cluster offers shared services, such as a distributed file system like Lustre [3].

[2]Latency specification only includes the network overhead, not the overhead of the endpoints.

[3]Bandwidth specification is raw on-the-wire bandwidth. 64% of the raw bandwidth is available to the application.

Some high performance fabrics, namely InfiniBand, employ mechanisms for secret key verification where the receiving network interface is able to drop packets that do not present the proper access key that corresponds to the requested resource. While this scheme provides a mechanism for coarse-grained security, it does not provide network isolation. As a result, the endpoints are susceptible to denial of service attacks, regardless whether they be accidental or malicious.

Cloud computing: In contrast to supercomputers, cloud computing facilities (e.g. Amazon Web Services [1], Microsoft Azure [14], Google Cloud Platform [10], Heroku [19], Joyent [12], etc.) are faced with the most malicious of tenants. These facilities run applications from many thousands of customers simultaneously, some as small as one machine and others as large as thousands of machines. These facilities must provide the highest level of security and isolation in order to protect their clients from each other. Furthermore, these facilities often have large sharable services that get used by their tenants for storage, caching, messaging, load balancing, etc. These services must also be protected from tenant abuse.

Network isolation mechanisms found in modern cloud computing facilities are network partitioning schemes (e.g. VLAN [17], VXLAN [13], NVGRE [21], etc.). These partitioning schemes are successful at completely isolating applications from each other, but just like the partitioning schemes found in supercomputers, they don't provide a mechanism for controlled interaction between applications. In efforts to bridge virtual LANs, network virtualization software like OpenStack Neutron [8] and VMware NSX [29] create virtualized routers that use access control lists to control the interactions between virtual LANs. This topic is covered in depth in section 2.4.

It is well known that cloud computing environments impose high network overheads and unpredictable performance on their clients [30, 6]. While we do not claim that all of these poor results are related to security and isolation, it is evident that modern network virtualization techniques cause significant overhead. A recent study [28] shows that 2 virtual machines communicating on the same host should expect 25-75µs of round-trip latency. Similarly, a virtual machine communicating with a native operating system connected to the same physical switch should expect 35-75µs of round-trip latency. The latency is significantly worse if the communication is forced to go through an intermediate host containing a virtual router in order to cross the boundary between virtualized networks.

Data center: The requirements of large singly-operated data centers, such as those by Facebook, Twitter, and LinkedIn, for example, lie somewhere in-between the requirements of supercomputers and cloud computing facilities. Due to cost sensitivity, these companies often choose to forego internal network security and isolation in order to increase the effective performance of the low cost commodity equipment they choose to deploy. In many cases this works fine with the assumption that the developers of the various services within

the company will comply properly to each other's APIs and expected usage rates. In general, there is no malicious intent between services, however, software bugs and system misconfigurations can, and do, cause detrimental disasters in the data center [22, 26, 11, 9, 2]. In cases where security is required, the isolation and security mechanisms found in cloud computing facilities can be used if the corresponding overheads are not too high for the applications' requirements.

2.4. Security and Isolation

The implicit privileges discussed in section 2.1 present the ideal level at which permissions should be enforced. As mentioned, these privileges are derived from the applications themselves and represent the actual intent of the underlying services on the network. The available security and isolation techiques in today's data centers use multiple layers of indirection before permissions are checked and enforced. This creates high operational complexity and presents many opportunities for misconfiguration. Even worse is that these systems lose information about the original intent of the application, thus, cannot enforce the privilege as it was intented. The lack of inherent identity authenticity within the network forces developers to use authentication mechanisms that incur high CPU overheads and are unable to properly guard against denial of service attacks. In this section, we will describe how current systems work and our proposal for a better solution.

To moderate network access, the majority of modern network isolation mechanisms use some form of access control list (ACL). In the abstract form, an ACL is a list of numerous entries each containing identifiers corresponding to some communication mechanism and represent a permissions whitelist. For access to be granted, each communication must match on one of the entries in the ACL. The most common type of ACL entry is derived from TCP/IP network standards. We will further refer to this style of ACL as a network-ACL or N-ACL. Table 1 shows an example of an N-ACL entry commonly referred to as a *5-tuple*. This entry states that a packet will be

Proto	Src IP	Src Port	Dst IP	Dst Port
TCP	192.168.1.3	123	10.0.2.10	80

Table 1: Example 5-tuple network-derived ACL entry

accepted by the network if the protocol is TCP and it is being sent from 192.168.1.3 port 123 to 10.0.2.10 port 80. Portions of a N-ACL can be masked out so that only a portion of the entry must be matched in order for a packet to be accepted by the network.

A comparison between the ACL whitelisting mechanism discribed above (N-ACL) and the privileges discussed in section 2.1 exposes many deficiencies of using any ACL system based on network-centric identifiers such as protocols, network addresses, or TCP/UDP ports. One important thing to notice is that the source entity is referenced by an IP address and optionally a port. For this system to work as desired, the system must know with absolute confidence that the source entity is the only entity with access to that address/port combination and that it is unable to use any other combination. This is hard to ensure because the notion of an IP address is very fluid. While it is commonly tied to one network interface controller (NIC), modern operating systems allow a single machine to have many NICs, a single NIC to have more than one IP address, and multiple NICs can share one or more IP addresses. There is no definitive way to determine the source entity based off a source IP address. Another issue is the use of UDP and TCP ports, which are abstract identifiers shared among all the processes on a given machine. Unless there exists a network-wide governing system that limits the use of these ports, these cannot be used for security and isolation.

ACL whitelisting has the right intent with its approach to security and isolation because of its inherent implementation of the principle of least privilege [20] and its ability to prevent denial of service attacks by filtering invalid traffic *before* it enters the network. However, using network-derived ACL entries is the source of security and isolation deficiency in modern networks. In order to design a better system, we propose creating ACL entries based directly from the privileges discussed in section 2.1. Our ACL entries exactly express the communication interactions of services and their APIs. We will further refer to this style of ACL as a service-ACL or S-ACL. Table 2 shows an example of an S-ACL entry. This

Src Service	Dst Service	Dst Process	Dst API
UserAccounts	Memcached	27	Set

Table 2: Example service-oriented ACL entry

service-oriented ACL entry references the source entity by its actual identity, the service. The destination is also referenced by the service along with the process identifier within the service and the API function to be used. In this example, the service *UserAccounts* has been given access to service *Memcached* using process #27 and API function *Set*. S-ACLs make reasoning about network permissions much easier and doesn't tie the permission system to any underlying transport protocol or addressing scheme. It simply enforces permissions in their natural habitat, the application layer.

In order for the S-ACL methodology to work as designed, the following requirements must be upheld:

1. The network is a trusted entity and no endpoint has control over it.
2. The network is able to derive the identity of a process and it is impossible for a process to falsify its identity.
3. The source (sender) identifier is sent with each message to the destination (receiver).
4. Messages sent are only received by the specified destination entity.

If these requirements are met, a tremendous amount of se-

curity benefits are available to the endpoints. This system inherently implements source authentication by which all received messages explicitly state the source entity's identification. Destination authentication is also inherent by the same logic. Combined, source and destination authentication remove the need for complex authentication software in the application layer. Furthermore, senders don't need to use nameservers to discover physical addressing for desired destinations as they only need to specify the destination by its actual identity (i.e. service ID, process ID, and API ID) and the network will deliver the message to the proper physical location.

3. Application and Security Model

3.1. Application Structure

With the insights gained in section 2, we define a new distributed system application and security model, called *Sikker*, that formally defines the structure of distributed applications. Sikker is strictly a service-oriented architecture and makes no attempt to justify the boundaries of *applications*. As a service-oriented application architecture, Sikker designates the *service* as the basic building block of distributed applications.

Sikker defines a service by its execution units and its application programming interface (API). The execution units of a service are simply processes. Each service contains a collection of similar processes that implement a common API. Each process within a service is assigned a numerical ID unique to the service.

The API of a service is divided into permission domains using abstract numerical IDs called *ports*. Each port represents a portion of the service's functionality with respect to a specific permission. Unlike TCP/UDP ports, ports in Sikker are not used for multiplexing, are not shared, and are only used to specify a destination. Each service has its own port number space, thus, two services using the same port ID is acceptable.

For an example of the port system, consider a simple key/value storage service that exposes functionality to perform data retrieval (e.g. "get" in memcached or Redis). Assuming that the service only allows users to access their own data and not data stored by other clients, the service would need to define a port for each permission domain (user) for the data retrieval function. Thus, for a system with three clients there would be three ports for the "get" functionality, one for each user's data.

3.2. Network Operating System

Sikker requires the existence of a network operating system (NOS) to act as a trusted system-wide governor. The NOS creates the services running on the network, establishes their permissions, and distributes the proper permissions to the proper entities in the system. The NOS is externally reachable such that users are able to start new services on the system and control existing services. While interacting with the NOS, the user is able to specify the structure of a new service in terms of processes and ports. Furthermore, the user is able to create fine-grained sets of permissions (processes and ports) which other services will be able to use. During runtime, services are able to contact the NOS for changes to their own structure and for permission changes. The specific placement, implementation, fault tolerability, and user interface of such a NOS is beyond the scope of this work.

3.3. Authentication

All communication in Sikker is explicitly authenticated to both source and destination. Similar to other networks, processes in Sikker reside at physical locations specified by physical addresses. However, in Sikker, processes are referenced by virtual addresses that specify both the service and the process. When a process desires to send a message on the network, it does not specify its own identity as the source. Instead, Sikker derives its identity, consisting of both service and process, and attaches it to the message.

When specifying a destination for a message, the source process specifies the destination by three things: a service, a process within the service, and a port within the service. Combined, the source and destination specifications are attached to every message transmitted on the network. Sikker guarantees that the message will only be delivered to the specified destination. Receiving processes are able to inspect the source specification in the message to explicitly know the source's identity.

Under the Sikker security model, processes need not be concerned about physical addressing in the network. Processes only use virtual service-oriented identifiers when referencing each other. Sikker performs the translations between virtual and physical addresses needed for transmission on the network. There is no need for name servers in Sikker.

3.4. Static Permissions

In Sikker, each service is given a set of permissions (or privileges) by the NOS with which it is able to communicate on the network. Each process within a service inherits all the permissions of the service to which it belongs. In order for a sending process to be able to transmit a message to a specific destination, the service of the sending process must have permission to access the specified process and port within the specified service. Intra-service messages are treated exactly like inter-service messages. Sikker performs permission checks before messages enter the network for every message.

The total state needed to represent the permissions of any permission scheme is proportional to the product of the number of entities that hold permissions and the number of resources being accessed. In contrast to process-oriented permissions (i.e. permissions given to processes directly), service-oriented permissions are highly scalable. For example, a large singly-operated data center might have millions of running processes but the number of services is only on the order of

hundreds or thousands. By assigning permissions to services instead of processes, Sikker is able to reduce the state needed by three to four orders of magnitude. Service-oriented permissions also make it much easier to logically reason about permissions within the system because developers and operations engineers most commonly think about communication between services, not individual processes.

3.5. Dynamic Permissions

The interaction policies of modern large-scale distributed systems are constantly in flux. Sikker allows processes and ports to be added and removed from services dynamically during runtime. When a new process is created, it inherits all the permissions of the service to which it belongs. Any time the permissions of a given service change, the change is reflected in all processes of the service.

To increase scalability and to fit better with large-scale multi-tenant systems, Sikker contains a mechanism for one-time-use permissions. This system is especially useful for request-response protocols and protocols where one service is allowed to operate on behalf of another service. In Sikker, a process can create a one-time-use permission to be given to any process it has permission to access. A one-time-use permission specifies a service, process, and port as a destination and can only be created using the permissions that the creating process already has. When a process receives a one-time-use permission from another process, it is stored by Sikker in a temporary storage area until it gets used by the receiving process, at which time Sikker automatically deletes the permission. Because a one-time-use permission fully specifies the destination, the process using it specifies the permission by its unique ID instead of specifying the destination as a service, process, and port.

3.6. Example

Figure 4 is shown to illustrate the concept of services and their interactions under the Sikker programming model. This graph diagram shows three services, each with a few processes and a few ports. Solid lines connect services to their corresponding processes and ports. Solid lines also connect processes to their corresponding hosts. As shown, and widely used in practice, processes from the same service and/or different services may overlap on the same host. Dashed lines show the permissions given to services. These lines originate at a service and end at either a process or a port. It is implied that all processes of a given service inherit the permissions of the service to which it belongs. Table 3 describes each service by listing its processes, process locations, and ports. Tables 4a, 4b, and 4c show the permissions of services 1, 2, and 3, respectively. As shown in these tables, service 1 has no permissions and as such can only receive messages from other services. Service 2 has access to itself and partial access to services 1 and 3. Service 3 has access to itself, partial access to service 1, and full access to service 2.

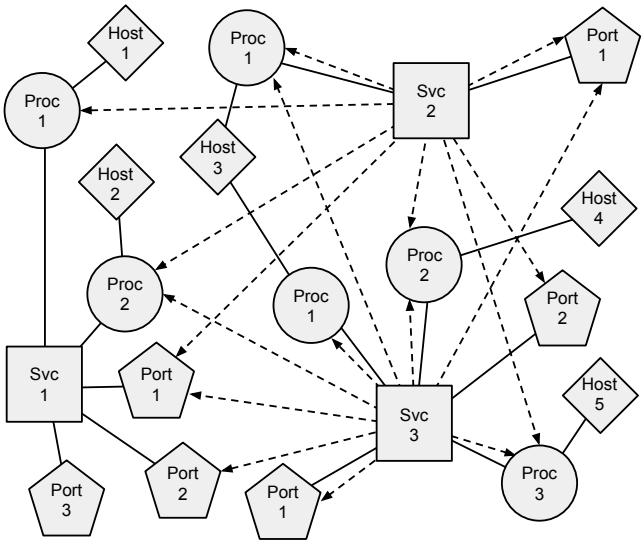


Figure 4: An example service interaction graph

Service	Processes@Host	Ports
1	1@1, 2@2	1, 2, 3
2	1@3	1
3	1@3, 2@4, 3@5	1, 2

Table 3: Service definitions

Service	Processes	Ports
1	-	-
2	-	-
3	-	-

(a) Service 1 permissions

Service	Processes	Ports
1	1,2	1
2	1	1
3	2,3	2

(b) Service 2 permissions

Service	Processes	Ports
1	2	1,2
2	1	1
3	1,2,3	1

(c) Service 3 permissions

Table 4: Service permissions

A successful transaction: For a typical network transaction, assume process #1 of service #2 desires to send a message to process #3 of service #3 using its port #2. Both figure 4 and table 4b show that service #2 has been given access to process #3 and port #2 of service #3. Since all processes inherit their permissions from their corresponding service, when process #1 attempts to send the message Sikker will allow the message to be sent to the destination. Sikker will translate the specified

virtual address into a physical address of the corresponding destination. When process #3 of service #3 receives the message, the message will specify the source of the message as process #1 of service #2. The message will also specify port #2 to be used in the destination process. The message transaction is fully authenticated because both the source and destination are guaranteed of each others identities which are unfalsifiable. Furthermore, the source is assured that Sikker will deliver the message to the specified destination and none else.

A failed transaction: If we take the same example as above and alter it such that process #1 of service #2 attempts to send to process #1 of service #3, this presents a permissions violation. It is also a violation if it attempts to use port #1. In these cases, the source process will construct the message just as it did before, however, the Sikker security system will deny access to the network for this message.

One-time-use privilege: Since service #1 doesn't have any static permissions, we can assume it is a service that only serves other services upon a request. This is a prime example of where dynamic one-time-use permissions work very well. For the sake of this example, assume that service #1 is a large service designed to serve the data storage needs for many services. Service #2 has some data stored in service #1 and needs to retrieve it. Since service #2 has permissions to access service #1, it constructs a request message to be sent to service #1, process #2, port #1. Its also creates a one-time-use permission to be used by service #1 so that it is able to send the response back to service #2. The one-time-use permission is created specifying the destination as service #2, process #1, port #1. Before sending the request message, the Sikker security system checks that service #2 has permission to access the request recipient (service #1, process #2, port #1) and the response recipient (service #2, process #1, port #1), which it does. The message is then sent.

Upon receiving the request message in service #1, Sikker stores the one-time-use permission in a temporary storage area and delivers the message to process #2 with a handle to the one-time use permission. Service #1 retrieves the data requested by service #2 and constructs a response message, however, it does not specify the destination of the message. Instead, it specifies the one-time-use permission to be used. Sikker then sends the message to the specified response recipient and deletes the one-time-use permission. It is important to note that while service #2 specified itself as the response recipient in this example, it could have specified any destination it has privilege to as the response recipient. For example, the response could have been sent to service #3, process #2, port #2.

4. Network Management Unit

4.1. Design

In this section, we present the Network Management Unit (NMU), a network interface controller (NIC) architecture. The NMU is the workhorse of Sikker as it provides each process

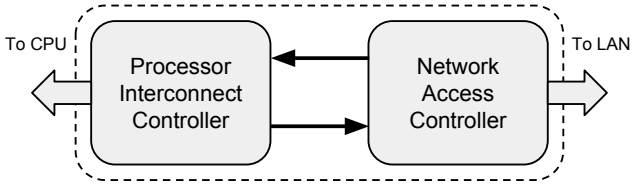


Figure 5: Basic NIC architecture

with high performance network access and implements the security model, described in section 3, under the direction of a network operating system (NOS). The standard NIC architecture, shown in figure 5, is simply an intermediate interface between the network and a host, including the CPU and memory. Thus, in its most basic form, a NIC is just a translation unit that converts messages from one interconnect protocol to another. The NMU can be viewed as an extension to the standard NIC architecture, however, in order for a NIC to be classified as an NMU it must have:

1. A method for efficient interaction between local processes and the network.
2. A method of determining the identity of local processes using the network.
3. A method for receiving and storing Sikker permissions.
4. A method for checking the permissions of outgoing messages and, if neccessary, blocking network access.

To implement high performance network access, from item #1, the NMU implements operating system bypass, commonly referred to as *OS-bypass*. As with most other OS-bypass implementations, the NMU allows a process and the NMU to read and write from each other's memory space directly without the assistance of the kernel.

The NMU's OS-bypass implementation has one major difference compared to other implementations, namely, it uses the memory interface to determine the identity of a communicating process, which fulfills item #2. The NMU contains many virtual register sets upon which the various processes interact with it. This corresponds to a large physical address space mapped to the NMU. When a new networked process is started, the NMU gives the host's operating system the base address of the register set that the process will use. The NMU contains an internal table that maps register set address to process identity. After the process is started, the register set is mapped into the process's memory space and the process is only able to use this register set for interaction with the NMU. The process never tells the NMU of its identity, instead, the NMU deduces its identity from the memory address used to communicate with it. Using this mechanism, it is impossible for the process to falsify its identity.

As mentioned in section 3.2, Sikker requires a NOS to coordinate service interactions and create permissions for the entire network. The NOS coordinates with every NMU in the network, which reside on each host. The NOS is responsible for creating permissions and distributing them to the proper NMUs. The internal data structure of the NMU has been

Index Map: {Service, Process} → {SourceIndex}
Info Map: {SourceIndex} → {Service, Process, ReplyKey, PrivilegeMap, ReplyMap}
Privilege Map: {Service} → {ProcessMap, PortSet}
Process Map: {Process, Address}
Port Set: {Port}
Reply Map: {ReplyKey} → {ReqService, ReqProcess, RecService, RecProcess, RecPort, RecAddress}

Figure 6: NMU internal data structures

carefully crafted such that all variable sized data is represented as nested hash maps[4]. Furthermore, the hash mappings and value placements have been meticulously optimized to keep the hash maps as small as possible in effort to produce low predictable search times. The elements of the NMU's internal data structure are listed in nested form in figure 6. This data structure is the NMU's fulfillment of item #3. For security reasons, the NMU contains its own memory subsystem that is inaccessible by the host's operating system.

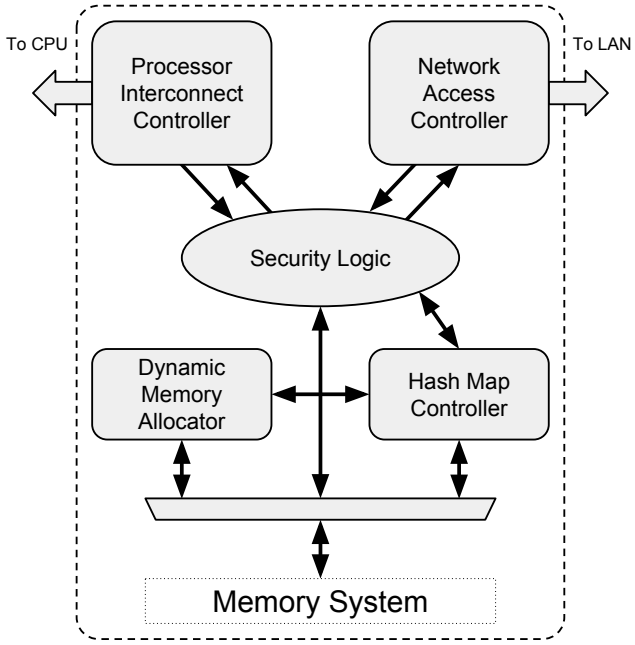


Figure 7: NMU architecture

To implement the NMU's internal data structures efficiently in hardware, the NMU's architecture has been designed as a data structure accelerator specifically for managing nested hash maps and hash sets. The high-level architecture of the NMU is shown in figure 7, which shows that the NMU is made up of 3 main blocks: permissions logic, hash map controller, and dynamic memory allocator. The combination of these logic blocks facilitates the management of the internal data structures.

[4]We consider hash sets the same as hash maps. A hash set is simply a hash map with a zero sized value.

Attached to the memory system of the NMU is the dynamic memory allocator which is a hardware implementation of a coalescing segregated fit free list allocator [4]. This allows both the permissions logic and the hash map controller to create, resize, and destroy dynamically sized blocks of memory. The hash map controller is a hardware implementation of a linear probed open addressing (a.k.a. closed hashed) [25] hash map controller. It connects to the dynamic memory allocator and directly to the memory system. Since the hash map controller handles all hash map and hash set operations, the permissions logic simply issues a set of operations for each NMU function.

As mentioned in section 3, Sikker requires the permissions of every message to be checked before entering the network. The NMU's main purpose is to fulfill this requirement efficiently. For each potential message being sent on the network, the permissions logic issues commands to the hash map controller that traverse the nested data structure to ensure that proper privileges exists. If proper permissions exist, the permissions logic translates the virtual address, consisting of a destination service, process, and port, into a physical network address. The message is then given to the network access controller to be sent on the network. When proper permissions do not exist, the permissions logic rejects transmission of the message and flags the process with an error code in its corresponding register set. This is the NMU's fulfillment of item #4.

To implement the one-time-use permission functionality discussed in section 3.5, the NMU contains a table for each resident process that holds temporary one-time-use permissions. Each permission in this table is indexed by a unique identifier that is given to the process receiving the permission. When the process desires to use one of its one-time-use permissions, instead of specifying the message's destination, the process specifies the one-time-use permission using its unique identifier. At this time, the NMU removes the permission from the table and uses the corresponding destination information to send the message. When a process desires to create a one-time-use permission to be sent to another process, the NMU checks to see if the creating process has the proper permissions to access the specified recipient of the response message.

4.2. Simulation

To explore the design space of the NMU and measure its performance, we developed a custom simulator, called *SikkerSim*, that models all the components of Sikker and the NMU. SikkerSim contains an implementation of a network operating system (NOS) that manages the permissions of all the NMUs on a network. It does this by creating a graph as shown in figure 4 and connecting an NMU to a selectable number of hosts. For each simulated NMU, SikkerSim models the internal logic elements of the NMU as well as various types of memory systems under design consideration. SikkerSim contains a synthetic system generator that loads the NOS with hosts, services, processes, and ports based on configurable

parameters.

The first matter of design space exploration is the size of memory system used in the NMU. We created a synthetic service interaction model to explore the memory size requirements of various distributed system deployments. Given a particular system size, this model expresses the size and number of services as well as the amount of interconnectivity between services. The connectivity parameters and 3 specifications are shown in table 5.

	Sparse	Normal	Dense
Processes per NMU	16	16	16
Processes per service	512	512	512
Ports per service	256	256	256
Service coverage	5%	10%	20%
Process coverage	65%	65%	65%
Port coverage	25%	25%	25%

Table 5: Connectivity parameters for service interaction models

These 3 models aim to model large data centers that utilize many large services with high amounts of interconnectivity requirements. The parameters are equal except for varying the amount service coverage, or the percentage of all services that one service interacts with. For each model, we simulated a loaded NMU on various network sizes ranging from 2k hosts up to 128k hosts and measured the amount of memory needed to store the corresponding permissions. As shown in figure 8, the memory size requirement varies proportionally to the size of the system, number and size of services, and the amount of interactivity between services. For the remainder of our analyses, we'll use the *Dense* model for our simulations since it presents a harder workload for the NMU's memory system.

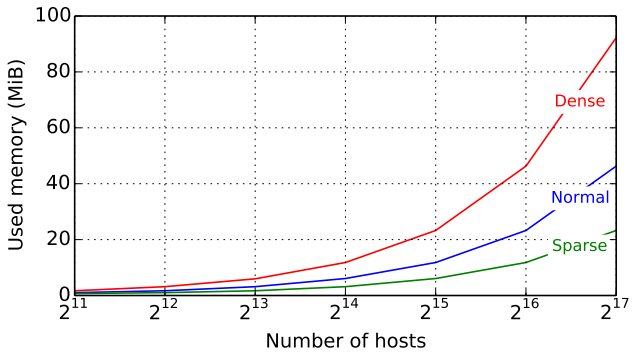


Figure 8: Memory usage vs. System size

The data structures of the NMU present abundant spatial locality to the memory system, and depending on the permission access pattern, significant temporal locality can also exist. SikkerSim contains a configurable synthetic permission access pattern that is placed on each simulated NMU. For each permissions check the permission access pattern selects a source and destination. The source specifies which resident process will be accessing the network. The destination specifies a service, process, and port that the source will be sending the message to.

The worst case access pattern is a uniform random selection across the source and destination possibilities. In this pattern, each permissions check randomly selects a resident process as the source, then randomly selects the destination service, process, and port from the corresponding source service's permissions. This pattern exhibits no temporal locality in the memory system.

The best case access pattern is repeatedly choosing the same source and destination. This pattern exhibits full temporal locality in the memory system. While this pattern is unrealistic for long durations, it is quite realistic for short permission access sequences. Similarly, a slight variant of this pattern would be repeatedly accessing the same destination service, while switching destination process and/or port.

Since both the worst and best case access patterns are somewhat realistic, we architected the synthetic permission access pattern in SikkerSim to reflect two common attributes that control the temporal locality in a realistic way.

Repeated Domains: The first attribute configures the amount of repeatability at each step of the selection process for the source and destination. There are several aspects that makes this realistic in practice. For instance, it is common for a process using the network to interact several times with the network interface before another process has the chance to. This can be caused by CPU thread scheduling or application-level network bursting. Also, it is common for a process to send multiple back-to-back messages to the same destination service or even the same destination service and process. The result is a higher level of temporal locality simply due to repeated accesses in the same domain.

Hot Spot Domains: The second attribute configures the selection process when the synthetic permission access pattern chooses a new source process, destination service, destination process, and destination port. This selection process can optionally choose using a uniform distribution or a Gaussian distribution. The uniform distribution models network traffic that is irregular and unpredictable while the Gaussian distribution models network traffic that contains hotspots both in terms of the source and destination.

Using these controllable attributes, we used SikkerSim's synthetic permission access pattern to create four access patterns that we use to benchmark the performance of the NMU. They are as follows:

- **Uniform Random (UR):** Every permissions check selects a source and destination with a uniform random distribution.
- **Uniform Repeated Random (URR):** Same as "Uniform Random", except that portions of the source and destination are re-used a configurable number of times.
- **Gaussian Random (GR):** Every permissions check selects

a source and destination with a Gaussian random distribution.

- **Gaussian Repeated Random (GRR):** Same as "Gaussian Random", except that portions of the source and destination are re-used a configurable number of times.

5. Performance

Since the NMU can be viewed as an extension to the standard NIC architecture, we quantify its performance by measuring the additional latency incurred by performing its operations. As figure 8 shows, a tremendous amount of permissions are able to fit in a relatively small amount of memory, even for very large system sizes and complexities. The logic of the NMU can be attached to any memory system and the performance of the NMU is largely affected by the type and size of the memory system chosen. To narrow down the design space, we used SikkerSim to model the NMU connected to various types of memory systems spanning from a single SRAM to a multi-stage cache connected to DRAM. Cacti [15] and DRAMSim2 [18] were used in connection with SikkerSim to produce accurate timing results for each case.

For the sake of performance analysis, we've chosen a memory system design that yields high performance while not incurring excessive cost. This design attaches the NMU logic to a memory system containing two levels of cache and a DRAM main memory. The first cache level (L1) is an 8-way set associative 32 kiB cache. The second cache level (L2) is a 16-way set associative 4 MiB cache. While we use the standard DRAM latencies as reported by DRAMSim2, the main memory of the NMU is small enough that it can be implemented as an eDRAM on the same die, a secondary flip-chip DRAM die, or a traditional off module DRAM.

5.1. Fixed Permissions

This section analyzes the NMU's performance while checking the permissions that are statically placed within the NMU. The operation of a permissions check is a traversal of the internal data structures listed in figure 6. When a resident process communicates to the NMU, the NMU inherently knows its *SourceIndex* from the memory address used for communication. That is then used in the first hash map search operation where the NMU searches the *InfoMap* for the corresponding source process's information. It then searches the process's *PrivilegeMap* using the specified destination service. Next the *ProcessMap* and *PortSet* are searched using the specified destination process and port, respectively. In total, there are four hash maps searched, but only three operations are sequential.

Figure 9 shows the average (mean) latency incurred by a single NMU permission check for each of the four permission access patterns described in section 4.2. As expected, the UR and GRR patterns represent the best and worst patterns, however, the UR pattern is only 25% worse than the GRR pattern. Figures 10a and 10b show the mean, median, 90th percentile, and 99th percentile latencies for the Uniform Random (UR)

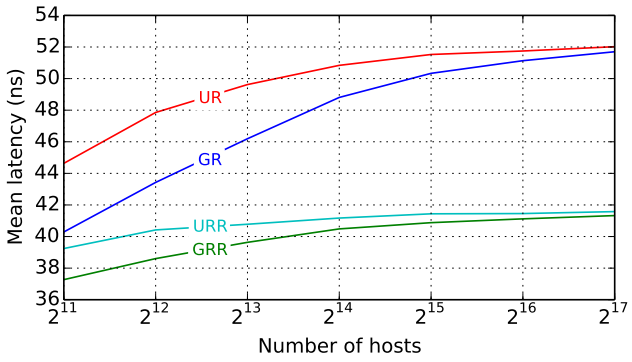
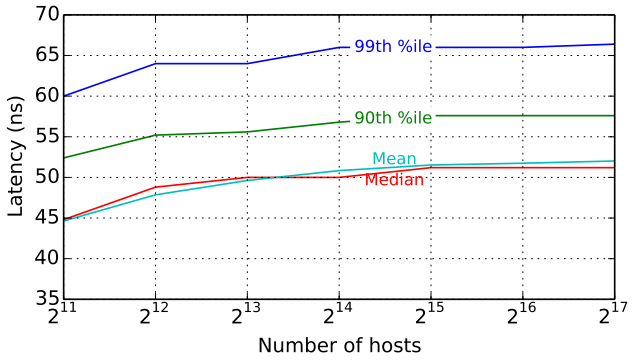
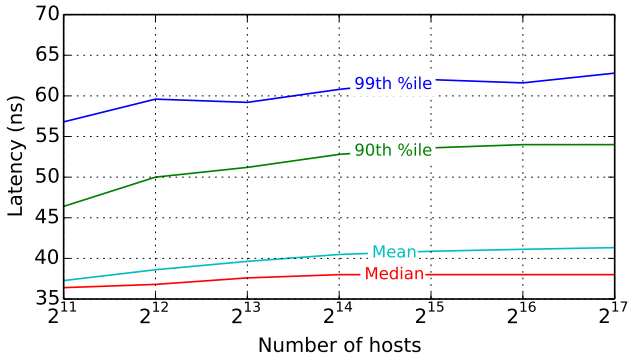


Figure 9: Mean latency of all 4 access patterns.



(a) Uniform random (UR) access pattern



(b) Gaussian repeated random (GRR) access pattern

Figure 10: Permissions check latency overheads

and Gaussian Repeated Random (GRR) access patterns. These figures show that even under extreme conditions, the NMU adds negligible overhead to network transactions. Under the worst access pattern (UR) and on a large system size (131072 hosts), the 99th percentile latency overhead is only 66ns.

5.2. One-Time-Use Permissions

The one-time-use permission feature of the NMU uses the same underlying mechanisms as tested in section 5.1 except that some operations bypass the *PrivilegeMap* and use the *ReplyMap* instead. When sending a message along with a

one-time-use permission, the NMU performs two permissions checks as described in section 5.1, one for the specified message destination, and one for the specified one-time-use permission. The expected latency for the this procedure is equivalent to the results found in section 5.1 because the two permissions checks are independent and can be performed in parallel.

When receiving a one-time-use permission, the NMU stores the permission in the corresponding process's *ReplyMap* as listed in figure 6. In terms of latency analysis, the only procedure that must be performed before the message is delivered to the process is determining the unique identifier that the one-time-use permission will be assigned in the *ReplyMap*. Fortunately, this is can be determined by only two hash map search operations, searching the *IndexMap* then the *InfoMap* to produce the *ReplyKey*. The one-time-use permission can be queued for storage while the message is given to the process.

When using a one-time-use permission that had previously been stored in a *ReplyMap*, the expected latency is again equivalent to the standard permissions check process. While the standard process performs a *PrivilegeMap* lookup then a *ProcessMap* and *PortSet* lookup, the one-time-use permissions check performs a remove operation on the *ReplyMap* to retrieve and remove the permission. The number of sequential hash map operations is reduced by one.

Relative the to standard permissions checking process, using one-time-use permissions incurs the same latency overheads with negligible differences.

6. Security and Isolation

The NMU implements all the security and isolation features of Sikker as discussed in section 3. This includes source and destination authentication, virtual-physical address translation, sender-enforced service-oriented permission checks, and permissions management. Sikker's security model is more straight forward than other approaches because the policies on which it is established are dervived directly from the applications themselves, instead of being tied to network transport mechanisms.

Sikker's sender-enforced isolation mechanism removes the ability for denial-of-service attacks between services that don't have permission to each other. This isolation mechanism creates a productive programming environment for developers since they can assume that all permissions checks were performed at the sender. In this environment, developers are able to spend less time protecting their applications from the network and more time developing core application logic. For services that do have permission to access each other, upon detecting an attack a service can immediately remove itself from the sending service's permissions.

The Sikker application model uses individual endpoint machines to host the processes of the various services (hence the name *host*). As such, Sikker relies on the host's operating system to provide process level isolation between the processes resident on that host. In general, Sikker assumes that the various host operating systems within the network are unreliable. For this reason, the NMU was designed to be explicitly controlled by a trusted network operating system rather than individual host operating systems. In the event that a host's operating system is exploited by a resident process, the process might be able to assume any of the permissions that have been given to *all* processes on that host. This is a large improvement over current systems that utilize the host operating systems for security. In those systems, an exploited operating system might be given access to anything in the entire network, not just the permissions resident on that host.

In Sikker, if a host's operating system cannot be deemed reliable and the tenants of the system are potentially malicious, it is recommended to co-locate processes only where an attack would not prove detrimental if one resident process gained access to another resident process's permissions. For all intents and purposes, under the Sikker application model, processes and virtual machines are synonymous. Instead of mapping only a single process to one of the NMU's virtual register sets, a hypervisor can map this address space to a virtual machine.

7. Conclusion

In this paper we have introduced a new distributed system architecture, called Sikker, with an explicit security and isolation model designed for large-scale distributed applications that run in data centers, supercomputers, and cloud computing facilities. Sikker is designed to be a high performance and scalable solution to enforce the permissions of the complex interactions of modern distributed applications. Sikker's service-oriented application model is an intuitive and effective alternative to network-derived ACL systems as it was derived directly from the interactions of current applications.

We've presented the Network Management Unit (NMU), a network interface controller that efficiently enforces the permissions scheme of Sikker. Working under the direction of a network operating system, the NMU provides network isolation through enforcing permissions at the sender and provides security through its inherent implementation of the principle of least privilege as well as source and destination authentication. Even when compared to the performance of the highest performing supercomputers, the NMU induces negligible overheads for network transactions.

Sikker and the NMU enable a new generation of distributed systems performing like supercomputers while operating with inherent service-oriented security and isolation. This new generation of computing supports large-scale multi-tenant computing platforms where system architects and application developers are able to access remote data quickly and spend less time writting tedious and error-pone security checks.

References

[1] Amazon.com. (2014) Amazon web services (aws). Available: http://aws.amazon.com

[2] T. S. Blog. (2013) Twitter site issue. Available: http://status.twitter.com/post/60295900552/twitter-site-issue

[3] P. J. Braam *et al.*, "The lustre storage architecture," 2004.

[4] R. Bryant and O. David Richard, *Computer systems: a programmer's perspective*. Prentice Hall, 2003.

[5] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The ibm blue gene/q interconnection network and message unit," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–10.

[6] J. Ciancutti. (2010, December) 5 lessons we've learned using aws. Available: http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

[7] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "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*. IEEE Computer Society Press, 2012, p. 103.

[8] O. Foundation. (2014) Openstack neutron. Available: https://wiki.openstack.org/wiki/Neutron

[9] A. Hern. (2014) Facebook unavailable during longest outage in four years. Available: http://www.theguardian.com/technology/2014/jun/19/facebook-unavailable-longest-outage-four-years

[10] G. Inc. (2014) Google cloud platform. Available: http://cloud.google.com

[11] R. Johnson. (2010) More details on today's outage. Available: https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919

[12] Joyent. (2014) High-performance cloud computing. Available: http://www.joyent.com

[13] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks," *draftmahalingam-dutt-dcops-vxlan-01. txt*, 2012.

[14] Microsoft. (2014) Azure: Microsoft's cloud platform. Available: http://azure.microsoft.com

[15] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[16] Netflix. Netflix cloud architecture. Available: http://www.slideshare.net/adrianco/netflix-velocity-conference-2011

[17] V. Rajaravivarma, "Virtual local area network technology and applications," in *Southeastern Symposium on System Theory*. IEEE Computer Society, 1997, pp. 49–49.

[18] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[19] Salesforce.com. (2014) Heroku. Available: http://www.heroku.com

[20] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[21] M. Sridharan, K. Duda, I. Ganga, A. Greenberg, G. Lin, M. Pearson, P. Thaler, C. Tumuluri, N. Venkataramiah, and Y. Wang, "Nvgre: Network virtualization using generic routing encapsulation," *IETF draft*, 2011.

[22] T. J. Team. (2014) Postmortem for outage of us-east-1. Available: http://www.joyent.com/blog/postmortem-for-outage-of-us-east-1-may-27-2014

[23] M. Technologies. (2014). Available: http://www.mellanox.com

[24] M. Technologies. (2014) Infiniband performance. Available: http://www.mellanox.com/page/performance_infiniband

[25] A. M. Tenenbaum, *Data structures using C*. Pearson Education India, 1990.

[26] B. Treynor. (2014) Today's outage for several google services. Available: http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html

[27] Twitter. Finagle: A protocol-agnostic rpc system. Available: http://4.bp.blogspot.com/-riUycEXusDA/TlU85PnE_dI/AAAAAAAAAB4/ZDjz80Qu7NU/s1600/Finagle%2BDiagram.png

[28] VMware. (2012) Network i/o latency on vsphere 5, performance study. Available: http://www.vmware.com/files/pdf/techpaper/network-io-latency-perf-vsphere5.pdf

[29] VMware. (2014) Nsx. Available: http://www.vmware.com/products/nsx

[30] F. Xu, F. Liu, H. Jin, and A. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, Jan 2014.