

Diversifying the Network Edge

Fred Kuhns, Michael Wilson, and Jonathan Turner

Department of Computer Science and Engineering

Washington University

St. Louis MO. 63130

{fredk, mlw2, jst}@arl.wustl.edu

Abstract: The Internet has become an indispensable part of our modern information society. Unfortunately, with success has come a growing ossification in the core technology at the heart of the Internet that severely limits our ability to make fundamental architectural improvements. A recent call to arms [1] advances a research agenda to create a *virtual testbed* to overcome three barriers to effective architectural research: live testing, deployment and broad-based solutions. *Network diversification* [2] seeks to make virtualization a central feature of the core network infrastructure. It has the potential to permanently eliminate the problem of network ossification, by creating an environment in which alternate network technologies can be introduced easily, and can co-exist and compete with existing technologies. In this paper, we focus on what network diversification means for the access portion of the network, in particular the end systems that connect to the network and the local area networks commonly used to provide access.

Index Terms—extensible networks, virtual networks, protocols, operating system extensions, overlay networks

I. INTRODUCTION

In a remarkably short time, the Internet has become an indispensable part of business and daily life. On a personal level, it has changed the way we interact with the world and organize our lives. However, despite its success, the core protocols at the heart of the Internet have significant limitations. Over the years, various ad hoc mechanisms have been created to cope with some of these limitations (e.g. network address translation, firewalls and intrusion detection systems). These mechanisms distort the original Internet architecture and make it difficult to address new challenges. Solutions to other architectural limitations (e.g. lack of support for QoS and multicast) have been developed, but have not been widely deployed, due to a variety of largely non-technical factors. There is a growing recognition that the Internet has become ossified [1]10 and this ossification threatens its future growth and development.

To address these issues, the authors of [1] propose the development of a large-scale *virtual testbed* to enable new network technologies to be developed and experimentally evaluated. The testbed would use virtualization to allow multiple network technologies to co-exist, without interference, on a shared infrastructure. This idea is carried one step further in [2], which argues that virtualization should

be a core capability of the underlying network technology. This would eliminate the large barriers to entry that now face new network technologies and would make it possible for innovative solutions to new challenges to co-exist with existing technologies, and be allowed to succeed (or fail) on their own merits. In this paper, we seek to develop these ideas further, concentrating on the issues that arise at the edge of the network.

The concept of network diversification (*ND*) borrows heavily from previous work. Planetlab [4] is an experimental overlay network that uses virtualization to enable the construction of a wide range of different overlay networks on a shared infrastructure. The X-Bone [3] software system enables the construction of overlay networks with similar characteristics and incorporates an explicit virtual link concept. Network diversification seeks to make virtualization a core capability of the underlying network infrastructure. It allows multiple networks to operate in parallel on a shared *substrate*. The substrate is a thin resource provisioning layer that allocates resources (link bandwidth and processing resources in routers) to different virtual networks. The substrate provides mechanisms to ensure that different virtual networks do not interfere with one another, but otherwise places no constraint on how they use the resources at their disposal.

Section II provides a more detailed description of Network Diversification Architecture (NDA) and Section III describes the LAN environment with a focusing on Ethernet. Then in Section IV a detailed description of the end-system model and required operating system modifications are provided. This is followed by a description of the related work in Section V. A general discussion of this approach and future directions is given in Section VI.

II. NETWORK DIVERSIFICATION

A virtual testbed is proposed in [1] for overcoming barriers to architectural research and deployment of new network technologies. The virtual testbed concept is not new; rather it's a refinement to the current approach of using overlay networks and multiplexed overlay nodes [4]. As implied by the name, virtualization is a unifying theme with obvious roots in earlier work in operating systems. In what follows concepts and terminology have been borrowed from PlanetLab [4] and X-Bone [3]

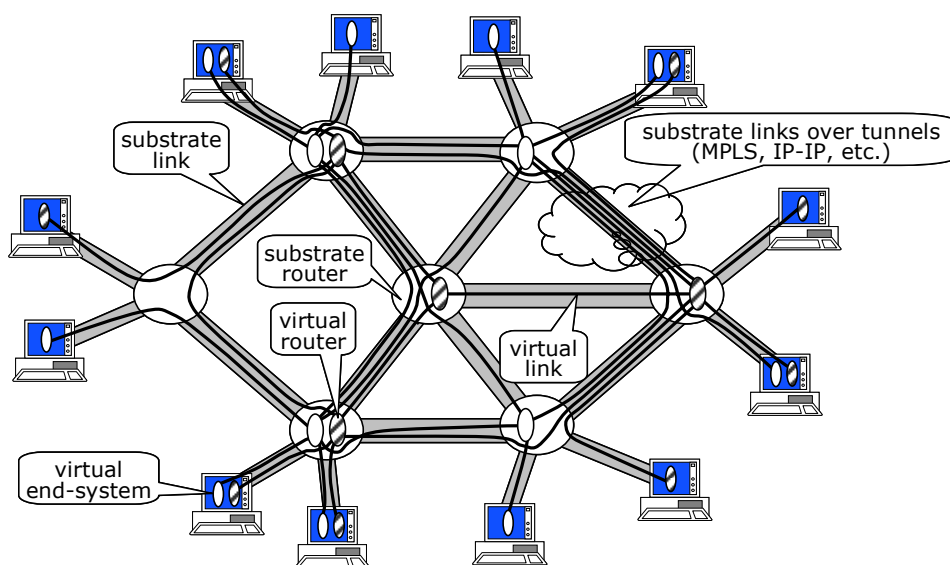


Fig. 1: The Basic Network Diversification Model

1) Comparison to OS Virtualization

In operating systems (OS), hardware resources (processor, memory and peripherals) are abstracted and managed in such a way as to present an extended virtual machine to clients. The OS exploits underlying mechanisms (hardware, software or both) to support *transparent isolation*, *controlled sharing*, *policy enforcement* and *accounting functions*. A program executes oblivious to the fact that system resources are multiplexed among unrelated processes and system tasks¹. Various policies are used to drive the allocation and scheduling of resources in order to meet quantitative or qualitative system metrics. Example policy measures are fairness, interactive responsiveness, throughput, or power consumption. Efficient virtualization and policy enforcement are made possible through a combination of hardware, software and protocol mechanisms.

Building on this theme, our proposed Network Diversification Architecture employs network virtualization to transparently support disparate network instances, i.e. virtual networks or simply *vNets*, all sharing a common internetworking infrastructure. Since networks are described in terms of protocol layers, the problem reduces to defining a common virtualization layer that sits between a desired network layer and existing data link layers. We refer to the mechanisms that implement this virtualization layer as the *substrate*.

2) Basic Virtual Networking Model

Operating systems export abstractions representing a physical machine, the virtual network model defines abstractions representing a physical network. Rather than processors and memory, virtual networks have the equivalent abstractions for links, routers and interfaces. So the basic building blocks for a virtual network are virtual links, virtual

routers and virtual interfaces. In its simplest form a *virtual link* models a unidirectional, fixed bandwidth link interconnecting adjacent virtual routers belonging to the same vNet. Virtual links originate and terminate at *virtual interfaces*. A *virtual router* implements a specific vNet protocol, forwarding algorithm and associated control protocols. A virtual router receives packets on input virtual interfaces and optionally forwards each packet to one or more output virtual interfaces.

Just as operating systems rely on indirection (for example virtual memory or the file system interface), so must the virtual network model, in order to separate the abstractions from their implementation. The X-BONE and Virtual Internet Architecture achieve similar indirection by using two layers of encapsulation (virtual link IP address and virtual network IP address) when defining virtual links thereby enabling a notion called revisitation [3], [5]. In our case, the substrate is responsible for creating the per virtual network illusion of dedicated resources.

The substrate layer creates the virtualized resource abstractions, enforces isolation and guarantees service levels for vNet instances. The substrate layer in turn is composed of substrate links, substrate routers and substrate interfaces. A *substrate router* provides an environment for hosting virtual routers, implementing virtual interfaces (i.e. the virtual link endpoint) and providing other virtualization services. *Substrate interfaces* define a context within which to enforce virtual link name spaces, bandwidth allocations and routing. In the basic model, substrate routers are interconnected by point-to-point *substrate links*, which terminate at substrate interfaces and are implemented using existing link or network layer protocols. If substrate nodes are physically adjacent then substrate links are typically implemented using the available link layer. When not adjacent, or when circumstances warrant, they can be implemented using various tunneling technologies such as PPP over SONET, MPLS, IP, IPsec, GRE or ATM. In summary, substrate links serve three purposes:

¹ Of course two or more processes may choose to knowingly share one of more resources with the operating system providing any necessary operations to permit coordinated access to said resource (controlled sharing).

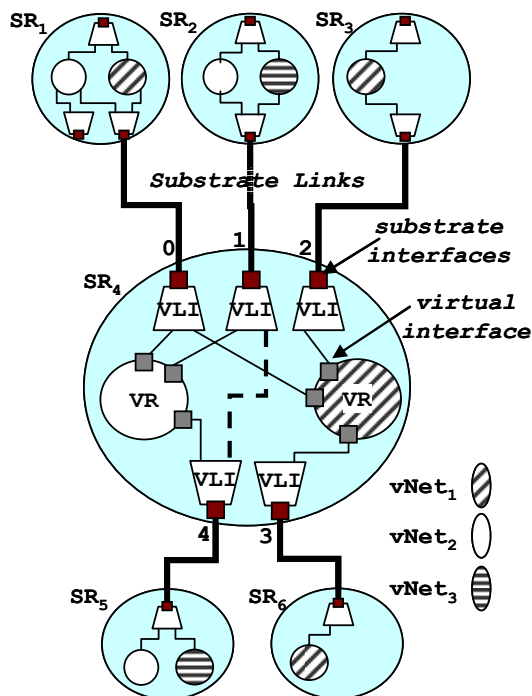


Fig. 2: Logical block diagram illustrating the relationship between substrate and virtual components.

1. Define a virtual link namespace scope.
2. Provide a convenient accounting entity for link bandwidth partitioning.
3. Create an indirection layer allowing for the tunneling of traffic across legacy nodes.

Virtual links can be defined within a substrate link and are identified by a packet header field called the *Virtual Link Identifier*. When defining a virtual network topology it is not necessary to include a virtual router on every substrate router. As with ATM and MPLS, the substrate layer uses the VLI field to route a packet to its destination virtual router or virtual end-system (i.e. virtual node).

Fig. 2 shows a typical substrate router, SR_4 , which hosts two virtual routers each with three virtual interfaces. There are a total of seven virtual links and five substrate links. The substrate link connecting SR_1 to SR_4 (substrate interface 0) carries two virtual links shown as the lines connecting the VLI demultiplexor to $vNet_1$ and $vNet_2$ virtual routers. The substrate link connecting SR_2 to SR_4 includes a virtual link which does not terminate at SR_4 , rather it continues to SR_5 . The substrate layer uses the VLI field in the substrate header to properly demultiplex packets to the correct virtual router or output interface.

3) Dealing with multi-access links

While point-to-point substrate links are suitable for connecting substrate routers to one another, they have limitations when applied to the access portion of the network, where multi-access local network technologies are prevalent. In order to facilitate the deployment of diversified networks, we propose to operate on top of existing access technologies such as Ethernet, and to co-exist with traffic that is not associated with the diversified network infrastructure. An

important test case for the diversified network technology is its ability to support IP within a virtual network. Because IP routers are designed to exploit characteristics of multi-access networks, it is important that a diversified network substrate enable IP to take advantage of these multi-access networks in much the same way.

One key issue raised by these considerations is how to provide QoS within a multi-access network. By their nature, multi-access links do not lend themselves readily to the provision of QoS. Fortunately, the growing use of virtual LAN (VLAN) technologies with priority-based queuing creates possibilities for delivering QoS that have not previously been available. We propose to use these mechanisms to implement point-to-point substrate links with reserved bandwidth within multi-access networks, so that virtual networks that require reserved bandwidth on their virtual links can get that capability.

On multi-access physical networks, we also provide a multi-access substrate link that can be used by any virtual network. We make no attempt to deliver QoS on this multi-access substrate link. We simply seek to provide virtual networks the same kind of capability that is provided by the underlying network. This is discussed in greater detail in the next section.

III. DIVERSIFYING THE LAN

The implementation of the virtual networking model in the LAN environment has the same concerns and goals as the core network. The primary difference is the prevalence of multi-access networks and virtual end-systems. As with the core network, substrate links emulate physical links (point-to-point or multi-access), interconnect adjacent substrate nodes and are defined in terms of either the underlying physical link or higher layer tunnels. The substrate exports a virtual link abstraction to the virtual network instances emulating physical links in terms of bandwidth constraints and access model. This link abstraction model is depicted in Fig. 3 where virtual links are exported to two virtual network instances, $vNet_1$ and $vNet_2$. In this example, $vNet_1$ virtual end-systems (instances of the $vNet_1$ protocol) use point-to-point virtual links to connect to their corresponding virtual router. While all $vNet_2$ nodes share a common multi-access virtual link.

1) Defining the LAN environment:

Local area networks vary in size and complexity from simple one or two computer home networks to large campus networks. Home networks are characterized by a relatively small number of end-systems, a lightly loaded LAN and one Internet connection using a DSL or cable model. In this environment the bottleneck is the broadband connection from the home user to the ISP. For the home network it is sufficient to have support within the end-systems and access router, there is no need for traffic isolation and rate control within the LAN's link layer devices. Traffic isolation and shaping is implemented within the end-systems and substrate routers.

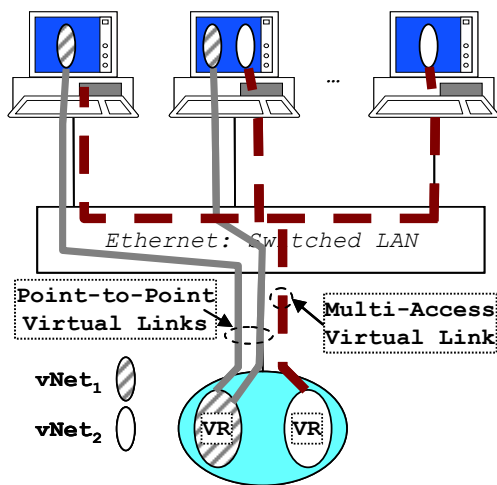


Fig. 3: Multi-access and point-to-point virtual links for an Ethernet LAN. All substrate nodes are members of the multi-access substrate link.

The campus network is different. In this case we expect the network to be large and complex and the LAN technology to be dominated by switched Ethernet. Further, enterprise class switches have extended support for VLANs and QoS using IEEE standards 802.1P/Q. It is not uncommon for these switches to support additional capabilities such as weighted round robin service disciplines or packet classification based on bit masks or higher layer fields. Consequently, for the campus network we assume the presence of LAN switches that support VLANs, per packet priorities (3-bits) and priority queues and priority based service disciplines. In this case, isolation is gained by placing candidate vNet traffic in a priority class higher than that of the legacy and best-effort traffic. It is the responsibility of the substrate layer on the end-systems and routers to correctly police and shape traffic entering the LAN.

2) Design Forces:

A fundamental goal of this work is to create a simple and intuitive virtual networking architecture which may be deployed within existing networks with minimal impact on current operations. We do not assume the ability to dynamically alter existing network device attributes such as VLAN membership. Rather, we assume substrate links are statically configured out-of-band, most likely by network administrators. However, virtual links are assumed to be under the complete control of the substrate layer and so may be created and destroyed dynamically.

In the LAN setting the model requires the following:

- End-systems are connected by a point-to-point substrate links to adjacent substrate routers.
- Neighboring substrate routers are connected by point-to-point substrate links.
- Substrate links do not connect end-systems.
- A multi-access LAN has a multi-access substrate link defined connecting all substrate nodes.
- Substrate links are defined statically by an administrative authority.

- All substrate links defined on the same physical interface use the same encapsulation mechanism.
- Virtual links are defined and managed by the substrate layer and require no outside authority for their creation or destruction.
- Traffic isolation and bandwidth guarantees are only available for point-to-point substrate links. Multi-access substrate links are treated as best-effort and handled similar to legacy traffic.
- On Ethernet LANs one VLAN identifier and one priority level are used for all point-to-point substrate links. A priority is assigned that is greater than LAN best-effort traffic (legacy and multi-access substrate link) to ensure isolation.
- Optimize for the simple case represented by an Ethernet switched LAN with a single substrate router and end-systems using point-to-point links for internetworking.

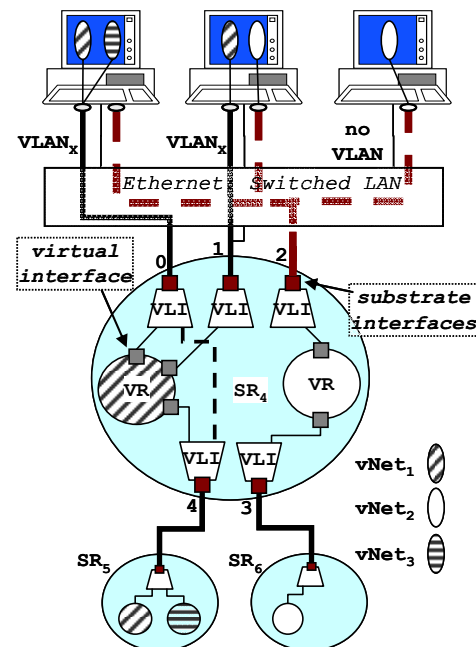


Fig. 4: LAN example showing relationship between point-to-point and multi-access links

3) Realizing the Link Abstraction Model:

Figure 4 expands on Fig. 3 to illustrate how virtual links are mapped to substrate links on an Ethernet switched LAN. In this figure all point-to-point substrate links share a common VLAN identifier (VLAN_x) and priority. Within each substrate link, the packets are tagged with a VLI to identify the virtual link and its associated vNet instance. The multi-access link extends to all substrate nodes and does not use a VLAN identifier. In this scenario packets contain an Ethernet header which includes the source address, destination address, type fields and optionally a VLAN identifier and priority. A new Ethernet type value is defined for virtual network packets allowing for their identification and processing of the substrate layers header.

On receiving a packet a substrate node first checks the type

field to determine if it should be processed as a virtual network or legacy traffic. If virtual network then the VLAN identifier, if present, is used to distinguish point-to-point substrate links from the multi-access link. In either case the senders Ethernet address is used to identify the substrate link context. Finally, the correct virtual link (and thus destination virtual router or virtual end-system) is obtained by combining the virtual link identifier in the substrate header with the substrate link context.

4) *Bandwidth Management in the LAN:*

Bandwidth management in the LAN is divided into traffic management across substrate links and vNet bandwidth allocation and policy enforcement across the LAN. Substrate link traffic management is primarily a local concern dealing with individual virtual link rates and node policy enforcement. Every node, end-system and router, includes an entity called a *traffic manager* (TM) responsible for enforcing rate limits of individual links and aggregate vNet limits. This is primarily an enforcement mechanism.

vNet bandwidth management however deals with global issues such as allocating LAN bandwidth to substrate links and ensuring the aggregate rate of vNet traffic does not exceed either the per vNet or total rate limits. A LAN has a vNet Bandwidth Manager (VBM) associated with it that is responsible for global policy issues, allocating resources and the efficient use of LAN resources. It communicates with the substrate node traffic managers to set node specific aggregate vNet rates.

Traffic Manager: The traffic manager on each substrate node is assigned an aggregate rate for resident vNets. This rate is an upper bound for the sum of all point-to-point virtual link *input* rates associated with that vNet. It is dynamically assigned by a VBM and is subject to change based on global LAN demand and policy. The traffic manager is free to assign rates to input virtual links as long it does not exceed their maximums and the total remains within this bound. Based on policy or demand the TM may request an increase or reduction in this aggregate rate.

Every point-to-point virtual *output* link has associated with it a current, maximum and minimum bandwidth allocation. In general the maximum and minimum allocations are set at virtual link creation and do not change. The current allocation varies between these extremes in response to demand. A traffic manager will initiate a current rate change in response to policy changes, demand or external requests. In order to change a virtual link's output rate, the local traffic manager must perform two operations. The first is to ensure the desired rate is less than the allowed maximum for that virtual link and it does not violate local policy. It then sends a message to the downstream TM requesting the rate change. Generally, a rate decrease will succeed. However an increase could fail for several reasons such as insufficient local resources or the node's current aggregate vNet allocation is insufficient.

When a node receives a request for a rate change (from an upstream node) it first applies any matching local policies followed by comparison against existing allocations and the

aggregate for the corresponding vNet. If necessary the node may send a message to the VBM requesting an aggregate increase/decrease which may or may not succeed. Assuming there is sufficient bandwidth available the virtual link's rate is increased and a reply sent authorizing the increase. In summary, a virtual link's rate may be changed for the following reasons:

- Local policy enforcement. For example, a policy may be to decrease a virtual link's rate to the minimum value if it is idle for a predefined interval.
- Increase or decrease in demand. If a virtual router or virtual end-system increases its sending rate and the current rate is less than the maximum rate then the traffic manager will attempt to increase the current rate by sending a request to the downstream node.
- External request. An upstream node may request a change in the current allocation of a virtual link.
- Aggregate rate change. A node may change the current allocations for the input virtual links of a vNet due to an aggregate rate change from the VBM. Upstream nodes are informed of these changes.
- Configuration changes. Configuration changes may result in vNet rate and membership changes.

vNet Bandwidth Manager: When a vNet is instantiated within a LAN it is assigned a percentage of the available bandwidth. The VBM then allocates the minimum aggregate vNet rates to substrate nodes on the LAN based on the minimum rates of the virtual links they support. Over time these allocations change in response to requests from node traffic managers, aggregate demand or policy enforcement. In general it will not be possible to allocate maximum rates to all virtual links. In this case each virtual link is initialized with its minimum allocation and additional bandwidth is assigned as needed.

For simplicity, the LAN is treated as though it is a single shared segment with bandwidth equal to that of the lowest bandwidth link on the LAN. While it may be possible to determine the VLAN spanning tree, for example by using SNMP, this is not supported by all switches. So rather than have special cases the basic model only supports this simple case.

IV. DIVERSIFYING THE END-SYSTEM

A *virtual end-system* is an abstraction representing an instance of a particular virtual network protocol on an end-system. Fig. 5 illustrates the architecture in block diagram form showing the vNet Framework and control daemons in relation to other major components of the network subsystem. The framework allows for vNet protocol instances to reside in either user or kernel space. Overall, the vNet framework on an end-system implements the link and end-system abstraction models providing the necessary isolation and traffic management functions to the protocol instances.

A. *Design Forces:*

The vNet end-system architecture implements the link

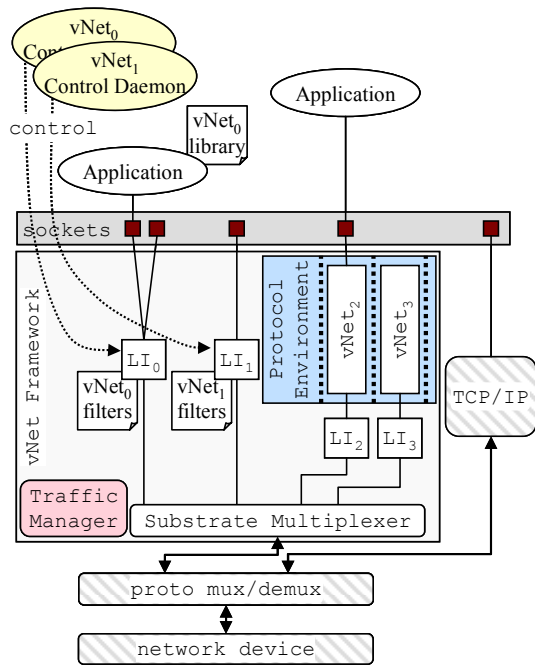


Fig. 5: Virtual End-System Block Diagram. Logical interfaces LI_{0-3} represent virtual link endpoints to the protocol instances.

abstraction model, provides traffic management functions and defines an OS extension framework without compromising system *safety*, *backwards compatibility* or *security*. In particular, an end-system continues to support current TCP/IP protocols and services without change. Three areas are addressed to support the vNet architecture within the end-system: (1) substrate layer support, (2) operating system extensions for new protocols and (3) the protocol development environment.

The addition or removal of a vNet is a relatively rare event. Network diversification targets the deployment of complete networks including router architectures, protocol implementations and applications. Given the scale and significance of the changes, deployment is necessarily protracted and includes a disparate collection of collaborators. It is not intended to support rapid creation of short term, application specific virtual networks. However, it does support dynamic membership in existing virtual networks.

It is assumed that end-system users make a conscious choice to add support for a new vNet and manually initiate local installs and or configurations. The effort is comparable to installing drivers for new hardware, possibly after downloading modules from a known source on the Internet. Changes to the end-system do not impact existing applications or the use of system programs and utilities. In principle, the addition of a new vNet on an end-system simply extends the list of available protocol domains through existing inter-process communications (IPC) facilities. An important consequence is the isolation mechanisms used for vNets do not extend to the application level. An application is free to open any combination of legacy IPv4 and vNet endpoints, implying it may also serve as a gateway between networks.

B. Substrate layer

The substrate layer has two primary functions: (1) multiplex virtual links onto substrate links and (2) traffic isolation and bandwidth management. The substrate layer defines OS dependent Logical interfaces² ($LI_0 - LI_3$ in the Fig. 5) to represent the two layers of indirection required to implement the virtual and substrate links. Virtual network protocol stacks are shielded from the implementation details, the substrate layer uses existing OS mechanisms to export an interface consistent with existing network devices. To the protocol instance the logical interface looks exactly like a physical interface. By hiding the details behind an operating system supported logical device porting existing protocols requires little or no change to table formats and programmatic interfaces.

Each logical interface is initialized with the state necessary to identify the corresponding virtual and substrate links. For example, a virtual interface is initialized with a VLI, substrate link reference and physical interface reference. During a send operation the logical interface verifies packet format (policy driven) and prepends a substrate header conforming to the type of encapsulation. For Ethernet encapsulation the VLI and frame length are included. If an IP tunnel is used then just the VLI is included. Encapsulation is the next step which depends on both the link's access type and encapsulating protocol.

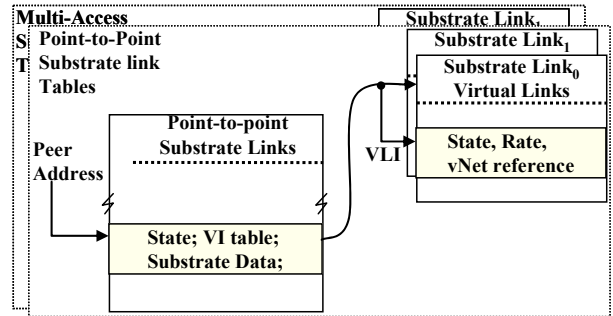


Fig. 6: Substrate Link Translation Tables

The substrate maintains a set of substrate and virtual link tables representing relationships and current state, see Fig. 6. The substrate link table store encapsulation data used for sending packets across a specific link. The logical device maintains a reference into the point-to-point substrate links table, see Fig. 6. The substrate data field includes all information necessary to correctly encapsulate packets for the given substrate link. For Ethernet encapsulation this includes the destination Ethernet address, VLAN identifier and priority. For IP encapsulation the source IP address and destination IP address are required.

Tables are also maintained for multi-access links, but in this case explicit link addresses for encapsulation are not required. Either ARP is used (dynamic address translation) or the sending protocol instance must supply an address translation.

² To avoid confusion we will use the phrase logical interface to represent an OS specific mechanism for creating hardware independent device drivers.

The substrate layer also enforces traffic isolation and bandwidth constraints. Fortunately, both Microsoft windows and many UNIX based operating systems provide traffic control mechanisms in the kernel. Linux has traffic control [6] built into the kernel allowing packets to be classified and associated with packet scheduling disciplines. There is support for class based queuing (CBQ) [7] and hierarchical token bucket (HTB) [8] packet schedulers. Microsoft Windows server 2003 has a traffic control service a generic packet classification driver and QoS packet scheduler (NDIS intermediate driver). An administrator is able to install classifier filters to map packets into traffic classes which are associated with a priority and shaping parameters [9].

C. Protocol Environment

The vNet end-system architecture defines a framework for dynamically extending the networking subsystem of an operating system. The vNet protocol extension framework maintains backward compatibility with existing programs and a consistent interface to support standard development patterns, idioms and paradigms. However, allowing dynamic extension of operating systems has very real consequences for system properties such as security, safety³, liveness and performance.

1) Operating System Environment:

A typical general purpose operating system consists of a base kernel, kernel extensions (e.g. third party drivers), system programs and interface libraries. The base kernel and extensions run in privileged mode and have unrestricted access to all system resources and so are trusted to correctly implement system functions. The privileged system programs and daemons operate with special capabilities entitling them to manipulate system tables and access restricted resources. Finally, the system libraries permit unprivileged applications to request the operating system to provide a service or to perform privileged operations on its behalf. The correct operation of the system depends on the kernel's ability to authenticate the user, authorize the request, validate parameters and enforce policies.

Extending operating systems with third party modules challenges the underlying trust model: I trust the operating system is *sufficiently correct*⁴, efficient and relatively free of bugs. However, adding third party extensions may compromise system behavior by introducing software bugs, unexpected interactions or malicious code. In Linux the frequency of errors in device drivers is 7 times that of the base kernel [10]. In our case, rather than a general OS extension the domain is restricted to network protocol extensions within the vNet framework. However the point remains true, it is reasonable to expect that extending OS functionality without appropriate constraints will increase the risk of adversely impacting system safety, security, liveness and performance.

³ While security may be considered a safety property we list it separately to highlight its importance.

⁴ By sufficiently correct we mean to convey the notion that users expect the OS to produce either correct or obviously incorrect results.

Not all OS extensions are equal; extensions applied directly to the OS kernel can be particularly problematic since they can have far reaching consequences. Kernel errors may result in unexplained system crashes, application memory corruption, resource starvation, poor performance or incorrect accounting. For example, the kernel (and therefore any of its extensions) is able to bypass memory protection and access process memory or it may subvert the authentication mechanism by allowing certain users to operate with increased privileges. Consequently an error in an OS kernel (or kernel extension) would likely result in negatively impacting system behavior.

Alternatively, system programs offer services to applications such as password authentication, electronic mail delivery, file management or a naming service. Since these services generally do not have unrestricted access to process or system resources, errors do not necessarily lead to violation of global system properties (that is, the errors are contained to the compromised service). Compartmentalizing a system service within a process effectively isolates it enabling the system to enforce the use of a restricted interface. Isolation prevents a large class of errors from having an impact on unrelated process or the base operating system.

When extending the functionality of an OS the question then is whether to extend the existing kernel or implement as user space libraries and service daemons. While a user space implementation does not necessarily improve safety or security properties for the extension itself (or the service it provides) it does simplify ensuring these properties for the system. Loading an unsafe or malicious protocol and installing in user space is much less likely to compromise system security than if it were loaded into the kernel. A running protocol instance is necessarily isolated to a single process restricting its ability to affect unrelated entities. If a control daemon is compromised than a system service or shared data may be compromised but this is isolated to the offered service.

2) vNet Protocol Framework:

The vNet protocol framework supports both user and kernel space protocol implementations, as shown in Fig. 5. The primary focus is on a dynamically extensible user-space protocol framework; however, the framework does support a migration path for protocol instantiation in the kernel. Assuming the kernel manages the interface hardware and buffers then the primary issues with this approach for protocol development center on efficiency, connection life cycles and managing global state.

The architectural models presented in [11], [12] provide a firm foundation on which to build a robust and efficient user-space protocol framework. We have adapted their work to the current environment and context; we refer the reader to the references for a more detailed discussion. In the virtual end-system architecture a virtual network instance includes a protocol library and control daemon. All data path operations are implemented within the library against which applications link. Any protocol control or security sensitive operations are

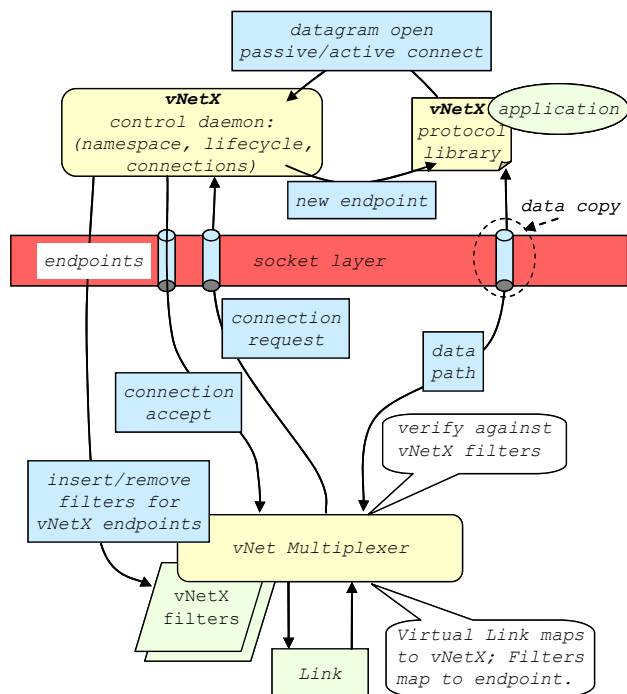


Fig. 7: User Space Protocol Architecture

implemented within a privileged system daemon which communicates with both the vNet framework's logical interfaces. Operations such as connection management, routing or authentication are handled by this vNet specific control daemon. Fig. 7 highlights the basic structure of the user-space protocol environment; the components are described in the following paragraphs.

vnet Multiplexer: The vNet multiplexer is part of the substrate layer and resides in the OS kernel. The multiplexer maintains separate configuration and state tables for each virtual network. A packet filter [15], [16] is used to properly demultiplex packets to the correct socket (i.e. endpoint) and optionally for validating outbound packets.

Control Daemon: Protocol operations which are likely to impact integrity, security or global state are placed in a common, privileged control daemon. For example, when a connection is established, kernel state must be created to associate the connection with the correct endpoint. If an application could arbitrarily establish this state then it would be able to masquerade or eavesdrop on other connections. So only a privileged system daemon is permitted to manipulate substrate state binding endpoints to vNet flows. Likewise, global state should only be trusted to a system process which requires a privileged user to install and configure. So the control daemon manages the namespace and other global attributes. System tables such as for routing and address translation, are kept in shared memory and exported read-only to application processes permitting efficient access.

Protocol Library: vNet specific protocol libraries are linked into applications and provide the necessary data path processing and interfaces to the vNet control daemon. The use of libraries for data path operations has two advantages: accounting and isolation. Accounting and isolation are

improved since protocol data path processing is charged directly to the application (no *hidden scheduling*). An implementation that consumes too many local resources (e.g. buffers or CPU) either because of an error or a greedy application is isolated to the application.

3) Kernel-space protocol implementations:

This is not the primary mechanism for implementing protocols, rather a migration option available when issues of efficiency or security warrant. A kernel extension can compromise the system through the unauthorized access or use of system memory, interfaces and resources. The substrate layer effectively isolates vNet network traffic and manages bandwidth allocations on the physical link. However, this is not sufficient to sandbox a particular vNet instance necessitating the use of additional constraints. Fortunately, protocols do not need to access general kernel data and interfaces, rather at a minimum they need only to exchange network packets and control data (from/to user space).

The run time environment of a protocol instance includes a call stack, heap memory and operations to manipulate packet data and control structures. Additionally, protocols are able to request periodic callbacks (e.g. timers); allocate and free packet data and control structures; use synchronization mechanisms when needed; and request the current process to be blocked if unable to allocate necessary resources. In summary, a vNet instance is executed within a *sandbox* with the following restrictions:

- **Memory access:** An executing vNet instance is allocated its own stack and heap segments enabling the run-time environment to conveniently isolate memory references. Access to memory is restricted to either an instance's allocated segments or select kernel interfaces and data structures. Permission to modify memory outside if the vNet instance is not permitted.
- **Accessible interfaces:** Restrict external interface access to just those exported by the substrate layer. The substrate layer provides wrappers for all exported kernel and substrate interfaces which implement parameter verification and constraint checking.
- **Resource use:** The kernel enforces resource limits for protocol instances. Exceeding CPU allocations cause an instance to be preempted and possibly unloaded. Access to other resources such as memory is explicitly limited at allocation time. Stack overruns result in a memory access exception and the protocol is suspended and unloaded from the system.

We adopt an extension framework for kernel resident protocols that is similar to the open kernel environment (OKE) [17]. In OKE a sandbox is created by using a type-safe variant of the C programming language (Cyclone) [18], wrappers for kernel interfaces and a run time environment to trap exceptions and enforce access constraints.

V. RELATED WORK

Related work is grouped into network and end-system virtualization categories.

A. Network Virtualization

1) Comparison to PlanetLab

Virtual testbeds have been proposed [1] as a solution to the current ossification of the Internet. The current approach to network virtualization uses Internet overlay networks connecting virtual nodes which implement a network service or distributed application. PlanetLab [4] is an example of a global scale testbed enabling the deployment of disruptive technologies. Currently IP tunnels are used but this is not a core architectural feature, rather the focus is on "broad-coverage services". Network diversification can be considered an implementation technology for virtual testbeds similar to PlanetLab[1]. For example, PlanetLab can be viewed as a set substrate routers interconnected by substrate links implemented as IP tunnels. A slice is then a set of virtual routers which constitute a virtual network. The only additional capability required of a Planetlab node is support for the virtual link/interface abstraction, i.e. demultiplex on the VLI to a particular virtual machine/service.

This has clear implications for a PlanetLab node collocated on an Ethernet LAN with other hosts. Exploiting the LAN diversification model vNet traffic is isolated from other LAN traffic enabling experimentation with a range of QoS sensitive technologies.

2) Comparison to the GX-Bone

The X-Bone [3], Virtual Internet and GX-Bone [5] projects are representative of overlay networks which focus on the network layer. While the Network Diversification Architecture shares some terminology and concepts with GX-Bone there are fundamental differences in intent and architecture. 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. Overlay networks can be constructed dynamically with little effort and on short time scales. The disadvantage is this adds overhead for the virtual routers and unnecessarily restricts deployed networks to use the existing IP infrastructure.

Network diversification makes no such assumptions. The substrate layer hides the details of how links are realized and permits any number of existing link and network layer technologies to be used. In the short term, diversified networks may in fact be realized using IP tunnels to interconnect PC based substrate routers. However, network diversification's abstraction layer enables low level isolation and protection so virtual networks are decoupled from the underlying implementation. This independence permits the live testing and deployment of broad-based solutions addressing fundamental architectural challenges.

A key feature of the GX-Bone is support for revisitation and recursion. Revisitation simply means that virtual routers for the same virtual network may be collocated on the same hosting node. This is directly supported by network diversification.

Network recursion for the GX-Bone means a virtual

network can be represented as a single virtual router in a higher level virtual network. This is similar to the role of autonomous systems and hierarchical routing protocols that define administrative boundaries and hide a network's internal topology. Since the GX-Bone depends on IP as an enabling technology (arguably any network layer protocol will do) this is a natural and important feature. However, network diversification provides only a link level abstraction and deliberately avoids assuming the existence of an underlying network layer protocol. Independent administrative domains define substrate routers at their edges and enable substrate links to peering substrate routers. Of course the substrate link itself could span many networks but this fact is invisible to the virtual networks. So we argue that recursion as a property, and as defined by GX-Bone is not necessary in the network diversification architecture.

B. End-system virtualization

Our motivation is to extend the network diversification architecture to include the end-system. A key enabling technology is a dynamically extensible networking subsystem that permits the safe and convenient installation of new protocols. The protocol environment leverages concepts and techniques from work in application-level networking [11]-[14] safe kernel extensions [10], [17], [19] and protocol extension [20]. The virtual end-system architecture and vNet framework differ from this prior work in two ways: (1) it defines an integrated framework for deploying both user and kernel space protocol implementations and (2) integrates a virtual networking model with an extensible end-system network subsystem.

1) User-space protocols

The user-space protocol environment is modeled on earlier work [11], [12] which used a protocol library linked to applications for data path operations and a trusted server for control, management and life-cycle issues. However, we do not enforce any particular functional partitioning between the library and daemon. Rather, the protocol environment simply provides a set of mechanisms to be used by the developer; it is up to the developer to decide how they are used. A developer is free to implement the entire protocol suite in the library by removing any restrictions on what capabilities are required to manipulate the corresponding logical device's state.

The Xok/ExOS exokernel [14] library is similar but the emphasis is on performance and implementation in an exokernel library-based operating system. We differ most from efforts focused on high-performance computing where a user-space protocol implementation leverages direct access to network interfaces and knowledge of the underlying link layer protocol to achieve low-latency high-throughput systems [13]. The virtual end-system architecture and vNet framework could not support this model since it requires the involvement of the substrate layer resident in the kernel. Of course substrate functions could also be implemented in the library but this diverges from the proposed architecture and is not supported.

2) Kernel-space protocols

Safely extending the kernel's networking subsystem requires protocol instances to be isolated such that incorrect behavior does not adversely impact unrelated processes, services or the base operating system. The Plexus networking architecture (part of the SPIN extensible operating system project) [19] relies on the type-safe programming language Modula-3 combined with link time constraints to ensure system safety. Type safety guards against improper memory access while linking restriction control access to kernel symbols (i.e. interfaces). Similarly the Open Kernel Environment (OKE) project [17] uses the type-safe language Cyclone [18] plus run-time restriction to kernel interfaces and system resources. The vNet kernel Framework is modeled after OKE but is further constrained to just interact with network specific interfaces and resources.

Rather than use a compiler to statically verify memory use, hardware or software techniques may be used. In software fault isolation (SFI) [21] instructions are inserted to dynamically verify memory accesses and jump instructions. Alternatively, hardware fault isolation in the form of processor priority levels and virtual memory techniques may be used to verify memory and interface use. We rejected using SFI or combining protection rings and virtual memory techniques for kernel extensions due to the added overhead. However, the Nooks project [10] is an interesting approach that combines the use of kernel wrappers, extension stacks and the virtual memory subsystem. It assumes the extension author is well intentioned and will not attempt to deliberately undermine system safety, security or performance. We are unwilling to make such an assumption and ultimately rejected this approach since an extension executes with kernel privileges and is able to circumvent the sandboxing mechanisms.

Another alternative is proof carrying code [22]. This was also rejected due to the complexity of generating proofs for non-trivial extensions.

In [20] the authors describe a system to dynamically upgrade network protocols using untrusted mobile code. The sandboxing of protocol extensions is compatible with OKE and our approach. The difference is they are upgrading the TCP protocol and make guarantees regarding its behavior on the network relative to other TCP flows. Our approach is to guarantee the isolation of virtual networks not the protocols that operate on a given vNet nor to replace individual protocols of a virtual network.

VI. DISCUSSION

Our desire to enable staged deployment without interrupting existing network services or placing unrealistic requirements on applications or operating systems weighed heavily on many of our decisions in developing the LAN and end-system architecture. For example, some LAN switches support topology discovery but we excluded that from consideration in order to accommodate less capable LANs.

We are actively developing this model and plan to

implement the virtual end-system architecture in the Linux and Windows XP OSes.

REFERENCES

- [1] Tom Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through Virtualization, in *Computer Magazine*. April 2005.
- [2] J. Turner and D. Taylor. Diversifying the Internet. To appear in *Proceedings of Globecom*, 2005.
- [3] J. Touch, Dynamic Internet Overlay Deployment and Management Using the X-Bone, *Computer Networks*, July 2001, pages 117-135.
- [4] L. Peterson, T. Anderson, D. Culler, and Roscoe, A Blueprint for Introducing Disruptive Technology in the Internet, in *Proceedings of HotNets-I*, Oct. 2002.
- [5] 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, pages 194-203
- [6] Linux Advanced Routing and Traffic Control, [Online], Available: <http://lartc.org/>
- [7] S. Floyd and V. Jacobson, Link-sharing and Resource Management models for Packet Networks, *IEEE/ACM Transactions on Networking*, Vol. 3, No. 4, August 1995, pages 365 - 386
- [8] Hierarchical token bucket for Linux, <http://luxik.cdi.cz/~devik/qos/htb>
- [9] Microsoft Corp., (1999, September) Quality of Service Technical White Paper, [Online]. Available: <http://www.microsoft.com/windows2000/techinfo/howitworks/communications/trafficmgmt/qosover.asp>
- [10] 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, Oct. 2003.
- [11] 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, p.554-565, Oct. 1993
- [12] 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.
- [13] 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*, p.196-205, 1995
- [14] 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, pages 49-83.
- [15] 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, pages 259-269.
- [16] D. Engler, M. F. Kaashoek, DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation, In *ACM SIGCOMM 1996*, pages 53-59.
- [17] Herbert Bos, Bart Samwel, Safe Kernel Programming in the OKE, *Proceedings of the fifth IEEE Conference on Open Architectures and Network Programming*, June 2002
- [18] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, Cyclone: A Safe Dialect of C, In *USENIX Annual Technical Conference*, Monterey, CA, June 2002, pages 275--288
- [19] Marc Fiuczynski, Brian Bershad, An Extensible Protocol Architecture for Application-Specific Networking, *Proceedings of the Winter USENIX Technical Conference*, pages 55-64, January, 1996
- [20] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, Tim Stack, Upgrading Transport Protocols using Untrusted Mobile Code, *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Pages 1-14, October 2003.
- [21] 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*, Pages 203-216, December 1993
- [22] G. Necula, Proof-carrying Code, In *Proceedings of the twenty-fourth Annual Symposium on Principles Programming Languages*, Pages 106-119, January 1997