RiaS – Overlay Topology Creation on a PlanetLab Infrastructure

Jens Lischka, Holger Karl
Paderborn Center for Parallel Computing
Paderborn University
33102 Paderborn, Germany
jeli@mail.uni-paderborn.de, holger.karl@mail.uni-paderborn.de

ABSTRACT

The PlanetLab testbed was originally built to develop new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables and query processing in a live-traffic environment. This allowed researchers to construct their own overlay network topologies on top of IP without any need for direct Layer 2 access.

While this structure is easy to use for many purposes, it does not lend itself directly to experiments with new routing protocols, which need a finer-grained control of where packets flow. Enabling such tests of new protocols and architectures on the PlanetLab infrastructure is the objective of this paper. To this end, a researcher must be able to build Layer 2 topologies upon the PlanetLab infrastructure and to have routing and forwarding protocols execute in such a defined infrastructure. Currently this is impossible due to the nature of PlanetLab's network virtualization.

This paper describes RiaS, a tool to create customized network topologies inside of PlanetLab slices. This enables researchers to evaluate and test new routing protocols on PlanetLab. We analyze the existing shortcomings of PlanetLab, identify the prerequisites to enable routing experiments, and propose our *Routing-in-a-Slice (RiaS)* system to overcome this impasse.

Categories and Subject Descriptors

C2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; C2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Network Testbeds, Overlay Networks, Experimentation, Network Testbeds, Routing

VISA 2010, September 3, 2010, New Delhi, India.

1. INTRODUCTION

PlanetLab [1] is a global research network that supports the development of new overlay network services on top of IP. This is also reflected in the way PlanetLab's network virtualization *VNET* [2] is realized. To safely share the network resources among many users PlanetLab forbids manipulations of the network stack (e.g. change routes, add new network interfaces) and direct Layer 2 access (e.g. ethernet packet sockets). This suffices for IP overlay experiments since there is no need for direct Layer 2 access but does not meet the requirements to evaluate new routing schemes.

Suppose a researcher developed a new, IP-independent routing scheme on his own private ethernet testbed. As a next step he wants to evaluate his routing scheme in a bigger, real-world environment and chooses PlanetLab. The researcher will probably do something like read incoming packets on one network interface card (NIC) (e.g. eth0) and forward them to one or more other NICs (e.g. eth1, eth2, etc.). To do so without modifying his code base he must be able to access multiple NICs on PlanetLab the same way he does on his ethernet testbed. In other words, he has to be able to setup his own customized Layer 2 topologies[1] on PlanetLab, which is currently impossible.

Our goal is to give researchers a tool that lets them create customized Layer 2 topologies on PlanetLab to run routing experiments in a huge, world-wide distributed live-traffic environment. In this paper we describe the current PlanetLab network virtualization in detail and in particular identify the problems related to routing experiments in the PlanetLab environment (Section 2). Next we propose several solutions to these problems and compare them in terms of:

1. Their deployability to the current PlanetLab kernel. Since PlanetLab is a world-wide distributed research network, changes to the PlanetLab kernel should be minimized to prevent downtime of the system and not affect its stability.
2. Their ability to grant (virtualized) Layer 2 access and to define multiple NICs.
3. Their performance; it should be possible to run multiple network experiments on the same hardware in a very efficient way.
4. Their ability to allow researchers to run routing experiments without the need to modify their source code.

[1]By Layer 2 topologies we mean that it is possible to send packets between neighboring nodes directly encapsulated in Layer 2 (e.g ethernet) frames

	Performance	Deployability	Layer 2 Access	Support existing software
Application Level	+/-	+	+/-	+/-
Full Virtualization	-	-	+	+
Container based	+	-	+	+
Multiple tun/tap interfaces	-	+/-	+	+

Table 1: Overview of Virtualization Solutions

For example, one should be able to run already existing routing software like XORP [3] or Quagga [4].

In Section 3 we propose an overlay network solution to the problem followed by some experimental results in Section 4. Section 5 concludes the paper and gives an outlook on our future work.

2. NETWORK VIRTUALIZATION

In the this section we discuss how network virtualization on PlanetLab currently works, what kind of problems arise related to routing experiments on PlanetLab, and how we can fix them.

2.1 PlanetLab network virtualization

PlanetLab is a global research network that supports the development of new network services. The PlanetLab infrastructure currently consists of 1085 nodes at 498 sites connected to the internet. In this context a site is a physical location where PlanetLab nodes are located and a node is a dedicated server that runs components of PlanetLab services. To setup experiments on the PlanetLab infrastructure a researcher allocates a *slice*, which is a set of allocated resources distributed across PlanetLab nodes. A set of allocated resources on a single PlanetLab node is called a *sliver*. Slivers are currently implemented as Linux-Vservers [5], which is a container-based virtualization approach (see Section 2.4) that allows several virtual linux hosts to run simultaneously on a single, shared kernel; no virtual host has direct access to the hardware. This concept allows to share the hardware resources of PlanetLab nodes among a large number of network experiments simultaneously in a very efficient way.

This container-based virtualization design has also a disadvantage. All virtual hosts share the same kernel and also share the same network stack. The issue is that routing experiments have to manipulate the central routing table and that

such a routing table manipulation would affect all other experiments running on this PlanetLab node since they all share the same kernel and in particular the same network stack. Suppose one wants to run his own routing software on a PlanetLab node. To do this one would have to add at least one more network interface to this node. This interface would be visible and configurable by all other slices on this PlanetLab node since it is added to the shared stack which is accessible by all slices. Clearly this is problematic for concurrently running experiments.

PlanetLab circumvents this problem by a very restrictive VServer network configuration setting. The VServers on PlanetLab nodes are configured in such a manner that the user owns no rights to change/add routing table entries or to configure/

add new NICs or tunnels. This makes it hard for researchers to build their own topologies on PlanetLab. In particular it is impossible to build Layer 2 topologies.

Currently, network virtualization on PlanetLab is done by *PlanetLab Virtualized Network Access (VNET)* [2]. VNET relies on *Connection Tracking*, which is part of Linux's *Netfilter* system [6]. VNET associates every inbound and outbound IP packet with a connection structure which ensures that slices send and receive only packets associated with connections that they own. That is, slices can only:

- Send packets associated with new connections or connections that they initiated.
- Receive packets associated with connections that they initiated or bound.

When an IP packet is sent through a socket, it passes through VNET and is associated with a new or existing connection. If the connection is not already bound to a slice, VNET allows the packet to pass through and binds the connection to the slice that sent the packet. If the connection is bound to a slice, and it is not the slice that sent the packet, the packet is dropped and an error is returned to the sending application.

When an IP packet is received by the stack, it also passes through VNET and is associated with a new or existing connection. If the packet was expected (that is, if the connection was bound by a slice or the connection was initiated by a slice) VNET allows the slice to receive the packet.

Hence, the problem of the current PlanetLab network virtualization to support the creation of Layer 2 topologies is that it forbids the creation and configuration of network interfaces by PlanetLab's VServer network settings. Thus, to create Layer 2 topologies some changes to the virtualization techniques that are currently in use are necessary. In the following, we propose possible solutions and discuss their pros and cons with respect to our needs and their deployability on PlanetLab.

2.2 Application level virtualization

Application Level or Overlay Virtualization has the great advantage that it can be adopted in PlanetLab without any modifications to the PlanetLab Kernel or VNET. Since PlanetLab consists of more than 1000 world-wide distributed nodes deployability is a key factor. But most current overlays (like X-Bone [7]) typically assume IP or a close relative as the architecture inside the overlay itself and are thus unsuitable for experimenting with new IP-independent protocols and architectures.

The big disadvantage of an overlay solution lies in its performance as we will see in Section 4. This is particularly important since the resources are shared among many concurrently running experiments.

2.3 Full virtualization

Using virtual machines (e.g. XEN [8], User Mode Linux (UML)[9]) instead of Linux Vservers would provide each slice with its own, fully virtualized network stack and enable

1. Layer 2 access,
2. creation and configuration of NICs,
3. run existing routing software/protocols without modification.

Several network virtualization projects like *PL-VINI* [10] or *VIOLIN* [11] make use of this approach. PL-VINI, for example, uses UML virtual machines to create virtual network topologies. UML allows to run a fully virtualized Linux kernel as an application within a normal Linux process. Concretely, PL-VINI runs the UML machines as user process inside the Linux VServers. The virtualization of the network stack is now done by the UML machine and the nodes are connected via UDP tunnels. This allows researchers to run existing routing software without modifications on PlanetLab nodes and provides virtualized Layer 2 access.

Using virtual machines with a fully virtualized kernel seems to be a solution to our problem. But running a virtual machine for each slice is not competitive in performance against the container-based VServer approach [12]. In addition the application of VMs on PlanetLab slices entails extensive changes to the current PlanetLab system and is therefore not easy deployable.

2.4 Container based virtualization extended by namespaces

Instead of a fully virtualized kernel it would suffice to serve each slice with its own virtualized network stack and continue to use Linux VServer for the host virtualization. Linux provides a network stack virtualization technology called Network Namespaces (NetNS) [13]. Network namespaces allow to assign a private set of network resources to one or several processes. These have their own set of network devices, IP addresses, routes, sockets, and so on. Other processes outside of the namespace cannot access these network resources.

Using network namespaces on PlanetLab would require some changes to the PlanetLab kernel to integrate the NetNS patchset and run each VServer with its own network namespace. Nevertheless this approach requires much less effort than the migration to full virtualization. In addition to that we do not lose the performance advantages of the container based concept.

Trellis [12], for example, is a network virtualization solution that makes use of NetNS by associating each VServer process with its own network namespace. This enables researchers to manipulate the network resources inside their own experiments without affecting other slices. Trellis has shown that the combination of Linux VServer with NetNS combines good performance (run many slices on the same machine, forwarding throughput performance) with our remaining requirements (Layer 2 access, creation and configuration of NIC/routing tables, run existing routing software/ protocols). Currently Trellis is running on the nodes of the VINI network which consists of around 40 physical nodes. Thus using Trellis to build huge topologies is not possible at this time.

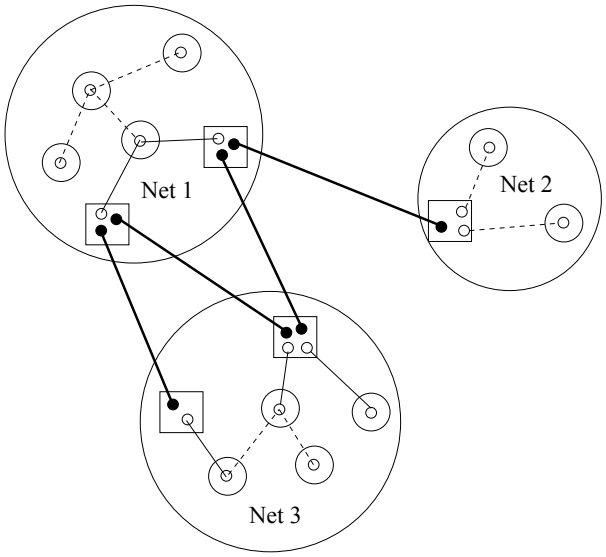

Figure 1: Sample Overlay Topology

2.5 Multiple tun/tap interfaces

tun/tap [14] provides packet reception and transmission for user space programs. It can be viewed as a simple Point-to-Point or Ethernet device, which instead of receiving packets from a physical media, receives them from a user space program and instead of sending packets via physical media writes them to the user space program.

tun/tap devices enable experimenters to implement their own custom network stack in user space. This would enable users to implement Linux packet sockets directly accessing these tun/tap devices and thus to run routing experiments on (virtualized) Layer 2. A disadvantage regarding to full and container based virtualization is that this approach does not allow the use of privileged Linux tools like ifconfig or route.

To create Layer 2 topologies with tun/tap devices we need to setup multiple tun/tap devices in PlanetLab slices and connect them via EtherIP tunneling [15]. Also tun/tap is integrated in the PlanetLab kernel, its tun/tap driver implementation currently supports only a single tun/tap device. To use multiple tun/tap one has to do some slight modifications to PlanetLab's tun/tap implementation but the effort of migration to PlanetLab is manageable.

Table 1 again summarizes the proposed virtualization approaches with respect to the four objectives performance, deployability, L2 access and their ability to support already existing routing software.

We have introduced four different virtualization techniques to fix the problems of PlanetLab related to run routing experiments. As we have seen, User space virtualization is a flexible solution to our problem since it is deployable onto PlanetLab without any modifications to the PlanetLab Kernel. Full virtualization seems to be a good approach but it is not easy to migrate to PlanetLab and it has some performance drawbacks. The idea of using multiple tun/tap devices provides users with a great flexibility for their routing experiments and would be relatively easy applicable on PlanetLab but it does not allow users to use Linux tools such as ifconfig or route. The best solution to our prob-

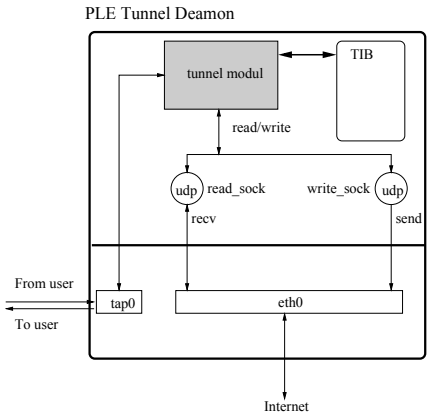

Figure 2: RiaS Tunnel Daemon on PlanetLab Node

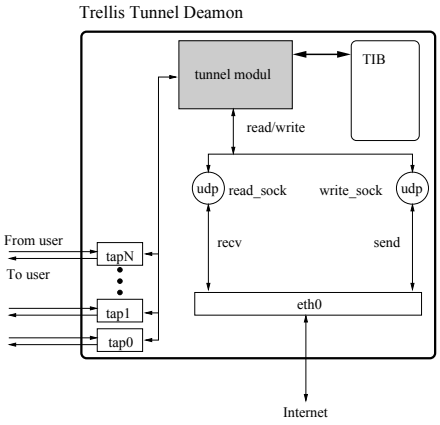

Figure 3: RiaS Tunnel Daemon on Trellis Node

lem seems to be the container based (Trellis) approach since it provides a fully virtualized network stack but still uses Linux VServers. Compared to full virtualization it is easier to migrate to PlanetLab but still entails some changes to the PlanetLab Kernel. Unfortunately there are currently only few nodes running Trellis in the VINI network available. To achieve our goal anyway we use a combination of Trellis and PlanetLab nodes which is described in the following section.

3. THE RIAS OVERLAY

The RiaS overlay topologies consist of router and host nodes. Since the router nodes need to have multiple network interfaces they are mapped onto VINI nodes that use the Trellis implementation which allows us to set up multiple network interfaces, whereas the host nodes are mapped to PlanetLab nodes.

There exist three kinds of connections:

1. Trellis to Trellis,
2. Trellis to PlanetLab, and
3. PlanetLab to PlanetLab

tunnels. The interconnection of the router (Trellis) nodes is done by the Trellis ethernet over GRE (EGRE) tunneling mechanism. Thus there is nothing more to do for us. For the remaining connections we use UDP tunneling. For the interconnection of router-and host nodes we were forced to use *UDP Hole Punching* [19], since the Trellis nodes use network address translation to communicate with the outside world.

To establish the tunnel connections, we run a tunnel daemon on each node of the overlay network. The role of the daemon is to establish Layer 2 tunnel connections which allow experimenters to send/receive Layer 2 frames to/from neighbor nodes.

In the following chapter we describe in detail how the RiaS tunnel daemons work.

3.1 RiaS tunnel daemon

The central part of the RiaS overlay network is built by the RiaS tunnel daemons. In principle these tunnel daemons build a VPN connecting PlanetLab and Trellis nodes. Each daemon consists of a *tunnel module* which is connected to a set of tun/tap interfaces on the one side, and two UDP sockets for write and read operations on the other side. Now each time a packet is put onto a tun/tap interface, it is given to the tunnel module, encapsulated inside an IP packet and sent to the internet through the write socket. The tunnel module gets the necessary address information of its neighbor nodes from the *Tunnel Information Base* (TIB). On the other side the packets are received by the read socket, decapsulated by the tunnel module and forwarded to the corresponding tun/tap devices.

Figure 4 depicts the way of an ethernet frame between two neighbor nodes in the RiaS overlay in more detail. The user application first creates a packet socket and binds it to the tun/tap interface tap0. When the application starts to send a packet it is copied from from the guest VServer $V1$ to the kernel space (1) of the host system, traverses the network stack (in case of a packet socket it is directly given to layer 2) and gets to the corresponding tun/tap interface (2). At this time the packet consist of an ethernet header with the destination MAC address of tap0 on host $H2$, src MAC address of tap0 on the local host $H1$, and the payload. Instead of calling the tansmission method of the real network card as a virtual network interface would usually do, the tun/tap interface writes the packet back to a file in user space (3). The RiaS daemon is listening on this file descriptor and adds a new IP header to each incoming packet which consists of the IP address of NIC eth0 of host $H2$ as destination address and of the IP address of the local NIC eth0 as source address. The tunnel daemon gets the necessary address information from its *Tunnel Information Base (TIB)* which was formerly configured by the topology creation process (s. Section 3.2). After encapsulation the daemon sends the packet back to the network stack (4). The packet passes the network stack again (5) and is send to the real network interface (6), which transmits it to the neighbor host over the internet (7).

On host $H1$ the packet is received by the real network interface (8) and passes the network stack (9) which emits the packet to the socket of the tunnel daemon inside of guest VServer $V2$ (10). The demon strips off the tunnel header and writes the packet to the file of the tun/tap daemon (11). This triggers the tun/tap interfaces receive function (12) which emits the (now decapsulated) ethernet frame again to the

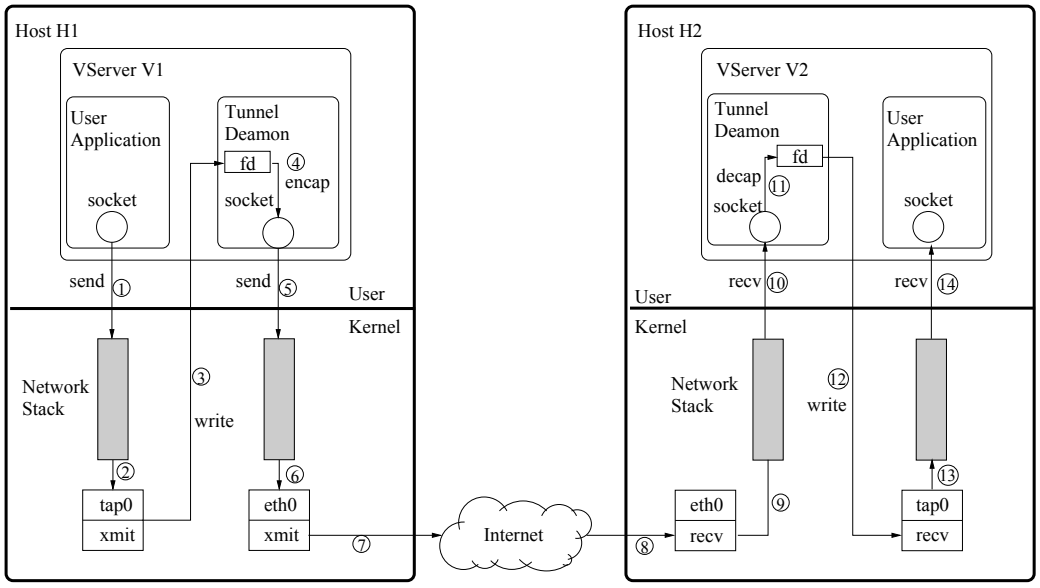

Figure 4: RiaS Tunneling Mechanism

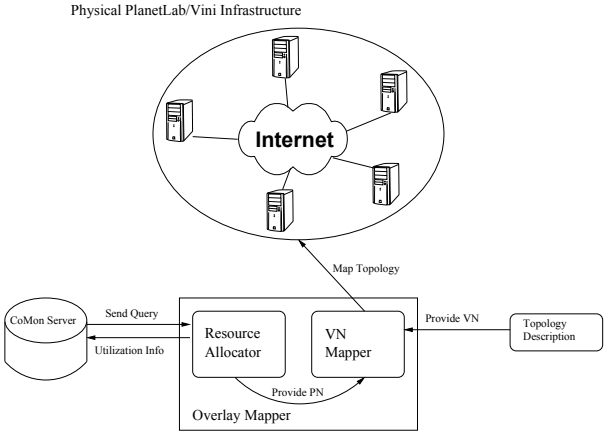

Figure 5: Topology Creation Architecture

network stack (13). As a last step the frame is delivered to the user application on the receiving side (14).

The tunnel daemons on PlanetLab and Trellis nodes differ for two reasons :

1. As already mentioned in Section 2.5 PlanetLab restricts us to use a single tun/tap interface. Thus the tunnel module of the PlanetLab tunnel daemon (s. Figure 2) listens on a single tun/tap interface.

2. Trellis nodes use NAT on the host system for the VServer guests to communicate with the internet. For our tunnels we used the hole punching algorithm proposed in [19] to traverse the NAT.

On trellis nodes we are able to setup a tun/tap interface for each neighbor node. Each tun/tap interface has a PlanetLab neighbor node as its counterpart (s. Figure 3). Based upon the information of the TIB the tunnel daemon can determine the destination address for outgoing packets, and the tun/tap interface to which the incoming packets are forwarded.

The definition of multiple tun/tap interfaces on the Trellis nodes allows researchers to route packets to different hops by choosing different tun/tap devices. On PlanetLab nodes we have to handle routing over MAC addresses since we have only one tun/tap interface. For this purpose the TIB holds a tuple of the IP of the neighbor host and the MAC address of its tun/tap device. Based on the MAC address the tunnel daemon can decide to which of its neighbors the packet will be sent. As an example consider three PlanetLab nodes $n1$, $n2$, and $n3$ connected by tunnels $(n1, n2)$ and $(n1, n3)$. If one wants to send a packet from $n1$ to $n2$ he adds the MAC address of the remote tun/tap interface of $n2$ as destination address of the packets L2 header and puts it on the local tun/tap interface of $n1$. The tunnel daemon will now lookup the corresponding IP address of $n2$ in its TIB and add it to the tunnel header.

3.2 Topology creation

Because the configuration of large overlay network topologies onto a physical network by hand is a rather tedious task we automated the process. In addition to topology configuration a researcher is also able to define a set of resource capacity constraints to the nodes of his overlay. For example if a researcher wants to run his experiment on a set of nodes with a CPU capacity of at least 100 MHz it is the role of Topology Creation to find a set of proper PlanetLab/VINI nodes. In literature this task is commonly known as Virtual Network Mapping (VNM). Our Overlay Mapper (OM) (s. Figure 5) consist of two components:

1. A Resource Allocator, and

2. a Virtual Network Mapper (VN Mapper)

component. The Resource Allocator is connected to a CoMon [16] monitoring server which provides a snapshot of the current resource utilization on the PlanetLab and VINI network. The Resource Allocator builds a Physical Network

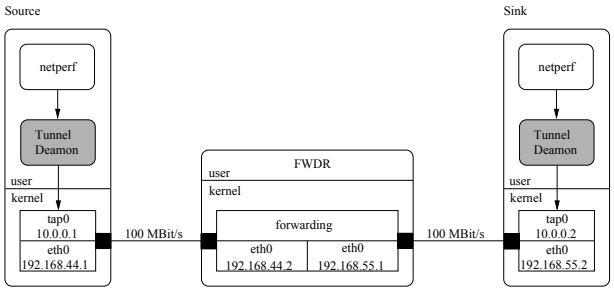

Figure 6: RiaS Tunnel Experiment A Setup

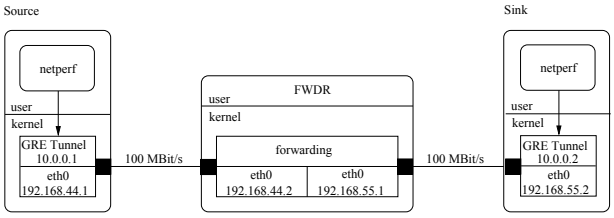

Figure 7: GRE Tunnel Experiment B Setup

(PN) description based upon these resource utilization information and hands it over to the VN Mapper. On the other side the researcher provides his topology description (VN) to the VN Mapper. The Mapper now allocates a set of physical nodes which meet the requirements of the user's description and configures the topology upon the PlanetLab/VINI infrastructure.

For the VN Mapping task We use vnmFlib [17], since it is able to handle large network topologies with any set of capacity constraints onto PlanetLab/VINI in a reasonable time.

4. EXPERIMENTAL RESULTS

The main advantage of our overlay solution which uses tunneling in user space lies in its flexibility and easy applicability on the PlanetLab and Trellis infrastructure. Its main disadvantage, however, is its bad performance. We have seen in Section 3.1 that tunneling in user space entails some additional overhead: The packet is copied back to user space by the tun/tap device, it gets an additional IP header, is copied back to kernel space and traverses the network stack a second time. In this section we investigate the impact of our tunnel mechanism on throughput and latency compared to a tunnel mechanism that works exclusively in user space.

Our benchmarks are run on the simple topologies depicted in Figure 6 and 7, consisting of three machines connected by 100 MBit ethernet links. The FWDR machine in the middle simply forwards packets from machine Source to machine Sink inside its Linux Kernel. Both Source and Sink machines are equipped with a Pentium 3 500MHz processor and 256 MByte memory, FWDR has two Pentium 4 2.6 GHz CPUs and 1 GB memory.

For the throughput measurement we used the *netperf* [18] network performance measurement tool at a confidence level of 95%.

For experiment A we run a RiaS tunnel daemon on Source and Sink machine (Figure 6). Experiment B (Figure 7)

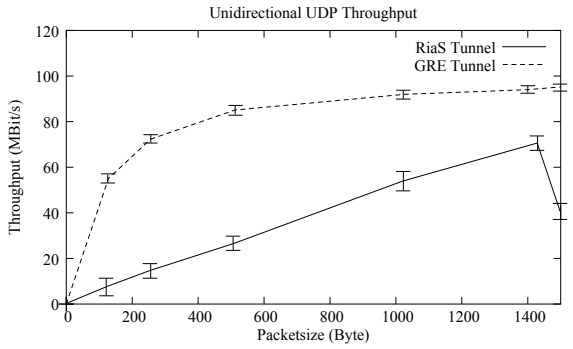

Figure 8: Unidirectional UDP Throughput

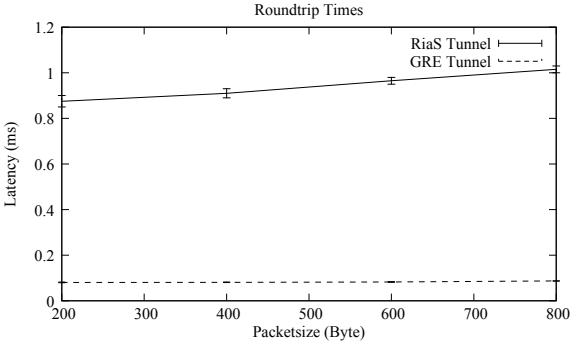

Figure 9: Latency Measurements

serves as a reference for our performance measurements and uses the Linux in-kernel GRE tunnel module.

Figure 8 shows the unidirectional UDP throughput for different packet sizes of our RiaS tunnel and the GRE in-kernel tunnel. With about 92 MBit/s the in-kernel tunnel achieved roughly the maximum throughput at a packet size of 1024 byte. Clearly our solution is not as efficient as the in-kernel tunneling but for a packet size of 1412 MBit it managed to achieve about 70% of the GRE tunnel performance. For bigger packets the performance decreases strong since our tunneling adds an extra header of 46 bytes to each packet. Together with the 42 bytes of the UDP, IP and ethernet header we get a maximum payload of 1412 bytes before we exceed the MTU of our ethernet cards. Since we want to emulate Layer 2 behavior inside our overlay we turned of IP fragmentation on our UDP tunnel. As a consequence when the packet size exceeds the MTU, the network card starts to fragment each packet into two ethernet frames of nearly equal size. This clearly leaves a lot of space in each ethernet frame unused and results in nearly 50% performance loss.

It is also striking that the throughput of our Rias tunnel decreases strongly for smaller packet sizes. For example, at a packet size of 128 byte our solution only reaches 13% of the in-kernel tunnel performance. Running *strace* on the RiaS daemon process shows that nearly 70% of the overall runtime is used by the *write* system calls whenever the packet is copied from kernel to user space and vice versa. This does not surprise since each incoming packet causes two additional write calls as we described in Section 3.1. Consequently since smaller packets entail more write operations they also entail more overhead.

To measure the latency values depicted in Figure 9 and

	400 Byte	600 Byte	800 Byte
RiaS	[0.89, 0.93]	[0.95, 0.98]	[1.00, 1.03]
GRE	[0.08, 0.082]	[0.081, 0.084]	[0.086, 0.088]

Table 2: Ping Test Results

Table 2 we used *ping -c 100 -s packetsize*. One can see that the latency of the RiaS tunnel is nearly ten times bigger than the GRE latency. The standard deviation also rose at a rate of about ten times. For example the standard deviation for a packetsize of 200 byte rose from 0.014 ms to 0.12 ms.

5. CONCLUSION

The application of routing experiments on PlanetLab requires the ability for researchers to configure their own customized network topologies. In particular one should be able to run existing routing software/protocols without the need to modify it.

To achieve this goal it is decisive to be able to setup multiple network interfaces on the PlanetLab nodes. In Chapter 2 we pointed out why this is impossible with the current network virtualization in PlanetLab and discussed several alternatives and their applicability on PlanetLab to overcome this problem.

We proposed an overlay network solution to the problem based on user space tunneling. Its advantage lies in its flexibility and easy applicability upon the PlanetLab infrastructure. But tunneling in user space also adds some additional overhead as we described in Section 3.1. To evaluate the price of this flexibility we compared our solution with in-kernel tunneling and observed that throughput is reduced at about 30% and latency is tenfold on average. Although this seems to be a big disadvantage, it can help researchers to compare different network protocols on a PlanetLab infrastructure rather than to obtain absolute performance values. In addition one should not underestimate the advantage of a solution in user space since PlanetLab is a world-wide distributed testbed of about 1000 nodes and must continue to support a large user base. For this reason changes to the PlanetLab Kernel should be avoided whenever possible.

For the future we plan to investigate more solutions to the problem which use PlanetLab nodes exclusively as for example the method described in Section 2.5. We also plan to test an in-kernel solution which uses a loadable kernel module to get better throughput and latency results while the impact on the PlanetLab kernel is still minimized.

6. ACKNOWLEDGEMENTS

We would like to thank Andy Bavier for his great support with the Trellis and VINI infrastructure. We would also like to thank Christoph Konersmann for helping us with his bash scripting skills whenever needed.

7. REFERENCES

[1] Planetlab. https://www.planet-lab.org.

[2] M. Huang. VNET: PlanetLab Virtualized Network Access. 2005.

[3] M. Handley, O. Hodson and E. Kohler. XORP: an open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33(1):53–57, 2003.

[4] GNU Quagga Project. http://www.quagga.org.

[5] Linux-VServer. http://linux-vserver.org.

[6] Linux Netfilter. http://www.netfilter.org.

[7] J.D. Touch, Y.S. Wang, V. Pingali, L. Eggert, R. Zhou, G.G. Finn. A Global X-Bone for Network Experiments. TRIDENTCOM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities.

[8] The xen.org Project http://www.xen.org.

[9] M.E. Hoskins. User-mode Linux. Linux Journal 145, 2006.

[10] A. Bavier, N. Feamster, M. H., L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. of SIGCOMM*, pages 3–14, 2006.

[11] D. Xu, X. Jiang VIOLIN: Virtual Internetworking on Overlay Infrastructure Proc. International Symposium on Parallel and Distributed Processing and Applications, 2004

[12] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, J. Rexford. Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. CoNEXT '08

[13] NetNS. http://lxc.sourceforge.net/network.php.

[14] M. Krasnyansky. Universal TUN/TAP Driver: Virtual Point-to-Point (TUN) and Ethernet (TAP) Devices, 2007. http://vtun.sourceforge.net/tun.

[15] R. Housley, S. Hollenbeck. RFC 3378 EtherIP: Tunneling Ethernet Frames in IP Datagrams, 2002. http://www.rfc-archive.org/getrfc.php?rfc=3378.

[16] K. Park, S.P. Vivek. CoMon: a mostly-scalable monitoring system for PlanetLab. SIGOPS Oper. Syst. Rev. 40(1), pages 65–74, 2006

[17] J. Lischka, H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures, pages 81–88, 2009.

[18] Netperf – Network Performance Measurement Tool http://www.netperf.org/netperf

[19] M. Holdrege, P. Srisuresh. RFC 3027 Protocol Complications with the IP Network Address Translator, Section 5.1 , 2001. http://www.rfc-archive.org/getrfc.php?rfc=3378.