ETTM: A Scalable Fault Tolerant Network Manager

Colin Dixon Hardeep Uppal Vjekoslav Brajkovic Dane Brandon
Thomas Anderson Arvind Krishnamurthy
University of Washington

Abstract

In this paper, we design, implement, and evaluate a new scalable and fault tolerant network manager, called ETTM, for securely and efficiently managing network resources at a packet granularity. Our aim is to provide network administrators a greater degree of control over network behavior at lower cost, and network users a greater degree of performance, reliability, and flexibility, than existing solutions. In our system, network resources are managed via software running in trusted execution environments on participating end-hosts. Although the software is physically running on end-hosts, it is logically controlled centrally by the network administrator. Our approach leverages the trend to open management interfaces on network switches as well as trusted computing hardware and multicores at end-hosts. We show that functionality that seemingly must be implemented inside the network, such as network address translation and priority allocation of access link bandwidth, can be simply and efficiently implemented in our system.

1 Introduction

In this paper, we propose, implement, and evaluate a new approach to the design of a scalable, fault tolerant network manager. Our target is enterprise-scale networks with common administrative control over most of the hardware on the network, but with complex quality of service and security requirements. For these networks, we provide a uniform administrative and programming interface to control network traffic at a packet granularity, implemented efficiently by exploiting trends in PC and network switch hardware design. Our aim is to provide network administrators a greater degree of control over network behavior at lower cost, and network users a greater degree of performance, reliability, and flexibility, compared to existing solutions.

Network management today is a difficult and complex endeavor. Although IP, Ethernet and 802.11 are widely available standards, most network administrators need more control over network behavior than those protocols provide, in terms of security configuration [21, 14], resource isolation and prioritization [36], performance and cost optimization [4], mobility support [22], problem diagnosis [27], and reconfigurability [7]. While most end-host operating systems have interfaces for configuring certain limited aspects of network security and resource policy, these configurations are typically set independently by each user and therefore provide little assurance or consistent behavior when composed across multiple users on a network.

Instead, most network administrators turn to middleboxes - a central point of control at the edge of the network where functionality can be added and enforced on all users. Unfortunately, middleboxes are neither a complete nor a cost-efficient solution. Middleboxes are usually specialized appliances designed for a specific purpose, such as a firewall, packet shaper, or intrusion detection system, each with their own management interface and interoperability issues. Middleboxes are typically deployed at the edge of the (local area) network, providing no help to network administrators attempting to control behavior inside the network. Although middlebox functionality could conceivably be integrated with every network switch, doing so is not feasible at line-rate at reasonable cost with today's LAN switch hardware.

We propose a more direct approach, to manage network resources via software running in trusted execution environments on participating endpoints. Although the software is physically running on endpoints, it is logically controlled centrally by the network administrator. We somewhat whimsically call our approach ETTM, or End to the Middle. Of course, there is still a middle, to validate the trusted computing stack running on each participating node, and to redirect traffic originating from non-participating nodes such as smart phones and printers to a trusted intermediary on the network. By moving packet processing to trusted endpoints, we can enable a much wider variety of network management functionality than is possible with today's network-based solutions.

Our approach leverages four separate hardware and software trends. First, network switches increasingly have the ability to re-route or filter traffic under administrator control [7, 30]. This functionality was originally added for distributed access control, e.g., to prevent visitors from connecting to the local file server. We use these new-generation switches as a lever to a more general, fine-grained network control model, e.g., allowing us to efficiently interpose trusted network management software on every packet. Second, we observe that many end-host computers today are equipped with trusted computing hardware, to validate that the endpoint is booted with an uncorrupted software stack. This allows us to use software running on endpoints, and not just network hardware in the middle of the network, as part of our enforcement mechanism for network management. Third, we leverage virtual machines. Our

network management software runs in a trusted virtual machine which is logically interposed on each network packet by a hypervisor. Despite this, to the user each computer looks like a normal, completely configurable local PC running a standard operating system. Users can have complete administrative control over this OS without compromising the interposition engine. Finally, the rise of multicore architectures means that it is possible to interpose trusted packet processing on every incoming/outgoing packet without a significant performance degradation to the rest of the activity on a computer.

In essence, we advocate converting today's closed appliance model of network management to an open software model with a standard API. None of the functionality we need to implement on top of this API is particularly complex. As a motivating example, consider a network administrator needing to set up a computer lab at a university in a developing country with an underprovisioned, high latency link to the Internet. It is well understood that standard TCP performance will be dreadful unless steps are taken to manipulate TCP windows to limit the rate of incoming traffic to the bandwidth of the access link, to cache repeated content locally, and to prioritize interactive traffic over large background transfers. As another example, consider an enterprise seeking to detect and combat worm traffic inside their network. Current Deep Packet Inspection (DPI) techniques can detect worms given appropriate visibility, but are expensive to deploy pervasively and at scale. We show that it is possible to solve these issues in software, efficiently, scalably, and with high fault tolerance, avoiding the need for expensive and proprietary hardware solutions.

The rest of the paper discusses these issues in more detail. We describe our design in § 2, sketch the network services which we have built in § 3, summarize related work in § 4 and conclude in § 5.

2 Design & Prototype

ETTM is a scalable and fault-tolerant system designed to provide a reliable, trustworthy and standardized software platform on which to build network management services without the need for specialized hardware. However, this approach begs several questions concerning security, reliability and extensibility.

- How can network management tasks be entrusted to commodity end hosts which are notorious for being insecure? In our model, network management tasks can be relocated to any trusted execution environment on the network. This requires the network management software be verified and isolated from the host OS to be protected from compromise.
- If the management tasks are decentralized, how can these distributed points of control provide consistent decisions which survive failures and disconnections? The system should not break simply because a user, or a whole team of users, turn off their computers. In particular, management services must be available in face of node failures and maintain consistent state regarding the resources they manage.
- How can we architect an extensible system that enables the deployment of new network management services which can interpose on relevant packets? Network administrators need a single interface to install, configure and compose new network management services. Further, the implementation of the interface should not impose undue overheads on network traffic.

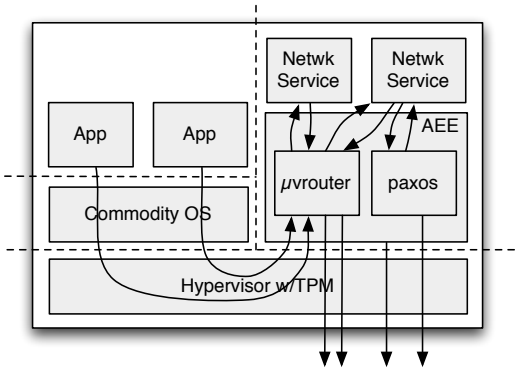


Figure 1: The architecture of an ETTM end-host. Network management services run in a trusted virtual machine (AEE). Application flows are routed to appropriate network management services using a micro virtual router (μvrouter).

While many of the techniques we employ to surmount these challenges are well-known, their combination into a unified platform able to support a diverse set of network services is novel. The particular mechanisms we employ are summarized in Table 1, and the architecture of a given end-host participating in management can be seen in Figure 1.

The function of these mechanisms is perhaps best illustrated by example, so let us consider a distributed Network Address Translation (NAT) service for sharing a single IP address among a set of hosts. The NAT service in ETTM maps globally visible port numbers to private IP addresses and vice versa. First, the translation table itself needs to be consistent and survive faults, so it is maintained and modified consistently by the *consensus* subsystem based on the Paxos distributed coordination algorithm. Second, the translator must be able to interpose on all traffic that is either entering or leaving the NATed network. The micro virtual router (μvrouter)'s *filters* allow for this interposition on packets sourced by a ETTM end-host, while the *physical switches* are set up to

Mechanism	Description	Tech Trends	Goals	Section
Trusted Authorization	Extension to the 802.1X network access control protocol to authorize trusted stacks	TPM	Trust	2.1
Attested Execution Environment	Trusted space to run filters and control plane processes on untrusted end-hosts	Virtualization, Multicore	Scalability	2.2
Physical Switches	In-network enforcers of access control and routing/switching policy decisions	Open interfaces	Standardization	2.3
Filters	End-host enforcers of network policy running inside the Attested Execution Environment	Multicore	Extensibility	2.4
Consensus	Agreement on management decisions and shared state	Fault tolerance techniques	Reliability, Extensibility	2.5

Table 1: Summary of mechanisms in ETTM.

deliver incoming packets to the appropriate host.[1] Lastly, because potentially untrusted hosts will be involved in the processing of each packet, the service is run only in an isolated *attested execution environment* on hosts that have been verified using our *trusted authorization* protocol based on commodity trusted hardware.

2.1 Trusted Authorization

Traditionally, end-hosts running commodity operating systems have been considered too insecure to be entrusted with the management of network resources. However, the recent proliferation of trusted computing hardware has opened the possibility of restructuring the placement of trust. In particular, using the trusted platform module (TPM) [39] shipping with many current computers, it is possible to verify that a remote computer booted a particular software stack. In ETTM, we use this feature to build an extension to the widely-used 802.1X network access control protocol to make authorization decisions based on the booted software stack of end-hosts rather than using traditional key- or password-based techniques. We note that the guarantees provided by trusted computing hardware generally assume that an attacker will not physically tamper with the host, and we make this assumption as well.

The remainder of this section describes the particular capabilities of current trusted hardware and how they enable the remote verification of a given software stack.

2.1.1 Trusted Platform Module

The TPM is a hardware chip commonly found on motherboards today consisting of a cryptographic processor, some persistent memory, and some volatile memory. The TPM has a wide variety of capabilities including the secure storage of integrity measurements, RSA key creation and storage, RSA encryption and decryption of data, pseudo-random number generation and attestation to portions of the TPM state. Much of this functionality is orthogonal to the purposes of this paper. Instead, we focus on the features required to remotely verify that a machine has booted a given software stack.

One of the keys stored in the TPM's persistent memory is the endorsement key (EK). The EK serves as an identity for the particular TPM and is immutable. Ideally, the EK also comes with a certificate from the manufacturer stating that the EK belongs to a valid hardware TPM. However many TPMs do not ship with an EK from the manufacturer. Instead, the EK is set as part of initializing the TPM for its first use.

The volatile memory inside the TPM is reset on every boot. It is used to store measurement data as well as any currently loaded keys. Integrity measurements of the various parts of the software stack are stored in registers called Platform Configuration Registers (PCRs). All PCR values start as 0 at boot and can only be changed by an extend operation, i.e., it is not possible to replace the value stored in the PCR with an arbitrary new value. Instead, the extend operation takes the old value of the PCR register, concatenates it with a new value, computes their hash using Secure Hash Algorithm 1 (SHA-1), and replaces the current value in the PCR with the output of the hash operation.

2.1.2 Trusted Boot

The intent is that as the system boots, each software component will be hashed and its hash will be used to extend at least one of the PCRs. Thus, after booting, the PCRs will provide a tamper evident summary of what happened during the boot. For instance, the post-boot PCR values can be compared against ones corresponding to a known-good boot to establish if a certain software stack has been loaded or not.

To properly measure all of the relevant components in the software stack requires that each layer be instrumented to measure the integrity of the next layer, and then store that measurement in a PCR before passing execution on. Storing measurements of different components into different PCRs allows individual modules to

[1] This is possible with legacy ethernet switches using a form of detour routing or more efficiently with programmable switches [30].

be replaced independently.

As each measurement's validity depends on the correctness of the measuring component, the PCRs form a chain of trust that must be rooted somewhere. This root is the immutable boot block code in the BIOS and is referred to as the Core Root of Trust for Measurement (CRTM). The CRTM measures itself as well as the rest of BIOS and appends the value into a PCR before passing control to any software or firmware. This means that any changeable code will not acquire a blank PCR state and cannot forge being the "bottom" of the stack.

It should be noted that the values in the PCRs are only representative of the state of the machine at boot time. If malicious software is loaded or changes are made to the system thereafter, the changes will not be reflected in the PCRs until the machine is rebooted. Thus, it is important that only minimal software layers are attested. In our case, we attest the BIOS, boot loader, virtual machine monitor, and execution environment for network services. We do not need to attest the guest OS running on the device, as it is never given access to the raw packets traversing the device.

2.1.3 Attestation

Once a machine is booted with PCR values in the TPM, we need a verifiable way to extract them from the TPM so that a remote third party can verify that they match a known-good software stack and that they came from a real TPM. In theory this should be as simple as signing the current PCR values with the private half of the EK, but signing data with the EK directly is disallowed.[2] Instead, Attestation Identity Keys (AIKs) are created to sign data and create attestations. The AIKs can be associated with the TPM's EK either via a Privacy CA or via Direct Anonymous Attestation [39] in order to prove that the AIKs belong to a real TPM. As a detail, because many TPMs do not ship with EKs from their manufacturers, these computers must generate an AIK at installation and store the public half in a persistent database.

To facilitate attestation, TPMs provide a quote operation which takes a nonce and signs a digest of the current PCRs and that nonce with a given AIK. Thus, a verifier can challenge a TPM-equipped computer with a random, fresh nonce and validate that the response comes from a known-good AIK, contains the fresh nonce, and represents a known-good software stack.

2.1.4 ETTM Boot

When a machine attempts to connect to an ETTM network, the switch forwards the packets to a verification server which can be either an already-booted end-host running ETTM, a persistent server on the LAN or even a cloud service.[3] On recognizing the connection of a new host, the switch establishes a tunnel to the verification server and maintains this tunnel until the verification server can reach a verdict about authorization.

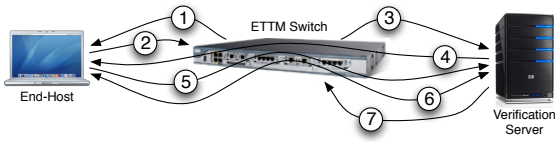


Figure 2: The steps required for an ETTM boot and trusted authorization.

If the host is verified as running a complete, trusted software stack then it is simply granted access to the network. If the host is running either an incomplete or old software stack, the ETTM software on the end-host attempts to download a fresh copy and retries. Traffic from non-conformant hosts are tunneled to a participating host; our design assumes this is a rare case.

Our trusted authorization protocol creates this exchange via an extension to the 802.1X and EAP protocols. We have extended the `wpa_supplicant` [28] 802.1X client and the FreeRADIUS [16] 802.1X server to support this extension and provide authorization to clients based purely on their attested software stacks.

This process is shown in Figure 2. First, the end-host connects to an ETTM switch, receives an EAP Request Identity packet (1), and responds with an EAP Response/Identity frame containing the desired AIK to use (2). The switch encapsulates this response inside an 802.1x packet which is forwarded to the verification server running our modified version of FreeRADIUS (3). The FreeRADIUS server responds with a second EAP Request Trusted Software Stack frame containing a nonce again encapsulated inside an 802.1X packet (4), and the end-host responds with an EAP Response Trusted Software Stack frame containing the signed PCR values proving the booted software stack (5). This concludes the verification stage.

The verification server can then either render a verdict as to whether access is granted (7) or require the end-host to go through a provisioning stage (6) where extra code and/or configuration can be loaded onto the machine and the authorization retried.

2.1.5 Performance of ETTM Boot

Table 2 presents microbenchmarks for various TPM operations (including those which will be described later in this section) on our Dell Latitude e5400 with a Broadcom TPM complying to version 1.2 of the TPM spec, an Intel 2 GHz Core 2 Duo processor and 2 GB of RAM.

[2]This is to avoid creating an immutable identity which is revealed in every interaction involving the TPM.

[3]We assume the existence of some persistently reachable computer to bootstrap new nodes and store TPM configuration state. Under normal operation, this is a currently active verified ETTM node.

Operation	Time (s)	Std. Dev. (s)
PCR Extend	0.0253	0.001
Create AIK	34.3	8.22
Load AIK	1.75	0.002
Sign PCR	0.998	0.001

Table 2: The time (in seconds) it takes for a variety of TPM operations to complete.

Operation	Wall Clock Time (s)
client start	0
receive first server message	+0.049
receive challenge nonce	+0.021
send signed PCRs	+0.998
receive server decision	+0.018
Total	1.09

Table 3: The time (in seconds) it takes for an 802.1X EAP-TSS authorization with breakdown by operation.

The time to create the AIK is needed only once at system initialization. The total time added to the normal boot sequence for an ETTM enabled host is negligible as most actions can be trivially overlapped with other boot tasks. Assuming the challenge nonce is received, the signing time can be overlapped with the booting of the guest OS as no attestation is required to its state.

Table 3 shows a breakdown of how long each step takes in our implementation of trusted authorization assuming an up-to-date trusted software stack is already installed on the end-host and the relevant AIK has already been loaded. The total time to verify the boot status is just over 1 second. This is dominated by the time that it takes to sign the PCR values after having received the challenge nonce.

2.2 Attested Execution Environment

In ETTM, we require that each participating host has a corresponding trusted virtual machine which is responsible for managing that host's traffic. We call this virtual machine an Attested Execution Environment (AEE) because it has been attested by Trusted Authorization. In the common case, this virtual machine will run alongside the commodity OS on the host, but in some cases a host's corresponding AEE may run elsewhere with the physical switching infrastructure providing an constrained tunnel between the host and its remote VM.

The AEE is the vessel in which network management activities take place on end-hosts. It provides three key features: a mechanism to intercept all incoming and outgoing traffic, a secure and isolated execution environment for network management tasks and a common platform for network management applications.

To interpose the AEE on all network traffic, the hypervisor (our implementation makes use of Xen [3]) is configured to forward all incoming and outgoing network traffic through the AEE. This configuration is verified as part of trusted authorization. Once the AEE has been interposed on all traffic, it can apply the ETTM filters (described in § 2.4) giving each network service the required points of visibility and control of the data path.

Further, the hypervisor is configured to isolate the AEE from any other virtual machines it hosts. Thus, the AEE will be able to faithfully execute the prescribed filters regardless of the configuration of the commodity operating system. [4] The AEE can also execute network management tasks which are not directly related to the host's traffic. For example, it could redirect traffic to a mobile host, verify a new host's software stack or reconfigure physical switches. It is even possible for a host to run multiple AEEs simultaneously with some being run on behalf of other nodes in the system. A desktop with excess processing power can stand-in to filter the traffic from a mobile phone.

Lastly, the AEE provides a common platform to build network management services. Because this platform is run as a VM, it can remain constant across all end-hosts providing a standardized software API. Our current AEE implementation is a stripped-down Linux virtual machine, however, we have augmented it with APIs to manage filters (described in § 2.4) as well as to manage reliable, consistent, distributed state (described in § 2.5).

While in most cases, the added computational resources required to run an AEE do not pose a problem, ETTM allows for AEEs (or some parts of an AEE) to be offloaded to another computer. In our prototype, this is handled by applications themselves. In the future, we hope to add dynamic offloading based on machine load.

2.3 Physical Switches

Physical switches are the lowest-level building block in ETTM. Their primary purpose is to provide control and visibility into the link layer of the network. This includes access control, flexible control of packet forwarding, and link layer topology monitoring.

- **Authorization/Access Control:** As described earlier, switches redirect and tunnel traffic from as of yet unauthorized hosts until an authorization decision has been made.
- **Flexible Packet Forwarding:** The ability to install custom forwarding rules in the network enables significantly more efficient implementations of some network management services (e.g., NAT), but is not required. Flexible forwarding also enables more efficient routing by not constraining traffic within the

[4] We make use of a VM other than the root VM (e.g., Dom0 in Xen) for the AEE to both maintain independence from any particular hypervisor and to protect any such root VM from misbehaving applications in the AEE.

traditional ethernet spanning tree protocol.

- **Topology Monitoring:** In order to properly manage available network resources, end-hosts must be able to discover what network resources exist. This includes the set of physical switches and links along with the links' latencies and capacities.

At a minimum, ETTM only requires the first of these capabilities and since we implement access control via an extension to 802.1X and EAP, most current ethernet switches (even many inexpensive home routers [31, 10]) can serve as ETTM switches. There are advantages to more full-featured switches, however. For instance, a physical switch that supports the 802.1AE MACSec specification can provide a secure mechanism to differentiate between the different hosts attached to the same physical port and authorize them independently, while denying access to other unauthorized hosts attached to the port.

Additionally, ETTM can better manage network resources when used in conjunction with an OpenFlow switch [30]. OpenFlow provides a wealth of network status information and supports packet header rewriting and flexible, rule-based packet forwarding. We currently leave interacting with programmable switches to applications. Many applications function correctly using simple Ethernet spanning tree routing and do not require control over packet-forwarding. Those that do, like the NAT, must either implement packet redirection in the application logic by having AEEs forward packets to the appropriate host or manage configuring the programmable switches themselves. We are in the process of creating a standard interface to packet forwarding in ETTM.

2.4 Micro Virtual Router

On each end-host, we construct a lightweight virtual router, called the micro virtual router (μvrouter), which mediates access to incoming and outgoing packets by the various services. Services use the μvrouter to inspect and modify packets as well as insert new packets or drop packets. The core idea of filters in ETTM is that they are the mechanism to interpose on a per-packet basis and their behavior can be controlled by consensus operations which occur at a slower time scale: one operation per flow or one operation per flow, per RTT.

The μvrouter consists of an ordered list (by priority) of filters which are applied to packets as they depart and arrive at the host. The current Filter API is described in Table 4. The filters which we have implemented so far (described in § 3) correspond to tasks that would currently be carried out by a special-purpose middlebox like a NAT, web cache, or traffic shaper.

The μvrouter is approximately 2250 lines of C++ code running on Linux using `libipq` and `iptables` to capture traffic. This has simplified development by allowing

`matchOnHeader()` returns `true` if the filter can match purely on IP, TCP and UDP headers (i.e., without considering the payload) and `false` if the filter must match on full packets
`getPriority()` returns the priority of the filter, this is used to establish the order in which filters are applied
`getName()` simply returns a human readable name of the filter
`matchHeader(iphdr, tcphdr, udphdr)` returns `true` if the filter is interested in a packet with these headers; undefined filters are set to `NULL` and behavior is undefined if `matchOnHeader()` returns `false`
`match(packet)` returns `true` if the filter is interested in the packet; behavior is undefined if `matchOnHeader()` returns `true`
`filter(packet)` actually processes a packet; returns one of `ERROR`, `CONTINUE`, `SEND`, `DROP` or `QUEUED` and possibly modifies the packet
`upkeep()` this function is called 'frequently' and enables the filter to perform any maintenance that is required
`getReadyPackets()` this returns a list of packets that the filter would like to either dequeue or introduce; this is called 'frequently'

Table 4: The filter API.

the μvrouter to run as a user-space application. However, the user-space implementation has a downside in that it imposes performance overheads that limit the sustained throughput for large flows. To address the performance concerns, we split the functionality of the μvrouter into two components—a user-space module supporting the full filter API specified in Table 4 and a kernel-level module that supports a more restricted API used only for header rewriting and rate-limiting. In applications such as the NAT, the user-space filter is invoked only for the first packet in order to assign a globally unique port number to the flow, while the kernel module is used for filling in this port number in subsequent packets.

The μvrouter enables an administrator to specify a stack of filters that carry out the data-plane management tasks for the network. That is, it handles traffic that is destined for or emanates from an end-host on the network. Traffic destined to or emanating from AEEs or physical switches constitutes the control plane of ETTM and is not handled by the filters.

2.5 Consensus

If network management is going to be distributed among a large number of potentially unreliable commodity computers, there must be a layer to provide consistency and reliability despite failures. For example, a desktop unexpectedly being unplugged should not cause any state to

be lost for the remaining functioning computers. Fortunately, there is a vast literature on how to build reliable systems out of inexpensive, unreliable parts. In our case we build reliability using the Paxos algorithm for distributed consensus [25].

We expose a common API which provides a simple way for ETTM network services to manage their consistent state including the ability to define custom rules for what state should be semantically allowed and ways to choose between liveness and safety in the event that it is required. We expose our consensus implementation via a table abstraction in which each row corresponds to a single service's state and each cell in a given row corresponds to an agreed upon action on the state managed by the service. Thus, each service has its own independently ordered list of agreed upon values, with each row entirely independent of other rows from the point of view of the Paxos implementation.

In building the API and its supporting implementation we strove to overcome several key challenges:

Application Independent Agreement: The actual agreement process should be entirely independent of the particular application. As a consequence, the abstraction presented is agreement on an ordered list of blobs of bytes for each application or service, with the following operations allowed on this ordered list.

- `put(name, value)`: Attempts to place `value` as a cell in the row named `name`. This will not return immediately specifying success or failure, but if the value is accepted, a later `get` call or subscription will return `value`.
- `get(name, seqNum)`: Attempts to retrieve cell number `seqNum` from the row named `name`. Returns an error if `seqNum` is invalid and the relevant value otherwise.

For example, our NAT implementation creates a row in the table called "NAT". When an outgoing connection is made an entry is added with the mapping from the private IP address and port to the public IP address and a globally visible port along with an expiration time. Nodes with long-running connections can refresh by appending a new entry. Thus, each node participating in the NAT can determine the shared state by iteratively processing cells from any of the replicas.

Publish-Subscribe Functionality: A network service can subscribe to the set of agreed upon values for a row via the `subscribe` API call. The service running on an ETTM node receives a callback (using `notify`) when new values are added to a given row through the `put` API calls. This is useful not just for letting services manage their own state, but also for subscribing to special rows that contain information about the network in general. For instance, there is one row which describes topology information and another row which logs authorization decisions. The consensus system invokes

- `subscribe(name, seqNum)`: Asks that the values of all cells in the row `name` starting with the cell numbered `seqNum` be sent to the caller. This includes all cells agreed on in the future.
- `unsubscribe(name)`: Cancels any existing subscription to the row `name`.
- `notify(name, value, seqNum)`: This is the callback from a `subscription` call and lets the client know that cell number `seqNum` of row `name` has the value `value`.

Balance Reliability and Performance: Invariably adding more nodes and thus increasing expected reliability causes performance to degrade as more responses are required. Thus, we allow for a subset of the participating ETTM nodes to form the Paxos group rather than the whole set. ETTM nodes use the following API calls to join and depart from consensus groups and to identify the set of cells that have been agreed upon by the consensus group.

- `join(name)` Asks the local consensus agent to participate in the row `name`.
- `leave(name)` Asks the local consensus agent to stop participating in row `name`. A graceful ETTM machine shutdown involves informing each row that the node is leaving beforehand.
- `highestSequenceNumber(name)` Returns the current highest valid cell number in the row named `name`.

Allow Application Semantics: While we wish to be application agnostic in the details of agreement, we also would like services to be able to enforce some semantics about what constitute valid and invalid sequences of values. Coming back to the NAT example, the semantic check can ensure that a newly proposed IP-port mapping does not conflict with any previously established ones and can even deal with the leased nature of our IP-port mappings making the decision once (typically at the leader of the Paxos group) as to whether the old lease has expired or not. We accomplish this by having network services optionally provide a function to check the validity of each value before it is proposed.

- `setSemanticCheckPolicy(name, policyhandler)`: Sets the semantic check policy for row `name`. `policyhandler` is an application-specific call-back function that is used to check the validity of the proposed values.
- `check(policyhandler, name, value, seqNum)`: Asks the consensus client if `value` is

a semantically valid value to be put in cell number `seqNum` of row `name`. Returns `true` if the value is semantically valid, `false` if it is not and with an error if the checker has not been informed of all cells preceding cell number `seqNum`.

Finally, each row maintained by the consensus system can have a different set of policies about whether to check for semantic validity, whether to favor safety or liveness (as described below), and even which nodes are serving as the set of replicas.

2.5.1 Catastrophic Failures

Paxos can make progress only when a majority of the nodes are online. If membership changes gradually, the Paxos group can elect to modify its membership. The two critical parameters that determine the robustness of the quorum are the churn rate and the time it takes to detect failure and change the group's membership. The consensus group can continue to operate if fewer than half of the nodes fail before their failure is detected. In such cases, since a majority of the machines in the consensus group are still operating, we have that set vote on any changes necessary to cope with the churn [26].

But if a large number of nodes leave simultaneously (e.g., because of a power outage), we allow services to opt to make progress despite inconsistent state. Each service can pick they want to handle this case for its row, deciding to either favor liveness or safety via the `setForkPolicy` call. If the row favors safety, then the row is effectively frozen until a time when a majority of the nodes recover and can continue to make progress. However, we allow for a row to favor liveness, in which case the surviving nodes make note of the fact that they are potentially breaking safety and fork the row.

Forking effectively creates a new row in which the first value is an annotation specifying the row from which it was forked off, the last agreed upon sequence number before the fork and the new set of nodes which are believed to be up. This enables a minority of the nodes to continue to make progress. Later on, when a majority of the nodes in the original row return to being up, it is up to the service to merge the relevant changes (and deal with any potential conflicts) from the forked row back into the main row via the normal `put` operation and eventually garbage collect the forked row via a `delete` operation. The details of this API are described in Table 5.

While, in theory, building services that can handle potentially inconsistent state is hard, we have found that, in practice, many services admit reasonable solutions. For instance, a NAT which experiences a catastrophic failure can continue to operate and when merging conflicts it may have to terminate connections if they share the same external IP and port, though most of the time there will be no such conflicts.

`setForkPolicy(name, policy)` Sets the forking policy for the row `name` in the case of catastrophic failures. The valid values of `policy` are 'safe' and 'live'.
`delete(name)` Cleans up the state associated with row `name`. Fails if called on a row which is not a fork of an already existing row.
`forkNotify(name, forkName)` Informs the consensus client that because the client asked to favor liveness over safety, the row `name` has been forked and that a new copy has been started as row `forkName` where potentially unsafe progress can be made, but may need to be later merged.

Table 5: API for dealing with catastrophic failures.

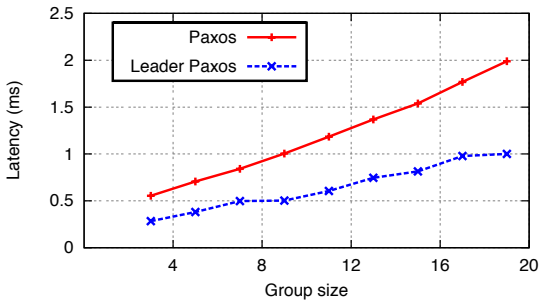


Figure 3: The average time for a Paxos round to complete with and without a leader as we vary the size of the Paxos group.

2.5.2 Implementation

Our current implementation of consensus is approximately 2100 lines of C++ code implementing a straightforward and largely unoptimized adaptation of the Paxos distributed agreement protocol. In Paxos, proposals are sent to all participating nodes and accepted if a majority of the nodes agree on the proposal. In our implementation, one leader is elected per row and all requests for that row are forwarded to the leader. If progress stalls, the leader is assumed to have failed and a new one is elected without concern for contention. If progress on electing a leader stalls, then the row can be unsafely forked depending on the requested forking policy. As nodes fail, the Paxos group reconfigures itself to remove the failed node from the node set and replace it with a different ETTM end-host.

Figure 3 shows the average time for a round of our Paxos implementation to complete when running with varying numbers of `pc3000` nodes (with 3GHz, 64-bit Xeon processors) on Emulab [15]. The results show that a Paxos round can be completed within 2 ms when there is no leader and within 1 ms with a leader. While the computation necessarily grows linearly with the number of nodes, this effect is mitigated by running Paxos on a subset of the active ETTM nodes. For example, as we

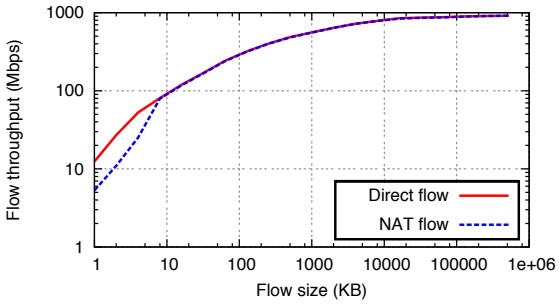


Figure 4: Bandwidth throughput of flows traversing ETTM NAT as we vary the flow size.

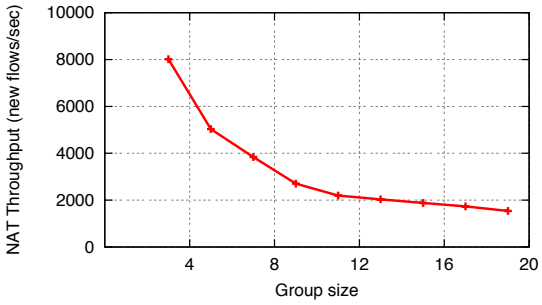


Figure 5: Throughput performance of ETTM NAT as we vary the Paxos group size.

will show in our evaluation of the NAT, a Paxos group of only 10 nodes—with new machines brought in only to replace any departing nodes in the subset—provides sufficient throughput and availability for the management of a large number of network flows.

3 Network Management Services

We next describe the design, implementation, and evaluation of several example services we have built using ETTM. These services are intended to be proof of concept examples of the power of making network administration a software engineering, rather than a hardware configuration, problem. In each case the functionality we describe can also be implemented using middleboxes. However, a centralized hardware solution increases costs and limits reliability, scalability, and flexibility. Proposals exist to implement several of these services as peer-to-peer applications on end-hosts [23, 38], but this raises questions of enforcement and privacy. Instead, ETTM provides the best of both worlds: safe enforcement of network management without the limitations of hardware solutions.

3.1 NATs

Network Address Translators (NATs) share a single externally-visible IP address among a number of different hosts by maintaining a mapping between externally visible TCP or UDP ports and the private, internally-visible IP addresses belonging to the hosts. Mappings are generated on-demand for each new outgoing connection, stored and transparently applied at the NAT device itself. Traffic entering the network which does not belonging to an already-established mapping is dropped. As a result, passive listeners such as servers and peer-to-peer systems can have connectivity problems when located behind NATs. Mappings are usually not replicated, so a rebooted NAT will break all connections.

In contrast, Our ETTM NAT is distributed and fault-tolerant. We store the mappings using the consensus API allowing any participating AEE to access the complete list of mappings. When the NAT filter running in a host's AEE detects a new outgoing flow, it temporarily hold the flow and requests a mapping to an available, externally-visible port. This request is satisfied only if the port is actually available. Once this request completes, the NAT filter begins rewriting the packet headers for the flow and allows packets to flow normally.

Handling incoming traffic is slightly more complicated. If the physical switches on the network support flexible packet forwarding (as with OpenFlow hardware), they can be configured with soft state to forward traffic to the appropriate host where its NAT filter can rewrite the destination address.[5] If the soft state has not yet been installed or has been lost due to failure, default forwarding rules result in the packet being delivered to some host which can appropriately forward the packet and install rules in the physical switches as needed.

Our NAT also works if the physical switches do not support re-configurable routing. Instead, we assign the globally-visible IP address to a specific AEE and have that AEE forward traffic to appropriate hosts. While this might appear to be similar to proxying all external traffic through an end-host, such an approach would be neither fault tolerant nor privacy preserving. In contrast, in ETTM the AEE allows for packets to be silently redirected to the appropriate host without those packets being visible to the user of the forwarding host. Also, the failure of that AEE can be detected and another can be chosen with no lost state. When selecting an AEE, we use historical uptime data as well as information about current load to avoid using unreliable hosts and to avoid unnecessarily burdening loaded hosts. While it is possible that a determined snoop might physically tap their ethernet wire to see forwarded packets, deployments that wish to prevent this could enforce end-to-end encryption using a combination of SSL, IPsec and/or 802.1AE MACsec to encrypt all traffic entering or exiting the organization.

Our NAT can be configured to allow passive connec-

[5] We implement address translation in the AEE despite OpenFlow support because some of our OpenFlow hardware has worse performance when modifying packets. Further, keeping translation tables reliably in AEEs keeps no hard state in the network.

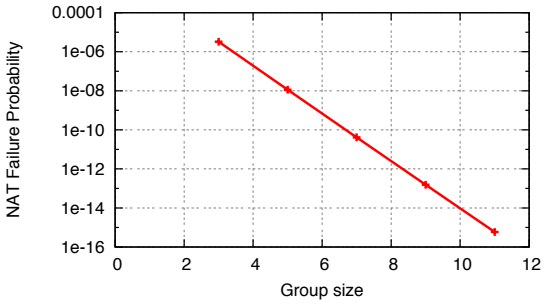


Figure 6: Availability of ETTM NAT as we vary the Paxos group size. Note the y-axis is in log scale.

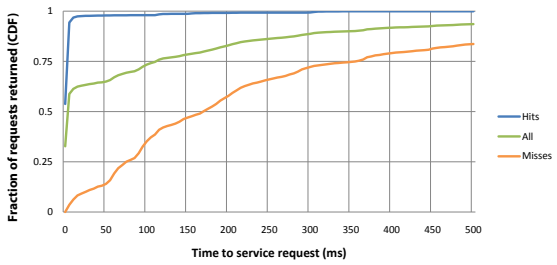
tions to establish mappings. We have implemented a Linux kernel module that can be installed in the guest OS to explicitly notify the NAT filter whenever `bind()` or `listen()` is called, triggering a request for a valid mapping to an external IP address and port. This allows the ETTM system to direct incoming connections to the appropriate host without having the administrator set up customized port forwarding rules. We attempt to provide passive connections with the same external port as its internal one; if this is not possible, the kernel module can be queried for the external port number.

Note that the ETTM approach for implementing NATs reinstates the fate sharing principle. We trivially support multiple ingress points to the network because there is no hard state stored in the network. A connection only fails if either endpoint fails or there is no path between them, but not if the middlebox fails. Even if the consensus group fails entirely, existing flows will still continue as long as one member of the group remains; of course, new flows may be delayed in this case.

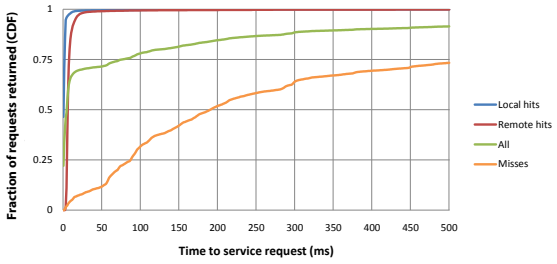
We evaluated the performance of our NAT module on a cluster of `pc3000` nodes on Emulab. Figure 4 depicts the flow throughputs with and without the NAT module for TCP flows of various sizes over a 1 Gbps LAN link. The NAT filter imposes some added cost in terms of the latency of the first packet (about 1-2 ms), which affects the throughput of short flows in the LAN. For all other flows, the throughput of the NAT filter matches that of the direct communications channel, and it achieves the maximum possible throughput of 1 Gbps for large flows.

Figure 5 plots the throughput of ETTM NAT by measuring the number of NAT translations that it can establish per second as we vary the size of the Paxos group operating on behalf of the NAT. While the throughput falls with the number of nodes, it is still able to sustain an admission rate of 2000 new flows per second even with large Paxos groups. Additional scalability would be possible if the external port space were partitioned among multiple Paxos groups.

We also model the NAT failure probability using end-host availability data collected for hosts within the Microsoft corporate network [12, 5]. The trace data has 81% of the end-hosts available at any time, and the median session length of these end-hosts was in excess of 16 hours. Figure 6 plots the probability of catastrophic failures assuming independent failures and a generous failure detection and group reconfiguration delay of 1 minute. As we can see from this analysis, a handful of end-systems would suffice for most enterprise settings.



(a) Latency by request type with a single centralized cache.



(b) Latency by request type with a distributed cache across 6 nodes.

Figure 7: The cumulative distribution of latencies by type of request with a centralized (Figure 7(a)) and distributed (Figure 7(b)) web caches.

3.2 Transparent Distributed Web Cache

It is common for large networks to employ a transparent web cache such as Akamai [1] or squid [38] to improve performance and reduce bandwidth costs. These caches exploit similarity in different users' browsing habits to reduce the total bandwidth consumption while also improving throughput and latency for requests served from the cache.

Even though a shared cache is often very effective, many small and medium sized networks do not use one because of the administrative overhead of setting it up and the potential performance bottleneck if the centralized cache is misconfigured. An alternative is to coordinate caches on each end-host [23], but this requires reconfiguration by each user and it raises privacy concerns since requests can be snooped by anyone with administrative privileges on any machine.

We implemented a distributed and privacy preserving

distributed cache. The cache runs as an ETTM network management service that is triggered by a μvrouter filter capturing all traffic headed to port 80. The service first checks the local AEE's web cache to see if the request can be served from the local host. If it cannot be served locally, the service computes a consistent hash of the request url and forwards it to a participating remote AEE based on the computed hash value. If the remote AEE does not have the content cached, it retrieves the content from the origin server, stores a copy in its local cache, and returns the fetched content to the requesting node. Note that the protocol traffic in ETTM is captured by the web cache filter and is not visible to any of the guest OSes. Also, communication between the caches can be optionally encrypted to prevent snooping. We adapted squid [38] to serve as the cache in each AEE and to provide the logic for interpreting http header directives, such as when to forward requests to the origin due to cache timeouts or outright disabling of caching.

We evaluated our end-host based web-cache implementation using a trace driven simulation. In order to generate trace data we aggregated the browser history of three of the authors and replayed the trace data on six nodes on Emulab [15]. In the centralized experiments, all clients but one have their cache disabled and were configured to send all requests to the one remaining active cache. In the distributed experiments each node runs its own cache. In the centralized case, the single cache is set to 600 MB, while in the distributed experiments the cache size for each of the six nodes is set to 100 MB.

Cache hit rates are similar in both cases. For brevity we omit detailed analysis of hit rates and instead focus on latency. The cumulative distribution of latencies for the centralized and distributed caches is shown in Figure 7. The latency for objects found in the other node's caches is at most a few milliseconds more than local cache hits, indicating that the distributed nature of our implementation imposes little or no performance penalty.

3.3 Deep Packet Inspection

The ability to filter traffic based on the full packet contents and often the contents of multiple packets—commonly called deep packet inspection (DPI)—has quickly become a standard tool alongside traditional firewalls and intrusion detection systems for detecting security breaches. However, the computation required for deep packet inspection is still limits its deployment.

The ETTM approach opens the door to 'outsourcing' the DPI computation to end-hosts where there is almost certainly more aggregate compute power than inside a dedicated DPI middlebox. Traditionally, the idea of running this DPI code at end-hosts would flounder because they could not be trusted to execute the code faithfully—a virus infecting one host could undermine network security. While no security is invulnerable, we offer a narrow attack surface similar to middleboxes, and also use attestation to be able to make claims about booted software and detect malicious changes on reboots.

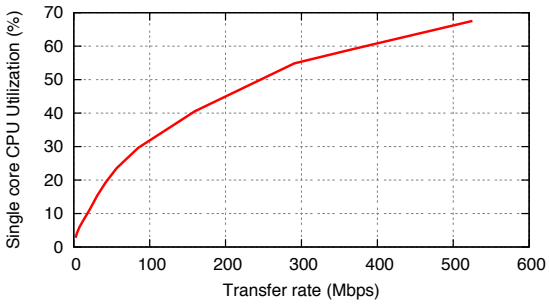


Figure 8: CPU load of ETTM DPI module as we vary the transfer rate of our trace.

Our implementation of DPI is based on the Snort [37] engine and renders decisions either by delaying or dropping traffic or by tagging flows with metadata. The DPI filter is run within the end-host AEE and inspects the flows being sourced from or received by the end-host. In addition, the DPI modules running on end-hosts periodically exchange CPU load information with each other. In situations where the end-host CPU is overloaded, as in highly-loaded web servers, the flows are redirected to some other lightly loaded end-host running the ETTM stack in order to perform the DPI tasks.

The two commonly used applications of DPI are to detect possible attacks and to discover obfuscated peer-to-peer traffic. In the case of detecting attacks, the filter releases traffic after it has been scanned for attack signatures and found to be clean. If a flow is flagged as an attack, no further traffic is allowed, and the source is labeled as being believed to be compromised. In the case of obfuscated peer-to-peer traffic, normal traffic is passed through the DPI filter without delay, but when a flow is categorized as peer-to-peer the flow is labeled with metadata. The next section describes how we can use these labels to adjust priorities for peer-to-peer traffic.

Figure 8 shows benchmark results from a trace-based evaluation of our DPI filter. We ran the ETTM stack on a quad-core Intel Xeon machine with 4 GB of RAM where each core runs at 2 GHz. However, we only make use of one core as `snort-2.8` is single-threaded. The traces are from DEFCON 17 "capture the flag" dataset [13], which contain numerous intrusion attempts and serve as commonly used benchmarks for evaluating DPI performance. We vary the trace playback rate from 1x to 1024x and measured the CPU load imposed by our DPI filter at various traffic rates. Figure 8 shows the load on the ETTM CPU to analyze traffic to/from that CPU. This demonstrates that running DPI on a single core per host is feasible. Stated in other terms, the ETTM approach

of performing DPI computation on end-hosts scales with the number of ETTM machines; centralizing DPI computation on specialized hardware is more expensive and less scalable.

3.4 Bandwidth Allocation

The ability for ETTM to control network behavior on a packet granularity provides an opportunity for more efficient bandwidth management. In TCP, hosts increase their send rates until router buffers overflow and start dropping packets. As a result, it is well-known that the latency of short flows degrades whenever a congested link is shared with a bandwidth-intensive flow. Many large enterprises deploy hardware-based packet shapers at the edge of the network to throttle high bandwidth flows before they overwhelm the bottleneck link. In this subsection, we demonstrate a backwardly compatible software-based ETTM solution to this issue; we use this as an illustration of how ETTM can be used to improve quality-of-service in an enterprise setting.

We call our bandwidth allocation strategy *TCP with reservations* or TCP-R; the approach is similar to the explicit bandwidth signaling in ATM. In TCP-R, bandwidth allocations for the bottleneck access link are performed by a controller replicated using the consensus API. Endpoints managing TCP flows make bandwidth allocation requests to the controller, which responds with reservations for short periods of time. We next describe the logic executed end-hosts followed by the controller logic.

Endpoint: Whenever a new flow crossing the access link appears and every RTT after that, the bandwidth allocation filter on the local host issues a bandwidth reservation request to the controller. The request is for the maximum bandwidth the host needs, that can be allocated safely without causing queueing at the congested link. The controller responds with an allocation and a reservation for the subsequent round-trips.

Once the reservation has been agreed upon, the filter limits the flow to using that amount of bandwidth until it issues a subsequent reservation. The amount of the new reservation is based on the last RTT of behavior. Let $A_f(i-1)$ be the bandwidth allocated to flow f in period $i-1$, and let $U_f(i-1)$ be the bandwidth utilized by it during the period. Then it makes a reservation request $R_f(i)$ based on the following logic; this preserves TCP behavior for the portion of the path external to the LAN, while allowing for explicit allocation of the access link.

- If the flow used up its allocation, it asks the controller to provide it the maximum allowed by the TCP congestion window ($R_f(i) = cwnd/RTT$).
- If the flow did not use up its bandwidth allocation in the previous RTT, then it issues a new request for the lesser of the bandwidth it did use and the TCP congestion window, relinquishing its unused reservation ($R_f(i) = min(cwnd/RTT, U_f(i-1))$).

Controller: The controller allocates bandwidth among the reservation requests according to max-min fairness. It publishes the results by committing its allocation decision across the various controller instances using Paxos. Note that the actual reservation amount can be less than what was requested.

Periodically the controller processes the bandwidth requests and makes an allocation using the following scheme to achieve max-min fairness. It sorts the flows based on their requested bandwidth. Let $R_0 \leq R_2 \leq R_3 ... R_{k-2} \leq R_{k-1}$ be the set of sorted bandwidth requests, L be the link access bandwidth, and $A = 0$ be the allocated bandwidth at the beginning of each allocation round. The controller considers these requests in increasing order and the requested bandwidth or its fair share, whichever is lower. Concretely, for each flow j, it does the following: $A_j = min(R_j, \frac{L-A}{k-j})$ and sets $A = A + A_j$. Note that $\frac{L-A}{k-j}$ is the fair share of flow j after having allocated A bandwidth resources to the j flows considered before it.

In practice, because it takes some time to acquire a reservation, we leave some fraction of the link (10% in our implementation) unallocated and allow each flow to send a few packets (4 in our implementation) before receiving a reservation. Because the time to acquire a reservation (a millisecond or less) is smaller than most Internet round trip times, this avoids adversely affecting flows with increased latency.

TCP-R has many benefits over traditional TCP. It does not drive the bottleneck link to saturation, thereby avoiding losses and sub-optimal use of network resources. In particular, latency sensitive web traffic can obtain their share of the bandwidth resource even if there are simultaneous large background transfers.

This implementation of bandwidth allocation assumes that we are only managing the upload bandwidth of our access link. In the future, we will to extend our implementation to handle arbitrary bottlenecks as well as the allocation of incoming bandwidth.

Evaluation: Our evaluation illustrates the ability of the ETTM bandwidth allocator to provide a fair allocation to interactive web traffic. On Emulab, we set up an access link with a bottleneck bandwidth of 10 Mb/s and compared the latency of accessing `google.com` with and without background BitTorrent traffic that is generated by a different end-host in the network. Figure 9 depicts the webpage access latency at different points in time. When there is no competing traffic, the average access latency is 0.68 seconds. When there is competing traffic (during attempts 11 through 30), the average access

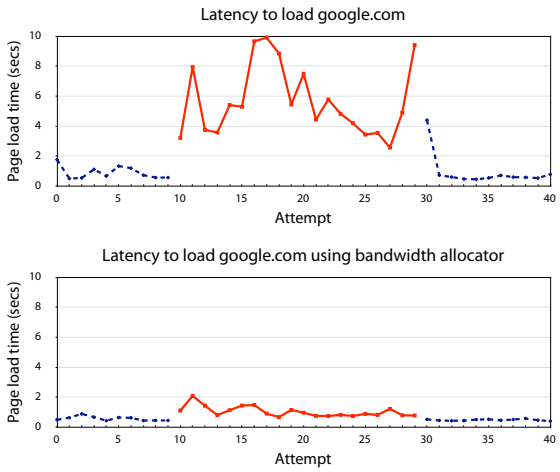


Figure 9: Webpage access latency in the presence of competing BitTorrent traffic with and without the bandwidth allocator. The solid lines depict the access latency when there is competing BitTorrent traffic.

latency is 5.67 seconds if we don't use the ETTM bandwidth allocator. With the ETTM bandwidth allocator, the interactive web traffic receives a fair share and incurs a latency of 1.04 seconds.

4 Related Work

Providing network administrators more control at lower cost is a longstanding goal of network research. Several recent projects have focused on providing administrators a logically centralized interface for configuring a distributed set of network routers and switches. Examples of this approach include 4D [34, 17, 42], NOX [19], Ethane [8, 7], Maestro [6] and CONMan [2]. Of course, the power of these systems is limited to the configurability of the hardware they control. While we agree with the need for logical centralization of network management functions, our hypothesis is that network administrators would prefer fine-grained, packet level control over their networks, something that is not possible at line-rate with today's current low cost network switches.

Other efforts have focused on building drop-in replacements for the the virtual ethernet switch inside existing hypervisors. Cisco's Nexus 1000V virtual switch [9, 40] provides a standard Cisco switch interface enabling switching policies to to the edge of VMs as well as hosts. Open vSwitch [33] accomplishes a similar feat, but provides an OpenFlow interface to the virtual switch and is compatible with Xen and a few other hypervisors. Still others are working to do hardware network I/O virtualization [32]. While all of these tools give network administrators additional points of control, they do not offer the flexibility required to implement the breadth of coordinated network polices administrators seek today. Instead, we are working to incorporate these standardized, simple points of control into ETTM to provide potentially higher performance some tasks and added control over the low-level network.

Other systems have tried to bring end-hosts into network management, though in limited ways. Microsoft's Active Directory includes Group Policy which allows for control over the actions which connected Windows hosts are allowed to carry out, but enforces them only assuming the host remains uncompromised. Network Exception Handlers [24] allow end-hosts to react to certain network events, but still leaves network hardware dominantly in control. Still other work [11] uses end-hosts to provide visibility into network traffic, but does not provide a point of control and assumes that the host remains uncompromised.

Other recent work has attempted to increase the flexibility of network switches to carry out administrative tasks. OpenFlow [30] adds the ability to configure routing and filtering decisions in LAN switches based on pattern matching on packet headers performed in hardware. A limitation of OpenFlow is throughput when packets need to be processed out of band, because there is typically only one underpowered control processor per LAN switch. In ETTM, we invoke out of band processing on the switch only for the initial TPM verification when the node connects, while still allowing the network administrator to add arbitrary processing on every packet.

Middleboxes have always been a contentious topic, but recent work has looked at how to embrace middleboxes and treat them as first-class citizens. In TRIAD [18] middleboxes are first-order constructs in providing a content-addressable network architecture. The Delegation-Oriented Architecture [41] allows hosts to explicitly invoke middleboxes, while NUTSS [20] proposes a novel connection establishment mechanism which includes negotiation of which middleboxes should be involved. Our work can be seen as enabling network administrators to place arbitrary packet-granularity middlebox functionality throughout the network, via validated software running on end-hosts.

Existing work has leveraged trusted computing hardware to avoid vulnerabilities in commodity software [35] as well as to ensure correct execution of specific tasks [29]. Our use of trusted computing hardware is complementary to these efforts.

5 Conclusion

Enterprise-level network management today is complex, expensive and unsatisfying: seemingly straightforward quality of service and security goals can be difficult to achieve even with an unlimited budget. In this paper, we have designed, implemented and evaluated a novel approach to provide network administrators more control at lower cost, and their users higher performance, more

reliability, and more flexibility. Network management tasks are implemented as software applications running in a distributed but secure fashion on every end-host, instead of on closed proprietary hardware at fixed points in the network. Our approach leverages the increasing availability of trusted computing hardware on end-hosts and reconfigurable routing tables in network switches, as well as the expansive computing capacity of modern multicore architectures. We show that our approach can support complex tasks such as fault tolerant network address translation, network-wide deep packet inspection for virus control, privacy preserving peer-to-peer web caching, and congested link bandwidth prioritization, all with reasonable performance despite the added overhead of fault tolerant distributed coordination.

Acknowledgements

We would like to thank our anonymous reviewers and our shepherd David Maltz for their valuable feedback. This work was supported in part by the National Science Foundation under grants NSF-0831540 and NSF-0963754.

References

[1] Akamai technologies. http://www.akamai.com/.

[2] Hitesh Ballani and Paul Francis. CONMan: A step towards network manageability. In *SIGCOMM*, 2007.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] Blue Coat Systems. Blue Coat PacketShaper. http://www.bluecoat.com/products/packetshaper.

[5] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS*, 2000.

[6] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A new architecture for realizing and managing network controls. In *LISA Workshop on Network Configuration*, 2007.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[8] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *USENIX Security*, 2006.

[9] Cisco Systems. Cisco Nexus 1000V Series Switches - Cisco Systems. http://www.cisco.com/en/US/products/ps9902/index.html.

[10] OpenFlow Consortium. OpenFlow >> OpenWrt. http://www.openflowswitch.org/wp/openwrt/.

[11] Evan Cooke, Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX*, 2006.

[12] D. Narayanan and A. Donnelly and R. Mortier and A. Rowstron. Delay Aware Querying with Seaweed. In *VLDB*, 2006.

[13] Defcon 17 ctf packet traces. http://www.ddtek.biz/dc17.html.

[14] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT), 1994.

[15] Eric Eide, Leigh Stoller, and Jay Lepreau. An experimentation workbench for replayable networking research. In *NSDI*, 2007.

[16] FreeRADIUS: The world's most popular RADIUS Server. http://freeradius.org/.

[17] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. In *CCR*, 2005.

[18] Mark Gritter and David R Cheriton. An architecture for content routing support in the internet. In *USITS*, 2001.

[19] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. In *CCR*, 2008.

[20] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*, 2007.

[21] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *CCS*, 2000.

[22] RFC 3220: IP Mobility Support for IPv4, 2002.

[23] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, 2002.

[24] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. Network exception handlers: Host-network control in enterprise networks. In *SIGCOMM*, 2008.

[25] Leslie Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.

[26] Leslie Lamport. Paxos Made Simple. In *SIGACT*, 2001.

[27] Ratul Mahajan, Neil Spring, David Wetherall, and Thomas Anderson. User-level Internet Path Diagnosis. In *SOSP*, 2003.

[28] Jouni Malinen. Linux WPA Supplicant (IEEE 802.1X, WPA, WPA2, RSN, IEEE 802.11i). http://hostap.epitest.fi/wpa_supplicant/, January 2010.

[29] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, April 2008.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. http://www.openflowswitch.org/documents/openflow-wp-latest.pdf, March 2008.

[31] OpenWrt. http://openwrt.org/.

[32] PCI-SIG. PCI-SIG - I/O Virtualization. http://www.pcisig.com/specifications/iov/.

[33] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *HotNets*, 2009.

[34] Jennifer Rexford, Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Geoffrey Xie, Jibin Zhan, and Hui Zhang. Network-wide decision making: Toward a wafer-thin control plane. In *HotNets*, 2004.

[35] Seshadri, Arvind, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[36] S. Shenker, C. Partridge, and R. Guerin. RFC 2212: Specification of Guaranteed Quality of Service, 1997.

[37] Snort. http://www.snort.org.

[38] squid : Optimizing Web Delivery. http://www.squid-cache.org/.

[39] Trusted Computing Group. TPM Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, August 2007.

[40] VMware, Inc. Cisco Nexus 1000V Virtual Network Switch: Policy-Based Virtual Machine Networking. http://www.vmware.com/products/cisco-nexus-1000V/.

[41] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[42] Hong Yan, David A. Maltz, T. S. Eugen Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D network control plane. In *NSDI*, 2007.