Mobility Support for Networked Applications built in the TCP/IP Stack

YaoShuang Wang

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-41

Mobility Support for Networked Applications built in the TCP/IP Stack

YaoShuang Wang

Supervisor: Yuri Ismailov, Ericsson AB,
Kista, Sweden

Examiner: Associate Professor
Vladimir Vlassov, ECS/ICT/KTH

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-41

Abstract

This work aims at providing mobility support inside TCP/IP stack for networking applications. Such approach allows true end-to-end mobility as opposed to usage of proxy servers and/or various types of forwarding agents. We propose an addition to the TCP/IP network stack to make mobility to be the natural feature of the stack, i.e. providing support for required changes dependently on the reaction needed from a system on various dynamic changes caused by user and/or device behavior. The implementation done in Linux kernel, conceptually provides that the simplified architecture applicable for “any” mobility scenario.

Acknowledgments

The author wishes especially to thank Dr Yuri Ismailov, Professor Vladimir Vlassov for their supervision and technical contributions to this thesis.
Thanks also to Petter Arvidsson and Micael Widell for their contributions.

Table of Contents

Table of Figures

1 Introduction

Initially, the Internet was designed with some particular set of requirements in mind. The requirements were affected by the fact that at that time computers were desktops and probably had only one network interface. Further on, with the introduction of mobile networks, mobility support viewed as the mobile objects – devices may change their point of attachment to a different IP network. The focus of mobility support is on single event (change of IP address) for a single interface. With the growing popularity of the Internet, requirements set by Internet applications and users broadened the mobile objects boundary. Besides network devices, the mobile objects on internet could be the content on a mobile hard disk, an ongoing conversation – a session, a service running on certain system, e.g. file transfer service. The mobile events could be triggered in different ways. It could be a wireless card changing the associated Access Point (AP), it could be the change of IP address of a desktop computer, it could also be a user requests to move a conversation from an Ethernet interface to a, for example, Bluetooth interface. If we look into the TCP/IP stack, from application layer down to network device interface, dynamic changes, caused by a variety of mobility events can happen at any layer. Even inside one layer, the changes could be diverse. The possible mobile objects are explained more in details in [1]. The old definition of mobile objects based on network interfaces is not enough to cover the dynamic events happening on internet today, it needs to be reconsidered. A proper definition of a mobile node will directly influence the mobility solutions based on it. In our project, we aim at provide mobility for application running on top of TCP/IP stack. Besides the reconsideration of definition of mobile objects, corresponding name resolution service is needed to be able to resolve the dynamic mobile object names to current addresses. There are different research interests on naming proposals, e.g. [2], [3] and [4].

We are designing and implementing a support for heterogeneous socket operations providing mobility management at a session layer, and downwards along the TCP/IP stack. Heterogeneous socket operations mean that the operations are still based on the client-server module but they provide some different functions from the traditional BSD socket operations. In our project, we are considering how could a single name resolution service be inclusive to handle the diverse mobile objects and flexible enough to react to the dynamic mobile events. The definition of targeted mobile objects together with correspondent naming methods will directly influence the mobility range. In our implementation, we take name resolution into

account but we focus more on providing mobility support for network applications inside the TCP/IP stack. Introduced in the report mobility management scheme is based on a broader view, which is based on the events generated according to the dynamic behavior of different types of mobile objects. Events examples are: interfaces going up and down; changing network addresses; switching between protocols; and even changing port numbers in the case another application instance has to take over ongoing communication. With the increased use of different types of network devices, especially for computers connected to the Internet, it provides the possibility of simultaneously connecting between two computers via many different ways. We implemented a heterogeneous socket interface for application to start connection with any of the possible simultaneous ways. The proposed approach provides intelligent choices of interfaces and configuration for establishment of communication according to the peer communicating host's configuration. It liberates the application from statically binding to source address, destination address, source port and destination port at the start connection time. After the connection is set up, during the communication time, mobile events could be triggered in many different ways. These events may affect any of the layers in between of application and network interfaces as shown in *Figure 1*.

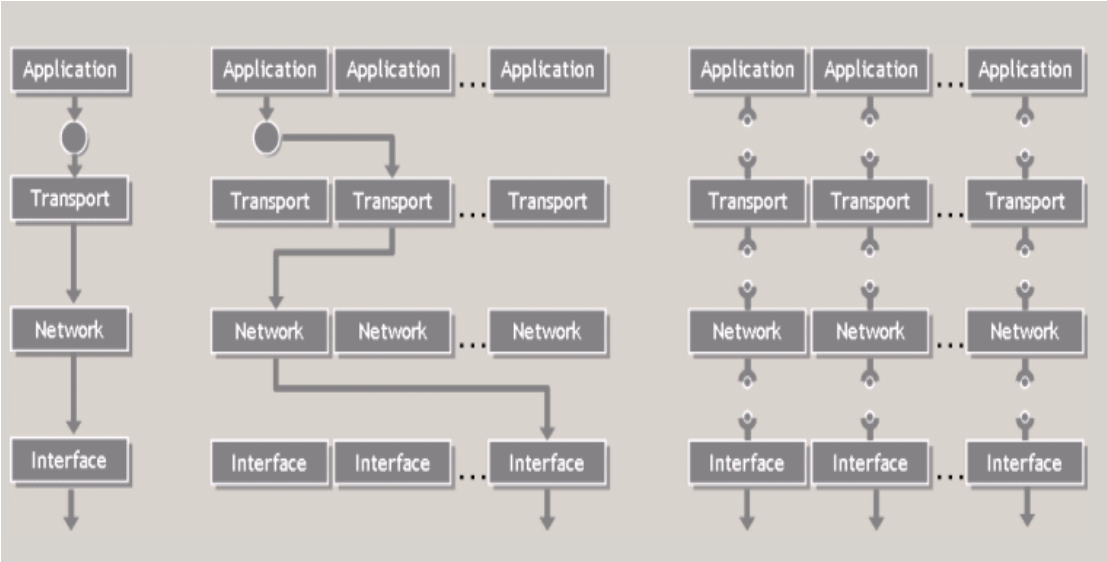

Figure 1 Dynamic change in the protocol stack

The traditional stack design is like the first column in *Figure 1*. For the application, after it is connected, it is statically bound from the top of the protocol stack to the very bottom of a single interface. This means that any event happens anywhere of the whole stack will break the current connection. But if we think more in the reality, we will find this static binding is not as simple as shown in the first column. It is more close to the picture at column two in

Figure (1). Application might use different transport layer protocols (TCP/UDP/SCTP), different network layer protocols (IPv4/IPv6) and different device interfaces (wireless/wired/Bluetooth). Any change, in the diverse binding between application and underlying protocol stack, will break current connection. Which means the application need to re-establish the connection and take care of the data consistency and be aware of the network situation. This means that applications even need to pay attention to the interface change. This is not efficient from application point of view. Here comes the need of revising the protocol stack design. How would a protocol stack be designed in a way to be compatible with all possible dynamic events happening in any place of the stack? This means that the application running on top such a protocol stack will not break when any dynamic change happens inside the stack. Besides being compatible with dynamic events, the protocol stack should be able to allow application to switch to use different protocols/devices in the same layer, like from IPv4 to IPv6 or from TCP to UDP or from one interface to the other. The ideal design of the protocol stack is shown at the third column in *Figure 1*. Inside this design module, the multiplexing point could be between any two layers. It is capable of react to any dynamic events inside the stack. It will allow application to rebind among different transport protocols. For the same transport protocol, the new protocol stack allows it to rebind among different network layer protocols. The new protocol stack will also allow application to rebind to different addresses or port numbers inside the same network protocols.

The goal of this project is to provide mobility support for network applications from inside the TCP/IP stack. The desired protocol stack will be able to support dynamic network changes by itself without need of external proxy servers or relay agents. The task of this thesis works focus on two parts. The first part is to free application from binding to specific address or port at the starting connection time. The second part is to make socket able to rebind to different address or port number. The evaluation of the socket rebind is to see if the socket could go on with send/receive message after rebinding.

1.1 Organization

The next chapter describes background and related works. Chapter 3 discusses our design choice and shows our implementation, how our heterogeneous socket is designed and how to rebind socket. Chapter 4 shows some test configuration and verification. Chapter 5 presents our conclusions. Chapter 6 discusses future work.

2 Background and related works

2.1 Background

Mobility support has been an interesting topic for many years. Many academic and industrial research groups are active in this area. The most popular solutions approach the mobility problem from the network layer. These solutions are designed to provide a seamless mobile communication service to transport and application layer entities, hence the basic requirement is to keep the IP address constant to the upper layers. Traditional view on mobile objects focuses on mobile devices like, for example, Mobile IP solution [14] , it focuses on solving the mobility problem at the network layer, using home IP address to forward packets sent to the mobile node. Mobility problem here means the problems caused by computers or devices moving around. However, the notion of mobility and "mobile node" has drastically changed because of the new features and functions evolving according to the requirements of applications and users.

The first observation is that multi-homing (multi-access) splits a singular presentation of a mobile node into a set of interfaces, each of which can act and be mobile independently of the others. Multi-homing (multi-access) means a node/host which has more than one network interfaces or points of access. Multi-homing creates the need to apply mobility management to the communication session itself, on each interface, rather than to the mobile node as a whole. Consequently, additional functions ought to be configured at least per interface rather than at a host level. The ultimate desire is of course to do this configuration for each application separately and according to each application's requirements.

Another observation is that mobility can occur without the change of an IP address on an interface or a host. Rather the IP address for the actual communication session changes. As an example, an application may wish to move a particular session between existing interfaces on a multi-homed node, whereas none of these interfaces changes its IP address. In this case, the initiator of mobility is an application rather than a networking event.

Apart from applications and the network, a user can trigger mobility as well. Assume that a person who wants to move from his office location to the lab with his/her laptop. The lab network and the network in the office belong to the same IP subnet. Moreover, the DHCP server recognizes MAC addresses and always allocates the same IP address for this laptop. There are a number of ongoing sessions the user started prior to the decision to move. Intuitively, the user's desire is to be able to suspend all sessions on the particular interface,

which will not have any access during the move. Moving may take quite a long time, discussions with colleagues on the way, having coffee, etc. After arriving to the lab, the mobile node, from an IP mobility point of view, will not change its point of attachment. However, mobility management in terms of suspending and resuming sessions or even more generally – session management, through some interface is required. This example shows "temporal" mobility as opposed to "spatial" mobility.

Various mobility events that may occur and their possible sources are depicted in *Figure 2*. These observations lead to the important conclusion that the "mobile node" is no longer of primary interest for mobility support, i.e. mobility can take place within the node without any changes occurring at the node level. The subject of mobility has to be revised, and must have a finer granularity than what we have today. The second important conclusion is that the number of events, which may lead to activation of mobility support, is much broader than just a change of IP address. The interpretation of "mobility" thus needs to be greatly enriched. However, at present, there is no mobility event dispatching mechanism in the system allowing correct processing of those events. Based on the above observations and conclusions we generally define mobility management as an expected system reaction on various mobility events.

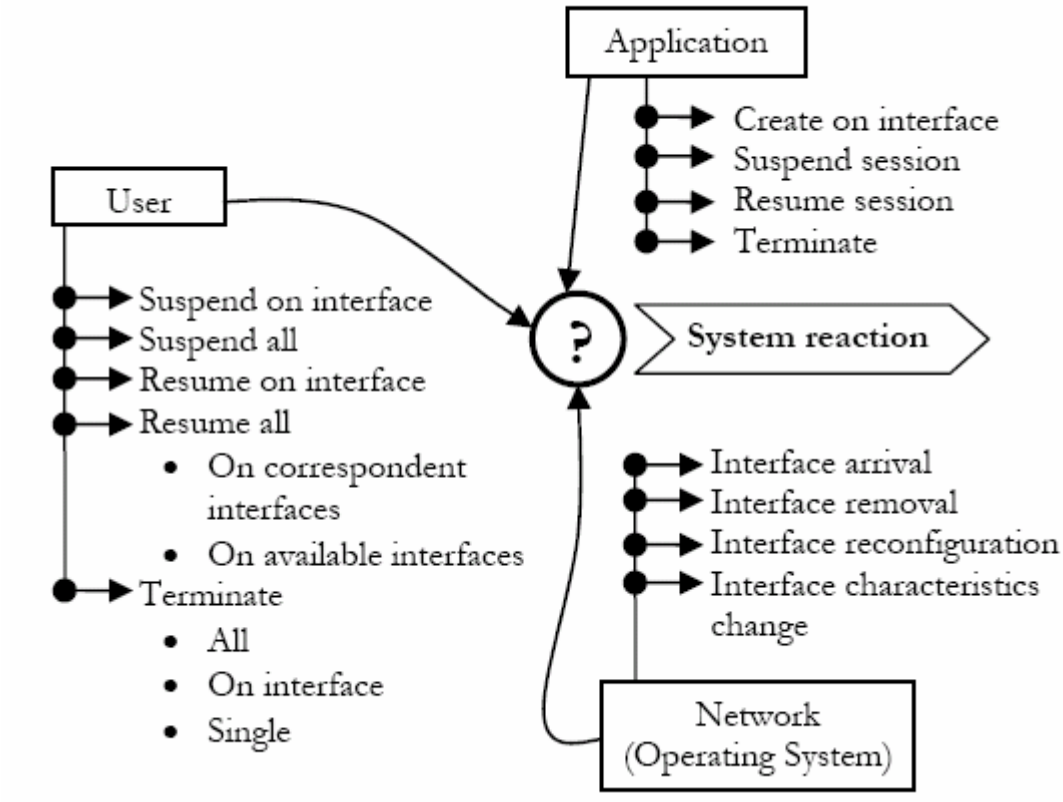

Figure 2 Mobility Events and Sources.

There are some similar work like [5] and [8] “A Mobile TCP Socket”/MSOCKS+ and [6] “Reconsidering Internet Mobility” by Alex C. Snoeren, [7] “Integrating Security, Mobility, and Multi-homing in a HIP Way” by Pekka Nikander. However, unlike these works, our design of the socket is no longer contains static bindings such as binding to the pair of IP address and port. Instead, it will dynamically bind to an entirely independent of underlying technology identifiers, which do not change during the communication lifetime. Our vision is to have a dynamic, fault-tolerant, delay-tolerant and disconnection tolerant socket.

There are two approaches presently, one focuses on network layer and tries to keep one virtual/permanent IP (Home Address) and Mobile Host will dynamically update with Home Agent with the current Care-Of-Address. The Home Agent will then route the packets destined to the mobile host to the real local address of the mobile host. This model requires special hardware to support Mobile IP. Different implementations differ more or less on hardware requirements and performance; they all consider mobile objects as mobile devices. The other approach puts emphasis on top of existing BSD sockets, either by implementing individual proxy server to relay network traffics, or by using cache server, which caches the network traffics, inside of socket implementation to provide TCP fault tolerance. This approach has a bit broader definition of Mobile Objects. It provides partly mobility support for network applications. For this project, we have chosen an approach for mobility management at high layers, which according to the analysis presented below provides high flexibility in mobility management and high degree of applications tolerance to various mobility related events.

2.2 Analysis of related work

In this section, we will go through related works with the following 4 criteria: What are the targeted mobile objects? How is naming resolution done? How much mobility is achieved? Does it support disconnect tolerance (suspend/resume)?

2.2.1 A Mobile TCP socket [8]

As stated in [8], “In our mobile TCP socket, a mobile mapping is introduced, which maps TCP associations to underlying TCP connections. The mobile mapping can be

implemented in the socket layer and on top of TCP/IP protocol layers." It implements TCP fault tolerance by providing TCP association and caching TCP packets together with Virtual Port Protocol. During the real TCP connection down time, the Virtual association will cache the ongoing TCP communications.

In this proposal, the targeted mobile object is a TCP connection. The name resolution of the connection is done by a 5-tuple [9]: (protocol, local-IP-address, local-port-number, remote-IP-address, remote-port-number). As stated in [8] "The mapping is from a TCP association (TCP, home-IPA, PortA, home-IPB, PortB) to a TCP connection {current-IPA, PortA, current-IPB, PortB}" happens every time the mobile node moves. Virtual Port Protocol and virtual identifier (V*id*) are employed to distinguish between the old connection and new connection. Virtual port is an additional port between socket and TCP port; it acts as a TCP port from the viewpoint of a socket.

This proposal provides partial mobility for a connection between any client and server by caching TCP packets in the socket layer. This is limited by the buffer size in the socket layer. The following figure is also from paper [8], it shows the relation between a association and a connection. This solution supports limited disconnect tolerance according to the server caching capacity.

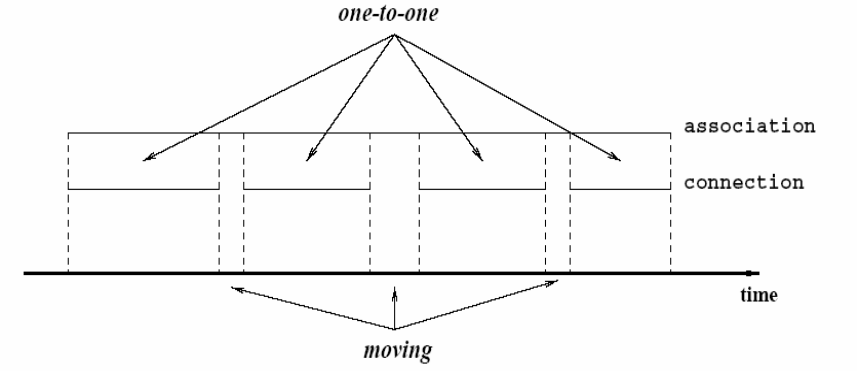

Figure 3 Mapping between connection and association [8]

2.2.2 MSOCKS+: An architecture for Transport Layer Mobility [5]

As stated in [5], "the MSOCKS+ architecture consists of three pieces: a user level *MSOCKS proxy* process running on a proxy machine; an in-kernel modification on the proxy machine to provide the *TCP Splice service*; and a shim *MSOCKS library* that runs under the application on the mobile node." This solution provides mobility between mobile node and static server by using the *MSOCKS proxy server*. The mobile object here is also a connection. The two endpoints of the connection must be a mobile client and a static server.
The name resolution is done by using a connection identifier of the logical sessions between

the mobile node and the proxy server. As stated in [5], “The MSOCKS proxy issues a new connection identifier every time a mobile node makes a BIND or CONNECT request to the MSOCKS proxy asking to be connected to a correspondent host. The connection identifier is sent to the mobile node along with the normal SOCKS reply message that indicates the success of the request. When the MSOCKS library wants to change the address or network interface that a TCP connection uses to communicate with the MSOCKS proxy, it simply opens a new connection to the proxy and sends a RECONNECT message specifying the connection identifier of the original connection.”

It provides connection based mobility between a mobile node and a static server. This proxy based mobility solution provides real-time roaming within Enterprise network territory with Packet Forward Latency. This solution provides disconnect tolerance.

2.2.3 Host Identity Protocol, HIP [6]

HIP introduces a new crypto-graphic name space and protocol layer between network and transport layer. It tried to solve two problems originally: Fake address updating (man in the middle) and Denial-of-service. Fake address updating could be decreased with reachable checking in certain degree.

The mobile object in HIP is a mobile host, which is Host ID, the network layer representation of the mobile host. The HLP/HIP approach provides new end-point names that *are* public keys. This secure solution is to provide a credential infrastructure binding addresses to public keys, thereby creating the possibility of binding nodes and addresses in a stronger sense with the use of IPSec Association. The transport layer sockets are bound to Host Identifiers instead of IP address. Additionally, since the communication context is bound to the end-point identifiers instead of IP addresses, the architecture also makes it easier to support several routing realms and to establish state with any node in the network.

This solution is compatible with any network layer mobile events. As the binding between Host ID and IP address is dynamic, the solution is capable of handling end-host multi-homing and mobility. This solution doesn’t provide fault-tolerance.

The following two figures are from [6]. The first one is an explanation of HIP architecture. The second one shows the difference between HIP architecture and normal BSD socket architecture. Such infrastructure will put high requirements on network layer. It is not very practical due to performance and implementation reasons. It does not support connection suspend/resume.

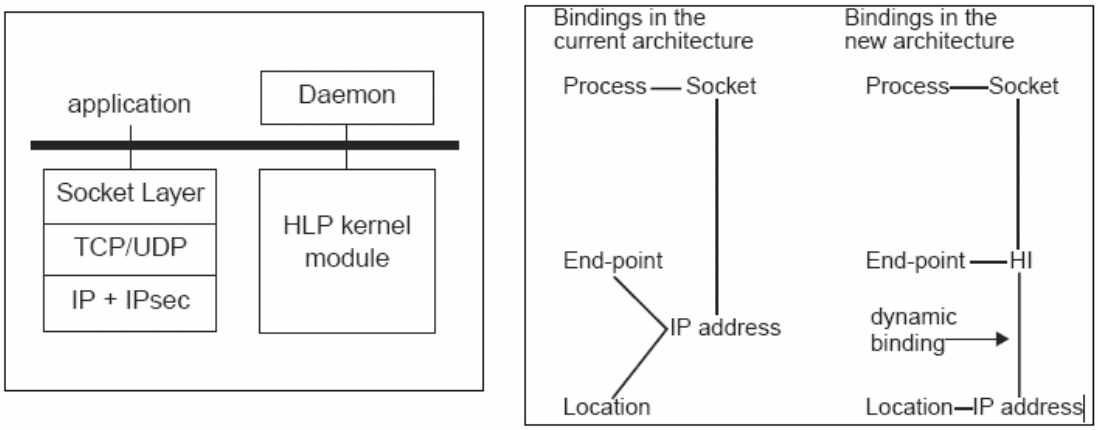

Figure 4 Binding between different architectures and HIP architecture [6]

2.2.4 Reconsidering Internet Mobility [7]

This proposal aims at "a comprehensive system architecture that efficiently addresses the needs of mobile applications. This is implemented with the Migrate approach to mobility, which leverages application naming services and informed transport protocols to provide robust, low-overhead communication between application end points. A session layer protocol handles both changes in network attachment point and disconnection in a seamless fashion, but is flexible enough to allow a wide variety of applications to maintain sufficient control for their needs." [7]

The mobile object here is mobile application. It engages in five fundamental problems: locating, preserving, disconnecting, hibernating and reconnecting of network communication. Name resolution is done by Session ID. Once established, a session is identified by a locally unique token, or Session ID, and serves as the system entity for integrated accounting and management. The session layer exports a unified session abstraction to the application. This work support full mobility for application. It supports disconnect tolerance.
Session layer additional functions: migrating session state between the system and application, and providing contextual validation of session state.

2.2.5 Internet Indirection Infrastructure (*i3)* [10]

[10] "*i3* attempts to generalize the Internet's point-to-point communication abstraction to provide services like multicast, any-cast, and mobility have faced challenging technical problems and deployment barriers. To ease the deployment of such services, *i3* has proposed an

overlay-based Internet Indirection Infrastructure (*i3*) that offers a rendezvous-based communication abstraction. Instead of explicitly sending a packet to a destination, each packet is associated with an identifier; this identifier is then used by the receiver to obtain delivery of the packet". *Indirection* that decouples the sending hosts from the receiving hosts. The purpose of *i3* is to provide indirection; that is, it decouples the act of sending from the act of receiving. The *i3* service model is simple: sources send packets to a logical *identifier*, and receivers express interest in packets subscribe to the identifier. Delivery is best effort like in today's Internet, with no guarantees about packet delivery.

The mobile object in this proposal is communication between two peers. It provides inborn support for mobility. Name resolution is done by a logical identifier. When a host changes its address, the host will update its trigger. It is not reliable but supports disconnect tolerance.

2.2.6 Mobile IP

Mobile IP is a mechanism to maintain transparent network connectivity for various mobile applications and equipments. Mobile IP supports mobility from network layer by using Home IP as a relative stable name. Every time when the mobile node's current address changes, it will update with the Home Agent of its new care-of-address. All packets sent to mobile node's home address will be forwarded to the mobile node's current address by home agent.

The mobile object here is the IP address of mobile node. The name resolution is done by translating the Home IP to current Care-of-Address. The mobility range in this solution is with in network layer. It doesn't support disconnection tolerance.

2.2.7 Comparison of different works

Compared to all the related works, our solution shares the most common with "Reconsidering Internet Mobility" [7]. But [7] only support application to move to different address in the same protocol family. We aim at providing full mobility support for mobile applications. Which means application could move to different addresses, different port numbers and even different address families. The following table presents the comparison of different related works according to the 4 criteria presented at the beginning of this chapter.

	Mobile objects	**Naming resolution**	**Mobility reached**	**Disconnection tolerant?**
A mobile TCP socket [8]	A TCP connection	*Vid* with A 5-tuple of protocol/address/port	Partial mobility for a TCP connection.	Yes / limited disconnection tolerance
MSOCKS+ [5]	A connection	Connection identifier	Mobile node VS static server	Yes
HIP [6]	A mobile host	Host ID	Network layer mobility	NO
Reconsidering Internet Mobility [7]	Mobile applications	Session ID	Full mobility for mobile applications	Yes
i3 [10]	A communication	Logical identifier	Not reliable	Yes
Mobile IP	IP address of mobile node	Mapping home IP to Care-of-Address	Within Network layer	No

Figure 5 Comparison between different related works

3 Design choice and implementation

3.1. Design Choice:

We share the similar design goal as [7], a comprehensive system architecture which will eliminate the static bindings between any layers inside the TCP/IP stack. This architecture will allow the upper layer protocol dynamically rebind with different lower layer protocols. Inside the same protocol, it also allows rebind to different configuration, like inside network layer to rebind to different address or port. Such system architecture will provide fault tolerance and disconnection tolerance. It will handle disconnections gracefully, by clearing according incoming/outgoing buffer and keep the current connection state. It also supports connection suspension and resumption. We achieve this goal by three steps. The first is to abstract the start connection from specifying destination address and port. Application will start a connection base on the remote endpoint name. This is the first step to eliminate static binding along the whole way down the stack. Application will bind to a name of the mobile endpoint, other than specific IP address and port number. This step allow the later rebinding of the mobile endpoint name to other address or port number, even different address family. The second step is to put a session layer on top of existing transport layer and below application layer. This session layer will dynamically collect network events in the underlying layers and also react to application layer suspend/resume requests. The session layer will also work as event dispatcher, according to the different property of specific event; it will trigger the corresponding functions to deal with it. After application started a connection, the session layer will keep a mapping between the connection and a socket identifier. The session layer will also provide data consistency. The third step is to extend existing socket implementation to allow rebinding inside the socket layer. Due to the limitation of thesis work time, we decided to focus on TCP IPv4 socket, to allow it to rebind to different address and port. Conceptually if a socket can rebind to different address and port without changing the current connection state, it means we could support natural mobility from the TCP/IP stack. This inborn mobility support will enlighten the possibility of designing an architecture to support full mobility for all network applications.

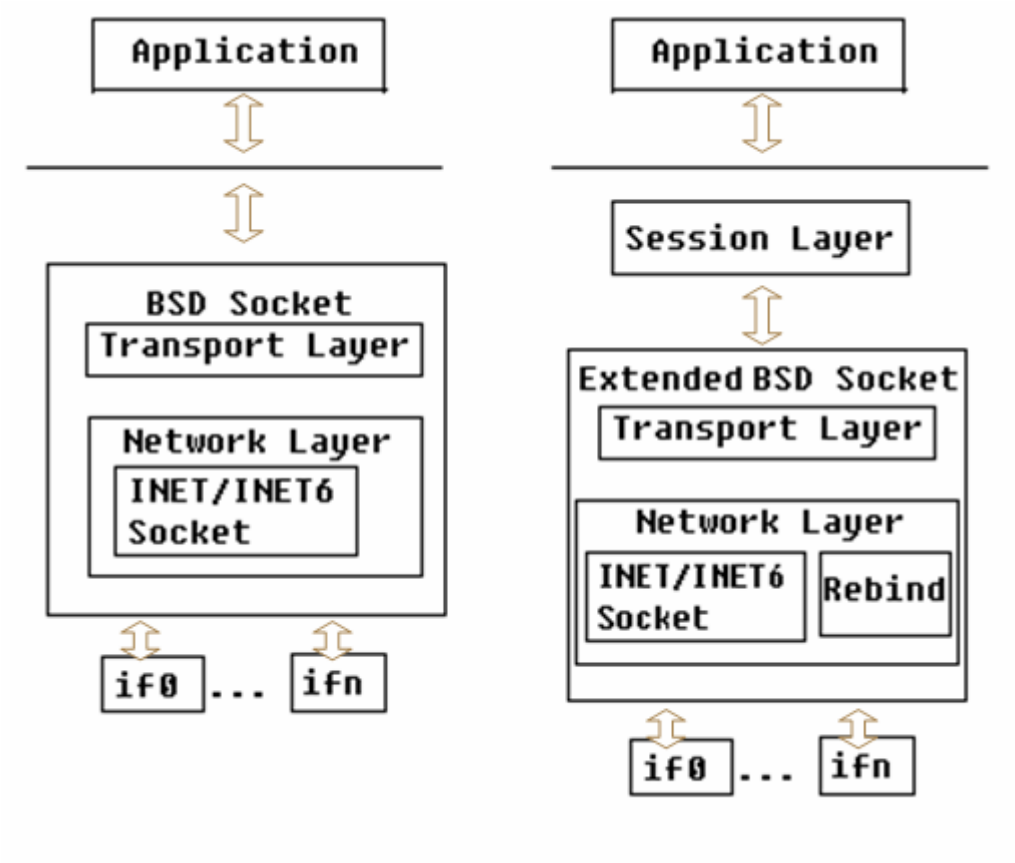

Figure 6 Comparison of our architecture with traditional BSD infrastructure.

This paper will not talk about session layer, which is the focus of the other group. Our implementation took place in two parts. First is to allow multiple ways to start communication. Second is to implement dynamic rebinding on socket. We decided to provide additional functions of BSD sockets. The original functions will remain untouched, which means if application doesn't use any our extended rebind functions, it will run as the same on any other BSD socket system. This makes our system compatible with any existing BSD socket applications. We are expanding it to have one more function named uss_socket_rebind, which is provided by adding one more kernel library named uss_rebind. Besides the wrapped up rebind function, application could also call individual rebind for source/destination address or source/destination port.

For the design environment, after some comparison we choose to use Linux kernel 2.6.15.

3.2. Heterogeneous socket:

The BSD socket implementation uses source address, destination address, source port and destination port to uniquely identify a connection. Here the name of a connection is the quadruplet of *[source address, destination address, source port, destination port]*. During the

communication time between two endpoints, if any of the four parameters changes, the current connection will terminate and a new connection needs to be set up for communication. Considering we are aiming at providing mobility support for communication, which means we think the communication didn't change as long as it was between the two same endpoints, even if the source/destination address or source/destination port might change. The old definition of identifying communication by source address, destination address, source port and destination port is not suitable any more. The first step we took is to choose a naming method to support our definition of mobility scope. During the life time of communication, the names of the two endpoints are relatively stable. There are several naming proposals for mobile objects, which will affect the mobility scope of the objects. Our first step is to liberate communication dependence on source and destination addresses by using a different naming method to identify the communication endpoints. We simply choose *[DNS names and port number]* to be the identifier of our mobile endpoints, which is relatively stable during the communication lifetime. Even any of the endpoints' address changes, the DNS name is still able to uniquely identify the communication. By taking this step, we allowed coordinative multiple ways of starting communication.

Nowadays, most computers have more than one physical network interfaces, like different speed wired Ethernet network cards, wireless network cards. The communication between two endpoints could be set up through different physical links, one endpoint could use wired Ethernet 100Mbps network card, the other endpoint could use 1000Mbps network card. It could also be one endpoint is using wired network card, and the other endpoint is using 802.11x wireless network cards. Besides the communication could be set up between various physical interfaces, different network protocols and transport protocols running on each endpoint also allow the communication between the two endpoints to be setup in more diverse ways.. With the same transport protocol, the communication could be set up through either IPv4 or IPv6 if both endpoints support the same network protocol. With certain gateway support, the communication could even be established while one endpoint uses IPv4 and the other endpoint uses IPv6. By using DNS name and port number to uniquely identify an endpoint is a proper naming method for us to allow setting up the communication between two endpoints in multiple simultaneous ways.

We implemented a heterogeneous socket library to support start connection dynamically from client side. In order to do so, we need a new data structure to keep the selected communication method information, like socket file descriptor, chosen address family and the

used socket address information. Here we defined a data structure named MIS_HSOCK_SESSION. The function hsock_startClinetSession takes four input parameters: server name, connection type, port number and a PMIS_HSOCK_SESSION session pointer.

```
typedef    struct MIS_HSOCK_SESSION {
               SOCKET hsock_socket;
               int hsock_family;
               union {
                        sockaddr_in hsock_sockAddr;
                        sockaddr_in6 hsock_sockAddr6;
                        }sockAddr;
               }HSOCK_SESSION;
typedef struct MIS_HSOCK_SESSION *PMIS_HSOCK_SESSION
```

int hsock_startClientSession(char *hostName, int type, char *clientPort, PMIS_HSOCK_SESSION pSession)

Parameters:

hostName	[in] A pointer to a char array which keeps the server name;
type	[in] Type specification of the connection. It's same as the type specification of Windows Socket. SOCK_STREAM or SOCK_DGRAM.
clientPort	[in] A pointer to a char array which keeps the port number.

Return value:

Return 0 if there is no error and the connection succeeds. The current connection information is in kept the data structure pointed by pSession. Return -1 if there is no local network interface configured or couldn't get local configuration information. Return -2 if server DNS name resolution failed. Return -3 if connection failed or there is way to set up connection.

Remarks:

In our startClinetSession function, we first do remote hostname resolution and get a list of registered addresses. Then we get local network interface information. At this point we do simple connection method decision and give priority to IPv6 if both ends support it. Applications using this startClientSession function could use the socket, which is kept inside the PMIS_HSOCK_SESSION structure pointed by the pSession, to go on to send or receive.

This function subtracts an application from specifying the address family during communication time. It also makes the underlying socket type and address type relatively transparent to application. This enables the possibility of changing socket or rebind socket while keep the application running on top of it undisturbed. With the heterogeneous socket library application could start independently underlying network, and keep on communication without knowing the changes below socket, the changes could be the current using interface going down or changing network address or disabling of certain network protocol (like IPv4) on one endpoint of the communication. After the first step we know that in order to make application compatible with network changes, we need to make socket to be able to dynamically rebind with network change. Right now, the socket is still statically binding to specific transport protocol (UDP or TCP), certain network protocol (IPv4 or IPv6) and port numbers. It is not enough to support dynamic network events. We need to change the socket to be able to bind to different address family and different port numbers. This leads to my second part of work.

3.3. Rebind socket

Sections 3.3.2 and 3.3.3 briefly describe Linux Kernel socket related functions implementation. This is needed for better understanding of my work aimed at manipulating and changing of those functions and correspondent data structures according to the needs and requirements set by mobility management. For more detailed description of the functions see [11]. Besides that, this material required significant work in order to understand the subject of those changes, which requires enhancements due to the above mentioned mobility management requirements.

3.3.1 Rebind socket design choice

Socket's statically binding to certain address family, type of communication and port number, limits us from allowing application compatible of different network changes. In order to support all possible changes trough the whole TCP/IP stack, we need a socket which is able to rebind to different network address family, different network address and different port numbers. With the limitation of thesis work time, we decided to focus on TCP IPv4 socket rebinding. This choice is done based on two facts. The first is that TCP is a protocol which keeps states; this makes it appear harder to support rebinding. The second is that IPv4 is the most widely used network protocol. Making changes in IPv4 socket could be interesting for many people. We choose to do our implementation on Linux kernel 2.6.15.

3.3.2. Current socket implementation survey

According to Linux kernel 2.6.15 [11], the networking part of the Linux kernel uses mainly two data structures, as show in *Figure 7*: one to keep the state related information of a connection, called *sock* (for "socket"), and another to keep the data and status of both incoming and outgoing packets, called *sk_buff* (for "socket buffer"). The following data structure of sock and sk_buff are from the kernel [11] source file source/include/net/sock.h and source/include/linux/skbuff.h.

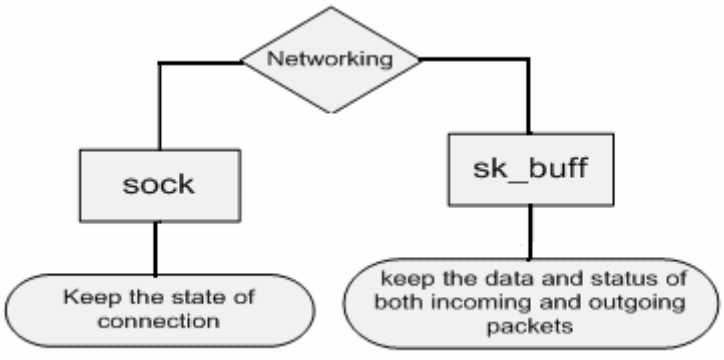

Figure 7 Socket data structure relation with network

The data structures in use:

```
struct sk_buff{
        struct sk_buff          *next;      /*Next buffer in list*/
```

```
        struct sk_buff          *prev;      /*Previous buffer in list*/
        struct sk_buff_head     *list;      /*List we are on*/
        struct sock             *sk;        /*socket we are owned by*/
        ……….
        struct net_device       *dev;       /*Device we arrived on/are leaving by*/
        ……
        transport and network layer headers;
        Socket control block;
        ……
        Data length
        Data
        ……
}
```

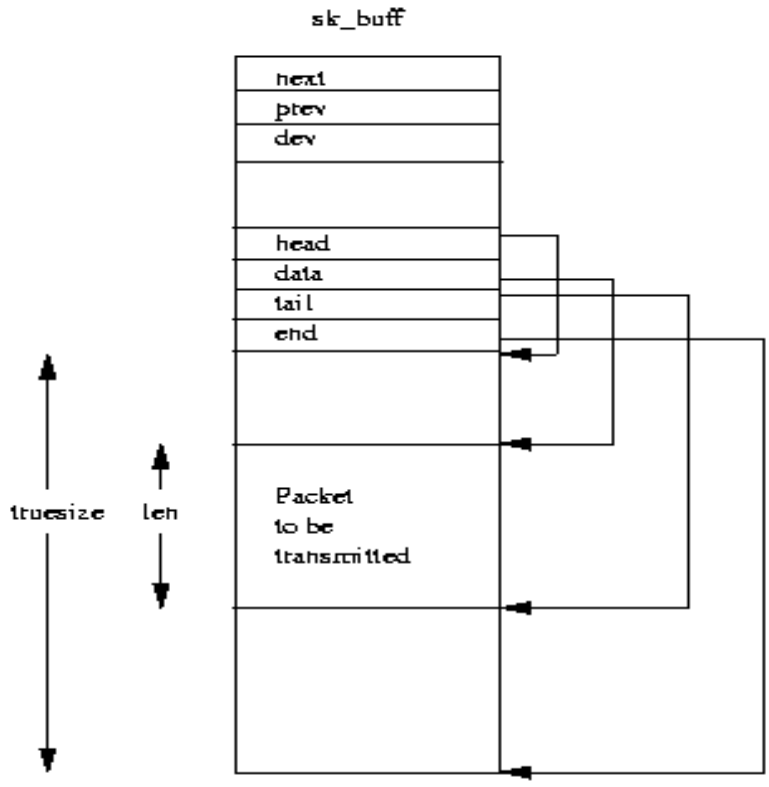

Figure 8 Sock buffer structure

```
*       struct sock_common - minimal network layer representation of sockets
 *      @skc_family: network address family
 *      @skc_state: Connection state
 *      @skc_reuse: %SO_REUSEADDR setting
 *      @skc_bound_dev_if: bound device index if != 0
```

```
 *      @skc_node: main hash linkage for various protocol lookup tables
 *      @skc_bind_node: bind hash linkage for various protocol lookup tables
 *      @skc_refcnt: reference count
 *      @skc_hash: hash value used with various protocol lookup tables
 *      @skc_prot: protocol handlers inside a network family
 *      This is the minimal network layer representation of sockets, the header
 *      for struct sock and struct inet_timewait_sock.
struct sock_common {
        unsigned short          skc_family;
        volatile unsigned char  skc_state;
        unsigned char           skc_reuse;
        int                     skc_bound_dev_if;
        struct hlist_node       skc_node;
        struct hlist_node       skc_bind_node;
        atomic_t                skc_refcnt;
        unsigned int            skc_hash;
        struct proto            *skc_prot;
};
```

```
  *     struct sock - network layer representation of sockets
  *     @__sk_common: shared layout with inet_timewait_sock
  *     @sk_shutdown: mask of %SEND_SHUTDOWN and/or %RCV_SHUTDOWN
  *     @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
  *     @sk_lock:       synchronizer
  *     @sk_rcvbuf: size of receive buffer in bytes
  *     ……....
  *     @sk_dst_cache: destination cache
  *     ……
  *     @sk_receive_queue: incoming packets
  *     @sk_write_queue: Packet sending queue
  *     @sk_sndbuf: size of send buffer in bytes
  *     @sk_flags:
  *   ……
  *     @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
IPV6_ADDRFORM for instance)
  *     @sk_ack_backlog: current listen backlog
```

```
 *        ……....
 *        @sk_type: socket type (%SOCK_STREAM, etc)
 *        @sk_protocol: which protocol this socket belongs in this network family
 *        ……
 *        @sk_send_head: front of stuff to transmit
 *        ……..
 *        @sk_state_change: callback to indicate change in the state of the sock
 *        ……..
 *        @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
 */
```

3.3.3 How each BSD socket function is designed?

int socket(int socket, int type, int protocol)

As shown in *Figure 9,* when a user invokes the socket() system call, this calls sys_socket() inside the kernel according to linuxkernel/net/socket.c). The sys_socket() function is made up of two parts. First part, it calls sock_create(), which allocates a new sock structure where keeps all the information about the socket/connection. Second part, it calls sock_map_fd(), which maps the socket to a file descriptor. With the file descriptor, the application can access the socket like a file.

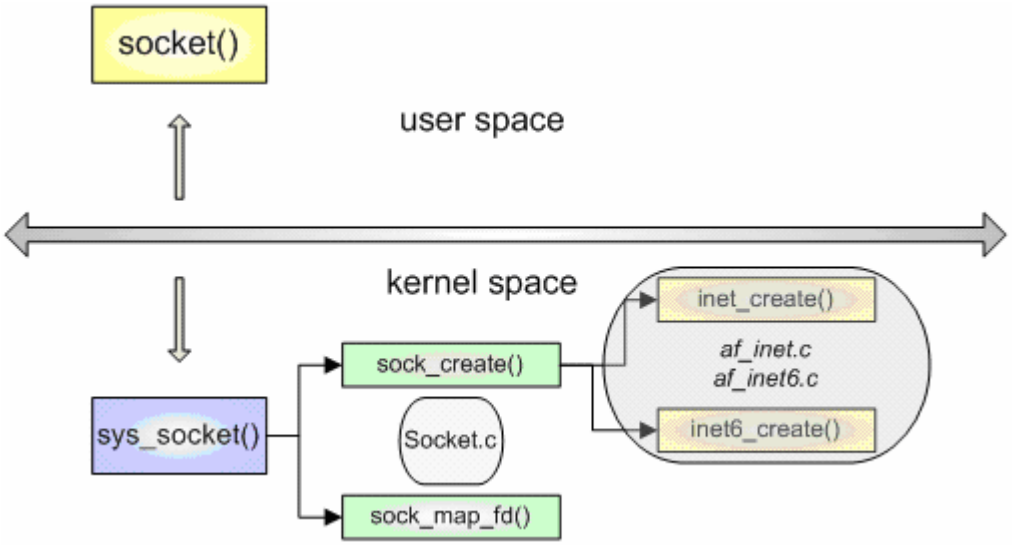

Figure 9 Structure of socket() function

int bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)

In the kernel the bind() system call is implemented by sys_bind(), which puts the send/receive address and device information into sock data structure. Inet_sk(sk)->rcv_saddr = inet_sk(sk)->saddr = inputaddr->sin_addr.s_addr and port numbers inet_sk(sk)->dport, inet_sk(sk)->sport = inet_sk(sk)-num. It will hash the address information and put it into bind hash table.

int listen(struct socket *sock, int backlog)

In the kernel the listen() system call is implemented by sys_listen(), which calls the appropriate listen function according to the protocol type, sk->ops->listen(sock, backlog). For TCP, it calls tcp_listen_start(). When an application calls listen(socket, backlog), the kernel sys_listen() is triggered and the listen function for according protocol family will be called eventually.

int connect(struct socket *sock, struct sockaddr *daddr, int addr_len, int flags)

In the kernel the connect() is implemented by sys_connect(), which calls the proper accept function according to the protocol type, sk->ops->connect()).When an application calls the connect() system call, the function sys_connect() is called inside the kernel and the accept function for according protocol family is called eventually. In the case of TCP, the accept function is tcp_connect(), which is called by sk->ops->connect() on the socket. There is no connect() implemented for UDP, because it is a connectionless protocol. The tcp_connect() function initializes several fields of the tcp_opt structure, and a sk_buffer is created at the end of function, which is a packet built in the format of a SYN packet (TCPCB_FLAG_SYN), which starts from sequence number 0 and ACK number 0.

int accept(struct socket *sock1, struct socket *sock2, int flags)

While for the server, a socket has been created, bound to a certain address and port number pair. The server has called listen(socket, backlog) to wait for incoming connections. This changed the state of the socket to LISTENING. When a SYN packet arrives, which is sent by the client connect() call, this is dealt with by tcp_rcv_state_process(). The server then replies with a SYNACK packet that the client will process in tcp_rcv_synsent_state_process(); this is the state that the client enters after sending a SYN packet. Both tcp_rcv_state_process() (in the server) and tcp_rcv_sysent_state_process() (in the client) have to initialize some other data in the tcp_opt structure. This is done by calling tcp_init_metrics() and tcp_initialize_rcv_mss().

Both the server and the client acknowledge these packets and enter the ESTABLISHED state. From now on, every packet that arrives is handled by tcp_rcv_established().

int write(struct socket *sock, struct char *msg, int flags)

Every time a user writes in a socket, this goes through the socket linkage to inet_sendmsg(). The function sk->prot->sendmsg() is called, which in turn calls tcp_sendmsg() in the case of TCP or udp_sendmsg() in the case of UDP.

int close(struct socket *sock)

When the user closes the file descriptor corresponding to this socket, the file system code calls sock_close(), which calls sock_release() after checking that the inode is valid. The function sock_release() calls the appropriate release function, in our case inet_release(), before updating the number of sockets in use. The function inet_release() calls the appropriate protocol-closing function, which is tcp_close() in the case of TCP. The latter function sends an active reset with tcp_send_active_reset() and sets the state to TCP_CLOSE_WAIT.
Figure 10 shows the relation between kernel files and BSD socket functions.

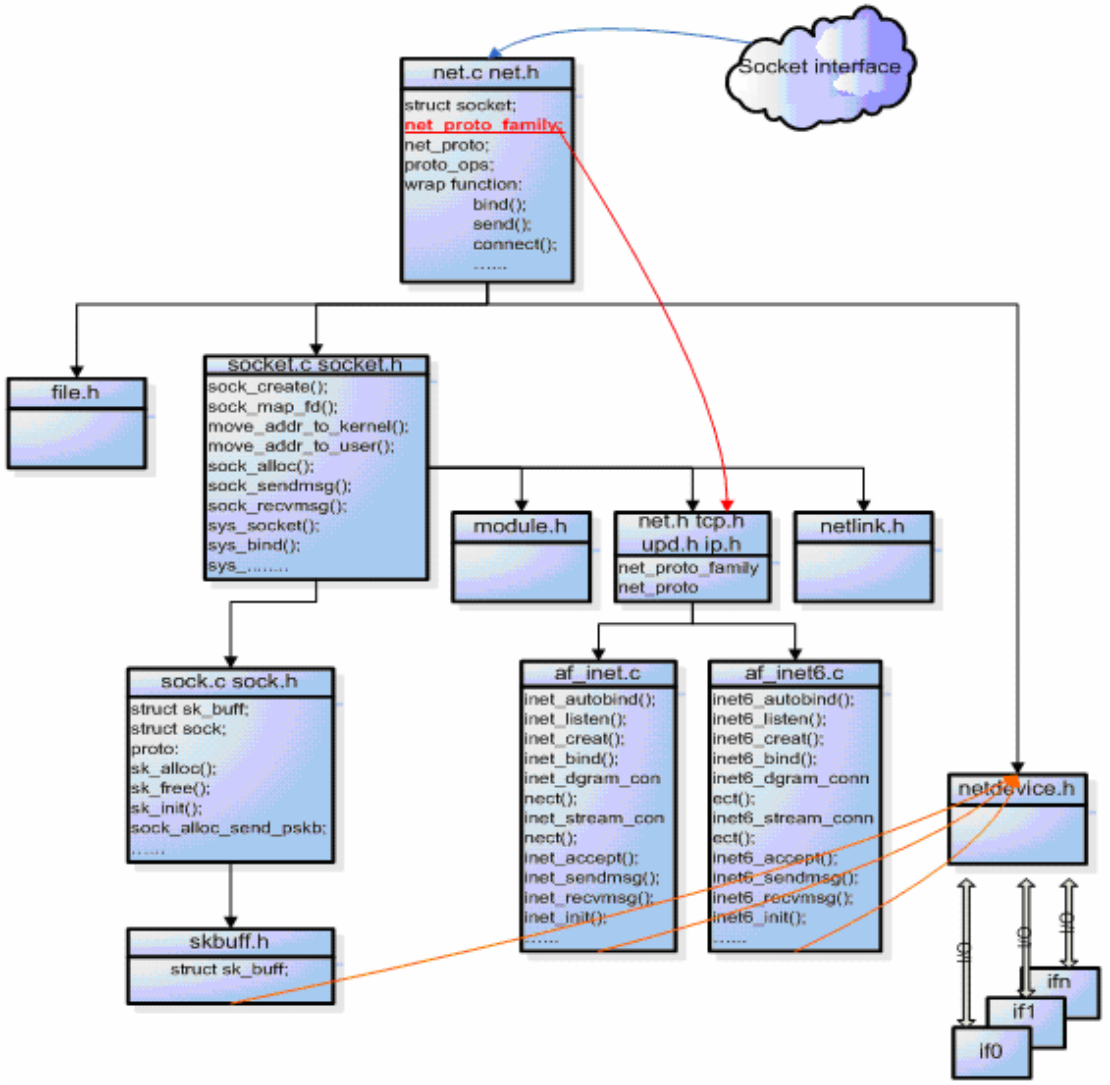

Figure 10 The relation between BSD Socket functions and kernel files.

The following packet reception process is referred to [12]. We modified it to fit the new kernel of 2.6.15.

Packet reception steps:

- Packet received by hardware;
- Receive interrupt generated;
- Driver handler copies data from hardware into fresh sk_buff;
- Calls netif_rx() to queue on backlog;
- Schedules net_bh() with mark_bh(NET_BH);
- net_bh() executes the next time the scheduler is run or a system call returns or a slow interrupt handler returns;
- net_bh() tries to send any pending packets, then de-queues packets from the backlog

and passes them to correct handler, say ip_rcv();

- ip_rcv() may call ip_local_deliver() or ip_forward();
- ip_local_deliver() results in call to tcp_v4_rcv() through the inet_protos list;
- tcp_v4_rcv() queues data at the correct socket's queue;
- When the socket's owner reads, tcp_recvmsg() is invoked through BSD socket's proto_ops;
- If instead the socket's owner had blocked on a read, that process will be woken using wake_up (wait queue);

The *Figure 11* is to group the fields of structure sock and sk_buff according into different layers of TCP/IP stack.

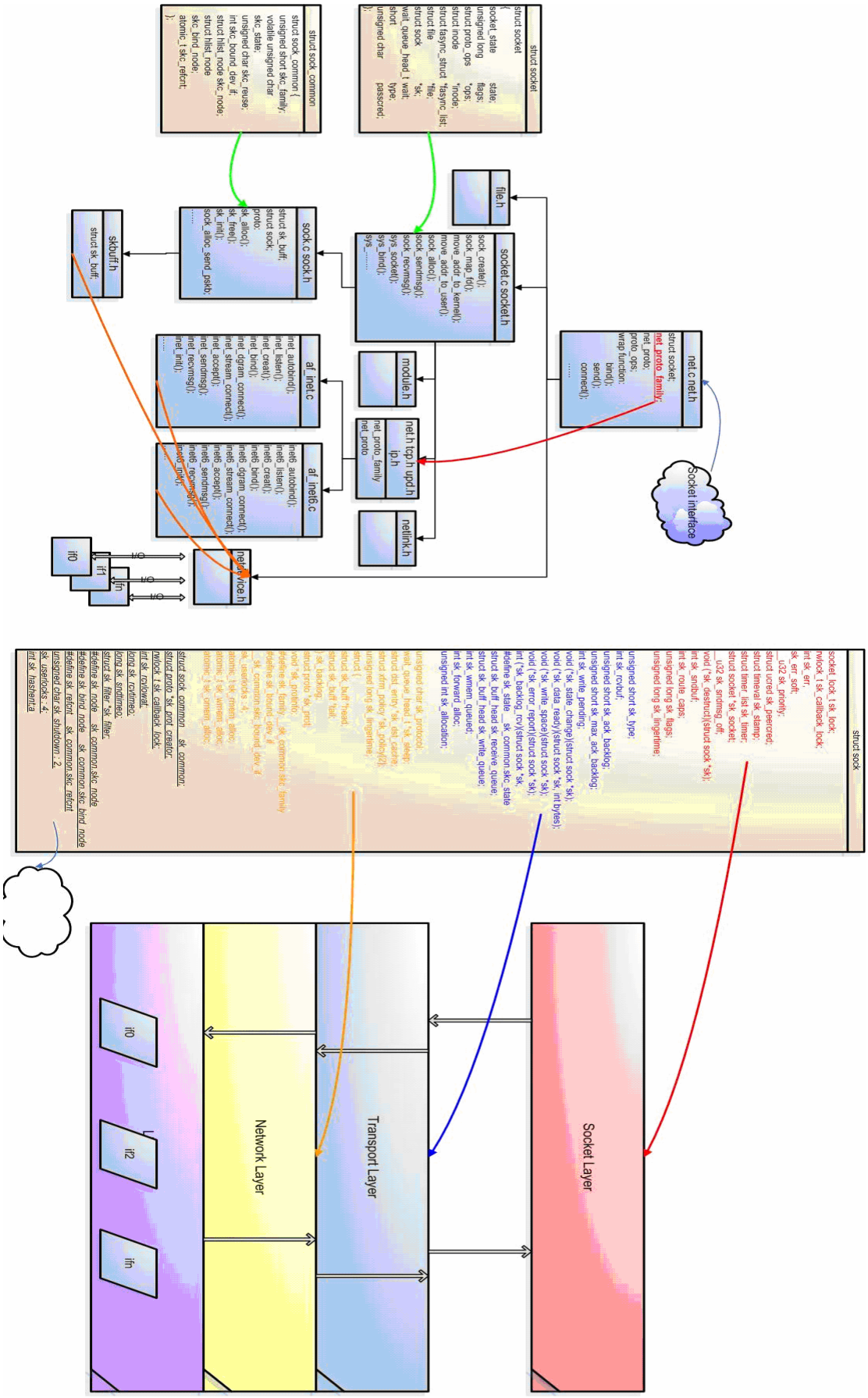

Figure 11 Mapping between data field and different layer in the stack

3.3.4 Rebind implementation

Based on the survey of Linux kernel sock structure, we implemented our socket rebind by using of several kernel functions: tcp_unhash(), tcp_hash(), ip_route_output_flow(), sk_setup_caps(), these four kernel functions details referred to [11] the linux-kernel-2.6.15 implementation. tcp_unhash(struct sock *sk) and tcp_hash(struct sock *sk) this two functions are used to update the hash tables correspondent with the sock state. ip_route_output_flow(struct rtable **rp, struct flowi *flp, struct sock *sk, int flags), this function returns a proper route for sock to use. sk_setup_caps(struct sock *sk, struct dst_entry *dst) , this function updates the destination cache for the sock and put the new route into it if they are different, in addition it will check the network capacibility.

We use both windows virtual machine 4.5.2 and user mode Linux as development environment.

3.3.4.1 Kernel Functions Details (Linux kernel 2.6.15 specific)

TCP unhash:

If the current sock is not hashed, it will just return.

- If the sock is in TCP_LISTEN or ESTABLISHED state, it will lock the current sock first. If this is a listening sock, it will lock TCP from listening on this sock.
- Unhash will detach it from the hash list and initialize the hlist_node sk_node to point to NULL.
- Decrease the number of protocols which are using the current sock.
- Unlock the sock.
- If this is a listening sock, before return it will wake up TCP to listen on this sock.

TCP hash:

If sock is not CLOSED, it will execute the following steps:

- Disable TCP bind hash, keep both current hash list information;
- If the sock is in listening state, insert its sk_node into listening hash table.
- If the sock is in established state, insert it into established hash table.
- Increase the number of protocols which are using the current sock.
- Unlock the sock.
- If this is a listening sock, before return it will wake up TCP to listen on this sock.

ip_route_output_flow(struct rtable **rp, struct flowi *flp, struct sock *sk, int flags);

Int __ip_route_output_key(struct rtable **rp, const struct flowi *flp);

It goes through the current (rcu: routing control unit) routing hash table. If there is a match with the current (protocol &&source address &&destination address &&source port && destination port), it will return the matching (rtable) routing table pointer. If there is no match, it will call:

Ip_route_output_slow(struct rtable **rp, const struct flowi *flp);

This function will check if there is outgoing device and route. If there is not outing going route, it will make a route and put it into the routing hash table with accordingly counter. The returned route will be put in to *rp.

sk_setup_caps(struct sock *sk, struct dst_entry *dst);

sk_setup_caps() calls __sk_dst_check() first.

__sk_dst_check(struct sock *sk, struct dst_entry *dst);

This function will put the destination entry pointed by *dst into sk->sk_dst_cache.

(I added an conditional sentence before this function if the new dst == sk->sk_dst_cache, this function will not change anything);

The function will check the routing capabilities of sending large packets and header length or if the interface device could offload TCP/IP segmentation.

3.3.4.2 Extended socket rebind functions

This section describes our extended kernel socket functions for socket rebind implementation. These functions are individual functions for single rebind functions begin with uss_ and the final wrapped up functions begin with session_.

Uss_socket_rebind(…) fuction details:

int uss_tcp_v4_rebind_daddr(struct socket *skp, struct sockaddr *newaddr);

Parameters:

skp [in] a socket pointer;

newaddr [in] a sockaddr pointer to the pass new destination address to bind with.

Return value:

Return 0 if the rebind succeed.

Remarks:

The function will change the old destination address in the sock structure which is inside the skp->sk to the new destination and delete the original hash entry in according hash table(listen hash or establish hash); And according to the current socket state en-hash the new address into the according hash table.

int uss_tcp_v4_rebind_saddr(struct socket *skp, struct sockaddr *newaddr);

Parameters:

skp [in] a socket pointer;

newaddr [in] a sockaddr pointer to the pass new source address to bind with.

Return value:

Return 0 if the rebind succeed.

Remarks:

The function will change the old source address and receive address in the sock structure which is inside the skp->sk to the new source address and delete the original hash entry in according hash table(listen hash or establish hash); And according to the current socket state en-hash the new address into the according hash table.

Int uss_tcp_v4_rebuild_header(struct sock *sk);

Parameter:

sk [in] pointer to the current sock structure.

Return value:

The function will return 0 if succeed.

Remarks:

The function will first call the **ip_route_output_flow** to get a new route for the destination address and port based on the source address and port. It calls __sk_dst_set via sk_setup_caps. The __sk_dst_set function will put the new route into the sk_dst_cache if it is different from the original one.

Int uss_rebind_daddr(struct socket *skp, struct sockaddr *newdaddr);

Parameter:

skp [in] pointer to a socket structure.

newaddr [in] pointer to the new destination structure.

Return value:

The function will return 0 if succeed.

Remarks:

This function will calls uss_tcp_v4_reselect_daddr() which will put the new destination address to the sock structure pointed by skp->sk. It will call tcp_unhash() to remove the according hash entry in the hashtable. Then it will use tcp_hash() to put the sock into hash table with new destination address. It will call uss_tcp_rebuild_header() to put the new route into the sock structure sk_dst_cache.

Int uss_rebind_saddr(struct socket *skp, struct sockaddr *newdaddr);

Parameter:

skp [in] pointer to a socket structure.

newaddr [in] pointer to the new source structure.

Return value:

The function will return 0 if succeed.

Remarks:

This function will calls uss_tcp_v4_reselect_saddr() which will put the new source address to the sock structure pointed by skp->sk. It will call tcp_unhash() to remove the according hash entry in the hashtable. Then it will use tcp_hash() to put the sock into hash table with new source address. It will call uss_tcp_rebuild_header() to put the new route into the sock structure sk_dst_cache.

Int uss_rebind_sport(struct socket *skp, _u32 sport);

Parameter:

skp	[in] pointer to the socket structure.
sport	[in] new port number in network bit order.

Return value:

The function will return 0 if succeed.

Remarks:

This function will calls will put the new source port number and locally chosen random port number to the sock structure pointed by skp->sk. It will call tcp_unhash() to remove the according hash entry in the hash table. Then it will use tcp_hash() to put the sock into hash table with new source address. It will call uss_tcp_rebuild_header() to put the new route into the sock structure sk_dst_cache.

Int uss_rebind_dport(struct socket *skp, _u32 dport);

Parameter:

skp	[in] pointer to the socket structure.
dport	[in] new destination port number in network bit order.

Return value:

The function will return 0 if succeed.

Remarks:

This function will calls will put the new destination port number to the sock structure pointed by skp->sk. It will call tcp_unhash() to remove the according hash entry in the hash table. Then it will use tcp_hash() to put the sock into hash table with new source address. It will call uss_tcp_rebuild_header() to put the new route into the sock structure sk_dst_cache.

session_socket_rebind(struct socket *skp, int operation);

Parameter:

skp [in] pointer to the socket structure.

operation [in] a integer of the type of operation.

```
#define USS_REBIND_SADDR 1
#define USS_REBIND_DADDR 2
#define USS_REBIND_SPORT 3
#define USS_REBIND_DPORT 4
#define USS_REBIND_SADDRSPORT 5
#define USS_REBIND_DADDRDPORT 6
```

Return value:

The function will return 0 if succeed.

Remarks:

This function will calls will put the new destination port number to the sock structure pointed by skp->sk. It will call tcp_unhash() to remove the according hash entry in the hash table. Then it will use tcp_hash() to put the sock into hash table with new source address. It will call uss_tcp_rebuild_header() to put the new route into the sock structure sk_dst_cache. If there is data in the current incoming/outgoing queue, the queues will be purged.

session_socket_rebuild(struct socket *skp)

This function will call the session_socket_tcp_v4_rebuild_header function and put the socket into correspondent hash table with the sock state.

session_socket_tcp_v4_rebuild_header(struct socket *skp)

This function will generate a new header for the socket accordingly and is called by the uss_socket_rebind. This function is mainly calling ip_route_output_flow() and sk_setup_caps().

3.3.5 Incoming and outgoing queues

After rebinding the socket to a different source/destination address or source/destination port, the data packets which are in the old send/receive queue of the socket are not changed. The tests we did showed that after rebinding on both end-points, the TCP convergence takes quite long time due to the packets left in the send/receive TCP queues with old pre-computed information. These packets will have a wrong checksum. As checksum is calculated with source/destination address information, the outdated information will make the checksum wrong. When the sender tries to send it out, the receiver will not accept it. These packets will be dropped after certain times of retransmission. While thinking about how to deal with the outdated data, we choose to clean up the buffer with the rebind function.

In the rebind function, it will check if there are data queued in the send/receive queues. If there are some, this will trigger the purge queue functions respectively. For the write queue, this is achieved by faking ACK for the packets in the send queue. We will detach each sock buffer from the sk_write_queue and free the sk_buffer. After freeing the write queue, the socket will reclaim the memory space used by the write queue and point the sk_send_head to NULL. For the receive queue, we put the responsibility of clearing the receive queue is left on the session layer to make a receive call before performing rebind. Data synchronization is performed by the session layer and involves some message exchange.

Status:

Rebinding of a socket will not influence the current TCP state (e.g. LISTENING, ESTABLISHED) and IO queues. After the rebind operation, TCP socket state will not be influenced, the relative sequence numbers will start from 0. According to [13] in TCP/IP “When a new connection is being established, the SYN flag is turned on. The sequence number field

contains the initial sequence number (ISN) chosen by this host for this connection. The sequence number of the first byte of data sent by this host will be the ISN plus one because the SYN flag consumes a sequence number". In our current stage this doesn't matter because according to our design plan we clear the existing TCP buffer.

The total amount of change for Linux kernel 2.6.15 is not big. It will include inserting a kernel module to provide the addition functions and minor changes in several kernel files. This means that the change could be easily port to another Linux system.

4 Test and Verification

The goal of this section is to verify the socket rebinding. After the rebinding, the communication must still go on normally, and the rebinding must not change the TCP states. If the TCP state has changed, after rebind the connection couldn't go on with send/receive message directly, it needs to re-establish the connection by sending out SYN and wait for SYN ACK. The tests, described in this chapter, are performed with the following configuration: The laptop computer is acting as the client, while the desktop computer is the server. Both computers are running the same Linux kernel 2.6.15 with the uss_rebind module in the kernel. Both client and server are in the same subnet of 192.71.20.224/28, address allocation is handled by a DHCP server running the other computer. All tests are performed in the following steps:

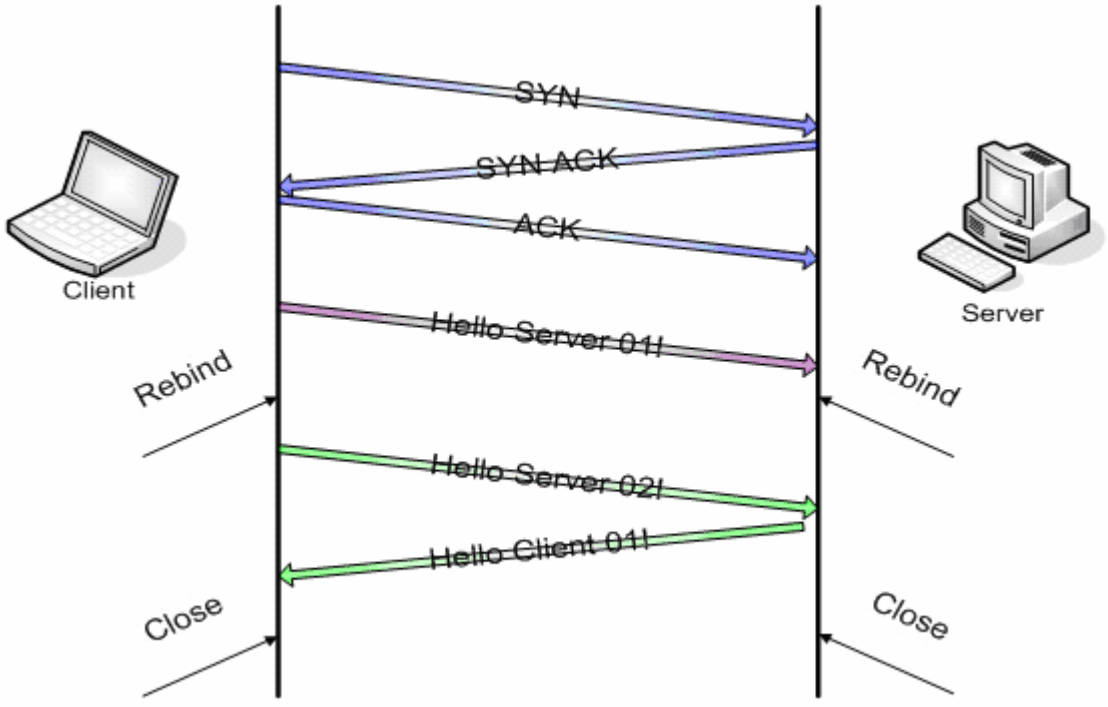

Figure 12 Test steps

Step 1: Server starts listening, client connects to server.

Step 2: If connection is successful, client sends a message of "Hello Server 1!"

Step 3: After client has sent out the first message, it will start rebinding the connected socket (this differs in different tests). After server received the first message, server will start rebinding (this also differs in different tests)

Step 4: If the rebind succeeds, the client will send to server the second message of “Hello Server 2!” On the server side if rebind succeeds, the server will try to receive the second message from client.

Step 5: After the second message received from the client, the server will send a message back to the client “Hello Client 1!” Client will try to receive a message from server after sending out the second message.

Step 6: Both server and client close sockets and return.

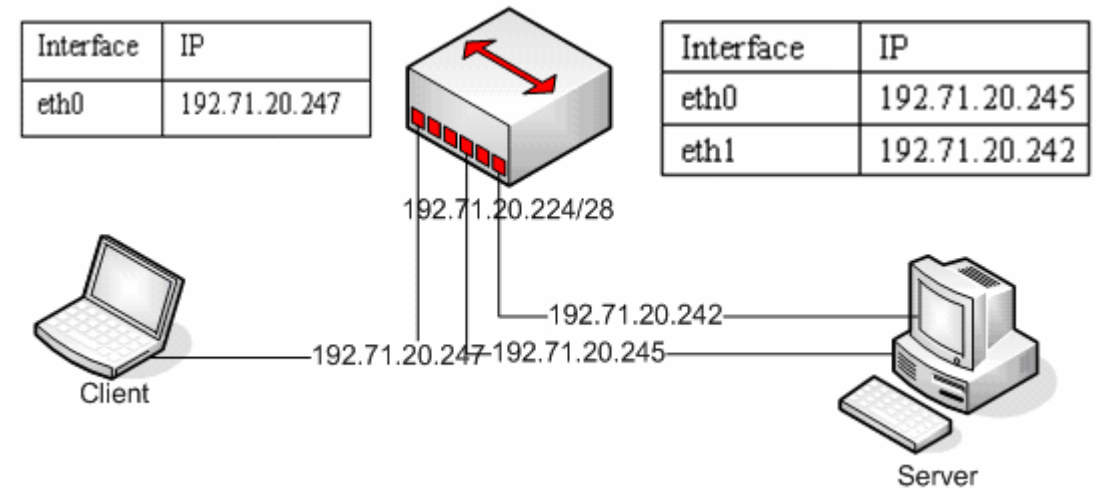

Figure 13 Test configuration

Both client and server are running on Windows virtual machine 4.5.2, the guest operating system is Linux kernel 2.6.15. The time of rebinding is synchronized manually in the test program by certain timer. If two ends are not performing rebinding at almost the same time, if one end sends packet to the other end, the connection will be reset. We also did some integrated test with the session layer which is currently under development. In that case session layer will take care of the rebinding synchronization.

4.1 Rebind destination address

In this test, after the connection is setup, the first message of “Hello Server 1!” from client to server is sent to 192.71.20.245, which is show in the following figure.

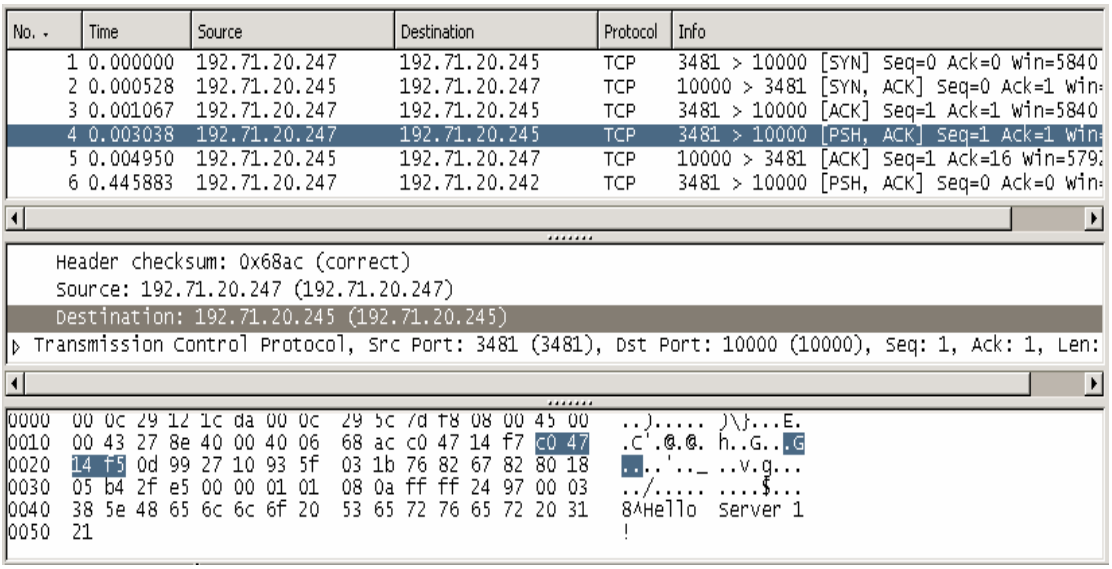

Figure 14 Before the rebind destination address

On the client side rebind happens after the client sending out the first message. Which means rebind happens after the 4th packet in the picture. After rebind, the client could send out the second data message of "Hello Server 2!" through the same socket. The message is sent to the new destination address 192.71.20.242, which is show in the following figure.

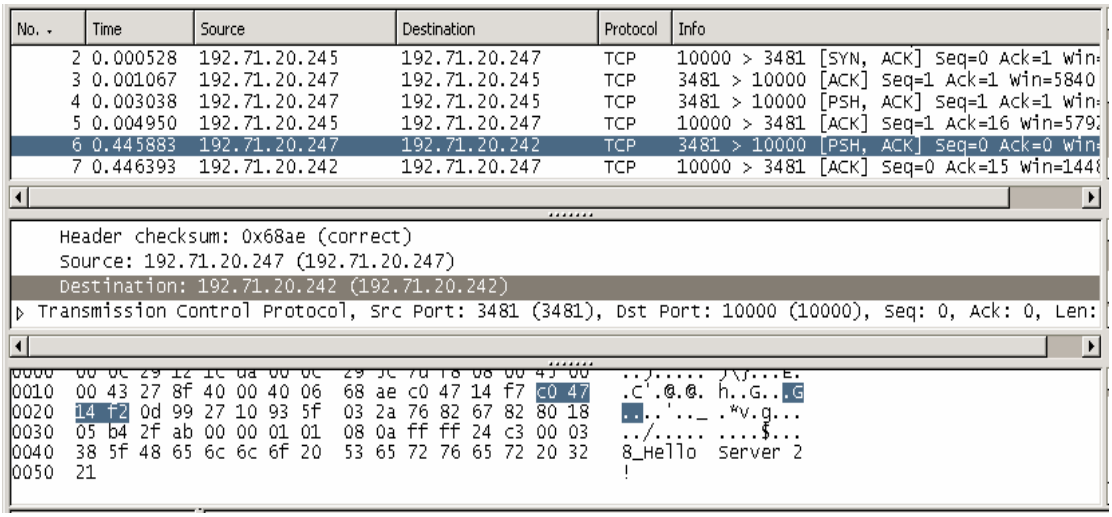

Figure 15 After the Rebind of Destination Address

The test shows that after rebind destination address, the socket state didn't change. The socket is still in Established state, data transmission could keep on.

4.2 Rebind source address

This test is to verify the rebinding the source address. Before the rebind, the server is using 192.71.20.245 as source address to send packets, which is shown in the following figure.

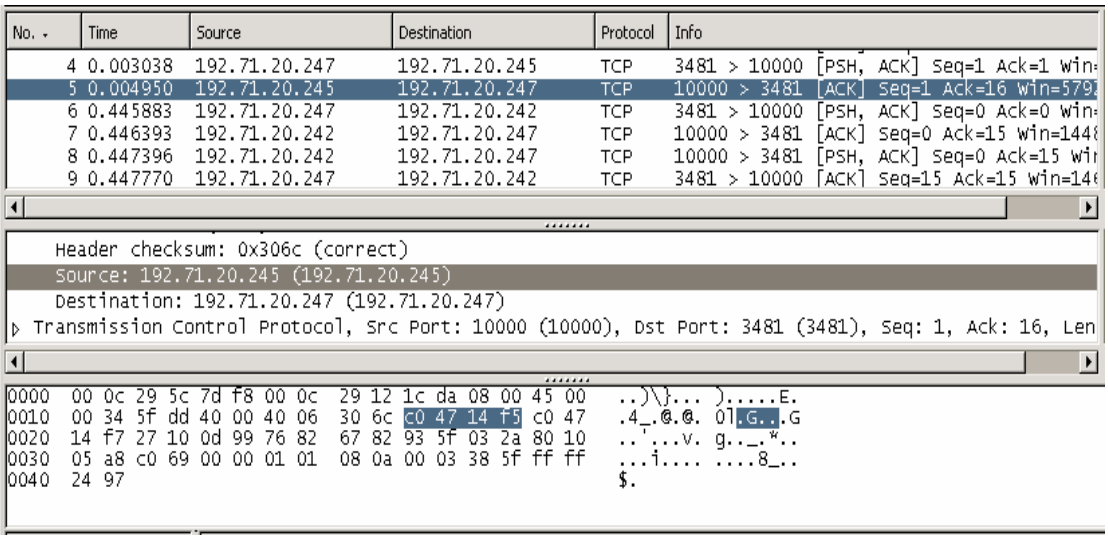

Figure 16 Before Rebind Source Address

On the server side rebind happens after sending out the ACK of the first data packet, which is the 5th packet in the picture. After rebind source address, the server will keep on receiving packets on the same source but with a different source address of 192.71.20.242. From the 7th packet in the picture, we can see TCP is still in ESTABLISHED state, data communication could keep on.

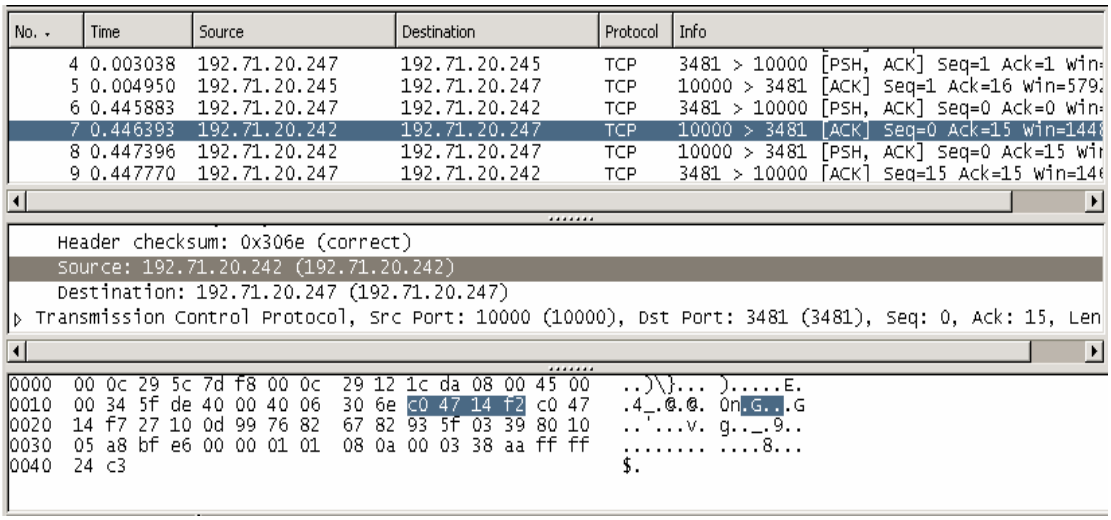

Figure 17 After Rebind Source Address

In the following figure, you can see the third message of “Hello Client 01!” from the server to the client.

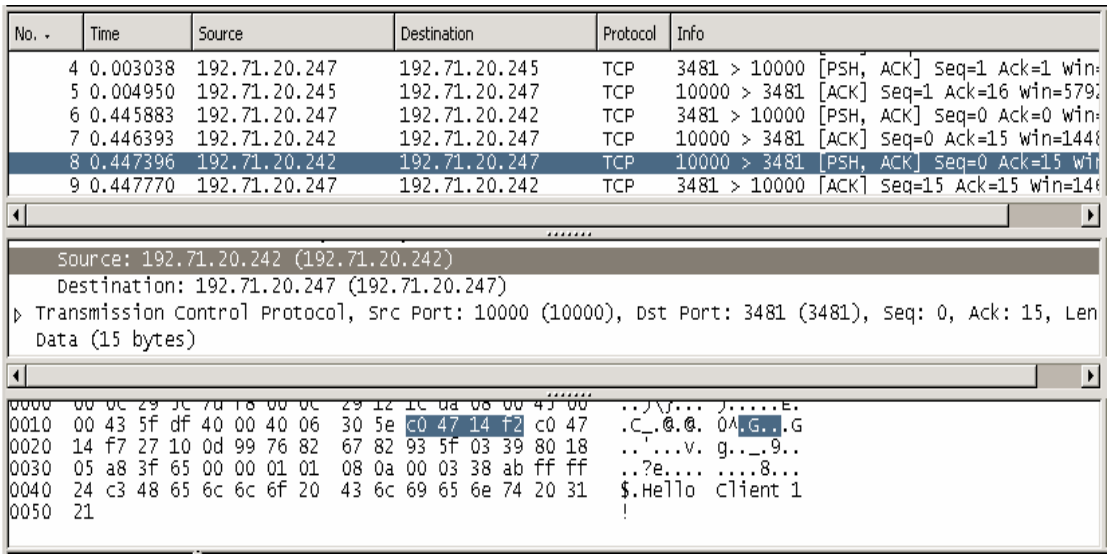

Figure 18 Send Message after Rebind Source Address

4.3 Rebind source and destination address

The following is the complete network traffic log of rebind destination address on the client side and rebind source address on the server side. After the rebind on both end after the send/receive of the first data packet of "Hello Server 01!" client could keep on sending data through the same socket but with different destination address, server could keep on receiving data through the same socket but with a different source address. This verified that the rebind didn't change the socket state. Both socket identifiers are in the bind hash table and established hash table.

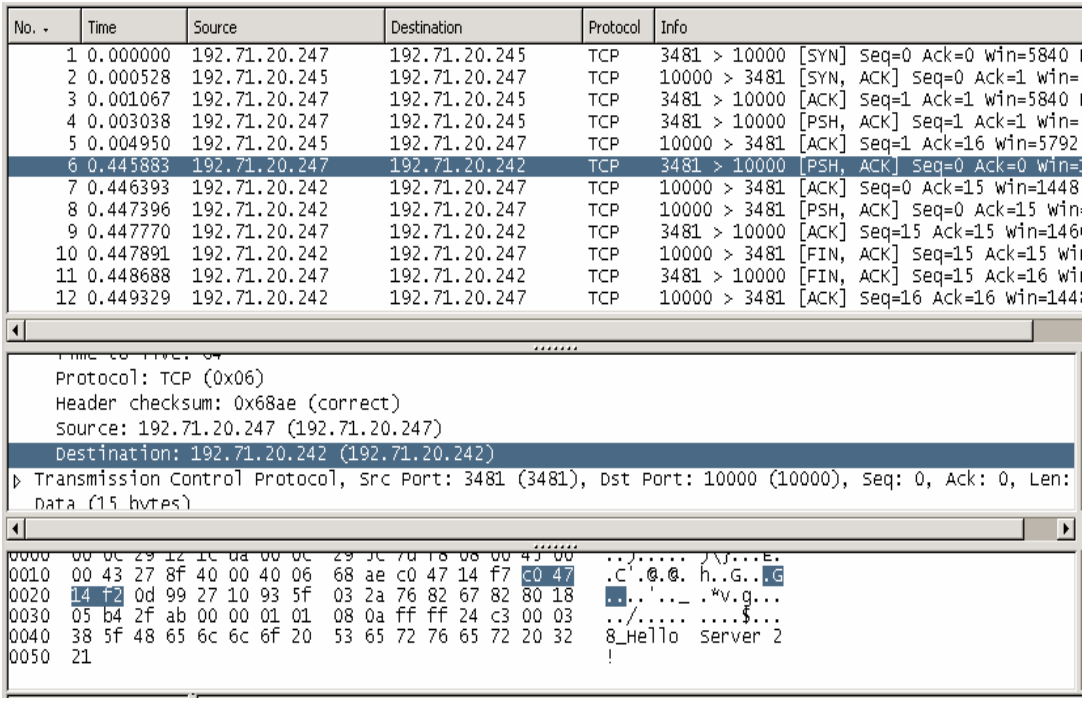

Figure 19 Rebind Source Address and Rebind Destination Address

4.4 Rebind destination port

In this test, the connection is setup by sending out the ACK of SYN ACK. The first message of “Hello Server 1!” from client to server is sent to 192.71.20.245 to port number 10000, which is show in the following figure.

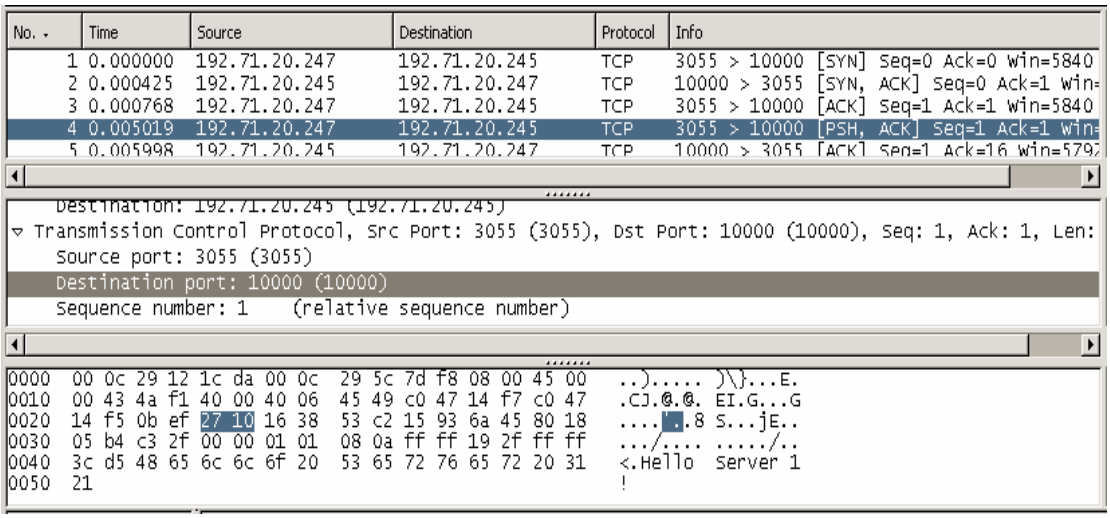

Figure 20 Before Rebind Destination Port

On the client side rebind happens after the client sending out the first message. Which means rebind happens after the 4th packet in the picture. After rebind, the client could send out the second data message of “Hello Server 2!” through the same socket. The message is sent to the same destination address 192.71.20.242 but with a different port number 1234, which is show in the following figure.

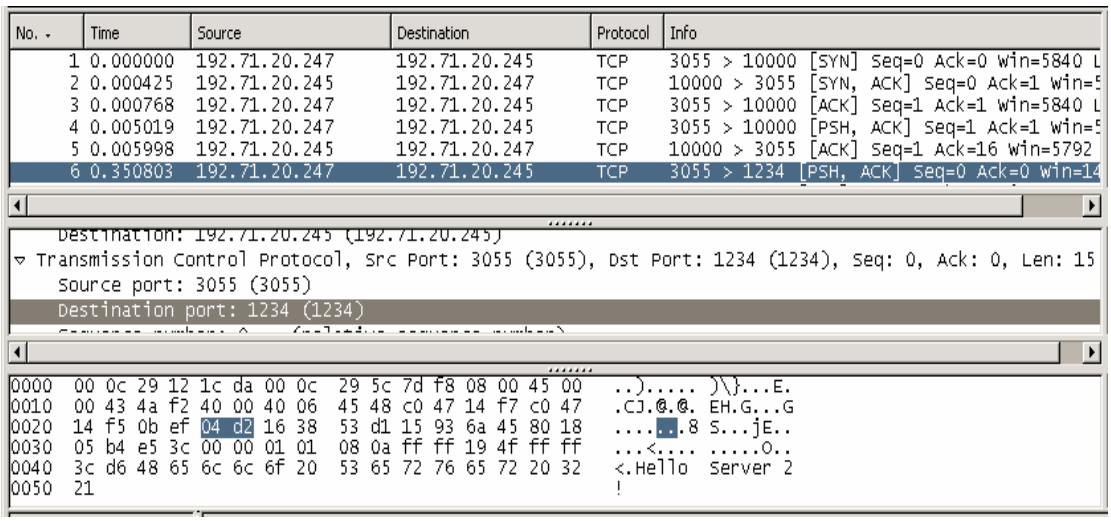

Figure 21 After Rebind Destination Port

The test shows that after rebind destination address, the socket state didn’t change. The

socket is still in Established state, data transmission could keep on through a different port number.

4.5 Rebind source port

This test is to verify the rebinding the source port, which is the listening port on the server side. Before the rebind, the server is using port 1000 as source port to send packets, which is shown in the following figure.

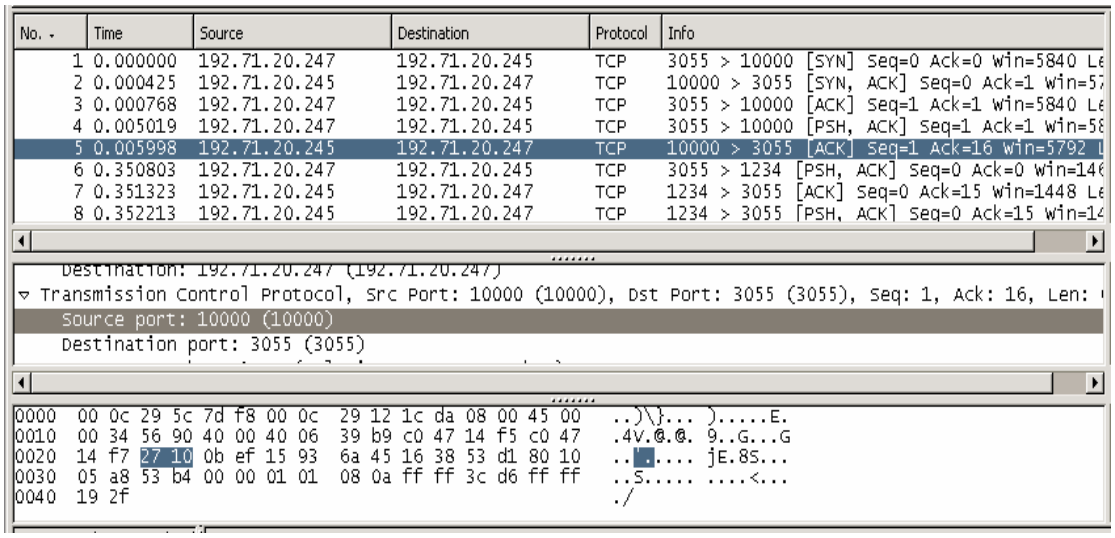

Figure 22 Before Rebind Source Port

On the server side rebind happens after sending out the ACK of the first data packet, which is the 5th packet in the picture. After rebind source port of the accepted socket, the server will keep on receiving packets on the same socket but with a different source port of 1234. From the 7th packet in the picture, we can see TCP is still in ESTABLISHED state, data communication could keep on.

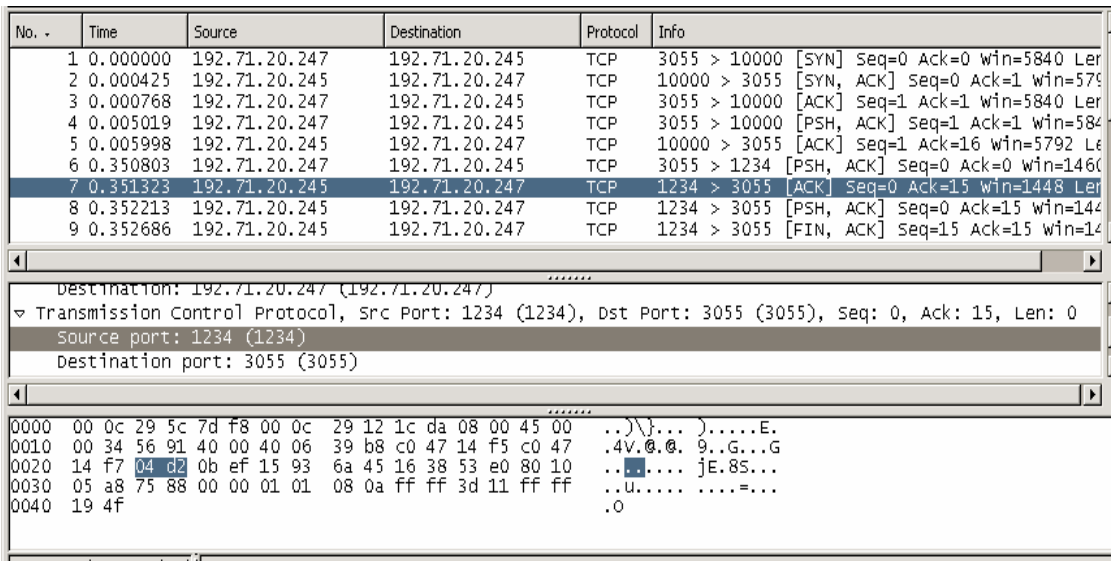

Figure 23 After Rebind Source Port

In the following figure, you can see the third message of “Hello Client 01!” from the server via the new port number 1234 to the client.

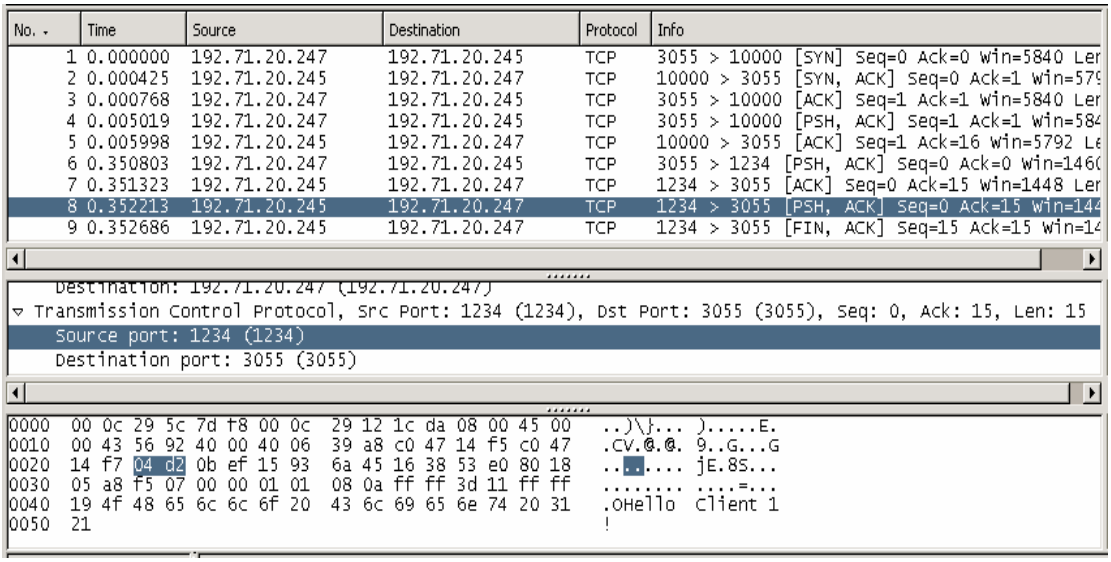

Figure 24 Send Data after Rebind Source Port

4.6 Rebind source port & destination port

The following is the complete network traffic log of rebind destination port on the client side and rebind source port on the server side. After the rebind on both end after the send/receive of the first data packet of “Hello Server 01!” client could keep on sending data through the same socket but with different destination port number, server could keep on receiving data through the same socket but with a different source port number of 1234. This verified that the rebind didn’t change the socket state. Both socket identifiers are in the bind hash table and established hash table. For the server side, it means the socket could keep in the original LISTENING and ESTABLISHED state after rebinding.

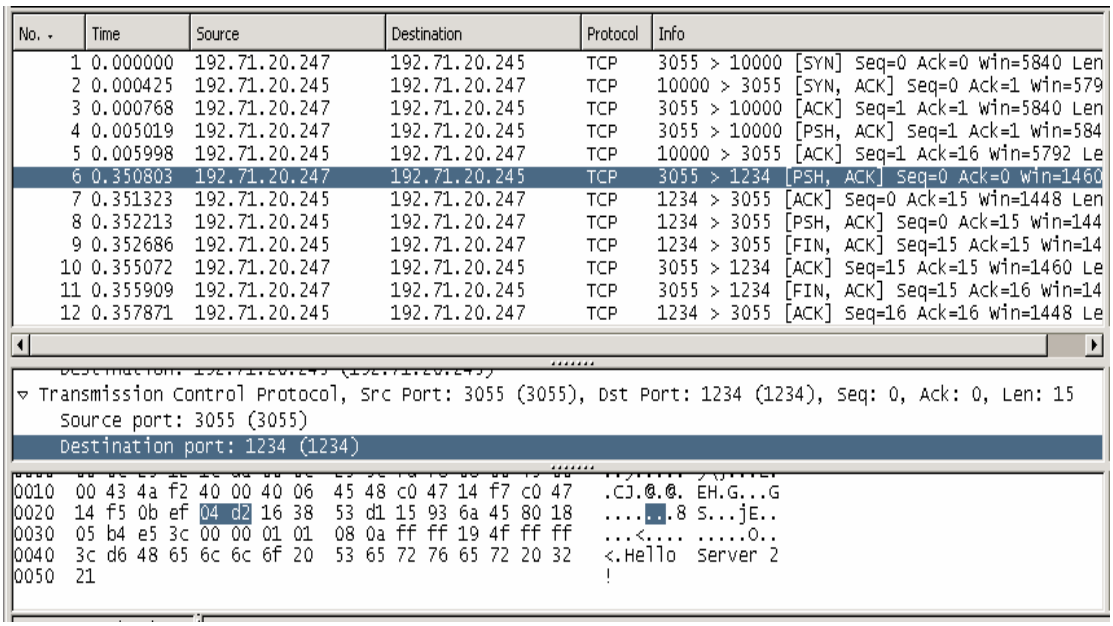

Figure 25 Rebind Source Port and Rebind Destination Port

4.7 Rebind source address/port & destination address/port

In the following test, after the client send out the ACK for the received SYN ACK, the connection is stabled. The ACK is the third packet in the following figure. From the client side, as shown in the fourth packet, the source address is 192.71.20.247, destination address is 192.71.20.245, source port is 1490 which is a randomly chosen local port, and destination port is 10000. In the data field, you can see "Hello Server 1!" The fourth packet is the first data message sent from the client to the server.

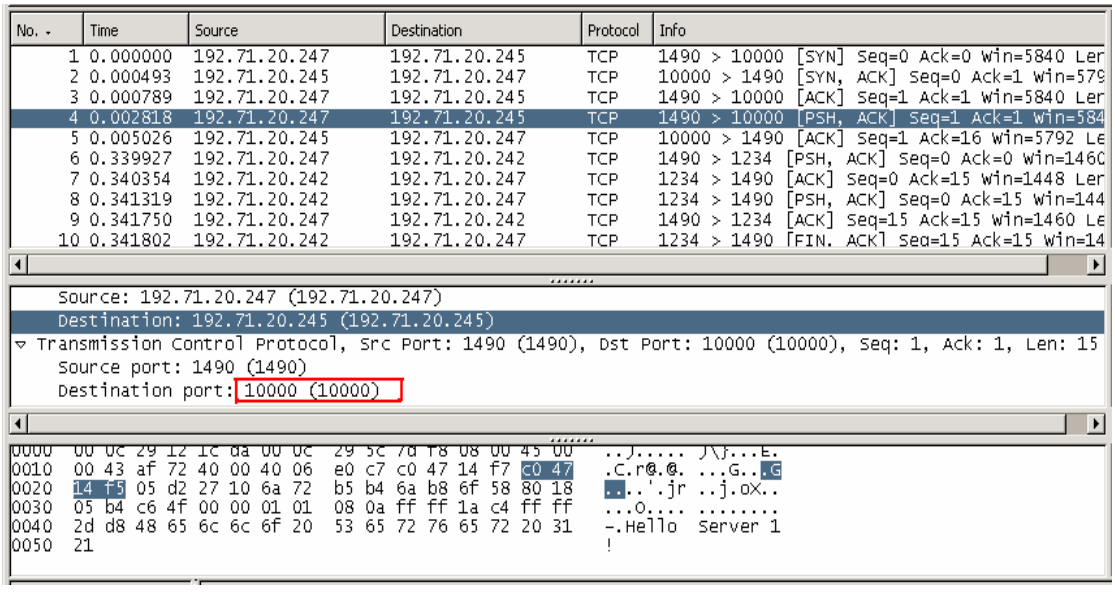

Figure 26 Destination Address/Port before Rebind

While for the fifth packet as shown in the following figure, it is an ACK of the previous

received data packet, sent from server to client. As shown in the packet header, source address is 192.71.20.245, destination address is 192.71.20.247, source port is 10000 and destination port is 1490. For the client, after getting the ACK for the first sent message, it will start to rebind to a new destination address (192.71.20.242) and a new port number (1234). For the server, after receiving the first data message, it will start to rebind to a new source address (192.71.20.242) and a new port number (1234).

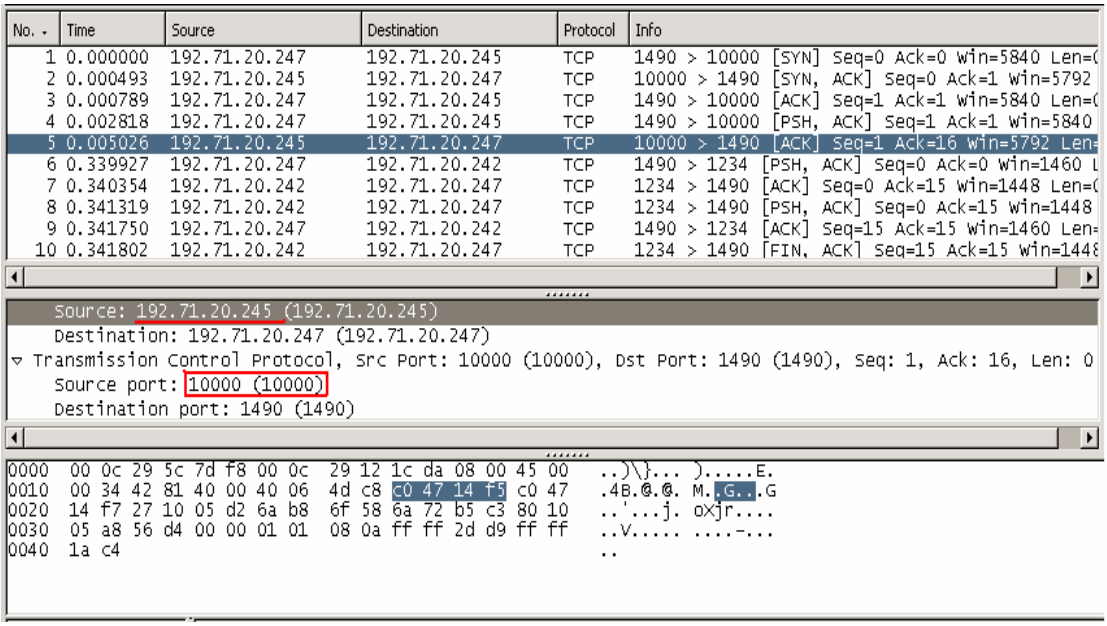

Figure 27 Source Address/Port before Rebind

After the client finished rebinding, it will send the second data message to the server through the same socket, which is shown in the sixth packet in the following figure. In the header field, it is shown that the new destination address is 192.71.20.242 and new destination port is 1234. From the log we can see that TCP is still in ESTABLISHED state, because there is no SYN or RST.

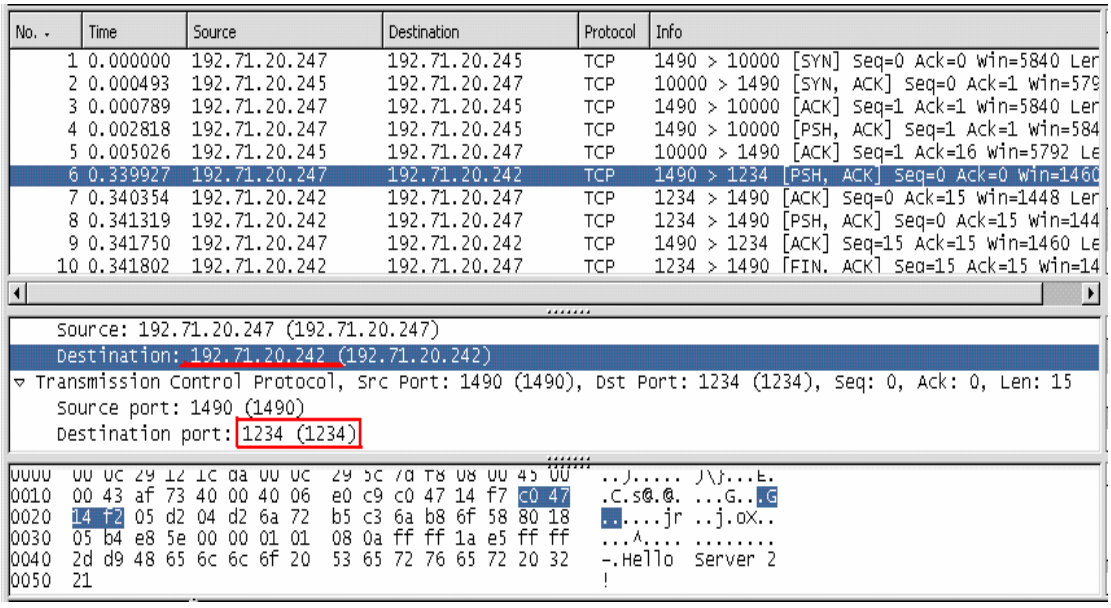

Figure 28 Destination Address/Port after Rebind

For the server, after rebinding to a new source address and new source port, it could keep on receiving data from the same socket. As show in the seventh packet in the following figure, a new source address of 192.71.20.242 and a new source port 1234 is shown in the header field.

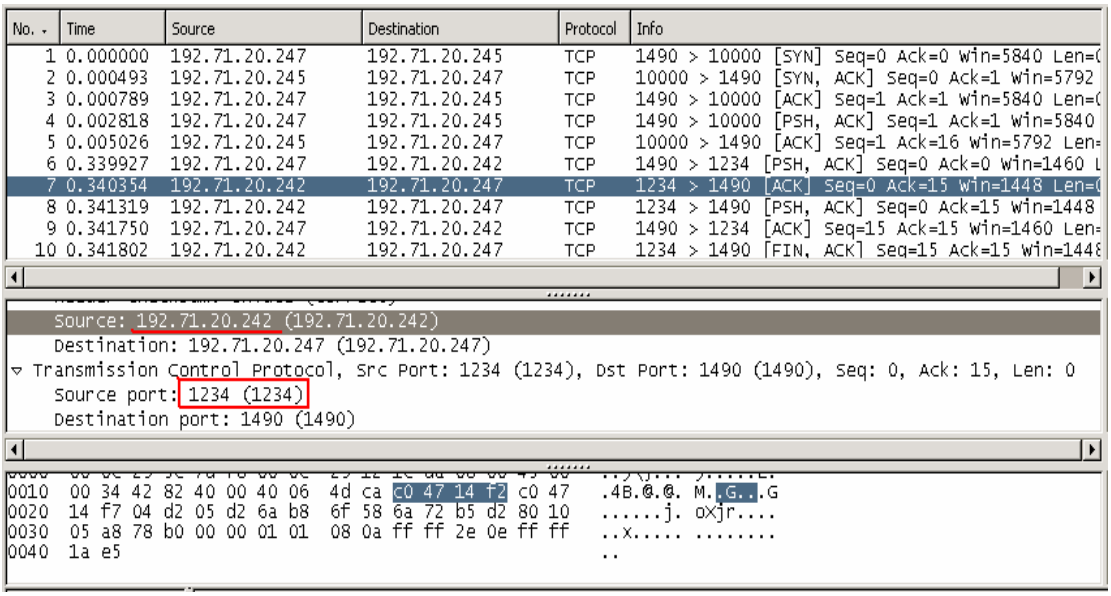

Figure 29 Sending ACK through new Source Address/Port after Rebind

After receiving the first packet after rebinding, the server is sending out one data message to the client to verify the other direction of data communication. From the following traffic log, we can see that the client could successfully receive from the server after rebinding as well.

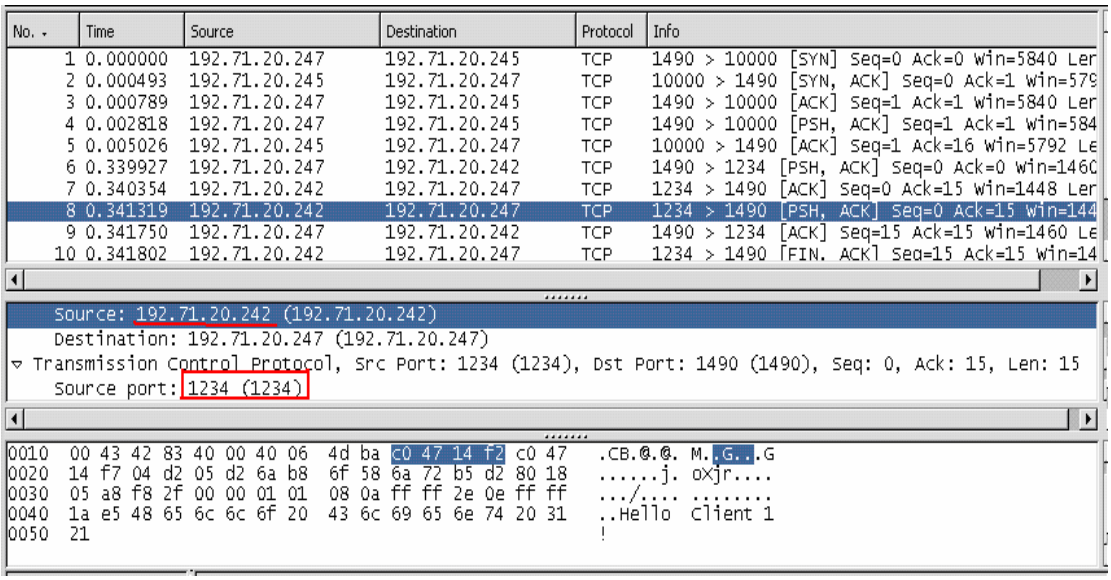

Figure30 Sending data through new Source Address/Port after Rebind

All the tests above shown that the rebind wouldn't change the TCP state, TCP state will be kept as before rebinding. The entire log is shown in the following figure, including the socket shutdown packets.

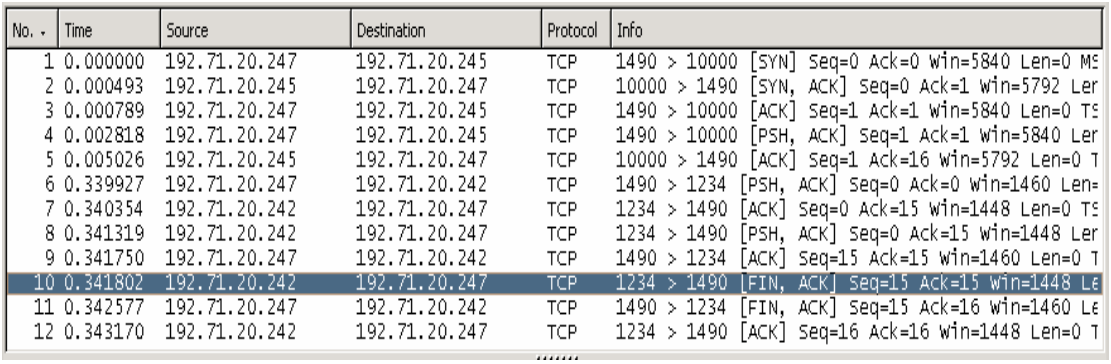

Figure 31 The full traffic log for rebind sadd/daddr/sport/dport

The above tests show that after any dynamic change of source/destination address or source/destination port of the socket, it won't change the socket state. The socket will keep in existing state before the rebinding and send/receive data could keep going as before.

5 Conclusions

Tests show that the extended kernel function of rebinding can successfully rebind the socket to a new address or port number without disturbing the connection state. These rebind functions work without impacting TCP/IP default behavior and will not influence applications which are not calling rebind functionalities. This means the network applications which are unaware of these extended rebind functions will executed as usual. There is certain limitation of the extended rebind functions. First rebind will not be able to know of the dynamic network changes. It doesn't have functions to collect or to react network events, like interfaces going up/down, address changes. The rebind functions need to work with certain event collector and dispatcher, which is the session layer in our proposal, to be able to make application be adaptive to dynamic network changes and as a result to provide communication continuity. The second limitation is rebind will discard certain data at the rebinding time. Due to the different reason of network changes; we decided to clear up the incoming/outgoing queue at the time of rebinding. In our system proposal, we put data continuity checksum in the session layer.

Based on the test results of our implementation of the Heterogeneous Socket and the extended socket rebind functions, together with the proposed session management framework, we can see the feasibility of the enhanced design of the TCP/IP stack architecture to support mobility for applications. The architecture will provide implanted mobility support into the TCP/IP stack without any proxy services or external servers. This infrastructure is also compatible with traditional BSD socket applications. Applications which are unaware of socket rebind functions will run as normal as before. With such mobility solution, mobility will be a natural property of any network application using TCP/IP stack. This architecture will dynamically react to the network changes and gracefully handle the unplanned disconnection or address changes. Application built on top this new architecture could suspend the connection at any time and resume it after any length of suspension. This architecture will also support the dynamic migration of service, as we could rebind services to run on different ports.

Compared to most related works (e.g. MIP, HIP, *i*3), our solution has a broader view of mobile events. We share the similar view of full range mobility with [6], but we took a step forward, the only focus on how to change IPv4 address. We aim at providing full range of mobility support inside the TCP/IP stack. Our implementation shows that our proposal is possible and the implementation will not break TCP protocol or standard BSD Socket semantics. Applications using TCP/IP protocol with original BSD socket functions will execute as before. Our work could be ported to any Linux machines with kernel 2.6.15.

6 Future work

This work has proved the concept to rebind TCP IPv4 socket to different address and port. But the rebind of IPv6 socket and between IPv6 and IPv4 protocols has not been done. This is planned as a future work. It is important to notice that Linux TCP/IP stack is designed for optimized performance and is difficult to use for the above purposes. Thus, another stack – FreeBSD operating system will be most likely used to continue studies and experiments for this approach. The rebinding between different transport protocols could also be considered in the future work. Security related issues should also be studied in deep details. Performance issue is very important and definitely should be a topic of the future work

7 List of Abbreviations

BSD: BSD Is the Berkley variant of UNIX, Berkley Software Distribution

BSD socket: A communications interface in Unix first introduced in BSD Unix

HIP: Host Identity Protocol http://www.ietf.org/html.charters/hip-charter.html

HLP: Host Layer Protocol [6]

MSOCK: An architecture for Transport Layer Mobility [5]

***i3*:** Internet Indirection Infrastructure [10]

sock: Linux kernel data structure, which is used to keep the state related information for a connection. It is mapped to the socket file descriptor in user space.

sk_buff: It represents sock buff, it is the kernel data structure used to keep the data to be send/receive and related state information.

Socket: A *socket* is one endpoint of a two-way communication link between two programs running on the network.

8 Reference

[1] “INTERNET MOBILITY: AN APPROACH TO MOBILE END-SYSTEM DESIGN”. Yuri Ismailov, Jan Höller. ICN22, April, 2006.

[2] E. Lear, Name Space Research Group, IRTF, draft-irtf-nsrg-report-02.txt “What's In A Name: Report from the Name Space Research Group”, Expires: August 14, 2002. http://search.ietf.org/internet-drafts/draft-irtf-nsrg-report-02.txt

[3] “Endpoints and Endpoint Names: A Proposed Enhancement to the Internet Architecture”. J. Noel Chiappa. Internet draft dated 1999 available at http://users.exis.net/~jnc/tech/endpoints.txt

[4] R. Moskowitz “Host Identity Payload and Protocol”, November 2001, Internet draft, http://search.ietf.org/internet-drafts/draft-moskowitz-hip-05.txt

[5] Pravin Bhagwat, David A. Maltz, Adrian Segall: MSOCKS+: an architecture for transport layer mobility. Computer Networks 39(4): 385-403 (2002)

[6] Alex C. Snoren, Hari Balakrishnan, and M. Frans Kaahoek, “Reconsidering Internet Mobility”. Proc. Of 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001.

[7] Pekka Nikander, Jukka Ylitalo, and Jorma Wall, "Integrating Security, Mobility, and Multi-Homing in a HIP Way," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03),* February 6-7, 2003, San Diego, CA, pp. 87-99, Internet Society, February, 2003.

[8] X. Qu, J. X. Yu, and R. P. Brent. *A Mobile TCP Socket*. In International Conference on Software Engineering (SE `97), San Francisco, CA, Nov. 1997.

[9] W. Richard Stevens. UNIX Network Programming. Pretice Hall, 1990.

[10] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, Sonesh Surana, "Internet Indirection Infrastructure," *Proceedings of ACM SIGCOMM*, August, 2002. http://i3.cs.berkeley.edu/

[11] Linux Source Kernel 2.6.15. http://www.kernel.org/pub/linux/kernel/v2.6/

[12] Miguel Rio, Tom Kelly, Mathieu Goutelle, “A Map of the Networking Code in Linux Kernel 2.4.20”, Technical Report DataTAG-2004-1, 31 March 2004 http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf

[13] ~Matthew G. Naugle,Van Nostrand Reinhold, “TCP/IP Illustration” Chapter 17. TCP: Transmission Control Protocol www.plinux.org/redgrid/books/tcpip_ill_1/tcp_tran.htm

[14] C. Perkins,”IP Mobility Support” RFC 2002, October 1996 http://www.faqs.org/rfcs/rfc2002.html