Node Containment In the Seattle Testbed

Cosmin Barsan
University of Washington

Abstract

Testbeds are a critical part of system evaluation, but usually lack in diversity and scale. To solve these issues we have developed Seattle, a testbed that obtains users in a community-driven, peer-to-peer manner. This effectively allows anyone to contribute or use resources on the testbed. However, given the openness of the testbed, it is necessary to provide a way to protect this service from malicious use, such as distributed denial of service attacks, SPAM, and participation in networks used to distribute illegal material. In this paper, we describe a containment system that allows machine participating in Seattle to enforce policy decisions about its interactions with systems outside of the testbed. To do this, each machine communicates with a set of servers under our control to build a complete table of member information. Each machine caches fragments of the table, and makes filtering decisions locally according to its policy. Whenever the status of a machine running Seattle changes, we use a gossip protocol to disseminate information describing the change that has occurred. We have found this solution is scalable and produces accurate filtering decisions. Our results indicate that there is very low delay, of less than a millisecond, associated with deciding whether to allow or deny traffic. Our model indicates that one server will be able to support approximately 100,000 clients, where each client machine has an average of 8 userkeys. We estimate that a farm of 16 servers will be capable of supporting on the order of 1 million nodes, each with 4 userkeys.

Introduction

Modern distributed systems and networking research utilizes testbeds to evaluate new ideas and concepts in practical settings. For instance, peer-to-peer systems are a popular research topic where new ideas have been validated on testbeds. However, existing testbeds lack several properties that are inherent in real world distributed systems. Testbeds lack the diversity and scale of existing distributed systems. Many popular testbeds, including PlanetLab, do not have different operating systems or architectures of devices. Testbeds also lack the network diversity of real users as they do not usually possess mobile nodes, nodes behind NATs, or nodes with wireless connectivity to the Internet. The scale of testbeds is also woefully inadequate to evaluate modern distributed systems. It is not uncommon for a modern distributed systems to have around 1 million nodes, while modern testbeds usually possess under 1 thousand nodes.

In the Seattle platform, we construct a testbed by using a portion of the resources on end user devices around the world. We run a lightweight virtual machine on end user devices that safely executes untrusted code. This allows end user devices to participate in our testbed without significant risk or penalty. Since our testbed software runs on diverse machines, our testbed also has operating system and network diversity. The current scale of our testbed is between seven hundred and two thousand computers, depending on the metric used to count the size. Our testbed is open for any developer to use and so presents unique security problems.

While the Seattle testbed presents unique opportunities for developers, the public availability of machines on this testbed presents us with the unique challenge of protecting this testbed from malicious use. Without technical mechanisms to protect against abuse, attackers can exploit the availability of the resources to create bot-nets, distribute SPAM, participate in illegal content distribution, or conduct distributed denial of service attacks. In this paper, we propose a system that will allow us to efficiently block machines in a testbed from sending or receiving traffic outside of the testbed. Machines can freely join the testbed at any time. This allows machines that want to communicate with testbed machines, perhaps to provide a service, to opt-in by joining the testbed.

The primary goal of this work is to ensure that the owner of a machine can control communication with machines outside of the testbed. The simple policy that we describe in this paper allows them to choose to allow outgoing traffic to non-testbed machines, incoming traffic to non-testbed machines, both, or

neither. More complex policy mechanisms could also be constructed on top of this, but we defer this to future work. There are also several secondary goals:

- It is important that there is no hidden delay in allowing or denying traffic as this may interfere with measurements or performance of applications on machines.
- The implementation must be secure against an adversary who wishes to circumvent the policy. For example, it should not be possible for a machine to opt-in another machine.
- The system must scale to over one million machines (the expected size of Seattle).
- The per-client overhead and infrastructure required must be realistic.

Our solution is to use a set of trusted central servers to host a service to manage addresses of machines that have opted in to receiving Seattle traffic. We call this service the Seattle Node Directory Service. All of the machines in the Seattle testbed are responsible for registering (and periodically re-registering) their IP address to continue receiving Seattle traffic. Within this paper, we will refer to machines that communicate with the Seattle Node Directory Service as **clients**.

Seattle machines make decisions about what traffic to allow or deny locally based upon their policy and their local cache of state from the Node Directory Service. The cache that is maintained, usually isn't a complete and up-to-date list of all nodes in the testbed because this isn't scalable. Instead each machine caches only those machines that have virtual machines that are being used by the same researcher. In most cases, two testbed machines will communicate only if the same researcher is using both systems. This allows a machine in the testbed to avoid caching the entire list of testbed IP addresses. However, we also white-list machines that provide services to the testbed. To protect for most malicious use scenarios, it will be sufficient to only restrict outgoing traffic. However, we will leave it to the administrator of each machine individually to decide what policy each node will implement. The policy choices available will be: filter outgoing traffic, filter incoming traffic, filter outgoing and incoming traffic, or no filtering.

If a program attempts to send a packet to an address that is not in the machine's cache, the virtual machine raises an exception but then sends a packet to the directory service asking if the address is valid. If the directory service replies positively, the address is added to the node's cache. The directory service will reply negatively if the client with the address in question did not perform a query registration. To maintain the node's cache of addresses as current as possible, the Seattle Node Directory Service sends out update packets describing changes to the list of addresses registered for each user key. The update mechanism uses a gossip protocol, with each node forwarding the update packet to a random set of other nodes. For each user key the server will generate a separate update packet containing address changes relevant to just that user key.

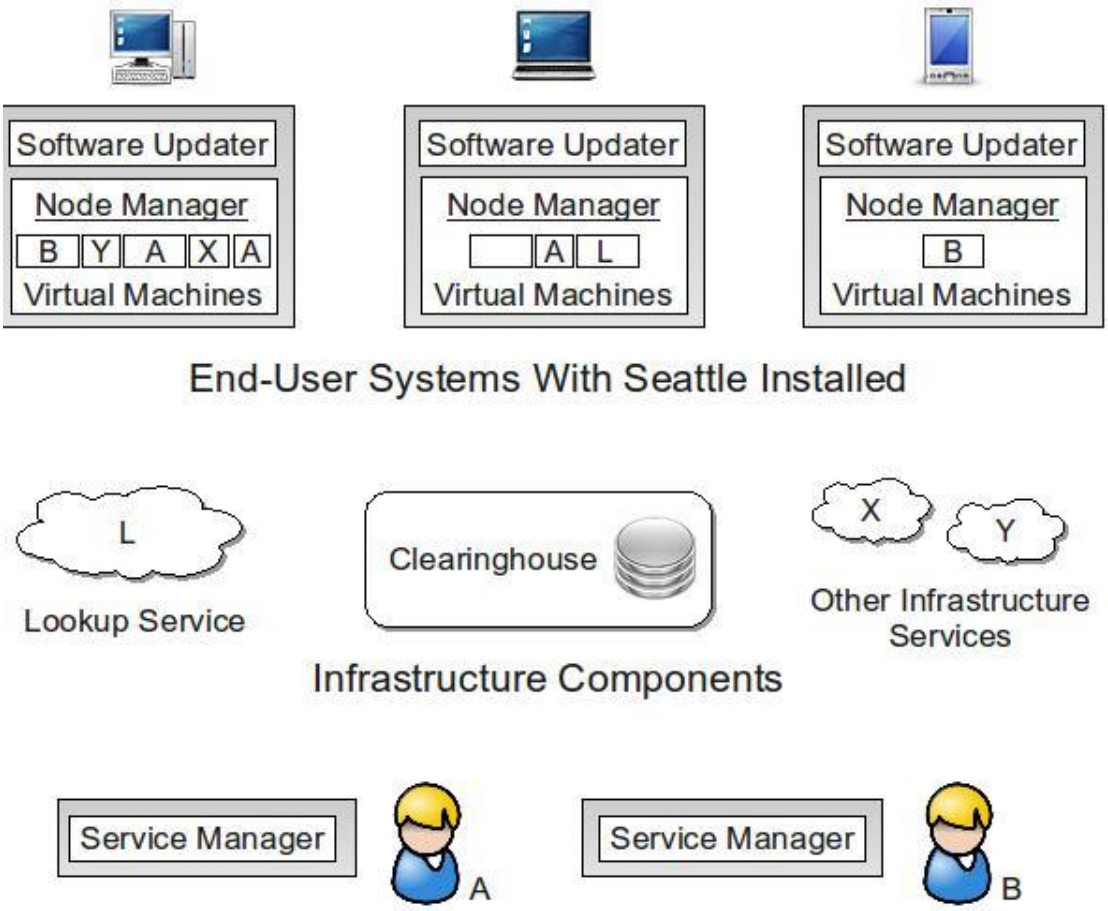
Unless traffic between the nodes and the Seattle Node Directory Service is protected, an attacker may be able to send false updates to nodes. It is therefore necessary to guarantee integrity of the packets exchanged between the directory service and nodes to protect against man-in-the-middle attacks. This is achieved by including a signed hash and a timestamp with each packet.

About Seattle and Repy

Seattle [1] is a platform that allows users to run code remotely on sandbox virtual machines. It has potential to be useful to researchers and educators as it provides a simple and low-cost way to run code on real systems all around the world. Seattle virtual machines are run on user-donated systems running either Windows, Mac-OS, Linux, or BSD. Each donated machine runs a number of such virtual machines, which we refer to as **vessels**. As mentioned in the introduction, we refer to each machine that contributes vessels as a **node**. Note, while all nodes will be clients, not all clients of the Seattle Node Directory Service will necessarily be nodes.

The most common use of Seattle involves a user first signing up on the SeattleGENI[3] site. This site provides an interface for account management, as well as for the management and reservation of vessels. Immediately after signing up, the user is granted 10 vessel credits, which limit the number of vessels that can be checked out at any one time. Each vessel credit allows for one vessel to be checked out at any given time. Signing up to use Seattle is opened to the entire public.

In this section we provide a brief overview of Seattle, and discuss the services and components that are relevant to the topic of this paper. For more information on the Seattle platform see the article "Building a Scalable and Diverse Testbed."



This diagram was taken from the article "Building a Scalable and Diverse Testbed" [6]. Note that each computer runs a Node Manager service, that manages the operation of each virtual machine running on the system. In this section, we will only discuss the aspects of Seattle that are relevant to the topic of the paper. For additional information about the Seattle Platform, see https://seattle.cs.washington.edu/wiki.

Seattle Platform Security:

Seattle is driven by user donated resources. A computer administrator can choose to donate a portion of a computer's resources by going to the Seattle home page and downloading the installer for the respective operating system. After Seattle is installed, it will automatically start up every time the machine is rebooted. In order to encourage users to donate resources to Seattle, for every machine a user donates resources on, the user is granted an additional 10 vessel credits on his/her Geni account. If a user has N credits, he or she is able to have up to N vessels reserved at a time.

When a user donates resources on a machine, strict controls are put in place to ensure that code executed in the sandboxes will not affect local system performance or cause security concerns. The total resource consumption of Seattle on a machine is limited to 10%. This is split among the vessels on the system. If any vessel attempts to exceed its resource limits, it will be paused as necessary by a monitoring process. In addition, each sandbox is limited to using only certain ports for communication. There are limits on memory use and disk use on each vessel. If code executing on a sandbox attempts to exceed these, the code will be stopped by the monitoring process. Code running in a sandbox is limited to writing files only its working directory (the directory corresponding to the vessel). More details about the security restrictions that make a vessel's execution safe can be found in other sources [1,6].

It is important we recognize that there is no restriction in place to prevent the owner of the machine from tampering with the Seattle vessels installed on their machine. For this reason, we cannot trust any Seattle node, as it may be compromised.

Repy Language:

A restricted form of python is used to write code that executes in Seattle vessels. This language is called **repy**. Standard python libraries are not available in repy. Instead, repy code can make use of provided Seattle libraries [2]. The 'import' statement is restricted, but instead users can use the dylink module [7], which provides an implementation of dynamic linking.

In addition to providing dynamic linking, the dylink module has the capability of generating a virtual namespace within which code is subsequently executed. A developer is able to transparently remap API calls accessible to the virtual namespace. We will be using this functionality to integrate our containment work within the Seattle platform, such that the appropriate network API calls, per policy selection of the host administrator, are remapped to functions we define that implement the filtering functionality desired.

There is a restrictions file that places limits on the privileges of each vessel. The restrictions file specifies which ports, how much disk space, how much memory, and how much cpu the vessel is allowed to use. It can also restrict the number of events, number of open connections, and types of file operations the vessel can perform. When launching a repy program, an accompanying restrictions file must be specified that will limit the activities the repy code may perform.

Repy code is portable on Linux, Windows, BSD, and Mac computers. In addition, while repy code is run within virtual machines on the Seattle platform, it is also possible for a user to run their repy code outside a vessel with no restrictions, provided they have appropriate privileges on the machine. The Seattle Node Directory Service was written in repy for these reasons. We are able to use any of the supported operating systems mentioned above to host the Seattle Node Directory Service.

Node Manager:

A node manager service runs on each node. The node manager provides APIs to allow users to execute operations on the vessels they have access to. Such operations include resetting vessels, checking the logs, uploading/downloading/deleting files, resetting/stopping vessels, and starting programs. In order to integrate our containment system into the Seattle framework, we will change the node manager such that when starting vessels, it uses the dynamic link module described above to remap the selected network API calls (openconn, waitforconn, sendmess, and recvmess) to versions of these functions that provide the filtering functionality. As mentioned before, the level of filtering on each node will depend on the policy selected by the host administrator. For example, if the host administrator chooses to only filter the sending of traffic, we will only remap the openconn and sendmess API calls. When an administrator installs Seattle and decides on the filtering policy to use, we can specify this choice through a setting in the node manager's configuration file.

Threat Model

In designing this containment system, we first determined what entry points a malicious user can exploit in our system. We assume an attacker is capable of spoofing the source IP addresses of UDP traffic. We assume that an attacker cannot spoof the source IP of TCP traffic. TCP traffic is harder to spoof because the design of the TCP protocol would require an attacker to first intercept and modify the initial handshake packets, and also predict the sequence id numbers on subsequent traffic. Though possible, most attackers will not have this capability. With this in mind, we require nodes that register with the Seattle Node Directory Service to first do so through TCP to ensure the source address of the registration request the server sees is in fact the true address of the client.

In addition, Seattle nodes cannot be trusted. Nodes are run on donated resources, and we must recognize any node can be compromised. For this reason, the Seattle Node Directory Service signs every packet it sends. The containment service we have built relies on nodes helping distribute messages from the server, but no node will create a packet and send it to another node for the purpose of containment functionality.

We do not expect our containment system to work on compromised nodes, but it is a requirement that compromised nodes do not prevent our containment system from working on non-compromised nodes. For this reason, our system is designed to work in situations where there exist non-cooperative nodes. Such nodes are not able to block cooperating nodes from receiving information in a reliable way. Even if packet delivery is blocked intermittently, we have a fallback mechanism which allows cooperating nodes to continue operating (see design section for details).

Design

For the purpose of containing vessel communication within Seattle, we use a distributed set of servers under our control. We refer to the service offered by these servers as the **Seattle Node Directory Service**. This system is based on the idea that nodes which share one or more userkeys are more likely to communicate with each other. Each node has a cache of addresses that the node's vessels are likely to communicate with. To minimize performance impact, we try to pre-emptively load addresses of nodes that share userkeys with the local node into the node's cache.

In this section, we will first discuss the setup of the Seattle Node Directory Service. We will then explain how the Seattle Node Directory Service tracks membership of addresses, and the client interaction (i.e. registration) involved in this process. Next, we discuss of the Seattle Node Directory Service uses a gossip protocol to pre-load relevant potions of its membership table to Seattle Nodes. In the Backup section, we discuss the implications of a Seattle Node Directory server failing and the safeguards in place to minimize the impact. Finally, in the security section, we discuss measures we have taken to make our containment system resistant to malicious parties.

Farm Setup:

The Seattle Node Directory service is a distributed system that can either be run on a single system, or on multiple systems. To deploy a Seattle Node Directory Service farm, we first must generate a key pair for each machine. Next, we create a file that specifies the IP address, port, and public key of each participating machine.

We next determine the range of keys that will be supported by each server. A key range is specified in terms of a lower bound and an upper bound, where each of these values may have at most 3 digits (positive numbers only). For example, the key range that will cover every userkey is 0-999. To verify if a userkey falls into a particular keyrange, we first hash the userkey to a 3-digit number, then compare this result against the bounds of the respective keyrange.

Finally, we start the Seattle Node Directory Service on each machine in the farm, specifying the communication port, key pair, and key range as arguments for each server. The Seattle Node Directory Service is written in repy. This gives us flexibility in deciding on which operating system to run the service on. In addition, it also gives us the capability to deploy the service on a set of Seattle vessels controlled by us, should we wish to do this in the future.

On start-up, each server parses the provided list containing the addresses of Seattle Node Directory Service machines and their userkeys. This information is saved in a table. Whenever a server receives communication from another Seattle Node Directory Service server, it verifies the signature and source address of the packet against the information saved in the table (the IP, port, and public key). If there is any discrepancy the packet is ignored. During startup, each server sends an announcement packet to every other server that was found in the provided list. This packet specifies the sender's key range, and also serves to inform the other servers that the sender is online. On receiving an announcement packet, a server saves the sender's keyrange information in a table and replies with its own keyrange information. This allows all servers to keep track of which keyranges are supported by each server. This becomes useful as it allows clients to be able to request keyrange information from any Seattle Node Directory Service server.

Keyrange assignments on servers are not static. The keyrange of a server may be changed, and affected clients will detect if their keyrange information is out of date, and request new keyrange data. If a single server fails, it is not necessary to restart the entire farm, only that particular server, which on startup will synchronize keyrange information from the other online servers.

Membership Tracking:

The Seattle Node Directory Service tracks node membership in two ways, by userkey and by node address. After the client retrieves the key range information from any of the Seattle Node Directory Service servers, it can use the information to determine which server to contact for a particular request. Each node performs two types or registrations, userkey (corresponding to userkey membership tracking), and query (for address-based lookups).

To perform userkey registration, a client node first determines the combined set of userkeys from all the vessels on the node. It then takes the sha hash of each userkey. We refer to this hash as the cnc-userkey. When registering, clients will communicate the cnc-userkeys rather than the raw keys to the

Seattle Node Directory Service. This is done for performance reasons. Next, the client will group the cnc-userkeys by keyrange, such that every cnc-userkey in a given group is covered by the same keyrange. Note, as mentioned earlier, we verify if a cnc-userkey is in a given key range by first taking the short hash of the key, then checking if the value falls between the lower and upper bound of the key range. Finally, one registration request per cnc-userkey group is sent to the appropriate server. The server processes such requests by storing the source address, source port, and time-stamp of registration in a dictionary indexed by cnc-userkey.

Query registration is similar to userkey registration, but instead of registering the local node as an asset of a particular cnc-userkey, we register the local node as a member in Seattle. To do this, we first determine which server to contact, by checking each keyrange against the short hash of the local IP address. The matching keyrange corresponds to the server that is responsible for handling query registration of this node. We send a registration request as before to this matching server, but instead of a cnc-userkey, we specify a string token that the server will associate with query registration. For the purposes of our system, we optimize this by allowing this token to be grouped with cnc-userkeys, thus potentially lowering the number of registration requests the node sends. Upon detecting a query registration token in a registration request, the server checks the source address of the request. The server checks the short hash of the source IP address and verifies it is covered by the local supported keyrange. If so, the server saves the source address in a dictionary separate then that used for userkey registration. We refer to this dictionary as the query table. The query table is indexed by IP address, and each entry contains information on the time-stamp of registration, as well as the source port.

For example, consider three servers, S1,S2,S3 with keyranges 0-332, 333-665, 666-999 respectively. Consider a client that has three userkeys, A, B, and C, where A and B fall into the 333-665 keyrange and C falls into the 666-999 keyrange. Suppose the client's IP address falls into keyrange 666-999 for the purpose of query registration. Then the client will send two requests each time it registers, one to server S2 and one to S3.

This diagram illustrates that a client may need to register with more than one server, but not necessarily every server in the Seattle Node Directory service. In this case, the client registers with servers S2 and S3, but does not resgister with S1.

User machines hosting Seattle are expected to be shut down and started up frequently. Since nodes hosted on these resources will come online and drop offline frequently, we require nodes to re-register about every 2 minutes. This number was chosen to provide a balance between performance, and accuracy of information. A longer interval between registrations implies less traffic for the servers to handle, but it will also take longer to detect if a node has gone offline.

It is essential for our system that the IP of the registering machine cannot be spoofed. If we do not satisfy this security requirement, it would be possible for malicious users to register an arbitrary IP address to receive Seattle communication, thus defeating the purpose of the service. For this reason, client nodes must use TCP when they first register. Whenever a client registers with a Seattle Node Directory Service server, the server replies with a randomly generated numerical 20 digit renewal key. The client node may then perform subsequent registrations with the server using UDP, as long as it provides the correct renewal key. Using UDP to renew registration allows us to gain a significant performance benefit, with little security risk. As we show in the Model of TCP-only Registration Load section under Evaluation, TCP-only registration would be too expensive to scale adequately.

Update Dissemination:

The Seattle Node Directory Service tracks the membership of addresses through the userkeys they provide at registration. When a node performs userkey registration, for instance specifying userkeys A and B, the server lists the address of the node as a member of the group A and as a member of group B. Essentially, through userkey registration, the node joins several groups, each group corresponding to a userkey. As part of registration, the node is enrolled to receive updates, information describing any changes that occur to any groups the node belongs to, such as other nodes joining or leaving. The update mechanism is the primary way we pre-load addresses into the node's cache that are likely for the node to communicate with. Whenever a membership change occurs within a group, an update packet is pushed out within 10 seconds. The exact delay depends on the amount of membership changes. If a sufficient number of changes occur such that the information will fill an update packet, the update is pushed as soon as this limit is reached. Otherwise, the packet is sent after 10 seconds from when the triggering membership change occurred.

During an update push, the update packet is sent to 4 nodes, selected at random from the resources of the respective userkey. Each of these 4 nodes send the update to 4 other nodes, selecting the addresses at random from the node cache under the respective userkey. To prevent load from accumulating, a client will only forward a given update packet to 4 other nodes the first time it receives it. If the client receives the same packet subsequent times, it is ignored. We choose to use a gossip protocol to disseminate this information for performance reasons. This requires low overhead. In addition, it has strong tolerance for failure, which is important because we cannot rely on all nodes reliably receiving and sending communication. With this mechanism, it is expected some nodes will not receive the update packet. To manage such cases, each update packet is stamped with an id, and nodes are thus able to recognize if they have missed one or more packets. If a node recognizes it has missed one or more update packets, it attempts to retrieve them directly form the Seattle Node Directory Service. The Seattle Node Directory Service caches the most recently sent update packets, and is able to provide the updates requested the majority of the time. If a node requests an update and the Seattle Node Directory Service fails to locate it in the cache (which happens in cases where the update packet is extremely old), the node will clear the cache entries for the respective userkey and retrieve fresh data from the server.

Each client processes update packets by applying the changes describes (addresses added and addresses deleted) to the local cache. A given update packet is applied only if it is more recent than the last update packet that was applied for the respective userkey. Update packets are also verified against a hard timeout. To protect from man-in-the-middle attacks, update packets are signed by the servers, and clients verify each packet before applying changes.

For the purpose of performance, IP addresses and ports are encoded in update packets, using 4 bytes to encode each IP and 2 bytes to encode each port. This allows us to pack larger amounts of data into each update packet, thus reducing traffic load.

Backup:

Prior to starting the Seattle Node Directory Service, we assign a backup server for each server through a configuration file. We do not have idle servers standing by to pick up load in the event of failure, but rather each server in the Seattle Node Directory Service acts as a backup for another server. Each backup server is permitted to handle requests from clients that are trying to register using keys that fall in the keyrange of the primary server. By default, clients attempt to use the primary server for each keyrange. In case contacting the primary server fails, clients will switch automatically to the backup server for the respective keyrange. If the backup server receives registration requests with userkeys that fall in the primary server keyrange, it will process them in the same way as the primary server and disseminate update packets.

In order to ensure data on the primary server is mirrored on the backup server, we use update packets. Whenever a server sends an update packet to clients, it also sends a copy to the server's backup. On receiving an update packet, a server applies the changes to its table, without disseminating updates for these changes.

We chose to use this method to backup our servers for performance reasons. Update packets are relatively cheap to send, and it is important to our scalability that we keep sent/receive traffic low.

In the event of backup failure or large scale failure, the short term impact on existing experiments will be minimal. The cached addresses on local nodes will remain in place until the server is restarted.

However, until the servers are back online, no changes in node membership will be processed, which would be problematic for starting new experiments.

Security:

All communication sent by the servers is signed. This is intended to protect from man-in-the middle attacks that would modify traffic to falsely convince nodes that arbitrary addresses are registered. Timestamps are also included in these messages to ensure that attackers are unable to replay old traffic to their advantage.

Server to server communication is also strictly controlled. A server will only process a request from another server if it recognizes its public key in the local table. We do this to ensure all servers are trusted and under our control. A malicious server is unable to claim control over a portion of the keyrange space.

Client-Server Communication Protocol:

The following API is used by the client library to communicate with Seattle Node Directory Servers. Note, that users of Seattle will not need to use this API or call into the client library, as this is handled using the dylink library covered in the Seattle Overview section.

Purpose	**Request**	**Description**
TCP Registration	*RegisterAddressRequest <user_key1>,<user_key2>,<user_key3>,... <timestamp> <signature>*	Registration request sent through TCP to the server. The server replies with a signed *RegisterAddressRequestComplete* message that includes a registration renewal key. The renewal key included allows the client to renew its registration through UDP. If one of the userkeys specified is not in the keyrange of the server, the server replies with *RegisterFailOutOfKeyRange.*
UDP Registration Renewal	*RenewAddressRequest <user_key1>,<user_key2>,<user_key3>,... <renew_key>*	Registration request sent through UDP to the server, a valid renewal key must be specified. The server replies with *RenewAddressRequestComplete*, in the event that the renewal request is processed successfully. In the event the renewal key is invalid, the server replies with *RenewFailedInvalidRenewalKey.* If one of the userkeys specified is not in the keyrange of the server, the server replies with *RegisterFailOutOfKeyRange.*
Requesting Full Address list for a userkey	*GetAddressesForUserRequest <userkey>*	Request the list of addresses for a particular user through UDP from a server. If the userkey is not found in the server's table, the server replies with *UserKeyNotFound*. Otherwise, the server replies with a *GetAddressesReply* message that includes the list of addresses for the specific key in an encoded format to reduce the size of the message. If the userkey has more than 1000 addresses, only a random subset of 1000 addresses will be included in the reply. If there are no addresses, "None" is given instead of the encoded string. The reply from the server is signed.
Requesting the userkey range table	*GetUserKeyRangeTables*	Request the key range table and query_server_table from a registration server, to allow clients to determine which

		servers handle a given key range. The server replies with a *GetUserKeyRangeTablesReply* message that describes each keyrange, and the address of each server associated with it.The reply from the server is signed.
Verifying if an address is valid (ie. query requests)	*VerifyAddressRequest <ip_address>*	Verify address request, asks server to verify if an IP address (no update port needed) is valid, sent through UDP. The server replies with a message that indicates whether the address is valid. The port the address used to register is also included in the reply, since this information is necessary to disseminate update packets. This reply from the server is signed.
Update Packets	*AddressListUpdate <userkey> <current_id> <size_add_data> start<address data added><address data deleted>end <update_server_public_key> <timestamp> <signature>*	Update packets sent by the server that describe changes in membership for a specific userkey, see the section on Update Dissemination for more information. We determine which port to send update packets to for each address by using the port each address used to register. Each update packet describes the addresses added since the last update, and the addresses removed since the last update for a specific userkey. This address data is sent in an encoded format to reduce the size of the packet. If there is no added address data, or no deleted address data, 'None' is used for the respective section. Update packets are signed by the server, so each client that receives an update packet, regardless of the source, can verify if it was sent by the server.
Keyrange Announcements between Seattle Node Directory servers	*CNCServerToServerAnnounce <lower_key_bound> <upper_key_bound> <public_key> <timestamp> <signature>*	Sent by each server to announce their properties to other servers (UDP). Each server replies with a *CNCServerToServerAnnounceReply* message that specifies its own keyrange information. This ensures each server has an up-to-date table that describes the keyrange of every server in the Seattle Node Directory Service.

Client Interface:

Each node runs a service that handles registration and address verification for all vessels on the node. A client library wraps the repy library functions to enforce the traffic policy. Depending on the traffic policy selected, either outgoing packets, incoming packets, both incoming and outgoing packets, or neither will be filtered based on whether the source/destination address is found in the local node cache (cache hit). If the address is not found in the local cache (cache miss), an exception will be raised, and a query request will be sent to the server with keyrange that covers the destination address, to verify if the address is valid. To protect the servers from having to handle excessive amounts of such requests, if a query request for a specific address is answered with a negative reply (address not valid), the client will not send another query request for the same address for at least 4 minutes. We chose this number as it provides a

good balance in the sense that it allows the node to recheck the information at a reasonable rate, while generating fairly low load. If a node is trying to contact an address that is not in the local cache (either because the update was missed or because it shares no keys with the local node), we want the node to be able to recheck the membership status of the address frequently enough that it would not impede the usability of our service, but not so frequently that the load will overwhelm the Seattle Node Directory servers.

This query mechanism serves several important purposes. First, it provides a way for nodes to get up to date information in the event they miss an update packet. Since we use a gossip protocol, it is expected that nodes will not receive updates with 100% reliability. This reliability may be further lowered in the case that an attacker is able to block update packet traffic or several compromised nodes choose not to participate in disseminating updates. Second, this mechanism allows two nodes to communicate in the event they do not share userkeys. If two nodes do not share userkeys, they will not have each other's addresses pre-loaded in their local caches.

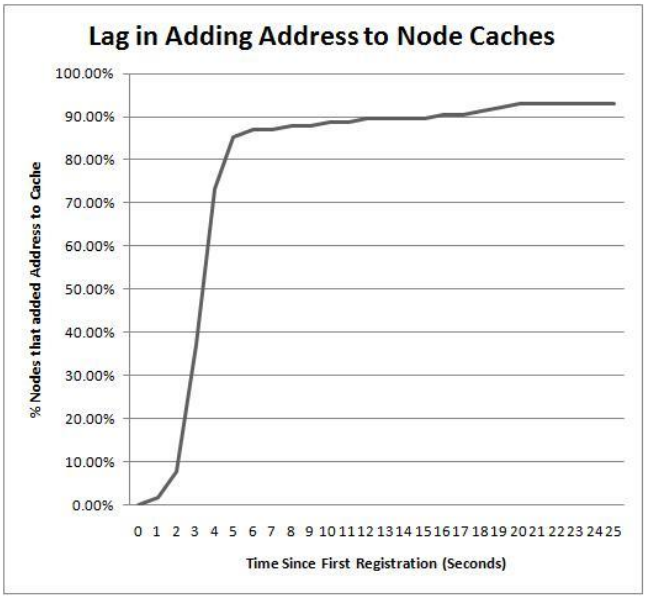
Evaluation

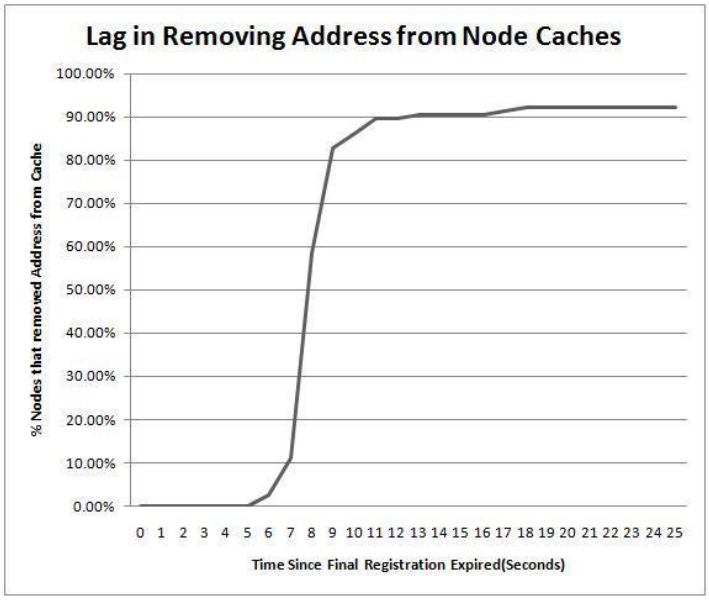
We evaluated our system through modeling and experiments to verify it achieves our objectives. First, we ran experiments to assess the performance of the mechanism we use to disseminate updates. We then setup and ran an all-pairs ping experiment, with all nodes registered under one userkey, and observed the rate of cache hits and misses. Afterwards, we built a model to estimate the load on Seattle nodes and the load on the Seattle Node Directory service for large numbers of clients. We then designed an experiment to measure load on the Seattle Node Directory Service using variable number of keys and variable farm configurations in order to assess how the performance of the system improves as we add multiple servers to the farm, and how performance is related to the number of userkeys per node. Finally, we measured the transparent delay that will be present when filtering incoming or outgoing traffic.

In evaluating our system, we assume the servers we will host the Seattle Node Directory Service will have an upload bandwidth of about 1 megabyte/second.

Lag In Loading and Removing Addresses from the Local Node Cache:

We experimentally measured the amount of time it takes, once a selected node has completed registration, for its address to be loaded into the local cache of every other node through the update packet mechanism. We then measured the amount of time it takes for a selected node that goes offline to be removed from the local cache of every other node (note, this is measured in seconds since the node's final registration). The selected node we used for our experiment was online for approximately 30 minutes. It registered about 5 minutes after the other nodes had come online. We show two graphs illustrating this data below.





Figures A.1 (left) and B.1 (right): In graph A.1, we show the delay in loading an address in the caches of 100 other nodes through update packets. In graph A.2, we show the delay in removing an address from the cache of 100 nodes using update packets.

The average time it took for an address to be added to the cache of other nodes after registration completion was 9.17 seconds, with the median at 3.38 seconds. This is not unexpected, since it is possible some nodes will receive the update packet after a long delay. It is expected a few nodes will miss the update packet completely, in which case the address will not be added to the cache until either the node attempts to contact the address, or until the next time the node retrieves the full address list from the server. It is worth noting that, after the node in question has completed its first registration, other nodes attempted and failed to send traffic to its address. This happened on different nodes at 19, 26, 60, and 121 seconds after the node in question had completed its first registration, resulting in the destination address being loaded to the cache of the respective nodes through query requests.

The average time it took for an address to be removed from the cache of other nodes after the last registration expired was 10.86 seconds, with the median at 7.83 seconds. The difference between these values and the values for the add delay is normal. The server disseminates updates following a rotating schedule, where it processes each userkey it manages every 10 seconds. Once a change is made to a node's membership status, it thus takes 0-10 seconds for the server to send an update describing the change. It is worth noting that in this particular experiment, 7 nodes did not receive the update packet indicating the removal of the address in question, so they are not included in the figure. The address in question will be removed from these nodes after each node retrieves the full address list form the server (which occurs about every 30 min), or when a subsequent packet is received and the node is able to detect from the id that it has missed the prior update.

During the 30 minute time-span when our selected node was online, we monitored the other nodes activity looking for cache misses on the address of the selected node. Other than the cache misses we note above in this section, all of which occurred at the start of the experiment, we did not observe any. This indicates that all other nodes were able to send traffic to our selected node during the time-span of the experiment, and our selected node was able to maintain registration with the Seattle Node Directory Service.

Accuracy:

	Cache Hit	Cache Miss
Address Valid	223814 (95.9%)	76 (0.0%)
Address not Valid	8684 (3.7%)	816 (.3%)

Table 1: This table shows the accuracy of the containment system in an experiment of 100 clients running an all pairs ping program for 20 minutes. All clients were in a single group and they each registered with a single userkey.

We assessed the accuracy of the Seattle containment system by setting up an experiment with 100 clients, all within one group, that register with one userkey each. They each also register for query. Each client ran an all-pairs ping program, where it periodically sent a poll to every other client in the group. Our results show that out of 233390 packets sent, the destination address of 223814 of the packets (95.9%) was found in the cache. For 816 packets (.3%), the destination address was not found in the cache, but in these cases the destination node was not registered with the Seattle Node Directory service at the time the packet was sent. For 8684 packets (3.7%), the destination address was in the local cache, but the destination node was not online at the time of registration. These false positives all occurred at the end of the experiment as the nodes were being stopped, and the majority of these cache misses (about 90%) occurred within two minutes of the destination node being stopped.

Model of Server Load:

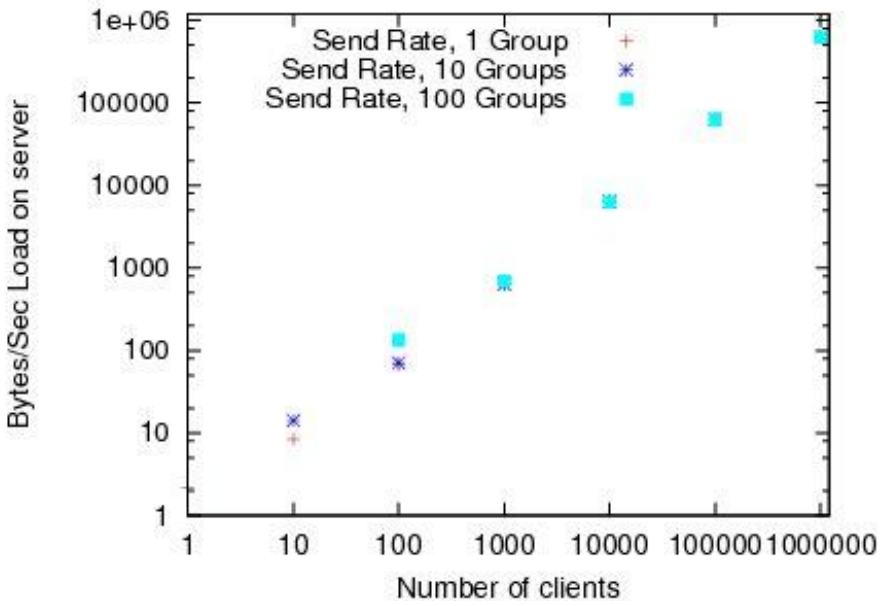


Figure B.1: We modeled the load that would be generated on the Seattle Node Directory Service, assuming it were run on a single server and with only one userkey per client. As we can see, for large number of clients, the load on the server does not appear to be significantly dependent on the way the clients are grouped (through userkeys)

To asses the scalability of the Seattle Node Directory service, we built a model to estimate load on the system for varying numbers of client nodes, and varying numbers of clients per group. Note, we define one or more clients as being in a group if and only if they share one or more or more userkeys. We built this model because it is not feasible for us to currently run experiments with 1 million nodes. Our model takes into account all forms of traffic that will be sent by the server, including update packets, registration replies, keyrange replies, and announcement messages.

We have validated our model of server load with experiments using 1, 10, and 100 client nodes. The load we observed in our experiments scaled in the same manner as projected by our model. The observed load is slightly higher than projected, about 25%-30%, which is not unexpected since the model assumes all nodes start at the same time. In our experiment, we attempted to approximate this, but we found it can take up to several minutes to start all nodes even when making use of parallelism. This longer start period means that additional update packets will be generated, each with less address information contained within.

We expect Seattle nodes each to have no more than 8 userkeys. This graph demonstrates a single Seattle Node Directory Service server can support on the oder of a million nodes that each have 1 userkey. In a subsequent section (Server Scalability), we will demonstrate the effect of multiple userkeys on server load.

Model of Client Load:

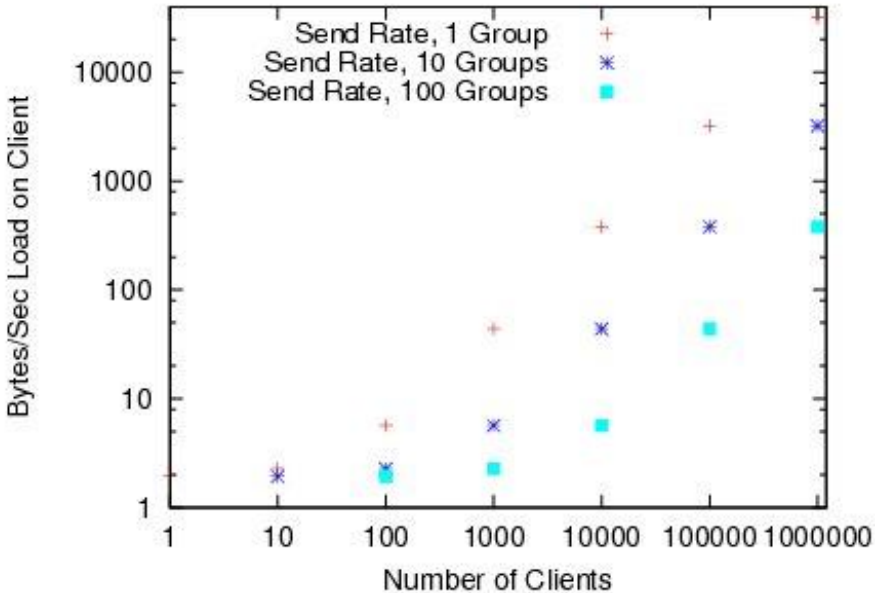


Figure B.2: *We modeled the load that would be generated on each client, assuming the Seattle Node Directory is run on a single server and with only one userkey per client. As we can see, the load on each client scales with the number of clients in the group, not with the number of clients total.*

We have modeled client load in the same manner as we modeled server load above. This was done because there are not sufficient nodes in Seattle to experimentally assess the scalability of the Seattle Node Directory service. Seattle nodes are often user donated machines, so it is important containment related load remains low.

From our model, we conclude that client overhead scales adequately. We expect most experiments will reserve around 100 vessels, and our graph shows client overhead will be low for groups of up to 10,000 clients. It is worth noting, as we can see in the above graph, that load on each client does not increase with the number of clients of the containment system, but rather client load increases only with group size. In other words, given a group of clients A, increasing the number of clients in a different group or adding more groups does not affect the individual load of clients group A.

We expect that most real-use scenarios will not involve all-pairs ping communication, so in most situations it will not be necessary for nodes to cache the full node lists.

Model of TCP-only Registration Load:

Using the same models above, we estimated the effect of using TCP to conduct all registrations (rather than allowing renewal through UDP). Our model used 1 userkey per client and 100 clients. We determined that requiring all nodes to register and renew registrations through TCP would result in about 84% more registration load sent by clients, and 96% more registration reply load sent by the Seattle Node Directory Service.

This results in approximately 35% increase in overall server send load, when using TCP for all registration renewal. By using UDP rather than TCP to renew client registrations, we decrease the send load on the server to about 74%.

Server Scalability:

The Seattle Node Directory service is designed to be a distributed system. However, there is overhead cost associated with running the server on multiple systems as illustrated by the two graphs below.

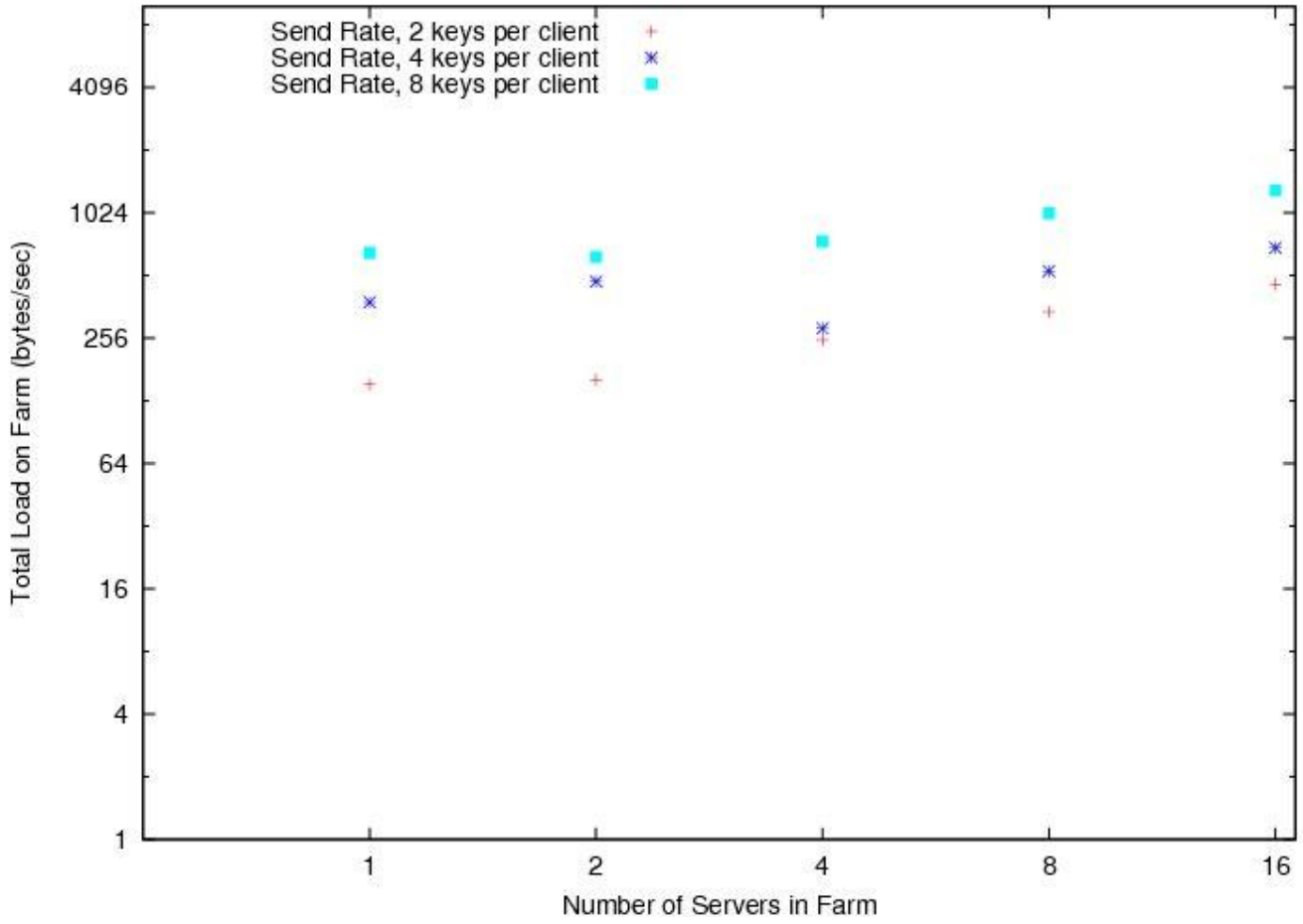


Figure C.1 : This graph depicts the effect of increasing the number of servers on Seattle Node Directory Service total load. The experiments were run using 100 clients, and with various number of keys. For the purpose of this graph, we treat the node's query registration as using an additional key (since this is the effect on registration load), so a client with one userkey that also performs query registration, is depicted as having two keys in the graph above.

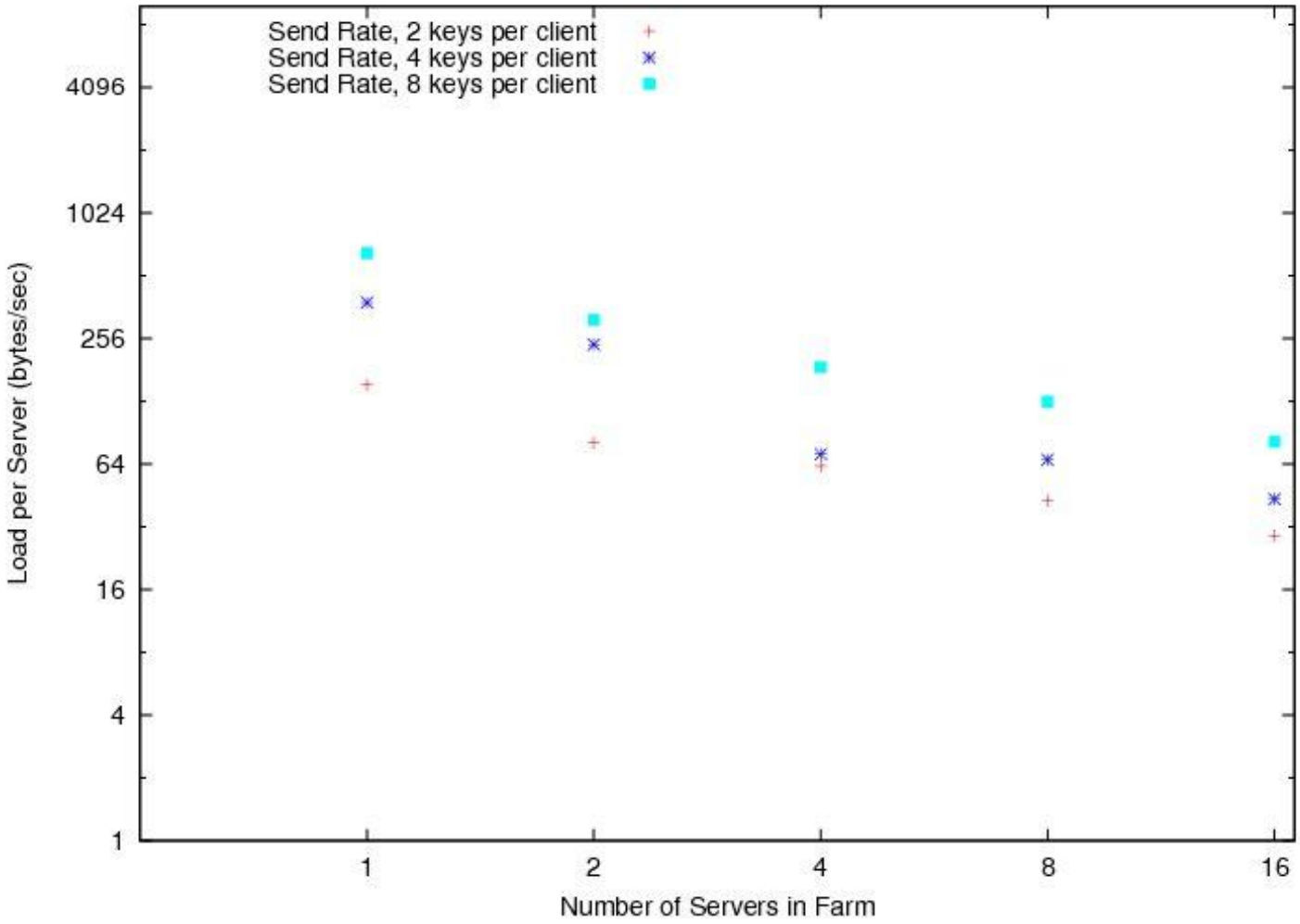


Figure C.2: This graph depicts the effect of increasing the number of servers on average individual server load. The experiments were run using 100 clients, and with various number of keys. For the purpose of this graph, we treat the node's query registration as using an additional key (since this is the effect on registration load), so a client with one userkey that also performs query registration, is depicted as having two keys in the graph above.

In figure C.1 we see a increase in total load as we add servers to the farm. This is expected due to the way clients register. Clients group the keys they need to register with such that they lower the number of registration requests they need to send. However, as the number of servers increases, each with a unique keyrange, userkeys are less and less likely to fall within the same keyrange, since each keyrange spans a smaller set of values. This results in clients needing to register with more servers. However, each client will not need to register more times than the number of keys it has plus one (the additional registration is for query, it is treated as an additional key in the graph C.1). Unlike registration load update packet dissemination load is split between the servers with no overhead as additional servers are added. For this reason we see in graph C.1 that the total load tends increases a lot initially as we add servers, but increases less and less once we reach a larger number of servers.

It is worth noting that there is additional overhead for managing the servers in the Seattle Node directory service associated with announce messages. As discussed in the design section, each server periodically sends an announcement indicating its keyrange to every other server in the farm. However, this load is negligible for small numbers of servers, and even though it is included in or results graph, it only accounts for a small fraction of the total load. However, announcement load is proportional to the number of servers squared, so for massive numbers of servers we may expect to see this as a larger component in total load.

Figure C.2 shows that as we increase the number of servers in the farm, the load per server tends to drop significantly regardless of the number of userkeys the client is using. It is worth noting that load per server is not directly inversely proportional to the number of servers. This is to be expected given that

adding additional servers increases load. However, it is unlikely we will need to run the Seattle Node Directory service on large numbers of servers (greater than 16). Based on the data for small numbers of servers, it appears that the load per server drops by a factor of 1/2 every time we increase the number of servers by a factor of 4. As a result, these graphs show that even though lead increases as additional servers are added, the average load per server decreases with increasing number of servers.

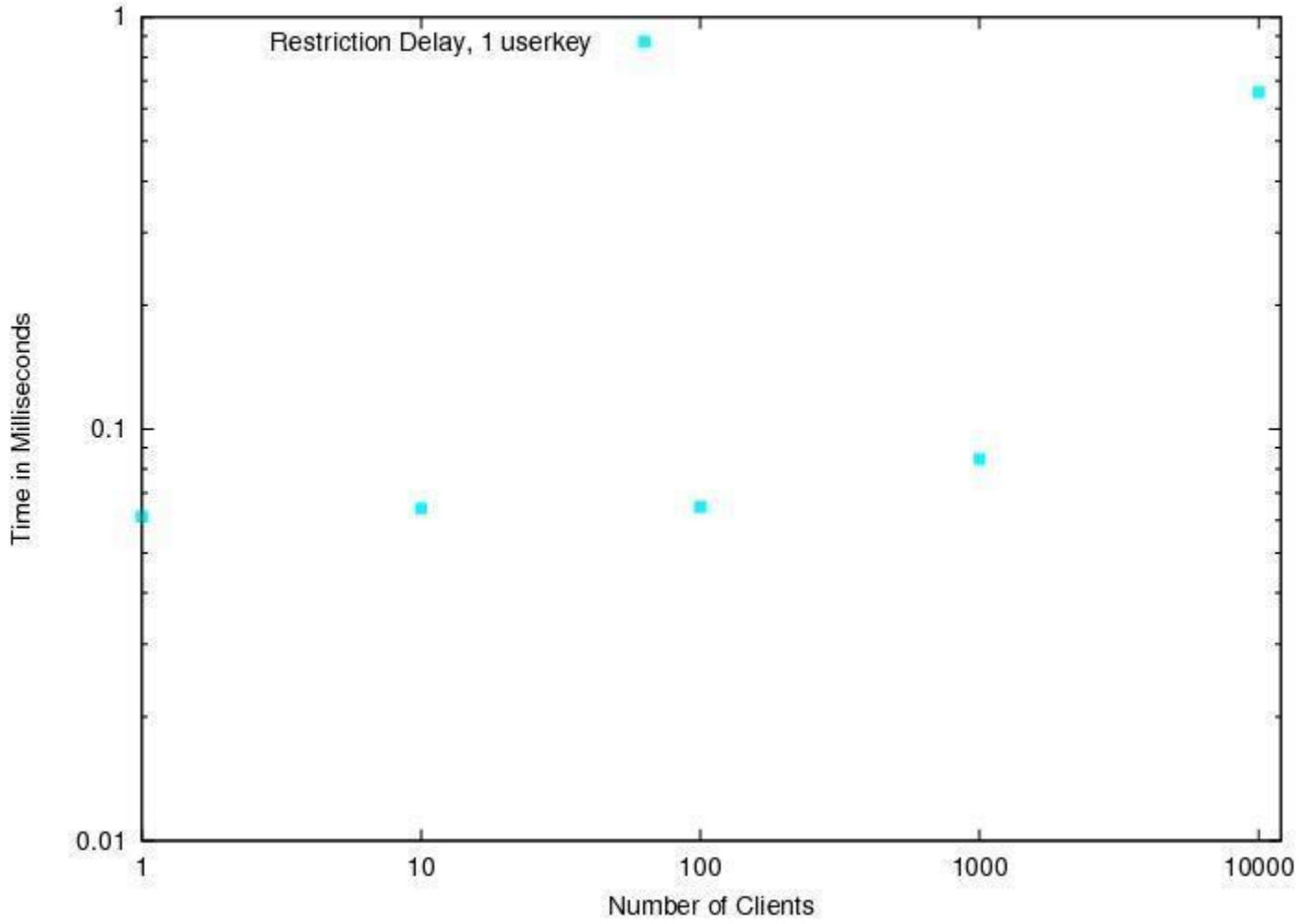
In both figures C.1 and C.2 we observe does not follow the curve perfectly, there is some variation. This is variation due to chance associated to the way we optimize registration performance. As described in the design section, nodes can register multiple userkeys simultaneously if the userkeys all fall within the same keyrange. For a farm of 4 servers, and clients with 8 userkeys, load would be significantly lower, if for any given node all userkeys fall within the same keyrange, then if two userkeys fell in each keyrange. The data point with 4 userkeys and 4 servers is an example of this phenomenon, where a high percentage of userkeys on any given node, fell within a single keyrange.

Taking into account the results from out model of the server load depicted in Figure A, we conclude that our system will be able to support about a million nodes, each having an average of 4 userkeys, if we use a farm of 16 servers.

Traffic Restriction Delay:

One of the goals of our system was to ensure that traffic restriction is done efficiently. To assess this, we timed how long it takes to verify whether an address is in the cache.

In the graphs below, we depict the effect of increasing the number of userkeys, and the number of clients on the delay associated with the restriction of traffic. As we can see in the graphs below, the delay in traffic remains low, though we expect to see variation in these numbers since they are highly dependent on the performance of the local system. We have seen situations where the restriction delay reaches 1 millisecond for experiments with 100 clients sharing a single userkey.



Figures D.1(left) and D.2(right): In the D.1, we hold the number of userkeys constant at 1 and vary the number of clients. In the D.2, we hold the number of addresses per userkey constant at 10 and vary the number of userkeys. We can see that the expected delay in looking up information in the node table remains low, even for cases where there are large numbers of addresses in the table or cases where the table has large numbers of userkeys.

Related Work

The role of our system is similar to that of a firewall. The role of a firewall is to inspect network traffic, and allow or deny it permission to pass based on some defined set of rules. In our system, this set of rules is dependent on the policy selected by the administrator of the host machine; we apply filtering to either outgoing traffic, incoming traffic, outgoing and incoming traffic, or no traffic. The next criteria on which we then determine if a given packet is filtered is whether the source/destination IP is present in the cache.

Firewalls are typically used to restrict incoming traffic, but in our system we may also want to restrict outgoing traffic. This alone is not unique since some corporate firewalls also restrict machines ability to send traffic to outside addresses. However, unlike such scenarios, in our testbed machines are not constrained to a small physical region, but are distributed throughout the world. In addition, the individual machines in our testbed cannot be treated as trusted entities, since they are run on user-donated resources and any could have been compromised by an attacker.

The most well known existing service that allows the collaborative sharing of resources is PlanetLab [5]. This system places a large amount of trust in its users. The first layer of defense PlanetLab has against malicious users is a system where each user is vetted. PlanetLab also makes use of an overlay auditing service called PlanetFlow [8] that logs the headers of all outgoing and incoming traffic. It does not restrict or block the sending of traffic, nor does it save the contents of each packet. However, once abuse has been detected, this service does allow quick identification of the service responsible for the

sending/receiving of the offending traffic. This is effective because PlanetLab users sign an Acceptable Use Policy and can be held accountable by their institution. These assumptions do not hold for Seattle.

Additional work has been done in attempting to protect virtual machine testbeds from misuse be attackers. WASCo [9] provides an infrastructure that allows full-trust, partial-trust, or untrusted users to run arbitrary code on surrogate virtual machines. Similar to Seattle, this infrastructure restricts the resources available to these virtual machines, for example limiting bandwidth to protect from participation in DDOS attacks. WASCo allows administrators to set specific bandwidth constraints for different traffic flows. WASCo uses logging on each virtual machine to detect and trace other forms of malicious activity such as illegal distribution of material or SPAM. However, it does not filter traffic sent by a given node provided the traffic does not exceed the bandwidth constraint.

In contrast, Seattle does not place any trust in its users. User code is run on sandbox virtual machines such that it cannot harm the host system. Unlike PlanetLab, Seattle does not require the user to be a member of a contributing institution. This makes the Seattle platform more accessible, so anyone with access to a computer may use Seattle and gain access to potentially large numbers of virtual machines. It is currently feasible for a malicious user to take advantage of the accessibility of the Seattle service, so we developed the containment system discussed in this paper.

Additional been done in membership tracking for distributed systems in the Census project [10]. In this system, the node population is subdivided into groups called epochs based on geographical location, and the system guarantees that nodes within an epoch will have synchronized membership lists. Update information is disseminated through the use of multi-cast trees, generated by a deterministic algorithm on each node. Assuming membership information is global, each node will generate approximately the same distribution tree. However, we chose not to use an implementation like this in our system because Seattle nodes frequently change their online status, which could result in large discrepancies between each node's generated distribution tree. In addition, such a system is high in complexity and would be far more difficult to implement.

Conclusion

The goal of this work has been to develop a way to prevent Seattle nodes from being used to send SPAM, distribute illegal content, or participating in DDOS attacks. Our requirements for this system are that it must be resistant to common forms of attack, it should scale to support about a million nodes, and it should not present a hidden delay to clients.

As a solution to this challenge, we have developed the Seattle Node Directory service, a a set of distributed servers that tracks the addresses of machines which have explicitly opted-in to receiving Seattle node traffic. Each Seattle node maintains a local cache of valid addresses. Using a gossip protocol, the Seattle Node Directory service will pre-load likely destination addresses to each node. We assumed any given two nodes are likely to communicate if they share one or more userkeys.

To assess the scalability of our system, we first build a model to estimate client and server load for variable number of nodes and variable grouping of nodes. We validated our model by experimentally verifying the data points for 1,10, and 100 client nodes. From this model we concluded that overhead of our system on client nodes increases with the number of nodes per group, not the total number of registered clients. We found that client overhead is acceptably low for groups of 10k nodes or less.

We conducted a second set of experiments to assess how load on the Seattle Node Directory service changes with variable number of userkeys on each client and variable number of servers in the Seattle Node Directory farm. Our results showed that for low numbers of servers in the farm, every time we increase farm size by a factor of 4, the load on the farm decreases by about a factor of 1/2. We found that as the number of user keys on a server increases, by a factor slightly less than that we increased the user keys by.

As a final step, we also verified that the process of clients checking the local cache before sending each message is low in cost, averaging less than 1ms for a group of 100 nodes.

Overall we have found we are able to scale Seattle to 1 million nodes if we use approximately 16 servers, and nodes have an average of 4 userkeys each. The current node load on Seattle can easily be handled by a single server.

References

[1] Seattle: The Internet as A Testbed. https://seattle.cs.washington.edu/wiki.

[2] Repy Library - Seattle - Trac. https://seattle.cs.washington.edu/wiki/RepyLirary

[3] SeattleGENI. https://seattlegeni.cs.washington.edu.

[5] PlanetLab. http://www.planet-lab.org.

[6] J. Cappos, I. Beschastnikh, J. Samuel, C. Barsan, A. Dadger, A. Honstain, E. Kimbrel, C. Meyer, A. Krishnamurthy, and T. Anderson. Building a Scalable and Diverse Testbed. Under submission.

[7] Dynamic Linking of Modules - Seattle - Trac. https://seattle.cs.washington.edu/wiki/DynamicLinkingModules

[8] M. Huang, A. Bavier and L. Peterson. PlanetFlow: Maintaining Accountability for Network Services. Operating Systems Review, vol 40. no. 1, pg 89-94, 2006.

[9] S.Goyal, and J. Carter. Safely Harnessing Wide Area Surrogate Computing or How to Avoid Building the Perfect Platform for Network Attacks. In *Proc. 4th WORLDS*, San Francisco, CA, Dec. 2004.

[10] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-Aware Membership Management for Large-Scale Distributed Systems. USENIX Annual Technical Conference 2009. San Diego, CA.