Prototyping User-Oriented Addressing Model in Linux Kernel

Maoke Chen, Taoyu Li, Xing Li
Network Research Center, Tsinghua University
Room 225 Main Building, Tsinghua University, Beijing 100084 P R China
mk@cernet.edu.cn, ldy03@mails.tsinghua.edu.cn, xing@cernet.edu.cn

Nicolas Mc Guire and Qingguo Zhou
Distributed and Embedded System Lab, Lanzhou University
Tianshui Nan Road No. 222, Lanzhou 730000 P R China
der.herr@hofr.at, zhouqg@lzu.edu.cn

Abstract

The emergence of slice-based computing model has raised high demand on replacing the interface-based addressing model with user-oriented addressing (UOA) paradigm. Assigning IP addresses to users instead of interfaces or hosts help us to realize high-granularity controls on port resource utilization, identity authentication, and quality of service.

In this paper, we propose a reference prototype implementation of Linux operating system, which enables an administrator to associate each user with a certain IPv6 (or IPv4) address. Processes of a certain user can only use the user's associated addresses, to send or receive packets. The system prototype has been implemented via three major works. First, we add a data structure in the operating system to record the user-address association. Second, the socket library in the kernel is modified in order that address usage permission is checked when process invokes *connect*, *bind* or *sendmsg* operations. Finally, a minimal set of administration tools has been developed so that the host administrator can configure the user-address association according to necessity. Basic experiments have shown that the prototype is working well without significant performance degradation.

We also would like to briefly discuss some controversy about user-oriented addressing model in this paper, but only the implementation level, but also in the context of architectural design.

1 Introduction

Traditionally, an Internet address is an identifier for a node, and also plays the role of global locator for routing packets destined to that node [2]. Exactly speaking, an IP address is assigned to an interface of the node, either host or router, which is connected in the Internet. This is the basic addressing model commonly applied in both IPv4 [11] and IPv6 [3, 6]. It is well working since decades ago up to now, when everyone has one or many personal computers, having communications with other peer ends among the Internet. However, some recent approaches are steadily changing the model of computing, and raising a challenge to the traditional interface-oriented addressing model of the Internet.

These changes are mainly happening in the network application area. Since the success of SETI@Home[1], it is believed that personal computers have much more resources than one user's need and can be shared to other users. Later on, the PlanetLab[2] project makes a community whose members are sharing their own computing resources to each other, and introduces a new computing model — multiple users are sharing their own personal computers among the Internet. The change of computing raises a call for the emergence of user-oriented addressing to obsolete the legacy interface-oriented addressing

[1] http://setiathome.berkeley.edu
[2] http://www.planet-lab.org

model. First, many users may have conflicting requirements of employing well-known port numbers, when they are deploying similar experiments or services on same hosts in the shared computing infrastructure. On the other hand, in some new architectures, such as NIRA [15] or LISP [5], traffic engineering goal can be achieved via properly selecting addresses in a multi-homed environment, and therefore it is also required to separate different users' addresses to meet their traffic engineering requirements out of their different considerations and understandings to the end-to-end performance. Finally, separating source addresses of users in each shared computer is very important to tracing and identifying real sources of malicious behaviours, which are very harmful for today's network and information security.

User-oriented addressing is also attractive for the operating system communities. As the UNIX designers play the game of setting user ID to make isolation for the security reason within the OS, we employ user-oriented address to provide isolation for the network-contexted security. For safety-related systems, such kind of granular isolation is much useful. For TCP connection migration (see, e.g. [14, 12]), user-level addressing provides a convenience to differentiated load balance in the migration support group.

All of these motivate the design of the user-oriented addressing or, briefly in this paper, the UOA model and its implementation. The rest parts of the paper is organized as follows. Section 2 will briefly explain the relationship between user-oriented addressing and the virtual machine approaches. Then, in Section 3, we identify what problems has to be solved for realising the user-oriented addressing model in operating system, and then describe the design details in the following Section 4. Before the concluding remarks, we discuss several controversial problems on the architecture level of thought.

2 Related Works

Isolating user network identifiers can be achieved by virtualisation with well-known software such as Xen, VMWare, vserver etc. PlanetLab does select virtualisation as its foundation stone for the slice-based user isolation model [8, 9]. In the PlanetLab, each user is assigned with a "slice" and a "slice" is a network of potential virtual machines, each of which, called as "sliver", is hosted by a computer node among the system. Each node hosts tens or hundreds of slivers, dynamically configured according to the number of concurrently active slices. User can apply a slice for large-scale development, experiment and deployment of planetary services. PlanetLab is commonly considered as a prototype of Global Environment for Network Innovation (GENI)[3]. and accordingly virtualisation is accepted as the foundation stone of GENI [1].

Rather than virtual machine approaches, we are trying an alternative solution for user isolation. On the other hand, user-oriented addressing is also useful in virtualised computing environment, supporting more granular isolation. Therefore, we proposed this idea and a first-step design to the Real-time Linux community in 2006 [16], and this time we focus on the design principles and the details in prototype implementation with the Linux kernel.

3 Requirements Analysis

To support user-oriented addressing in a UNIX-like operating system, it is necessary 1) to add data structure(s) enumerating available address(s) of each user ID, and 2) to mandate each user can use his available address(s) only when any processes of that user make connections, start waiting for connections, transmit data to or receive data from their peers. For the compatibility issue, it is definitely required that any change in the operating system is transparent to applications, i.e. ever-exiting applications can run in a system enabled with user-oriented addressing without need of any modifications.

3.1 Constraints of address utilisation

An user, actually UID in an operating system, can use and only use those address permitted for its usage. Detailedly speaking, the term "use" contains 3 aspects of meanings: being destination, being source, and selecting address to bind. Accordingly, we have three constraints which are treated as criteria for implementation.

1. (Destination Availability Constraint, DAC) When a packet is arriving, a user's process is forbidden to *receive* it unless its destination address is available to that user.

2. (Source Availability Constraint, SAC) A user's process is forbidden to *send* packets, unless the packet's source address is available to that user.

3. (Selecting Constraint, SC) A user's process is forbidden to establish a connection or to reserve a set of resources for future connections, unless the address it uses is available to that user.

[3] http://www.geni.net/

Moreover, to the consideration of scalability, the augmentation in operating system must be light-weight, without introducing too many overhead to the system.

3.2 Possible solutions

Obviously, complete virtualisation is a thorough solution for user isolation, including the network context. However, every user having its own virtual machine is a very resource-consuming approach, and makes the system unscalable. On the other hand, what the virtual machine isolate are mostly unnecessary to be isolated for our purpose. We only need to isolate the network context of users.

A possible way of making such a dedicated isolation is using net-filter facility, such as *iptables* in the Linux system [4]. *iptables* can do the owner-match for packet sender, and packets with unmatched senders will be discarded. The SAC constraint is satisfied in this way.

An variation of using net-filter facility is combining it with policy-based routing (PBR). According to a packet's owner, we can define a tag in it and set policy rules for that tag. Because selecting source address according to routing table is very popular for IPv4 and has become standard in IPv6 [4], the SC constraint can be satisfied as well. Unfortunately, with PBR or not, the *iptables*-based solutions cannot solve the DAC problem nonetheless.

Virtual interface is a potential solution as well. A user can be assigned with a certain virtual interface and limited to only use its own interface by specifying the interface's access rights. His available address is configured with its interface. However, limiting user's access rights to network interface devices cannot avoid modifications in the kernel.

In our work, we make a very direct design, with adding user-address correspondence check into the kernel's socket API library. Our work references the code of User-Level Networking (ULN) [10], which is a user-address management system for local area networks with addresses assigned via DHCP. However, UOA is quite more flexible than ULN. It can mandate user-oriented address usage either for simple LAN or multi-homed networks, with either dynamic or manually configurations. With the kernel modifications, we can completely satisfy the DAC, SAC and SC constraints.

4 Design and Implementation

Actually we just change the address assignment in the operating system kernel, but keep unchanged that all the addresses are configured to be attached with a certain interfaces. Therefore, we don't rewrite the whole protocol stack but extend it with new data structure for storing the user-address correspondence relationship, and new or modified functions to enable the availability check when user processes employ socket API. Both IPv4 and IPv6 are involved. For an overview, we list the major changed into the Table 1 and Table 2. All the changes are included in the Linux source code tree under the `/net` subdirectory.

An alternative view of the design separates the modifications into three major categories: socket-user interface (API), socket-protocol interface and administrative tools.

4.1 Socket API modification

The basic question for the user interface modification is: where to make the change? As we have ever defined in Section 3, the modifications must not be aware by ever-existing applications. On the other hand, we don't like the modifications are implemented in many places. Therefore, we choose the socket API which every network application will call as the key point of the modifications.

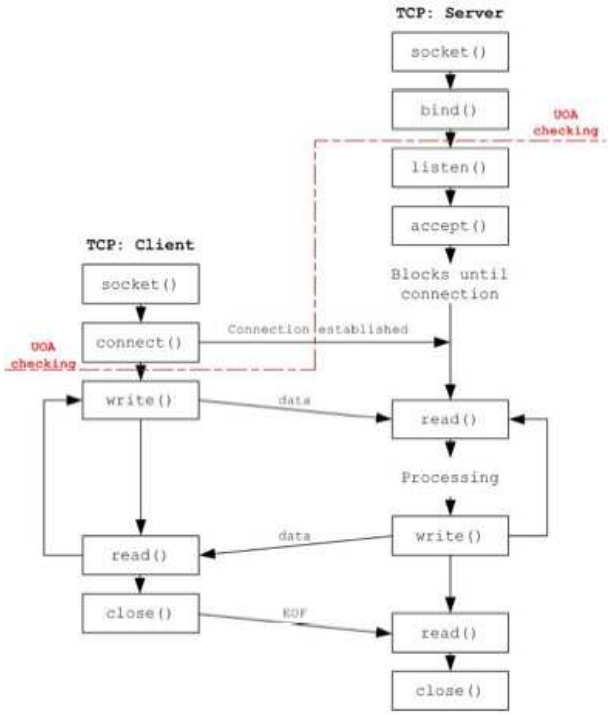

FIGURE 1: *TCP connect() and bind() do the checks.*

[4] cf. Netfilter core team, iptables project, http://www.netfilter.org/projects/iptables/

Table 1: Added Source Code Files

IPv4 src	IPv6 src	Role
ipv4/uoa_data.c	ipv6/uoa6_data.c	Data structure for user-address association
ipv4/uoa_ioctl.c	ipv6/uoa6_ioctl.c	ioctl interface
ipv4/uoa_check.c	ipv6/uoa6_check.c	Check the association data structure to get address availability for a user. Function is called by the system calls of socket API.

Table 2: Modified Source Code Files

IPv4 src	IPv6 src	Involved changes
ipv4/af_inet.c	ipv6/af_inet6.c	Common system call modifications for IP protocols
—	ipv6/autoconf.c	Auto-configured address selection in IPv6
ipv4/devinet.c	—	ioctl interface implementation
ipv4/inet_connection_sock	ipv6/inet6_connection_sock	TCP selects socket to accept packet
ipv4/inet_hashtable.c	ipv6/inet6_hashtable.c	Conflict detection for bind()
ipv4/raw.c	ipv6/raw.c	Raw socket related modifications
ipv4/tcp_ipv4.c	ipv6/tcp_ipv6.c	TCP related modifications
ipv4/udp.c	ipv6/udp6.c	UDP related modifications

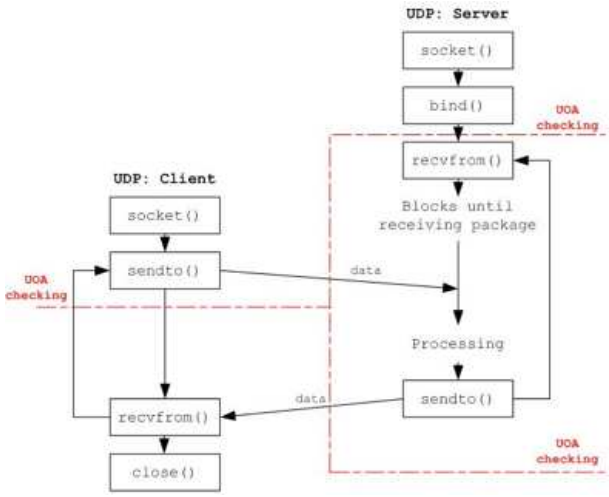

FIGURE 2: *UDP sendto() and bind() do the checks.*

Availability checking As is shown in Fig. 1, the checking code is embedded into `connect()` for TCP clients, and at `bind()` for TCP servers. This design can surely block the whole process of TCP connection, and accordingly the DAC and SAC constraints are satisfied. On the other hand, source address auto-selection is done when calling `connect()` or `bind()` without specifying a certain address [13], so this method can also solve SC problem as well.

For UDP (Fig. **??**), however, there is something different. Without a connection, address for either source or destination may various packet from packet. Therefore we must do the check where the packet is to be sent, i.e. in the `sendto()` system call. To ensure both DAC and SAC, we need to do a check at UDP `bind()` as well. Principle for raw socket modification is as same as in UDP.

Data structure of user address association The user address association is stored in a linked table. For IPv4 and IPv6, the linked table is almost same. For example, each item in the linked table for IPv4 user-address association has the structure in Fig. 3.

```
/include/net/uoa_data.h:
   struct ip4_token
   {
       struct list_head list_data;
       struct in_addr address;
       uid_t user_id;
   };
```

FIGURE 3: *Data structure for the linked table of IPv4 user-address association*

A group of functions are defined in `net/ipv4/uoa_data.c` and `net/ipv6/uoa_data.c` to manipulate the data structure with insert, update, delete and retrieval. These functions will be

called by system tools dedicatedly defined for the user-address association administration.

Communication with kernel mode The user-address association is maintained in the kernel, and therefore it must have an interface for the user modo to communicate with kernel in order to have accesses to the data structure. Two typical ways are available for this communication: `ioctl` and the `/proc` file system.

In our implementation, we add 5 new `ioctl` command descriptors into the system, as is depicted in Fig. 4. Indeed, these `ioctl` commands correspond to the data structure manipulation functions in the kernel. The user mode communicates with those functions via the `ioctl` commands. IPv6 case is quite similar and we omit the duplicated discussion.

```
/include/linux/sockios.h:
   #define SIOCAUOAMAP 0x89b0
      /* Add a UOA mapping    */
   #define SIOCDUOAMAP 0x89b1
      /* Delete a UOA mapping */
   #define SIOCCUOAMAP 0x89b2
      /* Check a UOA mapping  */
   #define SIOCUOADUSR 0x89b3
      /* Delete all UOA mapping of a user */
   #define SIOCUOADADDR    0x89b4
      /* Delete all UOA mapping of an address */

/include/net/uoa_data.h:
   struct in_ifreq_uoa {
      struct  in_addr  ifr4_addr;
      uid_t        ifr4_owner;
   };
```

FIGURE 4: *ioctl commands for kernel communication, and their common* `ioctl` *request structure.*

On the other hand, two files in the `/proc` filesystem, `/proc/uoa` and `/proc/uoa6` are available for administrator's convenience of reading current state of the associations for IPv4 and IPv6, respectively. Some typical line in `/proc/uoa` may look like as below, where the first field is the UID and the second is associated address:

```
1000: 166.111.203.101
1001: 166.111.203.102
...
```

With these user-address association data structure retrieval facilities, a series of UOA check functions are defined. TCP/UDP/raw socket system calls will call those check functions in order to check the address availability for any user processes. Due to the limit of page room, we have to remain the deep details of implementation within the source code.

4.2 Socket-protocol interaction modification

Basically, the socket API is modified with the checks for address availability and it's almost enough for our purpose. Only one big issue needs some change on lower level of the protocol stack — the support to ADDR_ANY. Traditionally, the ADDR_ANY represents any address assigned to the host, but now we need it to be associated with a certain user. We cannot limit each user have only *one* available address and make ADDR_ANY equivalent to the user's unique address, because a user at least should have access to localhost and one of the global addresses. In cases that multihoming or multicast are enabled, user having multiple addresses is a mandatory requirement for the networking facilities.

Because the user-address association linked table contains only the information of user-address relationship but ADDR_ANY actually involves the socket-address relationship, a socket's associated addresses are not possible to be known unless we know to what user the socket belongs. On the other hand, the wildcard ADDR_ANY is resolved when the the network protocol stack dispatches packets to sockets, and therefore it is necessary to change the semantics of ADDR_ANY there.

Data structure for socket-user association In order to tell the system which UID is attached to a socket, we add a data structure to the system in a seemly inverse but equivalent way — the user-socket association linked table. Each entry in the linked table contains an user ID and a pointer to the `inet_sock` structure, as is shown in Fig. 5. After a socket is created, the pointer is well defined.

```
include/net/uoax.h:
   struct sock4_token
   {
        struct list_head list_data;
        struct inet_sock * isk;
        uid_t user_id;
   };
```

FIGURE 5: *Socket-user association*

Socket-user association table manipulation The creation of new entry into the socket-user linked table is implemented as `insert_sock4_token` and `insert_sock6_token` for IPv4 and IPv6, respectively. They are embedded into the `inet_creat()` system call in the source file `net/ipv4/af_inet.c` and the `inet6_create` in `net/ipv6/af_inet6.c`.

For the table entry delete operation, `delete_sock4_by_sock` should be called in `inet_sock_destruct()` in order to prevent memory leaks.

Checking of address availability at packet dispatching When the transport protocol dispatches a packet to socket, it firstly finding in its own protocol socket list in order to get the port-matched socket. For raw socket, the operation ends up as soon as one matched socket is found; but for TCP and UDP, because a packet may have wildcards for source and/or destination addresses and/or ports, and the matched socket with least wildcards will win the patching. In the UOA enabled system, the wildcard semantics has changed, while we can just add our code for address availability checking to indicate whether the candidate socket's owner can use the address bound with the socket or not.

Address conflict detection in `bind()` In the legacy operating system, once a socket tries to bind with a port, it will detect whether the port of that address has been occupied by another socket. For ADDR_ANY, however, any duplicated employment of an ever-occupied port is not allowed. Now the semantics of ADDR_ANY has changed. It doesn't mean any address associated with the host but with a certain user. Accordingly the conflict detection behaviour should be changed as well.

For IPv4 TCP, the conflict detection is executed by `inet_csk_bind_conflict()`, while for UDP it is executed by `ipv4_rcv_saddr_equal()`, and for raw socket the conflict detection is not done at all. For IPv6 however, the conflict detection is performed by `ipv6_rcv_saddr_equal()` uniformly for both TCP and UDP. Therefore the conflict detection code should be inserted into these functions.

It is necessary to have a view on the code for the conflict checking. Fig. 6 shows the case for IPv4. In legacy system, the word "conflict" means two sockets have same address to be bound or one of them holds the address wildcard. Now, one socket holds an wildcard but another still can has an wildcard because the wildcard ADDR_ANY becomes user-associated. The code helps us to understand the "bind conflict" in a UOA-enabled system, i.e. bind conflict happens when either 1) two sockets try to bind same specific address, or 2) both belong to a same user and both try to bind with wildcard address, or 3) one of them holds an wildcard and the holder is associated with the other socket's address.

/net/ipv4/uoax.c:

```
int check_addr_conflict_uoa(struct inet_sock *isk1 ,
    struct inet_sock *isk2)
{
    uid_t uid1,uid2;
    if (isk1->rcv_saddr && isk2->rcv_saddr)
        return (isk1->rcv_saddr == isk2 ->rcv_saddr);
    if (!isk1->rcv_saddr && isk2->rcv_saddr)
    {
        struct in_addr tempaddr;
        tempaddr.s_addr=isk2->rcv_saddr;
        return (check_sock_with_addr(isk1,&tempaddr));
    }
    if (!isk2->rcv_saddr && isk1->rcv_saddr)
    {
        struct in_addr tempaddr;
        tempaddr.s_addr=isk1->rcv_saddr;
        return (check_sock_with_addr(isk2,&tempaddr));
    }
    uid1 = find_sock4_token_uidbysock(isk1);
    uid2 = find_sock4_token_uidbysock(isk2);
    if (uid1 == 0 || uid2 == 0) return 1;
    return check_2user_share(uid1, uid2);
}
```

FIGURE 6: *Conflict checking for socket bind (IPv4)*

For IPv6, the things are a little complicated. The key point is, for IPv4-mapped IPv6 address type, the IPv6 detection function must call that one defined for IPv4.

After changing the semantics of ADDR_ANY, it is not necessary any more to mandate that only root is privileged to use a port number below 1024.

4.3 Administrative tools

Well, all the system is almost done but we still need some tools to manage the user-address association. For administrative tools are defined: `uoamap`, `uoamap6`, `uoatool` and `uoaauto`.

The first two calls `ioctl` commands to add, change, delete, check or find associated user for a certain address. `uoatool` is an encapsulation for both `uoamap` and `uoamap6` to make the administration easier and simultaneously support IPv4 and IPv6.

`uoaauto` then reads the configuration file `/etc/uoalist` and invokes `uoatool` automatically to instantiate the configuration. An `/etc/uoalist` file maybe looks like that:

```
166.111.132.202 uoa
2001:da8:200:9002:250:4ff:fe98:6291 uoa
166.111.132.201 uoatest
...
```

5 Experiment

Here we briefly introduce some results from our experiment on the UOA-enabled Linux kernel. The environment for the test is a Linux computer patched

with UOA codes. The kernel version is 2.6.20.4. Two users, `uoa` (uid = 1000) and `uoatest` (uid = 1001) have been established on the host. The system has the file `/etc/uoalist` just like the sample shown at the end of Section 4.

With `uoaauto` tool, the system has all the user-address association configured.

```
root@nmstest:/home/uoa/bin# uoaauto
```

Then we can observe the results by check the files in `/proc`:

```
root@nmstest:/home/uoa/bin# cat /proc/uoa
1001 : 166.111.132.201
1000 : 166.111.132.202

root@nmstest:/home/uoa/bin# cat /proc/uoa6
1001 : 2001:0da8:0200:9002:3b1c:0000:0000:7305
1000 : 2001:0da8:0200:9002:3b1c:0000:0000:7309
```

Current the user `uoa` can have the access to any other site, e.g.

```
uoa@nmstest:~$ ping 59.66.122.66
PING 59.66.122.66 (59.66.122.66) 56(84) \
  bytes of data.
64 bytes from 59.66.122.66: icmp_seq=1 \
  ttl=60 time=1.04 ms
64 bytes from 59.66.122.66: icmp_seq=2 \
  ttl=60 time=0.435 ms
64 bytes from 59.66.122.66: icmp_seq=3 \
  ttl=60 time=0.422 ms

--- 59.66.122.66 ping statistics ---
3 packets transmitted, 3 received, 0% \
  packet loss, time 1999ms
```

Now we delete the address association for `uoa` but keep that for `uoatest` unchanged, and then we'll see that the network is not accessible for the certain user.

```
root@nmstest:/home/uoa/bin# uoatool \
  166.111.132.202 uoa del
    Delete success.
    Totally 1 of mappings between address \
  166.111.132.202 of user uoa(uid=1000) \
  deleted.

uoa@nmstest:~$ ping 59.66.122.66
connect: Resource temporarily unavailable
```

Due to the limited room of the paper, further experiments for TCP/UDP communications are omitted. Generally, applications can be running over UOA-enabled system without perceivable performance degradation.

6 Discussions

Somethings are still in open debate. An interesting problem of the implementation level is: can one transport layer protocol bypass the UOA checking mechanism? In practice, the possibility is very small, because almost all present applications are running over TCP/IP sockets. However, it is definitely possible in theory. The current implementation cannot prevent address misuse with an alien or new transport layer protocol, such as SCTP [7].

Another interested debate is of the architectural design. Considering that every user has its own address(es), it is reasonable to thinking of giving every user a route table. Sometimes it is necessary, esp. when user address does not only play the role of identifier for peer entity but also locator for routing, e.g. in the NIRA architecture [15]. However, user-oriented routing table may significantly increase the overhead of the system. One can have a doubt: if routing is also isolated for users, why not all the resources? Then the UOA would become to another virtual machine approach at all.

7 Conclusions

The innovation in computing model is calling the change of addressing model for networking. In this paper, we've comprehensively discussed the issue related to the addressing model for future Internet and future operating system, from the motivation, requirements to the details of the design and implementation of the user-oriented addressing model.

Experiments have shown that UOA can provide a granular isolation for user networking context with neither significant overhead nor application adoption. Ever-exiting applications can well run in a UOA-enabled operating system without any trouble. This makes the UOA design and implementation very attractive to both networking and operating system community.

For the computing in real-time systems, UOA provides a new way to achieve differentiated service oriented to users or, more precisely, to certain processes or threads which are identified with user IDs. The future work of this paper will involve this area, and combining user-oriented quality of service control with the UOA architecture.

References

[1] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. Technical Report GENI

Design Document GDD-05-01, GENI Planning Group, Apr. 2005.

[2] B. Carpenter, J. Crowcroft, and Y. Rekhter. IPv4 Address Behaviour Today. RFC 2101 (Informational), Feb. 1997.

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), Dec. 1998.

[4] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484 (Proposed Standard), Feb. 2003.

[5] D. Farinacci, V. Fuller, D. Oran, and D. Mayer. Locator/ID Separation Protocol (LISP). draft-farinacci-lisp-03.txt (work in progress), Aug. 2007.

[6] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), Feb. 2006.

[7] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May 2002.

[8] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets–I)*, Princeton, New Jersey, October 2002.

[9] L. Peterson, S. Muir, T. Roscoe, and A. Klingaman. PlanetLab Architecture: An Overview. Technical Report PDN–06–031, PlanetLab Consortium, May 2006.

[10] A. Pira, E. Tassi, and R. Davoli. User level networking-personal ip: assigning each user his/her own ip addresses in multiuser operating systems. In *ICN'04: Proceedings of the 3rd International Conference on Networking*, Feb 29 - Mar 4 2004.

[11] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[12] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 19–19, Berkeley, CA, USA, 2001. USENIX Association.

[13] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[14] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Connection migration for service continuity in the internet. *icdcs*, 00:469, 2002.

[15] X. Yang. Nira: a new internet routing architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 301–312, New York, NY, USA, 2003. ACM Press.

[16] D. Zhou, T. Li, M. Chen, and X. Li. Operating system modifications for user-oriented addressing model. In *Proceedings of 8th Real-time Linux Workshop (RTLWS 2006)*, Lanzhou, China, 2006.