2007-14

Network Access in a Diversified Internet

Authors: M.Wilson, F. Kuhns, J. Turner

Corresponding Author: mlw2@arl.wustl.edu

Abstract: There is a growing interest in virtualized network infrastructures as a means to enable experimental evaluation of new network architectures on a realistic scale. The National Science Foundation's GENI initiative seeks to develop a national experimental facility that would include virtualized network platforms that can support many concurrent experimental networks. Some researchers seek to make virtualization a central architectural component of a future Internet, so that new network architectures can be introduced at any time, without the barriers to entry that currently make this difficult. This paper focuses on how to extend the concept of virtualized networking through LAN-based access networks to the end systems. Our objective is to allow virtual networks that support new network services to make those services directly available to applications, rather than force applications to access them indirectly through existing network protocols. We demonstrate that this approach can improve performance by an order of magnitude over other approaches and can enable virtual networks that provide end-to-end quality of service.

Type of Report: Other

Network Access in a Diversified Internet

Michael Wilson, Fred Kuhns, and Jonathan Turner
Department of Computer Science and Engineering
Washington University, St. Louis MO. 63130
{mlw2, fredk, jst}@arl.wustl.edu

Abstract. There is a growing interest in virtualized network infrastructures as a means to enable experimental evaluation of new network architectures on a realistic scale. The National Science Foundation's GENI initiative seeks to develop a national experimental facility that would include virtualized network platforms that can support many concurrent experimental networks. Some researchers seek to make virtualization a central architectural component of a future Internet, so that new network architectures can be introduced at any time, without the barriers to entry that currently make this difficult. This paper focuses on how to extend the concept of virtualized networking through LAN-based access networks to the end systems. Our objective is to allow virtual networks that support new network services to make those services directly available to applications, rather than force applications to access them indirectly through existing network protocols. We demonstrate that this approach can improve performance by an order of magnitude over other approaches and can enable virtual networks that provide end-to-end quality of service.

1. Introduction

Today's Internet has grown far beyond the original design. New requirements have grown almost as rapidly as the scale of the Internet. Unfortunately, the Internet is owned by no single stakeholder, making it difficult or impossible to upgrade the underlying architecture. [1] As recognized in [3], the inability of the current Internet architecture to meet new needs has led to the development of numerous *ad hoc* solutions to legitimate problems. For example, Network Address Translation provides some measure of solution to network address depletion.

The Internet needs a means of deploying potentially disruptive technologies alongside existing technologies. Virtualization has been advanced as a way to meet this need. Virtualized networks and protocols could be deployed side-by-side but would be isolated by the virtualization mechanisms. The GENI [2] initiative seeks to use virtualization to create a national experimental facility for experimentation based on these very ideas.

Overlay networks have been proposed as one method of virtualizing the network. However, overlay networks exist on top of existing networks and protocols. We believe that overlay networks should be regarded as a temporary migration solution to allow legacy networks to participate in new services. We propose to make network virtualization as a core capability of a next generation *diversified internet* (in the remainder of this paper, we use the term diversification, in place of virtualization, because the "*V*-word" has been so overloaded, that it is often misinterpreted). In our diversified internet model, the underlying network provides a minimal set of services and a thin provisioning layer upon which new protocols may be developed. More details can be found in [4].

The fundamental abstractions for a diversified network are *substrate routers*, which are connected to each other by point-to-point *substrate links*; and *metarouters*, which are hosted on substrate routers and are connected to each other by point-to-point *metalinks* carried over substrate links. Collectively, a set of connected metarouters form a *metanet* exchanging *metaframes* adhering to a *metaprotocol*. We refer to the software components that support these abstractions as the Network Diversification Architecture (NDA).

In this paper, we focus on the impact of internet diversification on the access network and end systems. In section 2, we provide an overview of related work in this area. In section 3, we characterize the objectives and available features of the access portion of the network. In section 4, we present our design and prototype implementation of end-point diversification, and we present a preliminary evaluation of our prototype in section 5. We summarize our results and give a few words on future directions in section 6.

2. Related Work

Research in the area of network virtualization has focused on two general area: large-scale testbeds for development, testing, and experimental deployment of novel protocols, and overlay networks suitable for general deployment.

In the testbed arena, PlanetLab [5] is the most significant development to date. PlanetLab provides a shared in-

frastructure on which overlay services can be provided. Access to PlanetLab is handled purely through overlay connections and is either handled transparently (to support legacy applications) or handled explicitly by the application, with no system support in the end system. PlanetLab nodes provide transparent network traffic isolation using the VNET [6] module, which tracks and demultiplexes traffic. Existing Linux queuing disciplines such as Hierarchical Token Buckets (HTB) [7], [8] provide bandwidth allocation.

Closest to our work in design and spirit is PL-VINI [9], a virtualized network architecture implemented in the PlanetLab environment. PL-VINI is designed primarily to support networking experiments rather than deployment, but adapts well to both uses. PL-VINI leverages existing PlanetLab features for resource isolation and adds the novel concept of CPU reservations, where a slice is guaranteed a minimum percentage of the CPU despite fluctuations in other slices. Future work on PL-VINI is focused on improving experimental realism, including a non-work-conserving CPU scheduler to better enable experiment isolation on a single node.

In the overlay realm, the X-Bone [10], Virtual Internet and GX-Bone [11] projects are representative of overlay networks which focus on the network layer. These projects define a generalized Internet architecture and provide tools for the dynamic construction and management of Internet overlay networks. They assume an underlying IP network and rely on the existence of standard Internet services. This is distinct from our NDA, where the goal is to provide a new *underlying* architecture on which new protocols can be implemented.

Oasis [12] is an architecture for virtualized network access to overlay networks. It uses a virtual interface for packet interception in the kernel. Packets are routed to a user-space application which determines overlay membership and forwards to a second userspace process which manages that overlay access point. Oasis is designed to enable legacy applications to take advantage of overlay networks to obtain improvements in performance. It is not intended to bring novel network services to the endpoints.

3. Diversification of the Access Network

The access network provides the connection between a network endpoint and the first substrate router. We expect that Ethernet will continue to be one of the most common underlying technologies for access networks, and we focus our attention on the Ethernet context in this paper. In the future, we expect wireless connection to dominate the access, but we do not explicitly address wireless here. We also focus on the scenario when there is a substrate router connected to the access LAN. Our approach can also be extended to handle remote connection of hosts via IP tunnels, albeit with some loss of capability.

3.1. Objectives

The overarching objective for the access network in a diversified network infrastructure is to make it possible for end systems to take advantage of any network services that may be provided by metanetworks. This objective leads us to the following specific goals.

- *Enable provisioned access*. To make it possible for metanets to support applications with end-to-end QoS guarantees, it is important for endpoints to be able to reserve capacity for communication with specific metanetworks.
- *Enable dynamic reallocation of access capacity*. Traffic in access networks is inherently more dynamic than backbone traffic. This makes it important to allow adjustments in provisioned bandwidth to accommodate changing needs.
- *Support existing Internet protocols*. The existing Internet protocols should be able to operate within a diversified network environment with no loss of functionality and no significant performance degradation.
- *Support existing uses of multi-access LAN features*. The multi-access features of Ethernet are commonly used to implement important elements of the Internet protocol suite (e.g. ARP, multicast). Such uses should be possible within the diversified network environment.

3.2. Data Plane

The key to enabling provisioned access to metanetworks is the use of VLAN mechanisms in Ethernet networks. In the last several years, VLAN technology has become standard, even on inexpensive commodity Ethernet switches. Moreover, packets with specific tags can be assigned to high priority queues, effectively isolating them from the effects of congestion caused by packets with lower priority.

To enable provisioned access, we configure a high priority VLAN connecting all the endpoints to a local substrate router. The usage of this VLAN is restricted to diversified network traffic and endpoints are permitted to use it only to send to the substrate router (that is, packets destined for another endpoint on the same local network are required to pass through the substrate router). Packets sent on the access link include a substrate header that contains a *Metalink Identifier* (MLI). Each network endpoint is assigned an MLI for each metanetwork it is connected to and each MLI is used only for communication between its assigned endpoint and the substrate router.

The provisioned access link is configured with a certain amount of *assignable capacity*. The total traffic sent by the substrate router on the access link is limited to this assignable capacity, and the total traffic sent by the endpoints to

the substrate router is also constrained to the assignable capacity. The assignable capacity should be limited to some fraction of the bandwidth (say 50%) of the smallest inter-switch link used by traffic passing between the endpoints and the substrate router, to reserve capacity for lower priority traffic. In addition, there is a maximum endpoint capacity, which limits the rate at which any single endpoint can send on the provisioned access link. This will typically be set to some fraction of the slowest access link. In a typical network today, the total assignable capacity might be 500 Mb/s, while the maximum endpoint capacity might be 50 Mb/s. As 10 gigabit Ethernet becomes commonplace over the next several years, these numbers can be expected to grow by a factor of ten.

The substrate router directly controls the flow of outgoing traffic on each metalink. Each metalink has an assigned maximum bandwidth, and the sum of these may not exceed the assignable capacity of the access link; and the substrate router uses per metalink queueing to ensure that these limits are respected. In the upstream direction, the substrate router has no direct control over the sending rates, but since it does see all the traffic, it can monitor the incoming traffic on each metalink to ensure that it does not exceed the allowed maximum rate. Violations are reported through the network management system so that network administrators can take the appropriate steps to address them.

While a provisioned access link can meet the needs of metanetworks that require provisioned metalinks, it does not meet the needs of metanetworks that need to make use of the multicast features of the underlying LAN. In particular, IPv4 uses multicast to implement ARP and DHCP, as well as extending IP multicast to end systems. To enable IP-based metanets, it must be possible for an IP metarouter to use these features. Moreover, other metanetworks are likely to have similar uses for these features, making it essential that they be accessible in the diversified network environment.

These capabilities can be provided using a second VLAN. Metanetworks that require access to the multicast features of the underlying Ethernet network will send and receive data on the *multipoint access link* implemented by this second VLAN. This link can be used just like a normal Ethernet network, allowing metanets to implement protocols like ARP exactly as they are implemented today. Endpoints may communicate with each other directly over the multipoint access link, allowing local traffic to bypass the substrate router. To facilitate such direct communication, MLIs are assigned using *shared mode* in the context of multipoint access links. Specifically, each metanet is assigned a separate MLI and all endpoints sending data using that metanet use that MLI. Note that all traffic sent using the multipoint access link is purely best effort.

3.3. Control and Management

There are two primary control functions required for the access network. First, we need a mechanism to allow hosts to establish connections to the substrate and the metanets with which they want to communicate. Second, we need a mechanism to allow metanetworks to reserve bandwidth for provisioned access metalinks, and an accompanying mechanism to allow the substrate router to advise endpoints of their allowed sending rates. We only sketch these mechanisms briefly here.

When a host first connects, it starts by broadcasting a *substrate discovery* packet on its local network. The substrate optionally authenticates the endpoint (as determined by substrate domain-specific policies), and responds to the endpoint with its MAC address, the VLAN tags to use for communicating through the substrate router and the MLI to use for control communication. At this point, the endpoint can request connection to one or more metanets. The substrate router delivers each such request to the designated metanet, which may then request the establishment of a metalink to the endpoint. Once the access substrate router has been appropriately configured, it informs the endpoint of the MLI to use for accessing the metanet and for provisioned metalinks, the maximum sending rate they may use.

The access substrate router can adjust the bandwidth for provisioned access metalinks in response to requests from the associated metanetworks (depending bandwidth resource availability and local substrate policies). Metanets may include mechanisms that allow endpoints to request such changes, but such requests come to the substrate through the metanets. Access link bandwidth is provided on a *leased* basis, meaning that metanetworks must periodically renew their lease in order to retain the reserved bandwidth. Metanets may use either a *long-term lease*, or a *short-term lease*. Substrate routers will normally renew long-term leases as long as the metanet requests renewal. Short-term leases are provided to allow dynamic redistribution of bandwidth among metanets on a shorter timeframe.

4. Diversification of the Hosts

Host diversification mechanisms allow the introduction of new *Metanet Protocol Stacks* (MPS) that provide metanet-specific services to applications and users. These mechanisms include a common *substrate* which is independent of metanets, but can be configured on behalf of individual metanets.

4.1. Objectives

There are several key objectives that drive the design of the host diversification architecture.

- *Ease of adding new metanet stacks*. We envision a multiplicity of metanetworks, some of which may be tailored to specific applications or application classes.

While adding a new MPS is something that users will do infrequently, we want to minimize barriers to gaining access to a new metanet. The procedure for adding a new MPS should be no more difficult than installing an application program.

- *OS compatibility*. We can't expect users to use non-standard operating systems in order to use metanetworks. The software must run on standard OS platforms, including Linux and Windows and any OS extensions must make use of existing mechanisms.
- *Security*. The system must ensure that different MPSs cannot interfere with the operation of others. An MPS should not require special system privileges and should have no more ability to interfere with system operation than ordinary application programs.
- *Traffic Isolation*. Provisioned metalinks must be effectively isolated from one another and from other network traffic. This means that hosts must ensure that outgoing provisioned metalinks are able to get access to the full reserved bandwidth and that they are constrained to send no faster than a specified maximum bandwidth.
- *Enable fine-grained queue management*. Metanets should be able to associate multiple queues with their outgoing metalinks, and map outgoing traffic flows to queues in a flexible fashion.
- *Minimize constraints on metanets*. The software architecture should not limit the kinds of services that metanets can provide to users.
- *Close to native performance*. The performance of a metanet protocol stack should be at least roughly comparable to the performance that could be expected if the stack was integrated into the OS kernel.

Achieving all these objectives simultaneously is challenging but feasible. In the remainder of this section, we develop an approach to host diversification that we believe can achieve these objectives, and we describe a prototype implementation that demonstrates the most important elements of this approach.

4.2. Software Design

In most systems today, network protocol stacks are integrated within the OS kernel and accessed through an API defined by the socket interface. This implies that the network code is part of the system's trusted code base, since it has unprotected access to key kernel data structures. This is clearly unacceptable in the metanet environment. We expect many organizations to develop new metanets and MPSs. Requiring that new stacks be added to the OS kernel adds a significant barrier to adding new stacks and brings unacceptable security risks (in the context of existing popular operating systems).

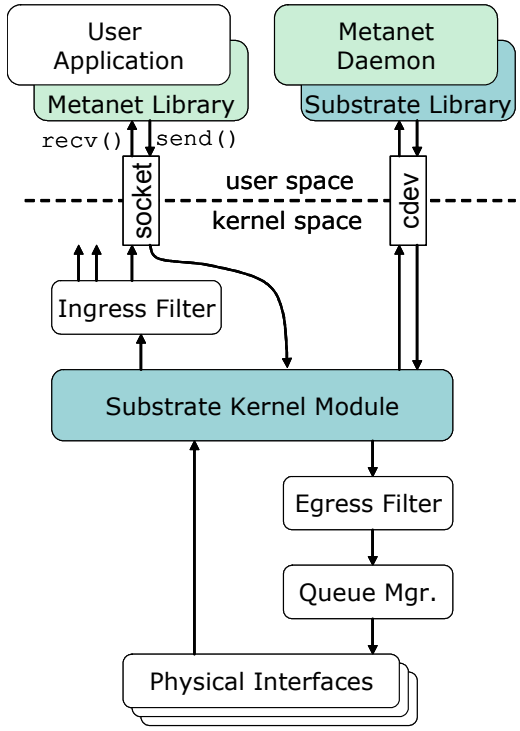

Fig. 1. Components of the end host Network Diversification Architecture

There is a rich body of work on alternate implementation models for network software [13]-[16]. The approach we take relies on user-space implementation of metanet protocols together with some generic (metanet-independent) OS extensions that are implemented by a loadable kernel module.

Fig. 1 is a block diagram showing the key components of the design for the Linux environment. The lightly shaded components are software components required for each metanet. The more darkly shared components are substrate software components, while the unshaded components are implemented using features of the standard Linux distribution.

The *Substrate Kernel Module* (SKM) is implemented as a loadable kernel module that must be installed on a one-time basis. It implements common substrate services, leveraging existing OS mechanisms as much as possible. In particular, the SKM uses kernel-resident packet filtering mechanisms to implement the ingress and egress filter functions and configures the Linux queue disciplines to regulate the traffic flowing into outgoing metalinks.

User applications send and receive data using a given metanet, using a metanet-specific library, which is linked to the application program. The library uses the standard socket interface, with the `PF_DIVINT` protocol family and a protocol number that identifies the particular metanet. Each metanet has a user-space daemon that implements

certain standard configuration functions and responds to requests for metanet-specific services from user applications.

4.3. Operation

Arriving packets use MAC-layer mechanisms to identify them as diversified internet packets. All such arriving packets are delivered to the SKM, which processes them based on the substrate header fields. In particular, it uses the MLI in the packet header to determine the associated metanet.

The ingress filter block determines the socket to which an arriving packet should be delivered, based on the MLI and the metanet packet header. These filters are defined using the Linux packet filtering mechanism, which is based on Berkeley packet filters. The kernel delivers entire metanet packets across the socket interface to the metanet library, which in turn delivers the data to the user application using a metanet-specific interface.

Outgoing data is passed to the SKM as complete metanet packets. The SKM associates packets with the proper metanet, based on the socket. Outgoing packets are filtered to provide a check on the validity of the metanet header and the appropriate substrate header is added. Packets are placed in Linux queue disciplines. Each metanet has an associated metalink with one or more queues, and a total reserved bandwidth, and a maximum bandwidth. When multiple queues are configured for a metalink, they can be assigned different shares of the outgoing bandwidth.

When an application opens a socket (with `socket()`) for an installed metaprotocol, the SKM handles the initial socket creation and establishes a set of default settings. The appropriate control daemon is notified of the socket creation request. Based purely on this metaprotocol, the daemon may refuse the request, allow it, allow it and attach initial packet filters, etc. Once a socket is opened, a typical applicaton sequence might be to `bind()` to a local address, `connect()` to a remote address, and `send()` application data. At `bind()`, the SKM would notify the control daemon, which might apply egress validation filters to enforce use of the local address in outgoing metaframes. At `connect()`, the SKM would notify the control daemon, which might supply any necessary routing information to the SKM. Finally, at `send()`, the SKM already has all the information needed to process the request without further recourse to the control daemon.

After the application finishes with the socket, the application calls `close()`. The SKM marks the socket as closed and notifies the control daemon. However, the SKM maintains the socket until such time as the control daemon notifies the SKM to actually deallocate it, thus supporting protocols like TCP where sockets linger after `close()`.

Before any application can open sockets for a given metaprotocol, that metaprotocol must be registered by a control daemon. The control daemon opens the SKM character device and registers the new metaprotocol with the SKM. The registration consists of a set of function calls to support operations on sockets within this metaprotocol. The substrate library spawns a reader thread that waits for messages from the SKM and dispatches these messages as upcalls to metaprotocol operations. The control daemon normally must request the SKM to create one or more metalinks. Later, at application `connect()` requests, the control daemon can attach a socket to a metalink for routing.

When the metaprotocol is shut down, the control daemon can simply close the SKM character device. The SKM is notified of the file release and closes all associated structures. Because the SKM is also notified by the OS even on abnormal termination of the control daemon, there is no potential for unattended metanets.

4.4. Prototype Implementation

Our initial prototype was developed on Linux 2.6.16. We currently support a subset of the socket operations. Some operations are not necessary to support a minimal metaprotocol. For example, we do not currently pass `listen()` and `accept()`. These can both be implemented directly in a metaprotocol library linked into the user application. `sendpage()` has no useful analogue, since it is designed for zero-copy sending, and copying the data into the control daemon defeats the purpose. We also do not currently support per-packet interception of `send()` or `recv()`, because this violates the model by returning the control daemon to the datapath instead of restricting it to management only. Metaprotocol developers should avoid the use of this functionality as much as possible to achieve maximum efficiency.

Our initial implementation is restricted to the SKM and substrate library. Bandwidth management relies on components already in place in the Linux kernel [7], and we manage the settings manually with `tc`. Similar functionality exists in Windows, the most popular desktop OS today [17]. Finally, while our model includes a variety of substrate link types, we currently only implement a point-to-point GRE tunnel with no multi-access substrate link.

4.5. Alternate Approaches

The main difficulty in implementing efficient frameworks for metaprotocol development is that metaprotocol code must be isolated from other metaprotocols, unrelated processes, or the base operating system, but metaprotocol data is opaque to the substrate. We have chosen to use Berkeley Packet Filters (BPFs) [18], an interpreted filtering mechanism, to perform very simple packet validation and demultiplexing. We encourage metaprotocol designers to accept

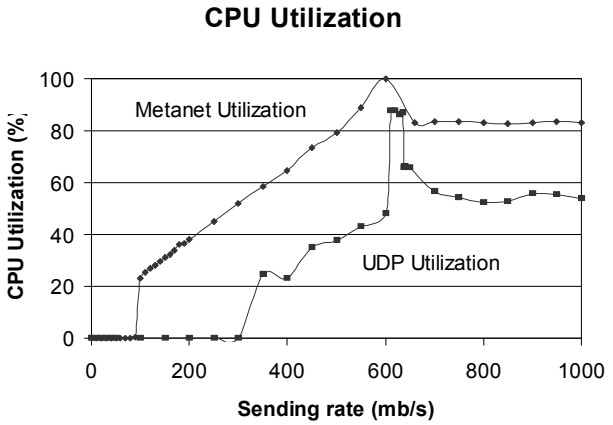

Fig. 2. CPU utilizations vs. sending rate as limited by the egress queues

this limited solution and to keep more complex functionality in the management daemon and not in the datapath.

In Oasis, the authors use user-space processes to determine routing in overlay networks on a per-packet basis. Packets are sent by a user application, then forwarded by the kernel to user-space processes that forward via an overlay network. While this allows for maximal flexibility in the overlay processing, the authors found that packet interception overhead resulted in a CPU-bound maximum sending rate of 3 Mb/s.

There are several approaches to enabling untrusted code to be added to the kernel in a safe way. In OS Sandboxing techniques, such as SFI [20], instructions are inserted to dynamically verify memory accesses and jump instructions. The Open Kernel Environment (OKE) project [22], SPIN [23], and Microsoft's Singularity [24] use type-safe languages and restricted access to kernel interfaces to enforce isolation. This requires careful language design and a method of validating that the extension code was compiled with the type-safe language. Palladium [25] is an architecture for safe kernel extensions that isolates extensions by preventing memory accesses outside the extension. Another alternative for isolating kernel extensions is proof carrying code [26].

The various mechanisms for adding untrusted code to the OS kernel offer promise for future systems in which metanet protocol stacks are more closely integrated with the OS. However, none of these approaches is directly applicable to existing operating systems, a clear requirement if we are to make endpoint diversification as painless as possible for users. We believe that the approach taken here can deliver acceptable performance and offer some preliminary evidence for that in the next section.

5. Preliminary Evaluation

5.1. Instantiating a new metaprotocol

Given an end system supporting the substrate, installing a new metaprotocol is very simple. A control daemon should be installed as a typical system daemon and added to the appropriate startup scripts. Because we may not unreservedly trust the control daemon, it can be run as a non-root user with an authorized group membership.

Bundled with the control daemon should be a metaprotocol library. Developers implementing applications that use this metaprotocol should just link with the metaprotocol library.

One complication is that every metaprotocol is identified by a number within the diversified network family. Because there is no central authority for assigning these numbers, user configuration on installing a new metaprotocol may include selecting an unused number. It would be simple to add a dynamic name to number mapping service to the system in the future.

Finally, the bandwidth settings for the new metaprotocol might be established by the user on installation. While our model assumes that a LAN-based bandwidth manager component will eventually deal with this process, we assume that the user should have some ability to specify bandwidth limits.

5.2. Performance of new protocol

To test the performance of the system, we created a minimal metaprotocol similar to a combined UDP/IP. We installed this metaprotocol and control daemon on a pair of 2.4 GHz machines running Linux 2.6.16 and connected via a 1000 Mb/s switch. To test the maximum available throughput, we wrote a simple sender/receiver application on top of the new metaprotocol.

With only a single metaprotocol, single socket and single application running, we tested our configuration at various bandwidth limits from 1 Mb/s to 1000 Mb/s. For each test, we ran the sender on a completely idle system and monitored the total system idle time with `top(1)`. A series of samples were taken at 3 second intervals, discarding the first 9 seconds of readings. From an average of these values, we could determine a total system utilization percentage.

We found that our maximum achievable bandwidth was 779 Mb/s. Because the CPU usage at peak was only 83% at this point, it is clear that the sender is I/O-bound, not CPU-bound. To confirm this, we ran a similar test using UDP from a user application, and achieved a maximum bandwidth of 780 Mb/s.

As shown in Fig. 2, our sender application CPU utilization is largely linear with respect to bandwidth consumed. An unusual phenomenon around 600 Mb/s is due to the

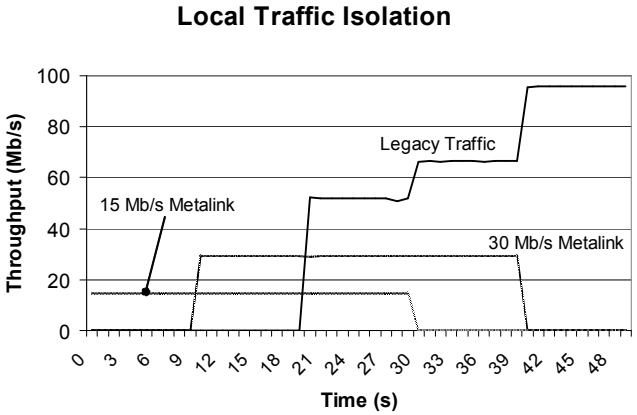

Fig. 3a. Isolation at the end system. Bandwidth is limited by egress queues. Metanet bandwidth is not impacted by other local traffic.

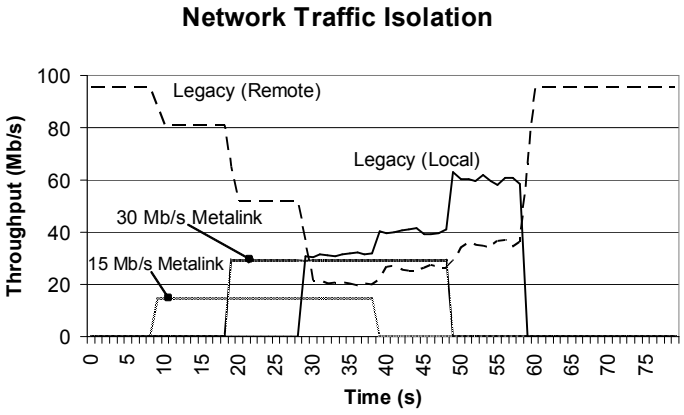

Fig. 3b. Isolation in the network. Legacy traffic (local and remote) is in low-priority queues. Metanet traffic is in high-priority queues and is not impacted by legacy traffic.

way in which the Linux Token Bucket is implemented. When there are insufficient tokens to allow sending traffic, the token bucket first dequeues the packet, checks the length, and sets a callback timer for when the queue will be ready, and finally requeues the packet for the next attempt. However, as an optimization, every time a packet is queued, a dequeue attempt is made (in case the packet is immediately ready). At speeds of 600 mb/s, this results in upwards of 50,000 failed dequeue attempts per second. At higher speeds, the queue never has a chance to run out of tokens.

We also ran a similar test with a small UDP sender. The UDP sender suffered the same problem near 600 Mb/s. More interesting is to compare the CPU utilization of our metanet application with the UDP application. While our utilization is always higher, the difference (83% vs 55% at peak bandwidth) is sufficiently small that our framework is competitive with native applications.

The performance of our prototype contrasts strongly with that of OASIS, which consumes the 100% of the CPU capacity at sending rates of under 10 Mb/s. While our prototype is less efficient than the native IP stack, it beats OASIS by well over an order of magnitude. Another comparative system is PL-VINI, which performs much better than OASIS, but still becomes CPU-bound near 200 Mb/s due to system call overhead. We avoid this by very strictly separating management of metaprotocols into the control daemon, data-related metaprotocol functionality in a metaprotocol library, and keeping the datapath clean of upcalls by allowing the control daemon to insert minimal vital services into the kernel in protected ways.

5.3. Traffic isolation

Our traffic isolation mechanisms consist of two pieces: we isolate metaprotocols from each other and from legacy traffic running current IP protocols on the local machine, and we isolate provisioned metaprotocol traffic from legacy traffic running current IP protocols within the network.

For the first case, we used two machines connected to the same 100 Mb/s switch as sender/receiver. Our sender was configured with two different metanet stacks, one with a provisioned 15 Mb/s metalink and another with a 30 Mb/s metalink. An additional flow was configured with legacy traffic running over current IP protocols. We began the flows 10 seconds apart in sequence and ran each one for 30 seconds. All bandwidth was measured at the receiver.

As shown in Fig. 3a, the 15 Mb/s flow begins and receives exactly the provisioned bandwidth of 15 Mb/s. The second flow enters at 10 seconds and also receives exactly the provisioned bandwidth of 30 Mb/s. At 20 seconds, the legacy flow enters and receives the balance remaining. At 30 seconds, the 15 Mb/s flow ends, and legacy traffic receives the remaining slack. At 40 seconds, the 30 Mb/s flow ends, and the legacy traffic receives the full link.

It is important to note that the sending applications did not have an established bandwidth limit. The only rate limit is supplied directly from the meta-interface. At no time did either legacy traffic or other metaprotocols interfere with allocated metaprotocol bandwidth.

For the second case, we added a third machine. The first machine ran the exact same flows in the same sequence. The second sender initiated a legacy flow to the receiver at 100 Mb/s.

The switch was configured with 802.1P/Q priority queuing. Because our provisioned metaprotocol traffic is marked at a higher priority than the legacy traffic, the metaprotocol traffic is sent preferentially to the bottleneck link, while legacy traffic is dropped.

As shown in Fig. 3b, the provisioned flows continue to receive their bandwidth allotments, while the legacy flows are reduced to the balance of the link.

5.4. Next steps

Several aspects of this system need further evaluation.

Because we have avoided placing the management portions of our architecture in the critical path, we have not undertaken performance analysis of these components. Particularly for applications that handle large numbers of connections, such as a web server, this overhead may become significant, and we need to analyze this futher.

We also have restricted our initial metaprotocol implementation to a minimal test case. Before we can confidently assert that this framework can meet the needs of an evolving Internet, we must validate it by implementing other protocol stacks and evaluating their performance. A complete IPv4 implementation would provide a strong endorsement of the framework. Likewise, analysis of existing transport and application protocols should be considered.

Our current schemes for ingress demultiplexing and egress validation use BPFs, interpreted code in the kernel. For the egress case, this limits the metaprotocol developer to those protocols which can be validated by purely stateless, single-frame filters. While of course a metaprotocol developer may eschew egress validation completely, this may allow a nefarious application developer to cause havoc within this metanetwork.

The ingress situation is more difficult. We suffer the limitation of a stateless, simplistic demultiplexing scheme, but we also have a potential performance impact from the ingress filters. Every packet arriving for a metanet will be checked against the ingress filter of every open socket for that metanet, an $O(N)$ operation. For end systems with a small number of open sockets per metanet, this is likely to have little impact on system performance, but for systems with many open sockets (such as servers), it could become a serious issue, and clearly needs to be evaluated. We are exploring general mechanisms to allow metanets to *preclassify* packets for comparison against a smaller set of filters, so as to reduce the number of filters that must be examined. We are also studying how performance might be improved by using a compiled filtering system, like DPF [19] in place of BPF.

6. Closing Remarks

In this paper, we have introduced a model for the diversification of the access network. We have created an initial prototype, and have presented preliminary evidence that performance considerations do not prevent serious adoption of network diversification. Our framework's strict adherence to keeping management and policy outside the critical path of sending and receiving results in near-native performance.

We have also demonstrated that existing quality of service mechanisms are adequate for enabling provisioned metalinks in the access network. This makes it possible to deliver network services requiring end-to-end QoS across appropriately designed metanets.

REFERENCES

[1] Anderson, Tom, Larry Peterson Scott Shenker and Jonathan Turner. "Overcoming the Internet Impasse through Virtualization," *IEEE Computer Magazine*, April 2005.

[2] GENI web site. http://www.nsf.gov/cise/geni/

[3] Report of NSF Workshop on Overcoming Barriers to Disruptive Innovation in Networking. [Online], Available: http://www.arl.wustl.edu/netv/noBarriers_final_report.pdf, January 2005.

[4] Turner, J. and D. Taylor. "Diversifying the Internet," *Proceedings of Globecom*, November 2005.

[5] Chun, Brent, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *ACM Computer Communications Review*, vol. 33, no. 3, July 2003.

[6] M. Huang, "VNET: PlanetLab Virtualized Network Access," Tech. Rep. PDN–05–029, PlanetLab Consortium, June 2005.

[7] Linux Advanced Routing and Traffic Control, [Online], Available: http://lartc.org/

[8] Hierarchical token bucket for Linux, [Online], Available: ttp://luxik.cdi.cz/ ~devik/qos/htb

[9] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. "In VINI Veritas: Realistic and Controlled Network Experimentation," SIGCOMM 2006.

[10] J. Touch, "Dynamic Internet Overlay Deployment and Management Using the X-Bone", Computer Networks, July 2001, pp. 117-135.

[11] J. Touch, Y. Wang, V. Pingali, L. Eggert, R Zhou and G. Finn, "A Global X-Bone for Network Experiments", In Proceedings of the TRIDENTCOM, February 2005, pp. 194-203

[12] Madhyastha, H. V., Venkataramani, A., Krishnamurthy, A., and Anderson, T. "Oasis: an overlay-aware network stack," *SIGOPS Oper. Syst. Rev.* 40, 1 (Jan. 2006), pp. 41-48.

[13] Chandramohan A. Thekkath , Thu D. Nguyen , Evelyn Moy , Edward D. Lazowska, "Implementing network protocols at user level", IEEE/ACM Transactions on Networking (TON), v.1 n.5, pp. 554-565, October 1993

[14] Chris Maeda, Brian Bershad, "Protocol Service Decomposition for High-Performance Networking", Proceedings of the 14th ACM Symposium on Operating Systems Principles. December 1993, pp. 244-255.

[15] Aled Edwards , Steve Muir, "Experiences implementing a high performance TCP in user-space", Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, pp. 196-205, 1995

[16] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney, "Fast and Flexible Application-Level Networking on Exokernel Systems", ACM Transactions on Computer Systems, Vol. 20, No. 1, February 2002, pp. 49-83.

[17] Microsoft Corp., (1999, September) Quality of Service Technical White Paper, [Online]. Available: http://www.microsoft.com/windows2000/techinfo/howitworks/communications/trafficmgmt/qosover.asp

[18] S. McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture", In Proceedings of the USENIX Winter Technical Conference, San Diego, CA. 1993, pp. 259-269.

[19] D. Engler, M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation", In ACM SIGCOMM 1996, pp. 53-59.

[20] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation", in Proceedings of the Fourteenth ACM Symposium on Operating System Principles, pp. 203-216, December 1993

[21] M. Swift, B. Bershad, and H. Levy, "Improving the Reliability of Commodity Operating Systems", in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, October 2003.

[22] Herbert Bos, Bart Samwel, "Safe Kernel Programming in the OKE", Proceedings of the fifth IEEE Conference on Open Architectures and Network Programming, June 2002

[23] Marc Fiuczynski, Brian Bershad, "An Extensible Protocol Architecture for Application-Specific Networking", Proceedings of the Winter USENIX Technical Conference, pp. 55-64, January 1996

[24] G. Hunt, *et al*, “An Overview of the Singularity Project,” Microsoft Research Technical Report MSR-TR-2005-135

[25] T. Chiueh, G. Venkitachalam, P. Pradhan. “Integrating segmentation and paging protection for safe, efficient and transparent software extensions,” Proc. ACM SOSP 1999

[26] G. Necula, "Proof-carrying Code", In Proceedings of the twenty-fourth Annual Symposium on Principles Programming Languages, pp. 106-119, January 1997