Network Exception Handlers: Host-network Control in Enterprise Networks

Thomas Karagiannis, Richard Mortier and Antony Rowstron
{thomkar, antr}@microsoft.com, mort@vipadia.com
Microsoft Research
Cambridge, UK

ABSTRACT

Enterprise network architecture and management have followed the Internet's design principles despite different requirements and characteristics: enterprise hosts are administered by a single authority, which intrinsically assigns different values to traffic from different business applications.

We advocate a new approach where hosts are no longer relegated to the network's periphery, but actively participate in network-related decisions. To enable host participation, network information, such as dynamic network topology and per-link characteristics and costs, is exposed to the hosts, and network administrators specify conditions on the propagated network information that trigger actions to be performed while a condition holds. The combination of a condition and its actions embodies the concept of the network exception handler, defined analogous to a program exception handler. Conceptually, network exception handlers execute on hosts with actions parameterized by network and host state.

Network exception handlers allow hosts to participate in network management, traffic engineering and other operational decisions by explicitly controlling host traffic under predefined conditions. This flexibility improves overall performance by allowing efficient use of network resources. We outline several sample network exception handlers, present an architecture to support them, and evaluate them using data collected from our own enterprise network.

Categories and Subject Descriptors: C.2.3 [Computer- Communication Networks]: Network Operations
General Terms: Management
Keywords: Enterprise networks, management, network exception handlers

1. INTRODUCTION

Enterprise networks have largely inherited management and design principles from the Internet. The Internet can be characterized as best-effort and application-agnostic, with hosts viewing the Internet as a black-box, and network-related decisions all being made independently from hosts. These characteristics have enabled the Internet to scale rapidly and have facilitated the deployment of diverse technologies and applications. However, hosts have been relegated to the periphery, playing almost no part in ensuring the smooth operation of the network.

As with an Internet Service Provider (ISP) network, managing an enterprise network is currently an expensive and time-consuming task, requiring highly skilled individuals on call 24/7. Large enterprise networks are at least as complex as tier-one ISP networks, providing numerous services with strict security and operational requirements to a large user base spanning several countries or even continents. However, enterprises differ from ISPs in two key aspects: all hosts are operated by a single administrative authority, and enterprises can estimate and express the intrinsic business value of the traffic generated by each application.

Enterprise network operators are forced to indirectly exert control over network decisions by deploying complicated mechanisms on top of pre-existing network services. For example, control is currently applied by using flow-level Equal Cost Multi-Path in order to achieve load-balancing in the network, or in extreme cases by running BGP with each router acting as a separate AS to achieve routing flexibility. Traffic management is often done crudely, for example, using port-based filters at routers to shape traffic. Artifacts like random port mappings at proxies and the use of arbitrary port numbers by applications such as Skype make port-based policies inefficient and inaccurate.

This leads to a tension between enterprise network management requirements and available functionality. We believe it is time to rethink the management and principles of modern enterprise networks to *include hosts in the overall "network equation"*. The aim is to provide administrators with a level of control over the network that they currently simply cannot have, by allowing hosts to participate in network-related decisions.

To achieve this we propose the concept of *network exception handlers* which are deployed at hosts, and are analogous to *program exception handlers*. A network exception handler describes an action, specified by the network operator in advance, to be executed on hosts when a network exception condition occurs. For example, a simple network exception handler could be: *if* backbone link X fails, *then* throttle, on all hosts, any traffic generated by application Y that would have been routed across the failed link. In order to implement network exception handlers, the network needs to expose to the hosts dynamic information about the network topology, potentially augmented with other information, such as link capacities and costs. This can be achieved without modifying the core networking functionality, such as the current generation of routers and networking infrastructure.

Network exception handlers allow hosts to play an active role in network operations, beyond the simple application of traditional transport-layer congestion control techniques. Modern hosts are typically powerful with substantial under-utilized processing and storage resources, and they are responsible for the traffic crossing the network and knowledgeable about the users and types of applications running on them. Hosts are ideally placed to help manage the complexity of the network, and to enable services and functionality that today's enterprise networks cannot support.

Network exception handlers can be used to improve the performance of the network and to broaden the range of functionality it provides. They enable per-user, per-host and per-application traffic management, features currently absent from enterprise networks. They can simplify network management, thereby reducing network operational costs and close the gap between network traffic engineering, application requirements, and management policies. Their benefits come from the extra context and control available at the hosts: traffic can be shaped in the network, but only hosts can correctly identify the applications, users and users' business roles generating the traffic, and thus implement policies that allow priority to be given to selected applications, users or roles. These policies are pre-defined and embody the application prioritization that enterprises already understand, based on their business requirements.

We propose an architecture to support the concept of network exception handlers. Our architecture can be incrementally deployable and supports the collection, synthesis and distribution to hosts of topology data together with other performance data from the network. Using trace data collected from our enterprise network, we justify this architecture and evaluate its overheads. We also use the collected data to motivate and evaluate examples of network exception handlers.

Our contributions can be summarized as follows:

- We challenge existing enterprise network architectures by describing network exception handlers and outlining potential uses (Sections 3 and 4).
- We propose an architecture that exposes network-related information to hosts and allows for informed, application-specific policy decisions to be taken at the edge of the network (Sections 5 and 6).
- We evaluate the feasibility, requirements and performance of the proposed architecture using topology and traffic data gathered from a large operational enterprise network (Section 7).

2. ENTERPRISE NETWORKS

To motivate the design of network exception handlers, it is first important to understand the characteristics and requirements of enterprise networks that drive specific design choices. In this section, we give a general characterization of modern enterprise networks, highlighting specific features that motivate and facilitate the concept of network exception handlers.

Enterprise networks support upwards of several hundred hosts, with the largest supporting many hundreds of thousands. Current host hardware consists of powerful desktop and laptop computers, *the majority of which are under-utilized, running modern commodity operating systems supporting complex traffic management features*. Hosts are distributed between multiple buildings with larger enterprises containing many hundreds of buildings spread through many countries and even across multiple continents. Larger enterprises operate one or more datacenters which may serve the entire enterprise or be designated to serve a particular continent, country, campus or building. Local connectivity within buildings is provided by the enterprise itself but backbone links providing wide-area connectivity between campuses and countries are typically leased from, or completely outsourced to, a network provider.

The purpose of the enterprise network is to support the many different networked applications that in turn support the business objectives of the company. Provisioning wide-area bandwidth to support all these application requirements forces enterprises to provision for peak loads, causing significant cost overheads. In contrast to an ISP, *a business can explicitly make a judgment as to the value of each application's traffic under normal and abnormal operating conditions*. For example, a business might value email and real-time video conferencing more than access to an archive of training videos. Therefore, under abnormal conditions due to, e.g., link failures, traffic on particular links may fall outside expected norms, making explicit prioritization of email access and real-time video conferencing over streaming training video data desirable. However, modern enterprise networks currently provide limited visibility and means of control; mechanisms such as MRTG[1], offer only aggregate information to IT departments, but provide no hints about how network resources are utilized by the various applications.

The enterprise network and all hosts attached to it operate as a single co-operative administrative domain. Host configuration is managed using a directory service, such as Active Directory.[2] This enables the IT organization to control the software installed and permitted to run on hosts, to control which network resources hosts and users are permitted to access, and generally to manage most aspects of host configuration and network access. By specifying *global policies*, IT operators manage the network's hosts as aggregates rather than needing to control the detailed configuration of each host. For example, in our network, all domain joined machines are required to present suitable authentication tokens when accessing domain controlled resources (file servers, printers, etc.). These tokens can only be gained after the domain has ensured that the host is compliant with global policy in terms of required patches applied, suitable local user and group settings, etc.

This sort of global policy system permits IT operators to make assumptions about the behavior of hosts within their network. Specifically, IT operators can assume that hosts will follow specified global policies because they expect that hosts cannot access network resources without having applied global policies.

A consequence of being a single administrative domain is that enterprises can more freely share network information. In general, enterprises run link-state routing protocols such as OSPF [13] as it is acceptable for all routers owned by the organization to understand the topology. Enterprises may also use protocols such as BGP often due to scalability issues, especially when the network is large enough that it cannot easily be handled by current link-state protocol implementations alone. Alternatively, use of BGP could be employed to express complex traffic management policies which require the extra control over route propagation that BGP provides.

Finally, many enterprises actively try to reduce their networking costs. Some outsource the wide-area connectivity to a network provider offering, for example, BGP-based MPLS VPN tunnels [17]. The outsourcing will include a service level agreement specifying the acceptable downtime and other performance parameters. From each of the enterprise's sites, the underlying network topology appears as a simple full-mesh topology between the enterprise's sites. Additionally, many enterprises with branch offices use WAN optimizers, for example from Riverbed[3], which optimize

[1]Multiple Router Traffic Grapher, `http://www.mrtg.com/`.
[2]`http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/default.mspx`
[3]`http://www.riverbed.com/`

```
Network Exception Handler: Backbone link failed
links = {   [10.39.11.40, 10.39.12.30, "SVC to SF"]
            [10.39.19.17, 10.39.22.12, "LA to SVC"] }
boolean Exception(NetworkState)
begin
    foreach link in links do
        if link not in NetworkState then return true
    return false
end
void Fire (NetworkState, HostState)
begin
    print "Link failure"
    foreach link in links do
        if link not in NetworkState then
            print "Link failed: " + link.description
    Register(Handler)
end
void Handler (NetworkState, HostState)
begin
    if not Exception(NetworkState) then
        RemoveAllRestrictions(); DeRegister(Handler);
        print "Link restored: " + link.description
        return
    foreach process in HostState.processes do
        if HostState.MachineRole is developer then
            if process.name is not "SourceRepositoryClient" then
                SetRestriction(process, maxBandwidth=0)
        else if "The Boss" in HostState.LoggedOnUsers then
            if process.name is "VideoStreamingClient" then
                SetRestriction(process, maxBandwidth=0)
            else if process.name is "EmailClient" then
                SetRestriction(process, maxBandwidth=300kbps)
            else if process.name is not "CallCenterApp" then
                SetRestriction(process, maxBandwidth=0)
end
```

Figure 1: Network exception handler for a backbone link failure.

traffic transmitted over WAN links with the aim of reducing link capacity requirements. This does not impact the network topology. Later in the paper we show how network exception handlers can be used to smooth bandwidth usage over time, potentially achieving the same benefits as a WAN optimizer without requiring additional dedicated networking infrastructure.

3. EXCEPTION HANDLERS

Network exception handlers allow network administrators to encapsulate network management policy that is to be enforced when a particular condition holds. Before discussing usage scenarios and the system architecture, we first describe the concept of network exception handlers.

For network exception handlers to be an efficient management mechanism, we require that they be simple yet flexible enough to capture a wide range of useful policies. Analogous to program exception handlers, a network exception handler comprises two parts: (*i*) the exception *condition* which triggers the handler, and (*ii*) an *action* which defines behavior to impose when the condition is met. Fig. 1 shows a simplified example of a network exception handler. In particular, the network exception handler is composed of three basic functions: *Exception()*, which embodies the *condition* that must hold for the exception to occur, *Fire()* which is called when the exception is first detected, and *Handler()* which is called during the period that the exception holds. The latter two functions together comprise the *action*. Information about the network is exposed in a single data structure, called *NetworkState*, which includes information about the network topology, link loads, and other information against which the network administrator writes exception conditions. Host information is exposed in a separate data structure called *HostState*, containing details such as the connections opened by each application, their network usage and the user running each application. In later sections, we describe how the information in these two data structures is maintained.

Conceptually, an exception handler is in one of two states: *passive* or *active*. While *passive*, Exception() is evaluated each time the NetworkState is modified. If true, the state becomes *active* and Fire() is called. When active, Handler() is invoked each time the NetworkState or HostState is modified. Handler() is required to explicitly deregister itself when the exception ceases to hold, placing the exception handler back in the passive state.

Exception() uses simple predicates involving measurable quantities of the network, e.g., link loads, router interface counts, etc. Fig. 1 is a simplified example demonstrating the general functionality network exception handlers expose. The exception becomes true when either of two backbone links are not present. When this occurs, Fire() displays a warning alerting the user and registers the Handler() function. Note that it is possible to register multiple, possibly different, Handler() functions. The registered Handler() function is then invoked each time the NetworkState or HostState is modified. In this example, Handler() checks if the exception still holds, and if the exception does hold, then Handler() enforces a set of policies at each host based on the role of the host, its users and its running applications. If the exception does not hold, Handler() removes all restrictions created by this specific exception handler, notifies the user that the link is restored and deregisters itself. This operation offers an effective mechanism for communicating to the user that network problems have been resolved, an operation which currently requires either explicit notification from a network administrator, or potentially user initiated actions (e.g., probing).

Locality and cascading events. We impose two important constraints on the potential actions of a network exception handler: *they should not impose non-local behavior* and *they should not require explicit coordination between hosts*. In many circumstances, it may be attractive to alter traffic patterns globally, for example by having hosts use loose source routing to reroute traffic when certain exceptions occur, such as a congested link. However, shifting traffic around the network by rerouting can lead to unexpected and highly undesirable persistent load oscillations, causing the triggering of further, potentially cascading, exceptions. Mitigating such effects would require complex real-time coordination between Handler() functions running on different hosts. Such policies would effectively increase the burden of management by introducing further complexity in both expressing and implementing such exception handlers.

Therefore, we want to ensure that exception handlers can only perform local operations: a handler should only be able to perform actions *that shape locally generated traffic*[4]*, or provide feedback to the user*. We will show in the next section that this practice still enables significant benefit and control over the network.

Time-scales of operation. Network exception handlers are not suitable for tracking very short-lived or bursty phenomena, such as instantaneous peaks on the load of a link. The time-scale of operation depends on the network exception handler under consideration,

[4]Note that shaping traffic locally can still have remote benefits, e.g., by mitigating congestion in the core of the network or at other remote sites, without leading to cascading events.

but in general handlers target exceptions that would significantly alter the network state and affect the user-perceived application performance.

It seems that it could be possible to build tools to perform static analysis of exception handlers, which might allow them to perform some, perhaps limited, non-local operations, but we leave this as an open question.

4. USES OF EXCEPTION HANDLERS

In this section, we present three sample usage scenarios for network exception handlers: (*i*) efficient response to link failures; (*ii*) application-aware traffic engineering; and (*iii*) non-transient congestion response. A combination of existing tools might offer potential solutions for some of these examples. However, network exception handlers provide a flexible framework that facilitates efficient and straightforward management for many scenarios through the same mechanism. For example, while an ICMP link failure may inform of an unreachable host or link (scenario (*i*)), such a mechanism cannot provide information for a link overload (scenario (*iii*)). Compared to existing tools, network exception handlers also largely automate the process of notifying the user of potential problems in the network and determining whether these problems persist, as information on the various exceptions is readily available at the host.

Efficient response to link failures. Network exception handlers allow detailed control of the network's response to failures. For example, consider an exception handler such as the following: *when links in the set $\{L_i\}$ fail, only applications in the set $\{A_j\}$ may transmit packets that will be routed across links in the set $\{L_k\}$*, i.e., when critical links fail, only a subset of applications may send traffic over the specified backup links. This ensures that critical traffic traversing these backup links does not experience congestion during what is already a troubled time for the network. This policy could be extended by restricting the set of hosts allowed to transmit. In contrast, current enterprise networks can only enforce inaccurate or coarse-grained control in response to link failures, such as applying filters to drop traffic using certain port numbers being transmitted or received by certain hosts, leaving hosts essentially unaware of the failures and thus unable to adapt their traffic demands appropriately. Effectively, *pushing topology information to hosts enables accurate control of application traffic dependent on the failure, at timescales that routing protocols cannot achieve.*

A key question underlying the utility of such a policy is *how frequent are link failures in enterprise networks?* Fig. 2 presents an initial analysis of the frequency of OSPF link failures, i.e., loss of OSPF adjacency, using a dataset collected from our global enterprise network. The dataset is described in detail in Section 7.1. The figure presents two dimensions of such failures (from left to right): Cumulative Distribution Functions (CDFs) of link downtime, and failure inter-arrivals per link, from the two different perspectives of our local stub area, and the regional backbone.

We observe that failures occur quite frequently in both cases (e.g., one failure per link every hour for 30% of the links in the stub area or for 60% of the links in the backbone area) but downtimes can also be significant: 20% of all failures last over 10 minutes. Link failures are triggered by the absence of OSPF traffic on the adjacency for 40 seconds [13]. In the figures, we consider failures that last over 30 seconds (i.e., the failed link was not advertised as active within 30 seconds after its withdrawal).

In the previous section, we presented an example of an exception handler that could be triggered in such failure cases (Fig. 1). Performing similar actions in the network to those in Fig. 1 is currently not possible, as not only is per-application packet shaping

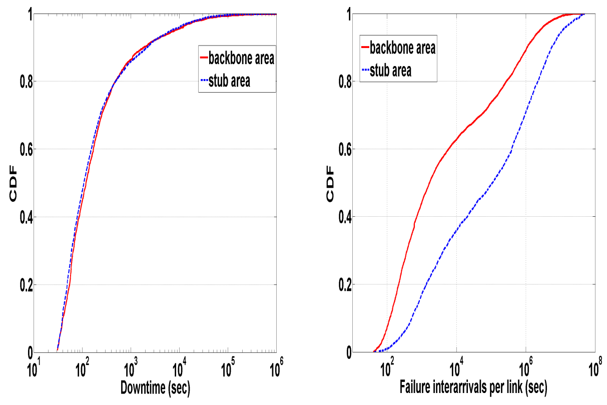


Figure 2: OSPF failure rates. From left to right: CDF of link downtimes, and the CDF of failure inter-arrivals per link. The median for failure inter-arrival per link is roughly a day for the stub area, while link downtimes lasting more than 10 minutes account for more than 20% of all failures.

not effective, but it also requires further information about the role of the host and the user of the application. Fig. 3 shows an exception handler triggered in the case of a failure of a building link. The exception handler proactively reports a message on each host within the building when the link fails, and each such host drops all traffic that would have traversed the link, effectively not generating traffic that would be dropped later in the network. When the link is restored, the exception handler reports this to the user. Similarly, all traffic to the disconnected subnets is dropped for hosts not in the building, and the users attempting to access resources in those subnets are informed. Only hosts that communicate over the specified link will be informed about the outage. Network exception handlers provide two advantages here: (*i*) traffic that would be dropped later in the network is not generated at the hosts, and (*ii*) users are informed during periods of disconnection, which is currently a nontrivial task. For example, in our enterprise, the standard practice would be to send an email to the affected users to inform them about the building link failure. However, since the email servers reside in our European Data Center which becomes inaccessible during such failures, the users are actually only informed once the link is again active!

Application-aware traffic engineering. Network exception handlers allow for efficient, detailed policies with regards to traffic engineering. For example, a policy for such a case might be as follows: *when traffic on link L reaches a predetermined threshold, T, non-critical applications in the set $\{A_i\}$ running on hosts in the set $\{H_j\}$ should be rate limited.* T here might be defined using the real economic cost of the link, e.g., the 95^{th} percentile should always be less than T Mbps. In Section 7, we study the feasibility and effectiveness of a network exception handler that applies such a policy. As previously noted, even highly controlled networks such as enterprise networks currently have no means of explicitly engineering their traffic with respect to applications. Several factors, from use of arbitrary port mappings at proxies to arbitrary application port usage and layer-3 or layer-4 encryption, prohibit fine-grained application control in practice. The net effect is that traffic can only be shaped accurately on a per-application basis at the host. *Enterprise networks explicitly do not want net neutrality, and this desire can be satisfied through network exception handlers.*

Fig. 4 shows an example of a simple policy that, during office hours, limits the bandwidth used by a particular application, in this

```
Network Exception Handler: Building link failed
links = { [ 10.39.11.40, 10.39.12.30, "Edinburgh to London"] }
Edinburgh_subnets = [ 10.39.18.0/24, 10.39.19.0/24 ]
boolean Exception(NetworkState)
begin
    foreach link in links do
        if link not in NetworkState then return true
    return false
end
void Fire (NetworkState, HostState)
begin
    if HostState.MachineLocation is "Edinburgh" then
        print "Building disconnected: do not panic"
    Register(Handler)
end
void Handler (NetworkState, HostState)
begin
    if not Exception(NetworkState) then
        RemoveAllRestrictions(); DeRegister(Handler);
        if HostState.MachineLocation is "Edinburgh" then
            print "Network connectivity restored"
        return
    SetRestriction(TrafficSrc() not in Edinburgh_subnets
                   and TrafficDest() in Edinburgh_subnets,
                   DropAndReport(TrafficDest()+" disconnected"))
    SetRestriction(TrafficSrc() in Edinburgh_subnets
                   and TrafficDest() not in Edinburgh_subnets,
                   Drop());
end
```

Figure 3: Network exception handler for a building disconnection.

case the *UpdateAndPatch* client. The handler would thus be active during the specified hours. Applying such a policy in the network would be feasible only if the application used a predetermined port number. Since many update applications run over HTTP on port 80, simple packet shaping would be insufficient as information about the source application is required.

Congestion response. In a related scenario, network exception handlers may allow hosts to directly alleviate congestion in the network. Currently, congestion avoidance is left to transport protocols such as TCP which operate over short timescales, e.g., loss of individual packets. Hosts operating such protocols must react in certain ways when congestion is observed: simplistically put, a host should linearly increase its network usage until it observes loss, when it should multiplicatively back-off so that all hosts sharing a path can obtain an approximately equal share of that path's resources.

While certainly necessary in the context of IP networks for maintaining the performance of the network under times of high load, this behavior is quite limiting and may not even be desirable in an enterprise network. It is based on a very limited signal from the network (observed packet loss) and permits only one, quite simplistic, response (decrease of the window of outstanding data on the flow observing congestion). By specifying policies with respect to link utilization, network exception handlers allow hosts to manage the usage of network resources more smoothly, over longer time periods, and even pro-actively, i.e., before congestion is directly experienced as with ECN [15].

For example, a policy may specify that a certain host's flows should be rate-limited when specific links are loaded to a predefined threshold, e.g., 80% utilization. In this scenario, a notification of impending congestion is provided before an actual loss event.

```
Network Exception Handler: Time-based traffic cap
links = { [ 10.39.15.60, 10.39.15.61, "Paris to Amsterdam" ] }
boolean Exception(NetworkState)
begin
    return NetworkState.LocalTimeNow in range 8AM to 5PM
end
void Fire (NetworkState, HostState)
begin
    Register(Handler)
end
void Handler (NetworkState, HostState)
begin
    if not Exception(NetworkState) then
        RemoveAllRestrictions(); DeRegister(Handler); return
    SetRestriction("UpdateAndPatch", maxBandwidth=50kbps)
end
```

Figure 4: Network exception handler to shape update traffic during local working hours.

Similarly, network exception handlers can prevent new *flows* from a host entering an overloaded link, a level of control unavailable to TCP-like mechanisms. This scenario does not require coordination of hosts, and network exception handlers can work together with ECN-type mechanisms, as they are orthogonal, operating on different time-scales. *Network exception handlers provide an early and efficient congestion avoidance mechanism unavailable through current transport-layer protocols.*

Summary. All these scenarios are examples of policies that rely on *per-application* knowledge that cannot be easily, if at all, supported in today's enterprise networks. Network exception handlers exploit the fact that hosts have both the knowledge of application requirements and the ability to apply per-application policies, enabling the handling of these exceptions. Network exception handlers are a generic mechanism enabling fine-grained control of the traffic generated by hosts.

5. DESIGN SPACE

The previous section described scenarios where network exception handlers could be employed to better manage the network experience. The underlying principle is that network and hosts should collaborate to better control the performance of the enterprise network. Network exception handlers require information about the network and hosts, denoted by *NetworkState* and *HostState* respectively in the examples. In order to create and maintain this state, information from the hosts needs to be distributed to the network devices controlling the network, or vice versa, information from the network needs to be available at the host. Both approaches have advantages and disadvantages which we now discuss.

In the network core, information about the current observed state of the network is available with low latency. However, making the host state information available in the network, either through inference techniques or through explicit signaling, even at the first hop router appears expensive, if not practically infeasible. Inferring applications in the network is hindered by (*i*) the increasing use of encryption, (payload and even TCP-headers might not be available with IPSEC, thus making deep packet inspection techniques impractical), and (*ii*) other obfuscation techniques, such as the widespread overloading of ports [12], a practice which is typical in enterprise networks. Additionally, network devices cannot infer other host-specific context information, like the user that generated the traffic, the user's business role, or the application's business value. For example, network devices may be able to detect

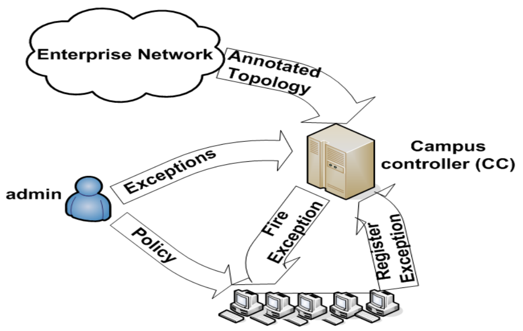


Figure 5: Information flow between the different entities of the architecture.

that media is being streamed to a host, but devices are agnostic as to whether the application is destined for a training video application or a video conferencing application. An alternative would be to implement such functionality at the edge routers but as multi-homing becomes increasingly popular, such solutions appear quite complicated. For example with laptops connected to both wireless and wired infrastructure, there is no longer a single edge network device which mediates access to the network for each host.

In contrast to implementing network exception handlers in the network, enterprise hosts have plentiful resources available and access to context of the network usage, including application names, process IDs, user IDs, etc. Hosts can further implement complex policies on local traffic easily, and policies can easily be functions of multiple applications generating traffic to different hosts over different network interfaces. This enables richer policies to be specified through exception handlers. Implementing the action at the host also implies that "unwanted" traffic potentially never enters the network. This is advantageous; for example, an administrator has the ability to deploy network exception handlers that are fired when users connect over VPN links to the enterprise, and block a set of non-essential business applications from transmitting. The enterprise would still incur the cost of the traffic should such a policy be implemented in the network, whereas a host implementation would block the traffic at its source. The challenge of realizing such policies on hosts however, comes from the fact that the network is virtually a black box to the host, and distributing all topology and other network information to each host in a robust way is expensive.

5.1 A Two-tier Architecture

This motivates us to advocate the implementation of network exception handlers in a two-tier architecture, similar to how existing enterprise services, such as directory services, are implemented. A set of servers are deployed across the network, referred to as *campus controllers (CC)*. In many enterprise deployments these would be collocated with servers operating other network-wide services, such as directory services, e.g., domain controllers supporting Active Directory. As an extreme, the *CC* functionality could be integrated with the functionality of the edge router. Each host authenticates with a *CC* nearby in the network, in the same way that the host authenticates and connects to an Active Directory domain controller. Each *CC* maintains an instance of the *NetworkState* object, representing the annotated network topology. In the next section, we describe in more detail how this NetworkState information is maintained. Each *CC* is responsible for interacting with a set of routers using for example SNMP, and aggregating the locally gathered information. It then disseminates this information to all other *CC*s using either IP multicast or an overlay. All information is associated with a lease, and if the set of *CC*s should become partitioned, the information contributed by unreachable *CC*s will timeout and be removed from the disconnected *CC*s.

Network exception handlers are distributed to the hosts, either exploiting the existing infrastructure available for host management in the enterprise or utilizing the *CC*s. The distribution and update of network exception handlers is thus expected to occur with low latency. Each host determines the set of network exception handlers that it is required to use, and then registers the exception condition of each with the *CC*. The way in which network exception handlers are defined means the exception condition is captured within the Exception() function, and this is parameterized only by NetworkState. This means that a host can easily delegate the evaluation of the set of Exception() functions it must use to the *CC*. When the *CC* detects that an Exception() has triggered, it informs the host which then executes Fire() and the registered Handler() locally. This requires that the *CC* exposes the NetworkState to any host that currently has active exceptions. This can be optimized in many ways, including having the hosts register filters on NetworkState to ensure that only relevant state-changes are propagated to the hosts. De-activating the exceptions occurs locally at the end hosts as part of the evaluation of the handler.

Each *CC* should be able to support a large-number of hosts so that it would be feasible for a single *CC*, for example, to support an average building with 500+ hosts. The overhead of checking the exception conditions is bounded on the total number of unique network exception handlers, rather than on the sum of the number of exception handlers per host. This implies that if the same exception handler is installed on all hosts associated with a *CC*, then the *CC* needs only to evaluate the associated Exception() handler once, and then trigger all connected hosts. Similarly, if any processing or filtering of the NetworkState is necessary, such a process will only be performed once, irrespective of the number of clients that have registered the exception; updates will then be distributed to all hosts with the corresponding exception handler active. As *CC*s are topologically close in the network to hosts, communication problems between the corresponding hosts and the *CC* are expected to be rare. Similarly, failure of the *CC* or loss of connectivity to the *CC* is easily detectable by hosts, which then would trigger an exception handler for such a situation, that would, for example, redirect hosts to an alternative *CC*.

Fig. 5 depicts the flow of various types of information towards the *CC*, and how the *CC* triggers and forwards information to its hosts. The network information collection and aggregation is performed at the *CC*s and the policies enforced at each host.

The two-tier architecture outlined here does not require the network administrator to explicitly understand where exceptions are being evaluated or enforced. Administrators describe the exceptions against a simple model of an object embodying the network state and an object embodying the per-host state. The network exception handlers are decomposed into exception detection and a set of actions to be enacted. We exploit this flexibility to enable a simple architecture that is similar to many network-wide services that are already deployed in enterprise networks today. A server, the *CC*, performs exception detection, reducing the amount of information that needs to flow to each host, and the hosts implement the actions. This clean separation allows for a scalable and efficient infrastructure that supports network exception handlers. The following section describes in more detail how the network topology information is extracted and aggregated, and how policy can be implemented at the hosts.

6. ENABLING EXCEPTION HANDLERS

In the previous section, we outlined a two-tier architecture for supporting network exception handlers in enterprise networks. In order for this architecture to be feasible, we need to be able to collect and synthesize the dynamic network topology information. We also need to be able to support traffic shaping and similar functionality on hosts.

6.1 Monitoring the network topology

Network exception handlers require dynamic network information. Enabling a diverse set of exception conditions requires a correspondingly rich dataset. We believe that a complete, dynamic link-level network topology, annotated with link capacities, costs and loads is both feasible and also necessary to support the full functionality that network exception handlers can offer.

Data concerning link capacities and costs is already maintained by an enterprise, and used by the network operators for billing, auditing and capacity planning purposes. This information is typically stored in a database and accessed via, e.g., web interfaces in our enterprise. The information changes slowly over time, when contracts are renegotiated or new links are provisioned. Therefore, distributing a copy of this information to the *CC*s is feasible and allows the network topology to be augmented with link capacities and costs.

Link load data is obviously far more dynamic than link capacity and cost information. However, this data is also available using existing mechanisms such as SNMP. Indeed, in most enterprises this information is already required by existing network management systems to enable capacity planning. It is thus feasible for *CC*s that are topologically close to such monitoring devices to extract this information, and then distribute it to all other *CC*s.

In contrast, the complete, dynamic link-level network topology is typically not currently available. We thus discuss here how to extract this topology information efficiently in the context of using OSPF [13]. We believe that OSPF is the most common routing protocol used in enterprise networks, including our own. In general, the techniques that we describe in the following sections should be easy to use with other link-state protocols, such as IS-IS [14, 5].

Collection

There are several methodologies for collecting OSPF data, including commercial solutions, such as those provided by Iptivia.[5] Shaikh and Greenberg [18] propose four basic methodologies to extract OSPF data in order to infer the network topology for a large ISP network. The first two involve forming a partial or full adjacency with a router, the third describes the *host-mode* where the IP multicast group used to distribute OSPF data in a broadcast network is monitored, and the fourth is the *wire-tap* method where a link between two routers is monitored.

In our two-tier architecture we assume that the *CC*s will collect the OSPF data. In a large enterprise network, the network will be configured into several OSPF areas, and it is necessary to extract the OSPF data for each area. To increase resilience, it is best to have multiple *CC*s per area. The wire-tap approach requires that *CC*s have physically proximity to the link being monitored, which may not always be feasible. The adjacency approach allows the *CC*s to act as 'dummy' routers that do not advertise any subnets but do participate in OSPF routing. This approach also has the advantage, that using GRE [9] tunnels, an adjacency between a *CC* and an arbitrary router in the network can be established, which permits OSPF data collection from routers when physical proximity is not possible. We have successfully used the wiretap method on a GRE tunnel configured to make an adjacency between two routers in the network visible to us. Host-mode would also be feasible in a network where IP multicast is supported, if the *CC*s support the necessary protocols allowing them to join IP multicast groups, and routers are suitably configured to allow them to join the relevant multicast group.

[5] `http://www.iptivia.com/`

LSDB MERGING:
1. Insert each non-summary LSA from the per-area LSDBs into the merged LSDB, tracking all prefixes for which such non-summary LSAs exist.
2. For each summary LSA, extract the single prefix it contains and retain it if there is *no* non-summary LSA for this prefix in the merged LSDB. If a summary LSA has already been retained for this prefix, then prefer according to the preference relationship on LSA types $1 < 2 < 5 < 3 < 4$.
3. Insert all retained summary LSAs into the network LSDB.
4. For all type-1 LSAs that are generated by each router that exists in multiple areas, construct and insert a new type-1 LSA, and remove type-1 LSAs with the same *lsid*.

Figure 6: OSPF topology synthesis process

Note that there are security implications to consider when not using either wire-tap or host-mode since standard router OSPF implementations do not permit a router to protect itself against bad routes inserted by hosts with which it is configured to form an adjacency. One of the advantages of the two-tier architecture is that the *CC*s are managed dedicated servers. Allowing these servers to act as dummy routers creates a smaller attack plane compared to allowing arbitrary hosts to act as dummy routers.

Collecting a live feed of OSPF data from each area and shadowing the current link-state database (LSDB) being maintained by routers in the area is therefore quite feasible.

Synthesis

Having collected the per-area LSDBs, they must be combined to synthesize a global topology. While this may be conceptually straightforward, practical implementation details can be complicated. To our knowledge, synthesis of global OSPF topology through merging of area LSDBs has not been described before and thus we provide here a detailed description of the procedure we followed.

The goal is to merge the LSDBs, each of which is simply a collection of link-state advertisements (LSAs) describing the links connected to a given link in the network. The complication arises from the fact that a "link" in OSPF can be a point-to-point link between two interfaces, a subnet being advertised by a router, a shared segment connecting many routers, or a subnet from a different area being summarized into this one. To account for this complication, we give preference to specific non-summary LSA types compared to the less specific summarization LSAs. Further, we must also disambiguate the LSAs describing an individual router in cases where that router exists in multiple areas. For completeness, given a set of per-area LSDBs to merge, Fig. 6 gives a detailed OSPF-specific description of the process.

After merging all LSDBs, the result is a single LSDB representing the entire network visible from OSPF. To gain a truly complete picture of the network's topology also requires the use of out-of-band data such as the router configuration files. Parsing configuration files reveals information not available through any particular routing protocol such as static routes and MPLS tunnels. This process can be automated by incorporating it into the configuration management system that any large network runs to manage its device configurations: when a device configuration is changed, the topology inference host for that part of the network should be automatically notified so that it can re-parse the relevant configurations. Other sources of routing data should also be considered although we note that, as discussed in Section 2, enterprise networks typically mandate use of proxies for external Internet access and use default routes to handle traffic destined outside the enterprise. Thus, we are not required either to infer topology from BGP data or to handle the high rate of updates usually observed in such data.

6.2 Control mechanisms at the host

Policy enforcement takes place at the hosts, with the exception handlers specifying actions to take when an exception is triggered. In order to implement network exception handlers at the host, access to local host state, e.g., the set of running processes, the application names, and the ability to shape the generated traffic are both required. Modern operating systems provide extensive support that enables this functionality. Event tracing frameworks, such as the Event Tracing for Windows[6] and other monitoring APIs, allow easy monitoring and access to the host state, e.g., application ids, user ids, TCP/UDP flows per process, without requiring any support from or modifications to existing applications. It is therefore easy to translate access to the host state by the network exception handlers into queries using these APIs.

Packet shaping functionality is also well supported in modern operating systems. For example, the TC/GQOS API[7] is provided with Windows XP and Windows Vista provides the QOS2 API[8] which both support traffic shaping on a per-process or application granularity. Indeed, blocking all traffic from a process, for example, is possible by configuring the host firewall, again at process or application granularity.

7. FEASIBILITY

In this section, we evaluate the feasibility of network exception handlers across two different dimensions. We first examine the overhead of distributing the annotated topology information across the enterprise network. We then present a concrete example of how network exception handlers might be employed to tackle one of the most important issues for IT departments of enterprise networks, namely traffic engineering to reduce bandwidth costs.

7.1 Datasets

The results presented in this paper are principally based on two datasets collected from the global enterprise network of a large multinational corporation. The first is a corpus of packet data collected from a single building connected to that enterprise network. The second is a corpus of routing data collected from two different areas of that enterprise network. Our local network within the monitored building comprises several subnets and a small datacenter (*DC*). The global enterprise network (*CORPNET*) contains approximately 400,000 hosts connected by approximately 1,300 routers spread across 100 countries and 6 continents. The particular building in question contains roughly 400 hosts used by a mixture of researchers, administrative staff, and developers. For scalability reasons, CORPNET contains 4 domains interconnected by BGP, each of which runs OSPF internally. Connectivity to the Internet is provided through proxies in a small number of large datacenters, roughly one per continent; these datacenters also host servers providing company-wide services such as email.

[6] `http://msdn2.microsoft.com/en-us/library/aa468736.aspx`

[7] `http://msdn2.microsoft.com/en-us/library/aa374472.aspx`.

[8] `http://msdn2.microsoft.com/en-us/library/aa374110.aspx`.

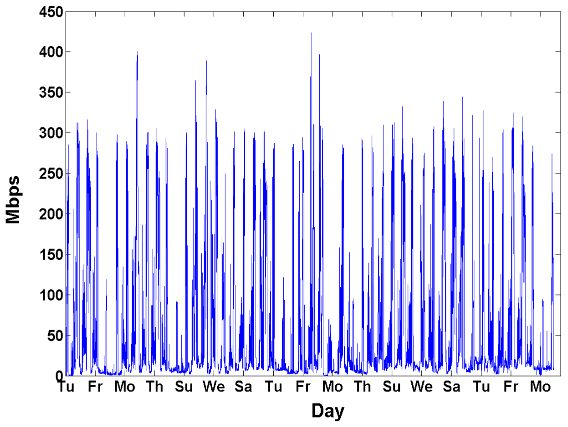


Figure 7: Aggregate traffic over time. The spikes correspond to regular backups, while traffic shows the expected diurnal patterns.

In terms of size, CORPNET is at the upper end of the scale of enterprise networks as it supports a large, multinational, technology-based corporation which makes extensive use of a variety of networked applications. As such, we use data from this network as an approximation to an upper bound on the complexity and scalability of network exception handlers. In smaller networks the overheads associated with network exception handlers should be lower. Small networks, such as small office/home networks, pose a quite different set of network management problems and would probably not significantly benefit from network exception handlers.

Data Traffic Dataset

The data traffic corpus was collected over a period of 7.7 weeks beginning on Tuesday February 13, 2007. Packets were captured using custom tools written against the WinPCap[9] packet capture library. Over the 7.7 weeks, 32.3 billion packets were captured, containing 32.8 TB of data. To analyze the collected trace we constructed flow tables corresponding to 5 minute time intervals with one record per uni-directional 5-tuple (source IP, destination IP, protocol, source port, destination port) flow observed in each 5 minute period. Each record contains the 5-tuple, the time of the first and last packets in the period, the number of bytes and packets observed, and the application inferred to have generated the traffic. We inferred applications by custom-made deep packet inspection tools, with less than 3.5% of packets not assigned to an application.

Fig. 7 shows a timeseries of the aggregate observed traffic bandwidth. The traffic pattern follows the usual diurnal patterns, with the exception of large spikes during the early morning hours of each day which correspond to backups taking place in the Data Center. We observed 66,697 unique IP addresses in the trace, 679 of which were local to the capture site. Of the observed addresses, 65,086 were sources (663 of which were local to the monitored building) and 49,031 were destinations (669 of which were local). The local addresses that received but never transmitted appear to be the result of automated security tools probing for active addresses, i.e., they appear as destination addresses only and never as source addresses.

[9] `http://www.winpcap.org/`

Table 1: Topology data. ***Core*** **links represent physical connectivity.** ***Leaf*** **links represent advertised prefixes, including summaries. An** ***event*** **is a link that fails, recovers or has its weight reconfigured.**

Area	Backbone	Stub
Date	2004-11-05	2004-06-16
Duration (weeks)	111	128
Gaps (days)	3	56
#Core links	175	42
#Leaf links	1123	1241
#Events (core)	183	720
#Events (leaf)	414090	270428

Topology Datasets

The network topology traces are extracted from OSPF data collected from two areas within CORPNET, one a backbone area and one a stub area. An autonomous system (AS) running OSPF internally contains a single unique *backbone area* which connects all the other *stub areas* in that AS; stub areas do not generally interconnect directly. CORPNET is split into four such ASs, with the AS numbers designated for private use and are not visible to the Internet at large. Each AS covers a large geographical region of the world and each runs a single OSPF backbone area. Both the backbone and the stub areas monitored are within the same AS. Data was collected until December 2006, beginning from the backbone area in November 2004, and from the stub area in June 2004. Data collection was almost continuous with about six short (roughly one day each) gaps per year in the stub area trace, and approximately one such gap per year in the backbone area trace. The stub area trace also experienced about four larger gaps (roughly one week each) over its entirety.

OSPF advertises *links*, which may represent either a prefix directly advertised by a router, a physical connection between two routers (point-to-point or shared media), or prefixes being summarized into this area from a neighboring area. Table 1 summarizes the characteristics of the topologies extracted from the OSPF data.

7.2 Overhead

We examine two types of overhead incurred by network exception handlers. First, the total traffic required to distribute the topology across the network. Second, the amount of traffic incurred by distributing the metrics the topology is annotated with, e.g., link load or link cost.

Live topology distribution. There are three components to the overhead of distributing the live network topology: (*i*) the receive bandwidth at each *CC* designated to gather the topology of its area; (*ii*) the transmit bandwidth at each such *CC* to transmit its area topology to other *CC*s; and (*iii*) the receive bandwidth at each *CC* required to receive the topologies of all areas.

The first component is very small, with median values of 71 Bps and 70 Bps for our core and stub areas respectively. Even the 99th percentiles for each are only 1.5 kBps (backbone) and 1.6 kBps (stub). This is a negligible overhead for a server-class machine to collect.

The second component is similarly negligible since most of the OSPF traffic does not actually report any changes to the area topology, being simply HELLO packets and refreshes of existing state. Only topology-changing *events*, i.e., addition of new links, failure or recovery of existing links, or link weight modifications, need to be redistributed. Examining events at the granularity of seconds, we see that only 0.002% of the total second-intervals for the stub area, and only 0.000007% of second-intervals for the core area, contain even a single event. The burstiness is similarly low, with the 99.99th percentiles at just 10 events for both areas, and the maximum number of events observed in any single second standing at just 91. Assuming events can be described in 25 bytes with 40 bytes overhead for TCP/IP headers, this corresponds to a worst case of around 2.5 kBps, and only 700 seconds over the three year period would require more than 1.5 kBps, i.e., more than a single packet per second.

```
SiteNames = ["Cambridge"];
Threshold = 100MB;
boolean Exception(NetworkState)
begin
    foreach site in Sitenames do
        siteLoad := NetworkState.Sites[site].UpStream.Load
        if siteLoad > Threshold return true
    return false
end
void Fire (NetworkState, HostState)
begin
    Register(Handler)
end
void Handler (NetworkState, Hoststate)
begin
    if not Exception(NetworkState) then
        RemoveAllRestrictions(); DeRegister(Handler); return
    foreach site in Sitenames do
        upStreamLoad := NetworkState.Sites[site].UpStream.Load
        ratio := Threshold / upStreamLoad
        foreach app in Hoststate.Apps do
            if app.isLowPriority then
                bwCap := app.UpStream.Load * ratio
                SetRestriction(app, maxBandwidth = bwCap)
end
```

Figure 8: Network exception handler to smooth upstream traffic by rate-limiting low priority applications at busy periods. The threshold may be specified by historical data.

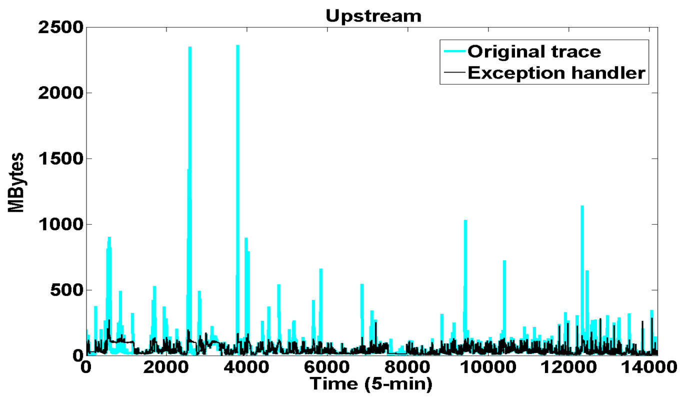
The third component is less negligible but is still very small. In our large enterprise network, there are approximately 62 distinct OSPF areas and 92 separate Active Directory domain controllers. Assuming that topology distribution was carried out using some efficient mechanism such as native IP multicast or a suitably configured application overlay, this corresponds to a worst case of 155 kBps being delivered to 92 receivers, and a 99.9th percentile of 4 kBps.

Per-link annotations. Distributing link-related information such as cost and load requires a higher overhead compared to the overhead of distributing the topology, but overall is still not significant. For example, assume that up to fifteen 4-byte counters per link are distributed to each *CC* every 5 minutes, using a similar IP multicast or application overlay mechanism. This corresponds to approximately 100 bytes per link every 5 minutes. Even in a large network with 10,000 links this is a total overhead of just 3 kBps, and could be reduced with more efficient encoding of the per-link values.

Thus, the total overhead of distributing the control information required to employ network exception handlers would be negligible even in a large enterprise network.

7.3 Managing bandwidth with exception handlers

We now evaluate a concrete example of network exception handlers using our enterprise network traffic dataset described in Section 7.1. In particular, we describe a hypothetical scenario where an exception handler is employed for bandwidth savings through



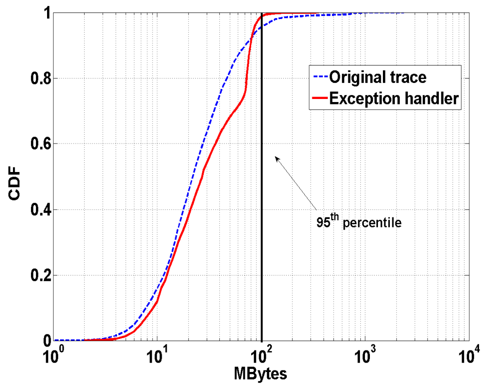


Figure 9: Bandwidth savings by applying the network exception handler in Fig. 8. LEFT: Upstream non-local traffic time-series before and after applying the exception handler, where most spikes are smoothed-out. RIGHT: Traffic CDF over 5-min intervals. The exception handler reduces the 95th percentile by 10%.

shaping of traffic caused by "low priority" user applications. Bandwidth requirements currently reflect one of the major sources of cost for enterprise networks as discussed in Section 2, and network exception handlers can provide a flexible and effective mechanism to mitigate this cost by letting hosts decide how to best utilize network resources, and specifically here bandwidth.

One of the possible charging schemes for bandwidth usage in enterprise networks is the 95th percentile of the upstream traffic. Bandwidth usage is calculated in 5 minute intervals, and at the end of the charging period which is usually a month, the cost is calculated by the provider at the 95th percentile of the corresponding cumulative distribution function (CDF).

Let's now assume that the operator of our monitored network aims at reducing this bandwidth cost by scaling down the 95th percentile of upstream traffic, but at the same time wishes to affect application performance and user experience as little as possible. A way to achieve this through network exception handlers would be to rate limit low-priority applications at time instances where traffic is over a threshold approaching the 95th percentile. This threshold could for example reflect the 90th percentile of last month's upstream traffic, as the overall traffic demands should be more or less stable over such periods. Low priority applications may correspond here to long file transfers and non-interactive applications such as backups, where users' behavior is not significantly affected, as it would be when rate-limiting HTTP traffic. An exception handler to implement such a policy is presented in Fig. 8.

To evaluate the effect of this exception handler in our enterprise network, we replayed our captured traffic employing the specified exception handler. We first extracted the traffic targeted out of the local building from the trace using the topology information, and then applied the exception handler in all 5 minute intervals for which traffic crosses over the specified threshold. Fig. 9(left) presents the upstream traffic timeseries before and after the use of the exception handler, where the handler is applied when traffic crosses over the 90th percentile of the total trace. The exception handler succeeds in eliminating most of the large spikes present in the traffic timeseries, by being able to smooth heavy periods caused by "low-priority" applications. Fig. 9(right) further highlights how the traffic distribution is affected by the exception handler by shifting traffic from heavier periods to periods where traffic is below the 95th percentile. Overall, by applying this simple exception handler, traffic never rises above the 96th percentile of the original distribution before activating the exception handler. Interestingly, WAN optimizers, which are increasingly being used in enterprise networks, attempt to reduce WAN link demands by compressing the traffic passing over the link of interest. This example shows the potential for network exception handlers to allow enterprises to reduce WAN demands by shaping traffic at the source, rather than compressing traffic in the network.

This example emphasizes the flexibility of traffic shaping at the sources through network exception handlers. Hosts are both allowed to decide how to prioritize applications subject to the specified policy, but also have the ability to rate-limit their own traffic as the application-context exists at the edge of the network. In a stricter scenario, certain applications could be instructed to completely back off. In such a scenario, exception handlers will prevent traffic from even entering the network (in contrast to dropping packets in the network, where network resources would still be used). Applying such policies of per-application throttling in the middle of the network appears practically infeasible in existing enterprise networks.

8. NETWORK EXCEPTION HANDLERS—A UTOPIA?

Network exception handlers demonstrate the potential of pushing part of the network decision-making process to hosts. Having highlighted the benefits and flexibility of network exception handlers throughout the paper, we believe that this mechanism should be a fundamental building block of modern enterprise networks.

Does this flexibility however imply that network exception handlers could be applied to every network operation, or that hosts could, or should, participate in all network-related decisions? Network exception handlers, while certainly powerful and flexible, are not the panacea of all enterprise management problems. They have limitations that make them unsuitable to express certain policies. Our proposed architecture also leaves a number of issues open for future research and we discuss the most important of these here.

Cooperation. The set of actions performed in a network exception handler use only the global network topology information and the local host state. There is no explicit cooperation or synchronization between hosts. This limits the kinds of policy that can be implemented. For example, using network exception handlers, it is difficult to implement the following policy: *allow application X running for users in Europe to consume no more than 50% of a specific link.* This would require all hosts concurrently running

```
Network Exception Handler: Server under SYN attack
boolean Exception(NetworkState)
begin
    return NetworkState.SYNAttackedMachines is not empty set
end
void Handler (NetworkState, Hoststate)
begin
    if not Exception(NetworkState) then
        DeRegister(Handler); return
    foreach host in NetworkState.SYNAttackedMachines do
        PID := SYNPktRateToDest(host, threshold)
        if PID != NULL then TerminateProcess(PID)
end
```

Figure 10: Network exception handler to terminate a process if the rate of SYN packets it generates to a particular destination is larger than a threshold.

application X to cooperate in order to potentially divide up that capacity. While attractive, the complexity of implementing this dynamic control over hundreds of thousands of geographically spread hosts is high.

Routing. Since the network topology is distributed across the network, hosts could potentially make routing decisions, and realize mechanisms such as multipath and loose source-routing [19], or load balancing techniques such as ECMP (equal-cost multipath). However, we chose explicitly not to allow hosts to make such routing decisions, because attempting to perform such operations at the host may lead to traffic oscillations across links. For example, if load-balancing is implemented at the host, it is hard to guarantee that all hosts will not shift their traffic from a congested link to a second link simultaneously, thus leading eventually to traffic oscillations due to congestion frequently shifting to new links. This is another instance where cooperation between hosts would be useful, so that a set of hosts would be able to coordinate to reduce the likelihood of oscillations. In general, hosts are better placed to perform per-application traffic management, and the network is better placed to perform routing or load-balancing across links.

Closing the loop. In our current design, information flows from the network to the hosts, but no information flows from the hosts to the network. If the loop were closed, then this would also provide a mechanism to facilitate better network management. For example, this could be a good mechanism to express and expose information to network administrators from hosts. At the moment, most host monitoring is simplistic, with a small set of attributes proactively monitored on each host, and aggregated into a single database. In our architecture, as described, when a network exception occurs, the user(s) on the host can be explicitly informed. This could be extended, so that the network administrator could also be informed. Therefore, the network administrator could gather the context of certain exceptions, i.e., both the HostState and NetworkState, when the exception triggers, in order to facilitate troubleshooting and management. Of course, there are a number of challenges with such an approach, such as handling the potential implosions of reports.

Security. Network exception handlers can expose powerful operations. An extreme scenario would even be to terminate a local process! Further, such a policy could be used in conjunction with network Intrusion Detection Systems (IDSs). For example, consider a SYN attack scenario. When the IDS detects an attack, all the IP addresses of the machines under attack are inserted in the NetworkState by the IDS. Each host then could run a network exception handler that triggers when this set is not empty. When triggered, the process producing the SYN packets would be terminated, if the rate of SYN packets to a particular destination were over a threshold. Fig. 10 shows an example of such an exception handler. Of course, this is a simplified case, since we assume for instance that the misbehaving process can actually be terminated, it will not restart once terminated, etc. Yet, this example clearly highlights the power and possibilities of providing the hosts with network state information. This is also another example where allowing feedback to the network administrators would be valuable.

However, exposing information to the hosts might be a mixed-blessing, should a host become compromised. Having all topology and traffic information pushed to the host, may enable informed attacks concentrated at specific "weak" points in the network. Having the *CC* filter the NetworkState information exposed to the hosts is therefore advantageous, and ensuring that the network operator signs all network exception handlers, and that *CC*s only execute signed network exception handlers is also important. In general, we expect certain levels of trust to already exist within an enterprise network, and that hosts will authenticate at least with the *CC*. Services such as Network Access Protection[10] may further ensure that systems are fully patched and are updated with the latest virus definitions before gaining access to the corporate network. However, protection from zero-day exploits, or detection and handling of compromised *CC*s are issues for future research.

Deployment. Network exception handlers may be partially deployed without affecting the performance of the enterprise network provided that *CC*s have access to annotated topology information. Some of the scenarios described throughout the paper may require that network exception handlers are deployed on all hosts in a domain for efficient handling of the specific scenario. In general however, even partial deployment is feasible and allows for localized management. This is possible as all local host decisions do not affect the performance of other remote hosts, and *CC*s may operate in isolation. In very small partial deployments some policies may have only limited impact. Network exception handlers do not rely on the deployment of new network capabilities that add to the complexity of network management (e.g., loose source routing) or modification of host applications.

9. RELATED WORK

Network exception handlers allow hosts to participate directly in the management of the network; they allow some traditional in-network functionality to be migrated to the host. Bellovin [3] proposed migrating firewall functionality to the hosts, to create a distributed firewall where firewall policy is pushed to the hosts to implement. This has now been widely adopted in enterprises. The mechanisms used to implement network exception handlers could potentially be used to implement a distributed firewall. However, enabling network management requires exposing of further dynamic information about the network.

Several host-based congestion control schemes have been proposed, e.g., PCP [1] and endpoint-based admission control [4]. These rely on probe packet sequences to determine the rate at which they should transmit. Bandwidth brokers use selected hosts in the network to maintain QoS management state so as to provide guaranteed services, essentially providing admission control on a path-basis by conditioning at selected points on the network's edge. Using network exception handlers potentially allows every host to be able to condition the traffic they introduce, but with much finer grained control.

[10]`http://technet.microsoft.com/en-us/network/bb545879.aspx`

There are many proposals to monitor/measure the network's performance, e.g., loss rate, throughput, or round-trip-time, for management purposes [7, 8, 21]. Network exception handlers allow policies to specify the desired reactions when particular behavior or performance is detected. Network exception handlers are almost the inverse of distributed network triggers [11, 10] which collect host measurements at a coordinator that raises an alarm when policy violation is detected. Network exception handlers push the policy towards the hosts and, by exposing information about the network's behavior, allow the hosts to implement the policy as required.

Several proposed network architectures improve network management by simplifying the network's configuration by providing more control over the network. Ethane [6] uses three high-level principles: policies are declared over high-level names, policies should determine the path packets follow, and packets are strongly bound to their origin. Network exception handlers effectively implement the first and third of these, but do so for performance rather than configuration. Tesseract [20] implements the 4D [16] control plane enabling direct network control under a single administrative domain. Tesseract configures all network switch nodes to impose defined policies on the network, so it is host independent. Network exception handlers enable policies to be imposed on the network *without* directly impacting network devices. Therefore, they can support richer policy, using information not exposed to the network.

CONMan [2] is an architecture for simplifying network device configuration and management. Each device is associated with a network manager, which can map high-level policy goals down to the capabilities of the devices. Network exception handlers allow the specification of a policy associated with a trigger, rather than a way to reconfigure the network to match overall policy requirements.

10. CONCLUSION

In this paper, we argued that within a single administrative domain such as an enterprise network, hosts should be more directly involved in the management of the network. To this end we introduced the concept of *network exception handlers*, where information is shared between the network and hosts so that when exceptional conditions are detected, hosts can be made to react subject to policies imposed by the operator. We described a simple programming model against which operators can specify such policies, giving several examples of its potential use. Finally, we described a concrete design and demonstrated its feasibility and effectiveness using data gathered from our own global enterprise network. Network exception handlers are a simple, yet powerful abstraction enabling enterprise network operators to gain significant control of their network's behavior. Our analysis suggests that changing our mindset about the architecture of enterprise networks is attractive, and demonstrates the feasibility of one possible such change.

11. REFERENCES

[1] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. PCP: Efficient Endpoint Congestion Control. In *Proc. ACM/USENIX NSDI 2006*, pages 197–210, San Jose, CA, May 2006.

[2] H. Ballani and P. Francis. CONMan: A Step Towards Network Manageability. In *Proc. ACM SIGCOMM*, pages 205–216, New York, NY, 2007.

[3] S. M. Bellovin. Distributed firewalls. *;login:*, pages 37–39, Nov. 1999.

[4] L. Breslau, E. W. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint Admission Control: Architectural Issues and Performance. In *Proc. ACM SIGCOMM 2000*, pages 57–69, New York, NY, 2000.

[5] R. Callon. Use of OSI IS-IS for routing in TCP/IP and dual environments. RFC 1195, IETF, Dec. 1990.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. ACM SIGCOMM*, pages 1–12, New York, NY, 2007.

[7] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proc. ACM SIGCOMM 2004*, pages 55–66, New York, NY, 2004.

[8] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors. In *Proc. USENIX 2006 Annual Technical Conference*, June 2006.

[9] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701, IETF, Oct. 1994.

[10] L. Huang, M. Garofalakis, J. Hellerstein, A. Joseph, and N. Taft. Toward Sophisticated Detection with Distributed Triggers. In *MineNet'06*, pages 311–316, New York, NY, 2006.

[11] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers. In *Proc. HotNets-III*, San Diego, CA, November 2004.

[12] A. W. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Sixth Passive and Active Measurement Workshop (PAM)*, Boston, MA, 2005.

[13] J. Moy. OSPF Version 2. RFC 2328, IETF, Apr. 1998.

[14] D. Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142, IETF, Feb. 1990.

[15] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, Sept. 2001.

[16] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-wide Decision Making: Toward a Wafer-thin Control Plane. In *Proc. HotNets-III*, San Diego, CA, Nov. 2004.

[17] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364, IETF, Feb. 2006.

[18] A. Shaikh and A. Greenberg. OSPF Monitoring: Architecture, Design and Deployment Experience. In *Proc. ACM/USENIX NSDI 2004*, pages 57–70, San Francisco, CA, Mar. 2004.

[19] A. Snoeren and B. Raghavan. Decoupling Policy from Mechanism in Internet Routing. In *Proc. HotNets-II*, pages 81–86, Cambridge, MA, Nov. 2003.

[20] H. Yan, D. A. Maltz, T. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D Network Control Plane. In *Proc. ACM/USENIX NSDI 2007*, pages 369–382, Cambridge, MA, May 2007.

[21] Y. Zhao, Y. Chen, and D. Bindel. Towards Unbiased End-to-End Network Diagnosis. In *Proc. ACM SIGCOMM*, pages 219–230, New York, NY, 2006.