MOBILE COMPUTING WITH VIRTUAL NETWORK ADDRESS TRANSLATION

PROJECT REPORT

Submitted in partial fulfillment of the requirements
for the award of the degree of

P-1199

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING

OF THE BHARATHIAR UNIVERSITY, COIMBATORE

Submitted by

K. RAJESH (0027K0193)
B. JEEVAKA PRABU (0027K0172)

Under the Guidance of

Mr.M.NAGESHWARA GUPTHA, B.E.,
LECTURER, CSE DEPARTMENT

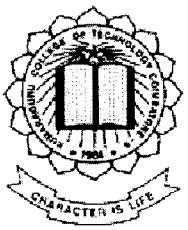
MARCH 2004

Department of Computer Science and Engineering

Kumaraguru College of Technology

(Affiliated to Bharathiar University)

COIMBATORE – 641 006



KUMARAGURU COLLEGE OF TECHNOLOGY

(Affiliated to Bharathiar University)
COIMBATORE – 641 006, TAMIL NADU, INDIA
Approved by AICTE, New Delhi - Accredited by NBA

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the project entitled

"MOBILE COMPUTING WITH VIRTUAL NETWORK ADDRESS TRANSLATION"

has been submitted by

K. RAJESH and B. JEEVAKA PRABU

in partial fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science and Engineering of the Bharathiar University, Coimbatore – 641 046 during the academic year 2003-2004

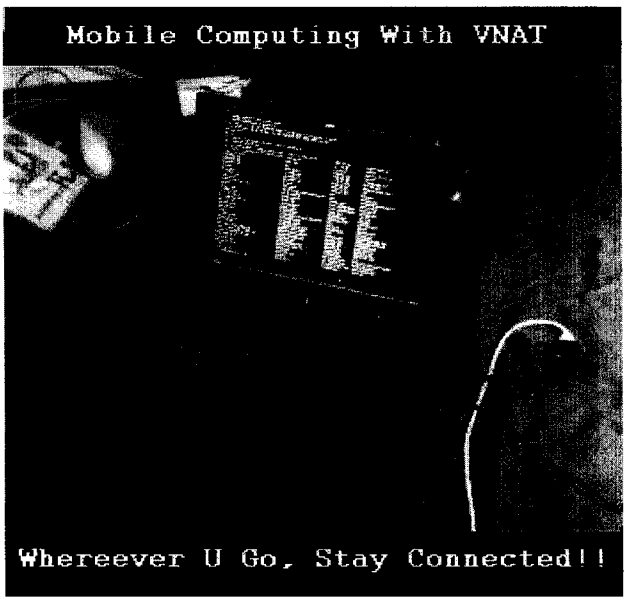
Head of the Department

Project Guide

Submitted for the university examination held on 23/3/'04

Internal Examiner

External Examiner



ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

We are greatly indebted to our revered Principal **Dr.K.K.Padmanabhan, Ph.D.**, Who has been the motivating force behind all our deeds.

We earnestly express our sincere thanks to our beloved Head of the Department Prof. **Dr.S.Thangasamy, Ph.D.**, for his immense encouragement and help and for being our source of inspiration all through our course of study.

We are much obliged to express our sincere thanks and gratitude to our beloved guide Lecturer **Mr.M.Nageswara Gupta, B.E.**, for his valuable suggestions, construction criticisms and encouragement which has enables us to complete our project successfully.

We gratefully thank Lecturers **Mrs.D.Chandrakala, M.E.**, and **Mrs.M.S. Hema, B.E.**, for extending their most appreciative and timely help to us.

We also thank all staff members of the Department of Computer Science and Engineering for all their encouragement and moral support.

We also extend our heartiest thanks to all our friends for their continuous help and encouragement throughout the course of study.

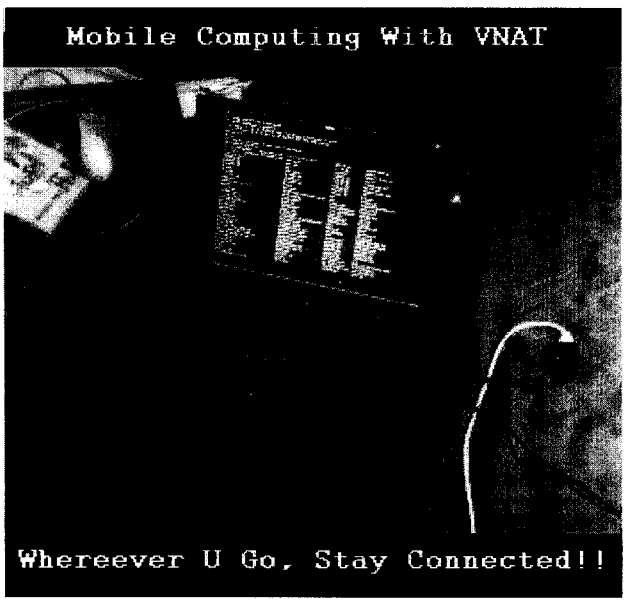
SYNOPSIS

This project work entitled "Mobile Computing with Virtual Network Address Translation (MC-VNAT)" is a novel architecture that allows transparent migration of end-to-end live network connections (i.e. the communicating host can move to any other network and get new IP address without shutting down the existing network connections). VNAT virtualizes network connections perceived by transport protocols so that identification of network connections is decoupled from stationary hosts. Such virtual connections are then remapped into physical connections to be carried on the physical network using network address translation.

VNAT requires no modification to existing applications, operating systems, or protocol stacks. Furthermore, it is fully compatible with the existing communication infrastructure; virtual and normal connections can coexist without interfering each other. VNAT functions entirely within end systems and requires no third party services.

CONTENTS

CONTENTS



INTRODUCTION

1.INTRODUCTION

Mobile computing is a coming reality, fueled in part by continuing advances in wireless transmission technologies and handheld computing devices. As systems are getting networked increasingly, mobility in data networks is becoming a growing necessity.

Examples of this demand include laptop users who would like to roam around the network without losing their existing connections, system administrators of network service providers who would like to move running server processes from one machine to another due to maintenance or load balancing requirements without service disruption, and scientific users who would like to move their long-running distributed computations off to another machine due to faulty processor or power failure without having to restart the computation all over again.

However, data networks today offer very limited support for mobility among communicating devices. One cannot move either end of a live network connection without severing the connection.

The lack of system support for mobile data communication today is due to the fact that the current de facto worldwide data network protocol standards, the Internet Protocol (IP) suite, were designed with the assumption that devices attached to the network are stationary. In addition, higher layer protocols such as TCP/UDP inherit this assumption. The key problem is that network connection properties are shared among many entities across network protocols, transport protocols, and applications.

For example, TCP/UDP uses IP addresses to identify its connection endpoints and applications use sockets, which are typically bound to IP addresses and TCP/UDP

port numbers, for their network I/O. Clearly, such information sharing makes it very difficult to change the network protocol endpoints without disrupting the transport protocols and/or the applications. A large amount of research has been conducted in an effort to overcome this deficiency; However, previous approaches either require changes to network or transport layer protocols, or suffer from substantial performance penalties which limit their deployment.

To effectively support efficient transparent migration of end-to-end live network connections without any changes to existing network protocols, Virtual Network Address Translation (VNAT) can be used. VNAT is a novel Mobile Computing architecture that enables connection mobility for a spectrum of computation units, ranging from a single process to the entire host. VNAT utilizes three key mechanisms to enable transparent live connection mobility: connection virtualization, connection translation, and connection migration.

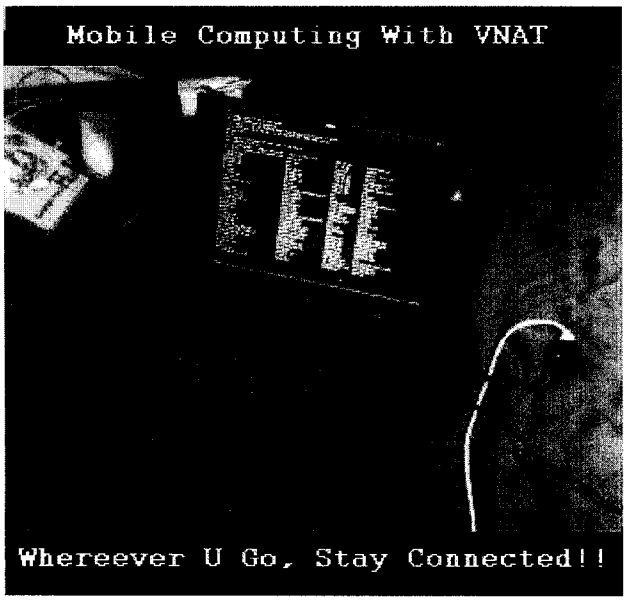
VNAT connection virtualization virtualizes end-to-end transport connection identification by using virtual endpoints rather than physical endpoints (e.g., IP addresses and port numbers). As a result, connection identifications no longer depend on lower layer network endpoints and are no longer affected by the movement of network endpoints.

VNAT connection translation translates virtualized connection identifications into physical connection identifications to be carried on the physical network. As connections migrate across the network, their virtual identifications never change. Instead, they are mapped into appropriate physical identifications according to the endpoints' attachment to the physical network.

VNAT connection migration keeps states and uses protocols to automate tasks for connection migration such as keeping connection alive, establishing a security key,

locating the migrated endpoint(s), and updating virtual-physical endpoints mappings.

VNAT is fully compatible with and does not require any modifications to existing networking protocols, operating systems, or applications. It can be incrementally deployed and operates entirely within communicating end systems without any reliance on third party services or proxies. VNAT assumes no specific transport protocol semantics and therefore can be easily adapted to any transport protocol. It also supports both client and server mobility and does not put any restriction on the mobility scope. We have implemented VNAT as an application in Linux Kernel 2.4.



EXISTING SYSTEMS

2.EXISTING SYSTEMS

A variety of approaches have been taken in previous work in providing communication mobility in current (IP) data networks. These approaches can be loosely classified as

- Network layer mobility mechanisms
- Transport layer mobility mechanisms
- Proxy-based mechanisms and
- Socket library wrapper mechanisms

Mobile IP:

Mobile IP is the most well-known network layer mobility mechanism. Mobile IP allows a host to move freely across the Internet without having to change its assigned "home" IP address. As a result, the movement of the host is transparent to layers above network layer. However, Mobile IP only provides communication mobility at the granularity of an entire host. It does not provide finer granularity mobility of individual end-to-end connection between two applications because network protocols are indifferent to higher layer "connections".

Unlike VNAT, Mobile IP uses a residual home agent that is a single point of failure and causes "triangle routing" where traffic destined to a mobile host must all go through its home agent when the mobile node is away from its home. Triangle routing incurs high traffic delay and wastes network resources. Although it can be alleviated by routing optimization, this solution requires additional changes to the non-migrating nodes.

VNAT incurs almost no overhead for new connections started after migration; Mobile IP incurs tunneling overhead for all traffic between the mobile node and the non-migrating node. Unlike VNAT, Mobile IP requires network layer protocol and infrastructure changes that are costly and make it very difficult to deploy.

Migrate:

Migrate is a transport layer mobility architecture that allows migration of individual end-to-end connections between two applications. Since traditional transport protocols are not built with mobility in mind, Migrate introduces a new TCP option to support suspending and resuming TCP connections. Migrate does not support migration of TCP connections for which both endpoints move simultaneously. Unlike VNAT, Migrate is TCP-specific and requires transport layer protocol changes which make it difficult to deploy.

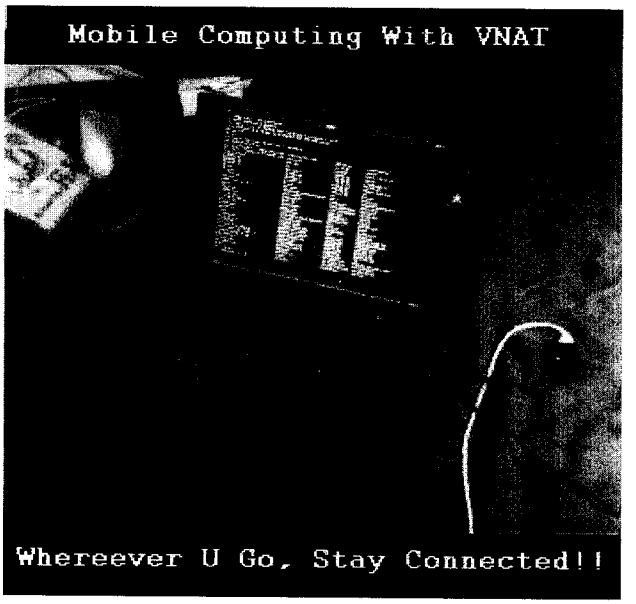
- Mobile IP and Migrate also provide mechanisms for mobile host location. Mobile IP uses the notion of home and foreign agents to also provide mobile host location technologies. Migrate, uses dynamic DNS updates. But VNAT focuses on the "tracking" (preserving an end-to-end connection once it is established) aspect of connection mobility while being compatible with and taking advantage of existing mobile host location technologies, such as those used in Mobile IP and Migrate.

MSOCKS:

MSOCKS is a proxy-based mobility architecture based on the TCP Splice technique. Essentially, a single TCP connection between a mobile client and a stationary

server is spliced by a proxy in the middle into two separate TCP connections. The proxy handles the disconnecting and reconnecting of the client-proxy half of the TCP connection when the mobile client moves and makes the single TCP connection between the mobile client and the stationary server appear to be intact.

Due to its reliance on TCP Splice, MSOCKS assumes TCP as the transport protocol. MSOCKS is designed to allow client mobility only; and the mobility is usually confined within the subnet for which the proxy is acting as the gateway. The use of a proxy avoids transport protocol changes but can limit scalability and performance.



VNAT ARCHITECTURE

3.VNAT ARCHITECTURE

The VNAT architecture is based on the surprisingly simple idea of introducing a virtual address to identify a connection endpoint. In current IP networks, it is impossible to keep end-to-end transport connections alive when one or both connection endpoints move because physical network protocol endpoints are used by transport protocol to identify its connections.

VNAT uses virtual addresses to break this tie between the transport protocol and network protocol by virtualizing the transport endpoint identification. Once the transport endpoint identification is made independent of network endpoint identification, the lifetime of a transport connection is no longer limited by changes in network endpoints.

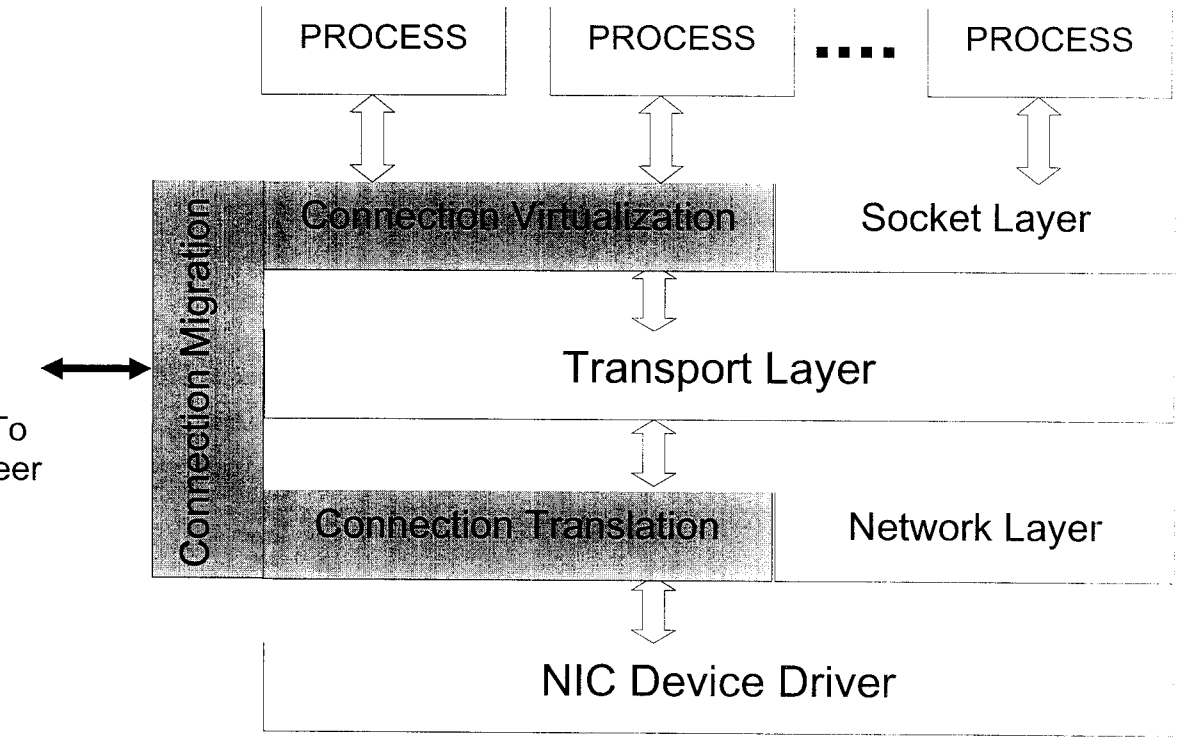
The VNAT architecture can be decomposed into three components, as shown figure.

- Connection Virtualization
- Connection Translation
- Connection Migration

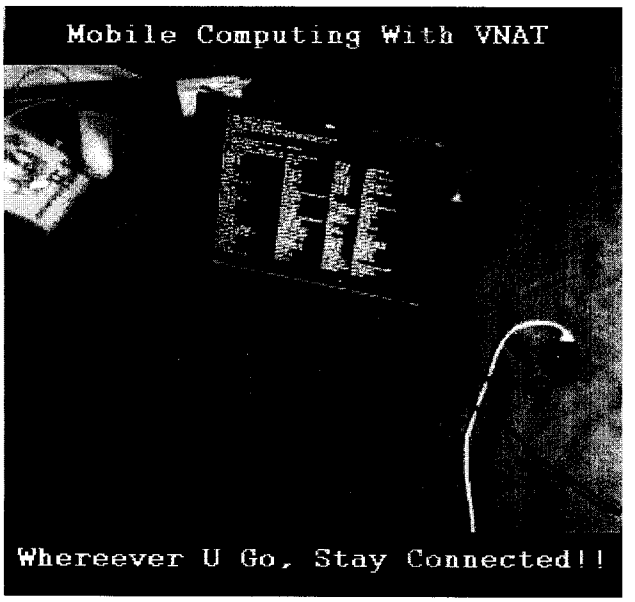
VNAT connection virtualization is the mechanism used to allow virtual rather than physical addresses to be used for the connection endpoints. VNAT connection translation is the mechanism used to maintain proper association and mapping between the virtual and the physical identifications because only real network endpoints can be used on the physical network to carry packets. VNAT connection migration facilitates the automation of securing the migrating connection, keeping alive connections during migration, and updating virtual-physical address mapping after migration.

These components can be implemented in a single module that is simply

downloaded, installed and executed on end systems without any need to modify or reconfigure the network infrastructure.



VNAT ARCHITECTURE



CONNECTION VIRTUALIZATION

4.CONNECTION VIRTUALIZATION

The function of VNAT connection virtualization is to virtualize the endpoints used by the transport protocol to identify its end-to-end connections. An endpoint is virtualized by identifying it with a virtual identification, which is a fictitious identification not tied to any real physical endpoint. We refer to an end-to-end transport connection identified by a pair of virtual endpoint identifications as a virtual connection, while a connection identified by a pair of physical endpoint identifications a physical connection. In VNAT, virtual endpoint identifications do not change during the lifetime of a virtual connection, even if the physical endpoints of the underlying physical connection change. Since a virtual connection is not tied to specific physical endpoints, it can be moved freely among physical endpoints without changing its virtual endpoint identifications.

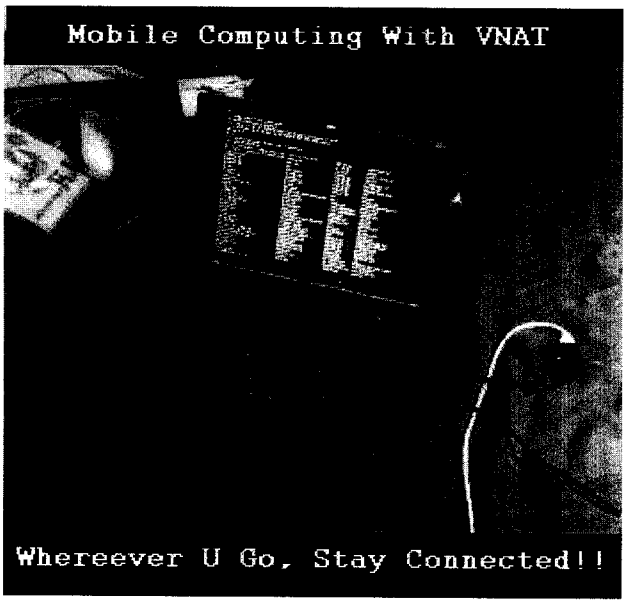
Depending on the specific transport protocol, a virtual identification may take different forms. For example, with TCP/UDP, a virtual identification is the combination of a network IP address and a transport port number, both of which are virtualized by VNAT. Throughout the paper, we use the generic term "virtual address" to refer to a virtual identification of a combined virtual IP address and virtual port number. However, we left out the virtual port number for simplicity. The same holds for the term "physical address", which is the combination of a physical IP address and a physical port number.

VNAT connection virtualization provides a simpler approach than previous mobility approaches such as proxy based mechanisms and socket library wrappers. All VNAT does is to convince TCP to use virtual IP addresses and ports rather than physical IP addresses and ports for connection identification. TCP treats a virtual connection exactly the same as any other physical connections. In fact, TCP does not even know the connection is virtualized. All TCP semantics apply equally to the packet flow on a virtual connection. Also note that the virtualization is done completely transparently to both the

application and the transport protocol and requires no modification to either party. Unlike previous approaches that strive to hide physical IP address changes from applications when connections migrate, the philosophy behind VNAT is to avoid such transport layer changes in the first place.

Although theoretically the virtual addresses can be anything that is accepted by the transport protocol, careful selection of the virtual addresses can greatly simplify the system. Since both parties to a connection must be aware of the same virtual address pair, there needs to be some way for each party to inform the other of its virtual address. If an arbitrary choice of virtual addresses is used, additional communication and delay will be incurred for every connection so that both parties to a connection can learn the virtual address chosen by the other side. This extra delay would be excessive if it was required for all connections, especially for short-lived connections in wide-area networks that never migrate.

The extra delay can be avoided by simply selecting the virtual addresses to be the initial physical addresses associated with a connection. In this way, no extra communication is required because the virtual addresses are essentially known beforehand. In effect, VNAT treats all physical connections as initially "implicitly" virtualized, with the virtual addresses for the connections being the same as the physical addresses. When a connection endpoint moves to a different physical endpoint, the virtual address for the endpoint does not change and is still the same as the initial physical address, not the new physical address.



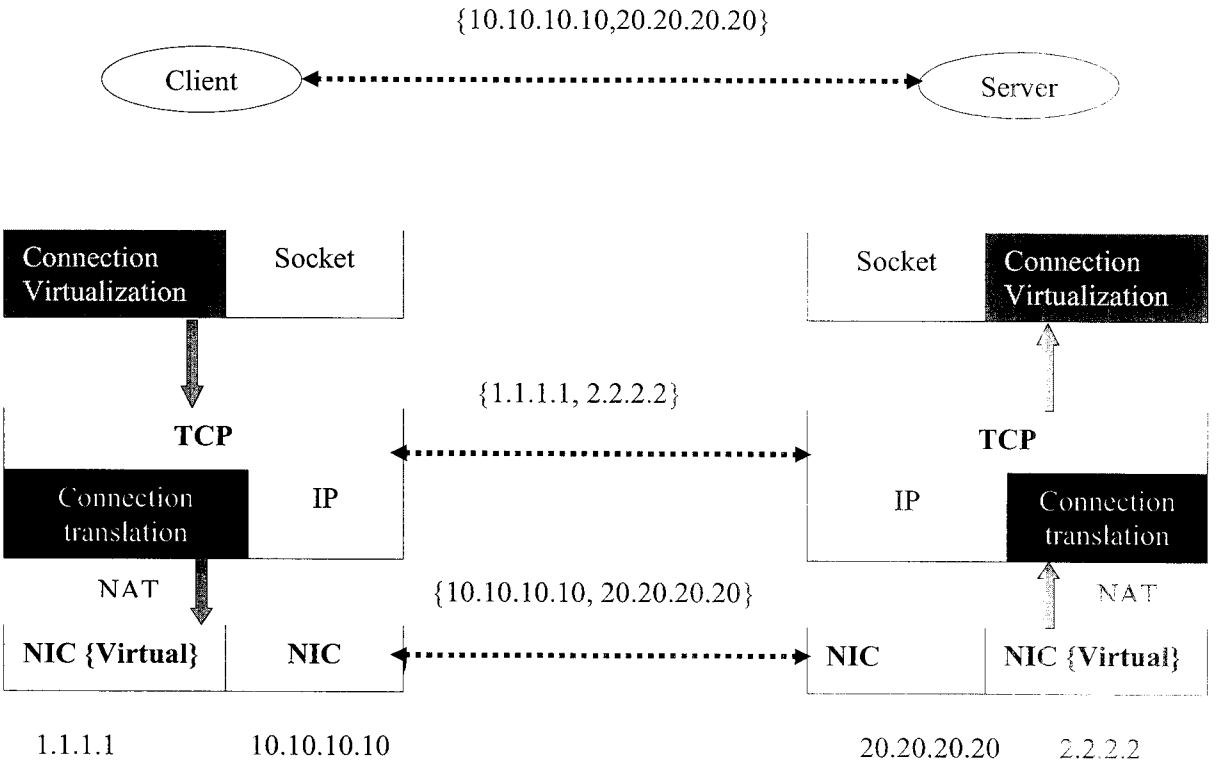
CONNECTION TRANSLATION

5.VNAT CONNECTION TRANSLATION

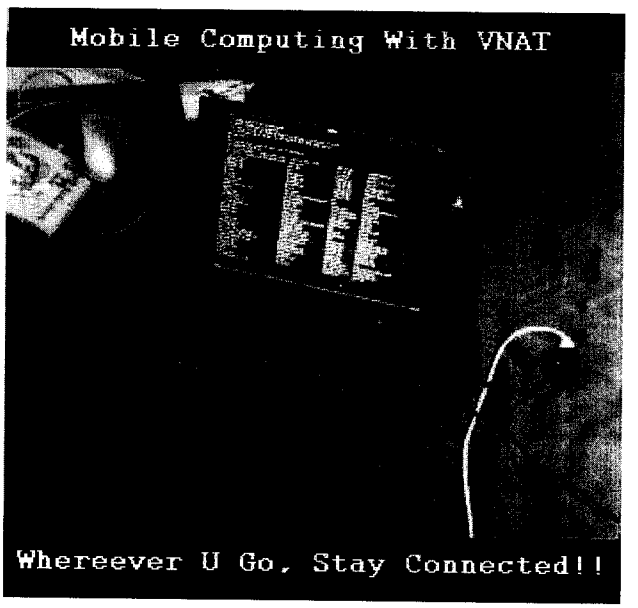
Once a TCP connection is virtualized, it is ready to be migrated anywhere without paying any attention to the physical IP addresses to which the connection endpoints are attached. But connection virtualization alone is not yet sufficient to allow packets to flow over a virtual connection. VNAT connection translation makes it possible to communicate over virtual connections by translating a set of virtual addresses associated with virtual transport endpoints to and from a physical address associated with a physical network endpoint. VNAT connection virtualization creates the virtual addresses while VNAT connection translation maintains the proper association and mapping between the virtual addresses and the physical network addresses.

VNAT connection translation is done using well-known Network Address Translation (NAT) technology, which is commonly used in the network layer today. However, instead of translating a set of "private" addresses on the LAN side to and from a "public" address on the WAN side, VNAT uses NAT concept to translate between virtual and physical addresses. Note that VNAT connection translation is done transparently below the transport protocol and therefore requires no modification to the transport protocol.

Using the initial physical addresses of a connection as its virtual addresses has benefits for VNAT connection translation as well. Because the virtual and physical addresses are the same for a connection that does not migrate, there is no need to perform connection translation for connections that have not migrated. As a result, no translation overhead will ever be imposed on a connection so long as it does not move. Connection translation is only necessary for connections after they migrate, so only migrated connections will incur any connection translation overhead.



CONNECTION TRANSLATION



CONNECTION MIGRATION

6.CONNECTION MIGRATION

VNAT connection migration builds on VNAT connection virtualization and translation to provide the mechanisms necessary to actually move a connection from one machine to another. VNAT connection virtualization and translation make an end-to-end transport connection "migratable" (can be freely moved) and "alive" (packets can flow). VNAT connection migration enables connections to be suspended at one location and resumed at another.

To suspend a connection, VNAT does not need to do anything at all. But it does provide optional functionality to establish a secret key for security protection and to activate mechanisms (called connection migration helpers) that keep the migrating connection alive. To resume a suspended connection, VNAT verifies the security protection key if it is available, updates the appropriate virtual-physical endpoint mappings, and deactivates the connection migration helper. The various functions that can be included in typical connection migration are described in the following sections.

Suspending a connection:

VNAT is designed to work with a variety of mechanisms for suspending and migrating a connection endpoint. A connection endpoint may move when the hardware associated with the connection moves its network location or when the process associated with the connection moves from one machine to another. For example, the connection endpoint may move because its host laptop is suspended, disconnected from the network, and moved and resumed in another place.

Alternatively, the endpoint may move with a process that has been moved via an operating system process migration mechanism. Yet another way in which a connection

endpoint may move is to simply unplug the network cable of a host and move the host. VNAT simply needs to be notified of the event of suspending a connection. Because a connection may be suspended and migrated without any notification as in the case of unplugging the network cable of a host and moving it, VNAT is designed to provide connection migration without any required processing or saving of state at the time a connection is suspended. VNAT can perform all of its necessary processing for connection migration when a connection is resumed.

However, VNAT can provide additional benefits if it is able to perform some functions when a connection is suspended. The various functions may be,

- Establishing a protection key

After a connection endpoint migrates, it needs to inform the other endpoint to update the virtual-physical address mapping for a virtual connection. This potentially leaves the door open for a malicious process to "hijack" the network connection of another process. For example, the malicious process can send a fake update message to a server, causing the server to map a virtual connection to a physical connection that is destined to the malicious process. Thus traffic intended for the original process is now being sent to the malicious process.

To address the problem of connection hijacking, VNAT provides the ability to protect each virtual-physical address mapping for a virtual connection by a secret key shared between the two endpoints. The key is established between the two endpoints at the time when a connection is suspended for migration. At the time of resuming a migrated connection, exchange of virtual-physical address mapping update messages is protected by the mutual authentication of the two endpoints through the secret key.

- Keeping the connection alive

When one endpoint of a live network connection is suspended for migration, it may be necessary to provide additional functionality at the non-migrating endpoint in order to preserve the migrating connection. Both the transport protocol and the application may have their own mechanism to keep alive what they perceive as a "connection". These mechanisms may need to be disabled if a connection is suspended for a long time in order to preserve the connection beyond the timeout limit of these mechanisms.

In order to keep the connection alive, the following techniques can be used,

- Change the timer limits to infinity
- Stop the protocol timer
- Suspend the migrating process, so that it doesn't get kernel time

Resuming a Connection:

Resuming a connection is the reverse of suspending a connection. If a connection endpoint is migrated by check pointing a process, the saved the process states are restored and the process is restarted. If an entire host was suspended, the states of the entire host are restored and the host is resumed. If it is just the network cable that was unplugged, one can simply just reconnect the network cable. VNAT simply needs to be notified after the appropriate states have been restored but before the process or host is resumed.

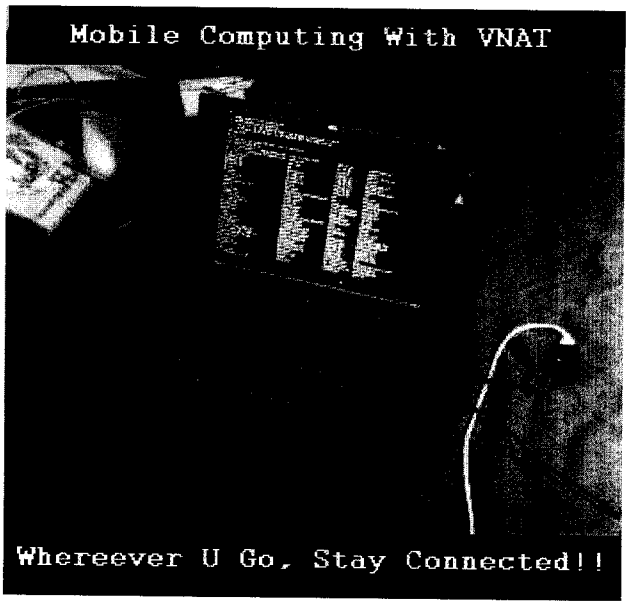
- Verifying protection key

When only one endpoint of a connection migrates, it is trivial for the migrated endpoint to find its peer because the existing connection states tell where its peer is. After

the migrated endpoints locate each other and before any virtual physical address mapping update for virtual connections can happen, the two endpoints must verify, for every virtual connection to be updated, the security protection key they established at the time when the connection was suspended. The exact process obviously depends on the particular security mechanism in use. If no security protection key was established when the connection was suspended and if VNAT is not configured to guarantee security, then there is no security key to verify and the connection is simply resumed.

- Updating Virtual-Physical End point Mappings:

When a connection endpoint migrates to a new location, its virtual address stays unchanged and therefore the virtual connection will stay intact. However, this virtual address now has to be mapped to and from a new physical address for the continued flow of packets over the virtual connection.



IMPLEMENTATION DETAILS

7.IMPLEMENTATION DETAILS

We have implemented VNAT as a user application which runs in Red Hat Linux 9.0 (Kernel Version 2.4.20-8). The whole VNAT architecture is divided into the following modules and then integrated to final application, so that changes can be easily made. The different modules are

- Kernel Timer Tuning
- Virtual Address Manipulation
- Network Address Translation
  - Source NAT
  - Destination Nat
- VNAT Peer

7.1. Kernel Timer Tuning:

In order to keep the TCP connections alive during the roaming period, three different techniques were discussed in the previous sections. Out of those three we have chosen the timer modification technique. The three parameters in the Linux kernel which governs the connection timeout declaration are

- net.ipv4.tcp_keepalive_time
- net.ipv4.tcp_keepalive_probes
- net.ipv4.tcp_keepalive_intvl

1.1 Net.ipv4.tcp_keepalive_time

This parameter specifies the delay after which the status of the connection has to

checked, when the packet loss exceeds a certain limit. The default value of this parameter is 7200 Seconds (i.e. approximately after two hours). Suppose if the kernel observes certain amount of packet loss, it will wait for two hours and then send a probe to check the presence of the connection. If the probe fails, then the connection will be declared 'timed out', after a certain number of probes as specified by the other two parameters.

So, in order to prevent this connection timeout during the roaming period, we have to change the value of this parameter to a very large value which will count for hours or days.

net.ipv4.tcp_keepalive_time = 7200 [2 hours]

To a value something like,

net.ipv4.tcp_keepalive_time =999999 [12 days]

1.2 Net.ipv4.tcp_keepalive_probes

This parameter specifies the number of times the status of the connection has to be checked before confirming the 'connection time out'. The default value of this parameter is 9 times. Thus by default the kernel will probe the connection status nine times including the first probe sent after the period mentioned by the net.ipv4.tcp_keepalive_time parameter. We have changed the value of this parameter to a very large value in addition to the change in net.ipv4.tcp_keepalive_time, so as to achieve maximum roaming period.

net.ipv4.tcp_keepalive_probes = 9

To a value something like,

net.ipv4.tcp_keepalive_probes =999

1.3. Net.ipv4.tcp_keepalive_intvl

This parameter specifies the interval between the two status probes. The default value of this parameter is 75 seconds. (i.e., kernel sends the status probes after delaying the number of seconds mentioned by this parameter, as many times as specified in net.ipv4.tcp_keepalive_probes. We have changed the value of this parameter also, in addition to the above mentioned parameters.

net.ipv4.tcp_keepalive_intvl = 75

To a value something like,

net.ipv4.tcp_keepalive_intvl =999

- These parameters are modified with the help of the 'sysctl' system call.

1.4 'Timeout' Declaration:

1.4.1 Default:

By default the total time for which the connection won't be declared as 'timeout', even after observing the packet loss is

Timeout =net.ipv4.tcp_keepalive_time + (net.ipv4.tcp_keepalive_probes * net.ipv4.tcp_keepalive_intvl)

=7200 + 9*75

= 2.5 Hours

1.4.2. Modified:

After modifying the value of these parameters we are able to achieve a very large delay, for timeout declaration. Since in normal conditions this will was the system resources and decrease the performance, the value of these parameters are restored to the normal values once the connections are resumed.

Timeout =net.ipv4.tcp_keepalive_time + (net.ipv4.tcp_keepalive_probes * net.ipv4.tcp_keepalive_intvl)

=999999+999*999

= 23 days.

Thus we have tuned our VNAT application to achieve a roaming period of about 23 days. This can even be increased with the increase in the value of these kernel parameters. But in normal situations this much lager roaming period is not necessary.

7.2. Virtual Address Manipulation:

This deals with the manipulation of the mappings between the virtual and the real IP addresses. Suppose if the user is going to connect to a new system, he has to inform the VNAT peer about the IP address of the destination machine.

The VNAT peer will create a mapping in its database with the virtual address as the same as the real IP address for simplicity and once the IP address of any communicating machine changes, then the mapping will be altered with the new IP address. The VNAT peer will immediately inform the destinations VNAT peer and inform its IP address, so that the remote machine also adds a map in its database. These mappings will be changed whenever any IP address changes and when there comes a request from any other VNAT peer.

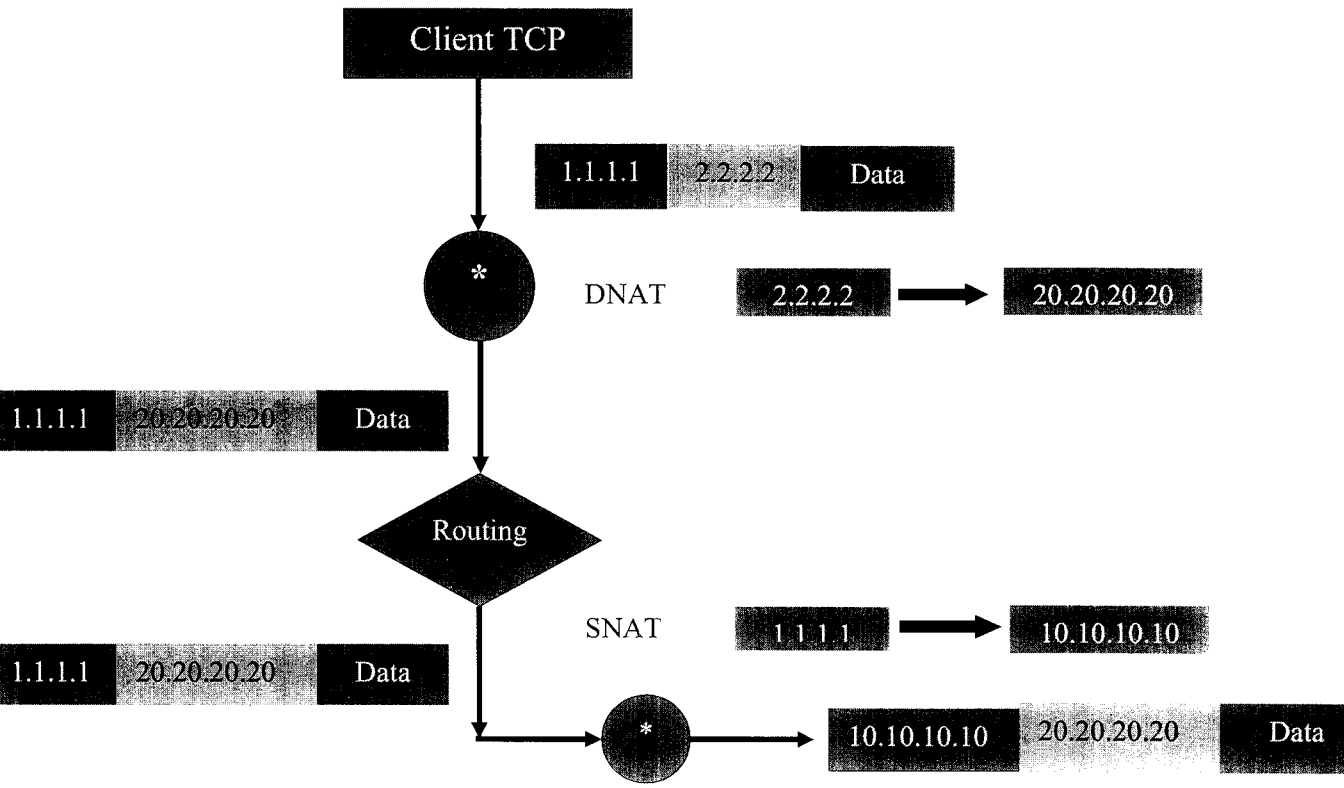
7.3. Network Address Translation:

The major implementation issue in the VNAT implementation is the network address translation. Since the connections above the IP Layer are virtualized, the addresses in the packets leaving the system should be changed to the real IP addresses, in order to route the packets to the correct destinations. The network address translation has two roles, one dealing with the source (SNAT) and the other dealing with the destination address (DNAT).

The Network Address Translation in Linux is controlled by the kernel service called 'IP Table' or 'NAT Table'. NAT allows us to modify the source or destination address or port of a packet, allow it to be redirected, or so that it appears to come from another system. The most popular use of NAT is for IP Masquerading, where we have a LAN of systems using private IP addresses, which have network connectivity through a gateway which performs NAT on their packets so that all connections appear, at least to the outside world, to have come from the gateway system.

The NAT table has three different chains, each of which is checked at a particular position in the routing of a packet. The first is PREROUTING, whose rules are checked for a match before the system attempts to route the packet anywhere. PREROUTING is where we perform destination NAT, or DNAT. We can also perform DNAT within the OUTPUT chain, which is for packets which originated locally, as they are never checked by PREROUTING. Finally we have POSTROUTING, which is the last chain checked for a rule match, which is where source NAT, or SNAT is performed.

The source network address translation is performed by the roaming system in order to change the source IP address in each of the outgoing packet with the IP address used to establish the connection, whereas the Destination network address translation is used by the static system in order to change the destination address destined for a roaming host with the newly assigned IP address.

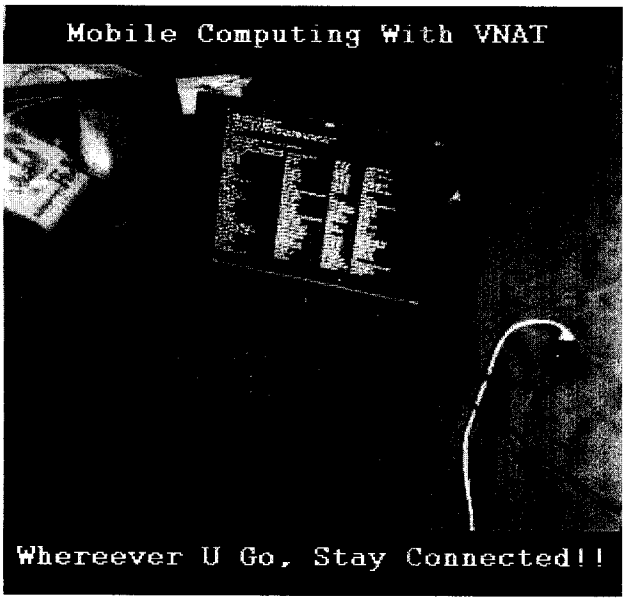


NETWORK ADDRESS TRANSLATION

7.4. VNAT Peer:

We have implemented the VNAT Peer service as an integration module of all the above individual modules. The special thing about the peer's implementation is that, peer will act both as a server and also a client. This is achieved through the multithreading concept in Linux. The server will run as a separate thread and the client will run as the main program thread.

The client thread will does the work of interacting with the user and at the same time communicate with the server thread in the other side, whereas the server thread will run as a silent module receiving the data from the client and configuring the VNAT database according to the information received.



OTHER ARCHITECTURAL ISSUES

8.OTHER ARCHITECTURAL ISSUES

Support for Connectionless Protocols:

So far we have implicitly concentrated on connection-oriented transport protocol (TCP) because the vast majority of network applications today are based on TCP. It is also important to understand the relative merit in how to support connectionless transport protocols such as UDP, as used in increasingly popular multimedia applications. Even though there is no concept of a "connection" with connectionless transport protocols, applications using these protocols often maintain by themselves some notion of a "connection" at the application level; although the applications usually do not expect either end of the "connection" to move.

Because the "connection" is maintained by the application itself rather than the transport protocol, it is necessary to hide from the application the current physical host location in order to virtualize such application-level "connection" without any modification to the application. VNAT will hide from the application the fact that its location or its peer's location has changed. This is simply done by returning the unchanging virtual address to the application instead of the physical address; therefore enable transparent migration of such UDP based "connections".

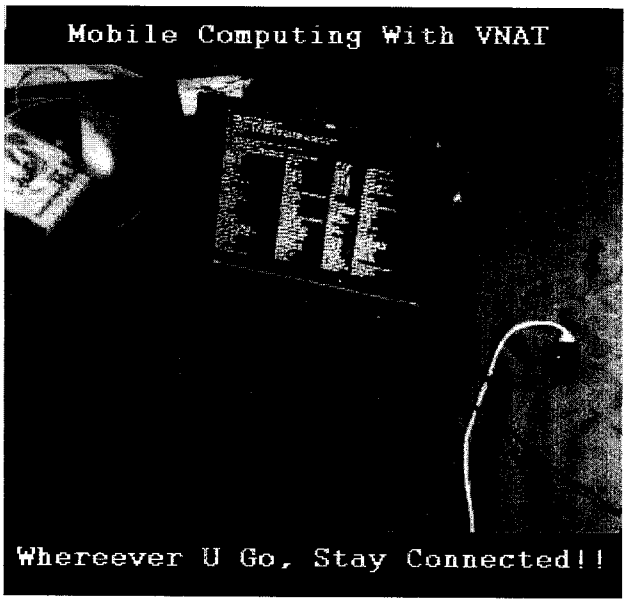
VNAT is committed to be compatible with existing networking protocols and therefore will not by default hide host location change from applications.

Moving Both the End Points Simultaneously:

The design of VNAT focuses on the common case of migrating one endpoint of a connection at a time. We have not addressed the problem of migrating both endpoints of

a connection simultaneously, where the endpoints are migrated to different locations. To do this, a mechanism would be needed for the two endpoints to inform each other their new location if neither one is aware where the other party is migrating beforehand.

There are several potential solutions to the problem. For example, one approach can use a well-known server that is consulted by both endpoints to find out the new location of each other after migration. Another approach is for both endpoints to leave new location states at their original location.



INCREMENTAL USABILITY

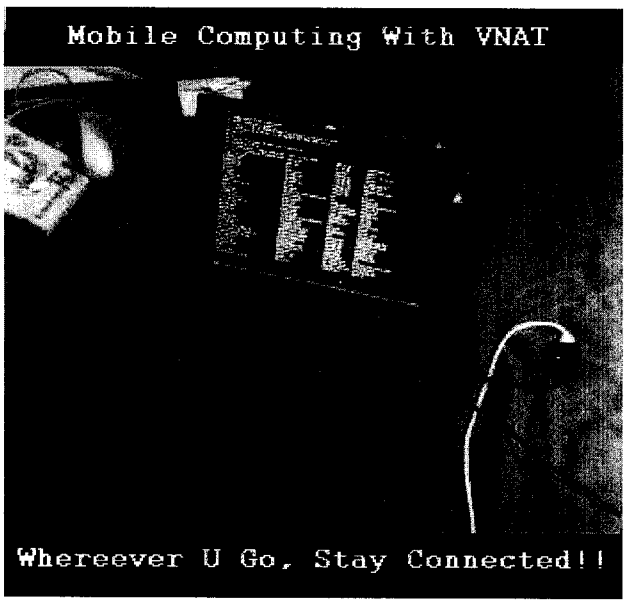
9.INCREMENTAL USUABILITY

An important underlying design principle in VNAT is the idea of incremental usability. VNAT provides a core set of functions to support connection mobility, but it also provides additional features which can be used incrementally. For instance, developers and users do not need to do anything to allow existing applications to work with VNAT without modification.

However, VNAT enables applications to provide richer functionality during connection migration by providing interfaces and mechanisms to support application-specific helper functions. Similarly, VNAT provides other functions that can be used when a connection is suspended to improve security and performance, but these functions are optional and need not be used to provide connection migration functionality. We didn't concentrate on these things during implementation.

VNAT also provides incremental usability in terms of deployment. Not only does VNAT facilitate easy of deployment by not requiring changes to applications, operating systems, or network protocols, but its architecture also facilitates deployment of its functions in an incremental fashion. VNAT can be locally installed on any subset of systems to provide connection mobility within those systems. It does not need to be installed in an entire administrative domain to operate and is compatible with existing network infrastructures.

Furthermore, not all aspects of the VNAT architecture need to be deployed when not all of its functionality is required. For example, VNAT can be implemented as a loadable kernel module or an application that does not even require a system to be rebooted when VNAT is installed, which makes it easier to deploy on shared servers that attempt to minimize downtime. Furthermore because of how it selects the initial virtual address, VNAT can be used to provide connection mobility to connections that already exist even before VNAT is installed.



EXAMPLE MIGRATION SCENARIO

10.EXAMPLE MIGRATION SCENARIO

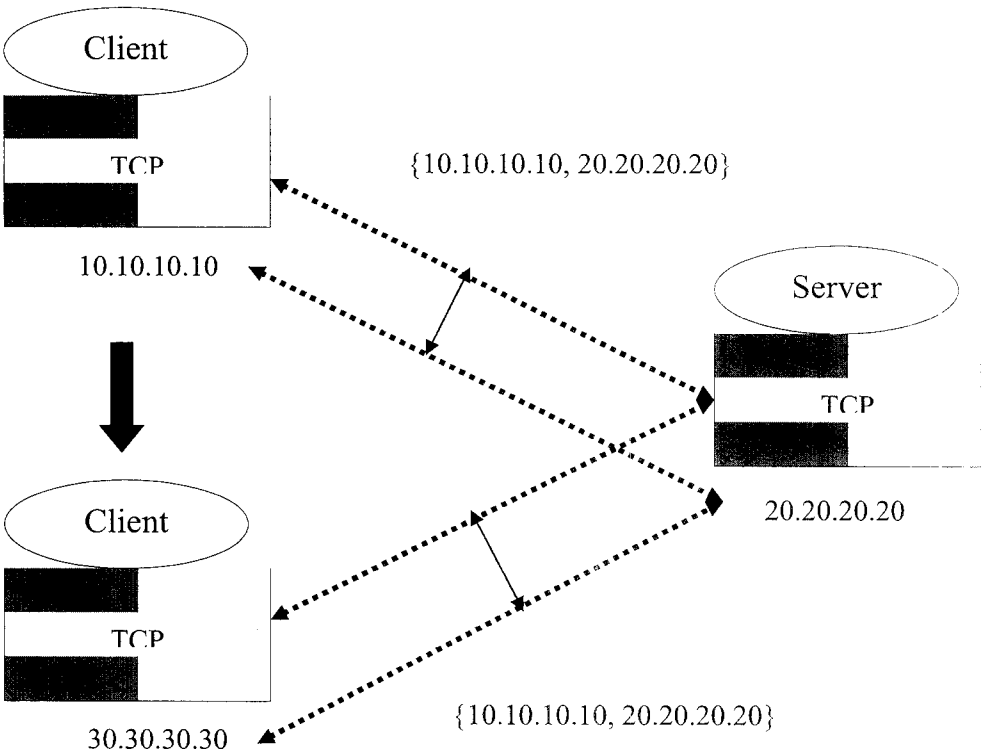
Let's now put all the pieces together and describe a typical migration scenario and see how VNAT migrates a live network connection.

Assume a client on host 10.10.10.10 opens a TCP connection to a server on host 20.20.20.20. The connection is virtualized by VNAT and perceived by TCP on both the client and the server as {10.10.10.10, 20.20.20.20}. Note that here we are using the initial physical addresses as the virtual addresses. In this example, we assume the client migrates and it doesn't matter whether a process or the whole client host migrates.

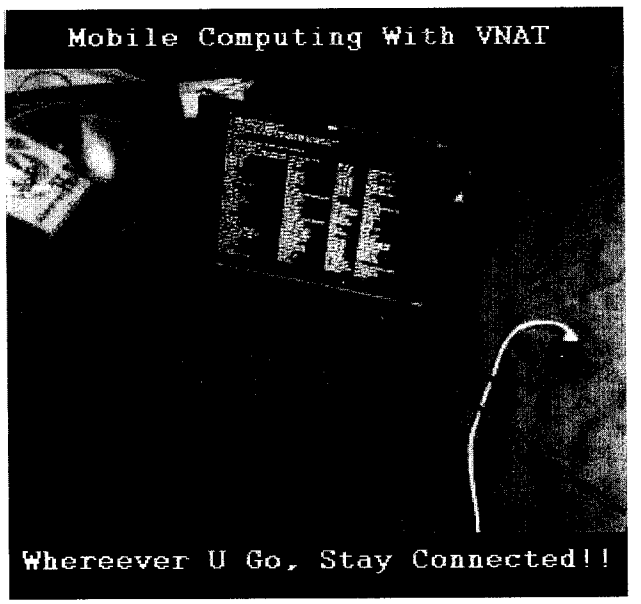
At the time of suspending the connection, the client will attempt to send a message to establish the secret key between the client and the server. In addition, a connection migration helper may be activated on the server for the migrating connection. Note that all of the above steps are optional and the suspension will work without any of them, but with the drawback of not having the benefit of those VNAT functions.

At the time of resuming the connection, the client VNAT at the new location will trivially locate the server location using the existing connection state. After verifying the secret key, the client will update the server with its new physical address 30.30.30.30. And both the client and the server will start translating the virtual connection {10.10.10.10, 20.20.20.20} to and from the physical connection {30.30.30.30, 20.20.20.20}. Note how the virtual connection {10.10.10.10, 20.20.20.20} perceived by the client TCP and the server TCP stays intact across the migration.

Either the client TCP or the server TCP is completely unaware of the change of the underlying physical address of the client. So with the addition cost of translating a virtual connection to and from a physical connection, VNAT will seamlessly migrate a transport end-to-end connection regardless of where the client moves.



EXAMPLE MIGRATION SCENARIO

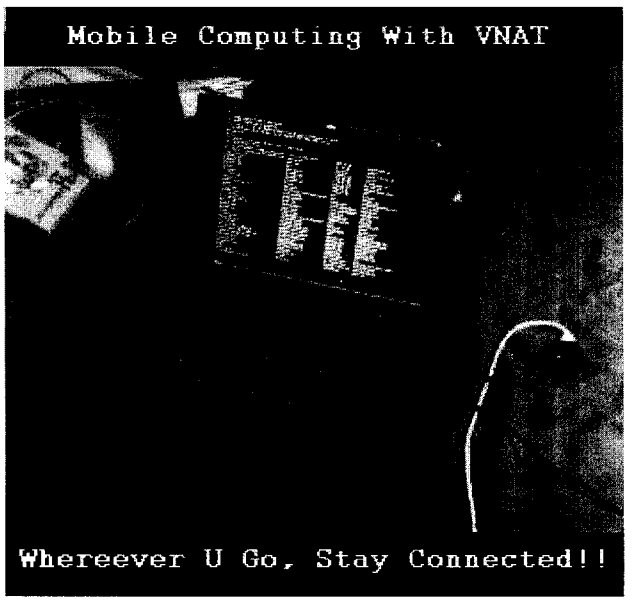


SCOPE FOR ENHANCEMENT

11.SCOPE FOR ENHANCEMENT

We have implemented only the basic architectural model of the VNAT system. We didn't implement the helpers and other optional components. So adding the left out things will itself serve as a separate project. We list out the possible extensions and enhancements below.

- Porting the VNAT from user space to kernel space as loadable kernel modules.
- Support for migration of both the end points at the same time.
- Establishing security mechanisms when suspending a connection.
- Automating the connection virtualization capturing the system calls.
- Process level migration in addition to the host level migration.



TESTING

12.TESTING

Unit Testing:

We have tested each and every module independently in order to ensure the correct data flow between the modules. The various code modules tested are

1. Kernel Parameters Modification

Here the kernel parameters were modified using our code and the modification was ensured by checking the file in which these data are stored by the kernel.

2. Network Address Translation

We have ensured the correctness of the source and destination network address translation by checking the rules added to the 'iptables', used by the kernel for doing network address translation.

3. Virtual Address Manipulation

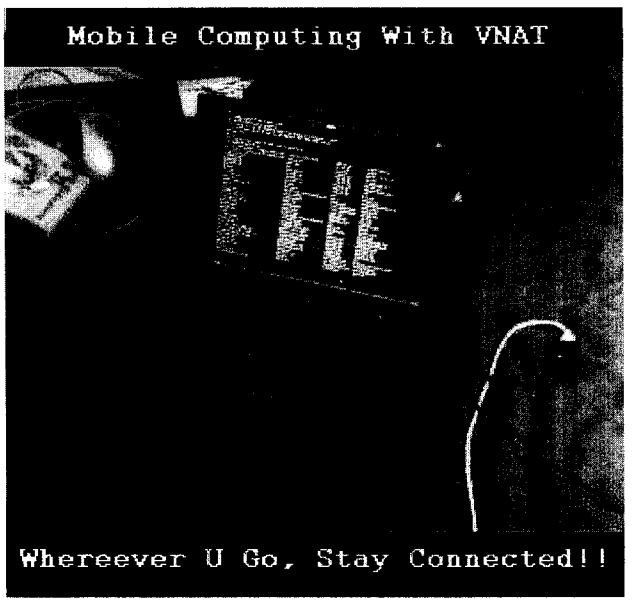
The code needed for virtual address manipulation were checked and ensured by tracking the changes made in the database whenever a command is issued.

4. The Peer Service Module

This mainly includes the testing of the server and the client threads, which operates on sockets. First the client and server were tested independently and then they were put together into the peer. Finally the integrated module is ensured with the correctness of data flow.

Integration Testing:

After testing each and every module independently we have started building the final infrastructure, side by side by side we have conducted this integration testing in order to ensure the correctness, which helped us to achieve the exact result we aimed at.



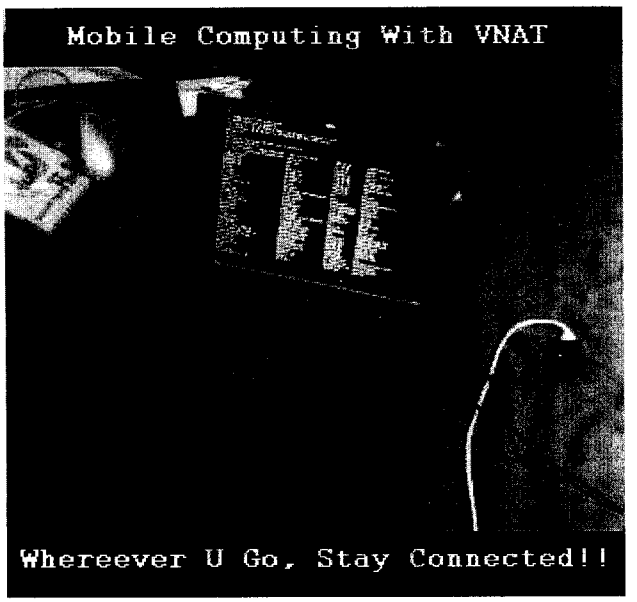
CONCLUSION

13.CONCLUSION

We have implemented VNAT, a novel architecture that enables transparent migration of live network connections associated with a spectrum of computation units. VNAT is based on the simple idea of virtual addresses and employs connection virtualization, translation, and migration to achieve its goals. VNAT supports migration of live end-to-end transport connections when either one or both endpoints of the connections migrate. VNAT provides incremental usability and does not require any modification to existing applications, operating systems, or networking protocols, which enable the system to be more easily deployed and used.

With the rapid increase of distributed networked systems and ubiquitous mobile computing devices, it is becoming a pressing need for developing new networking functionality to support these systems. However, developing and deploying new networking infrastructure is often a long and enduring process. We hope that our work can give insight in how such new networking functionality can be developed and deployed while allowing existing legacy applications to take advantage of the tremendous benefits offered by the coming reality of ubiquitous mobile computing and communication.

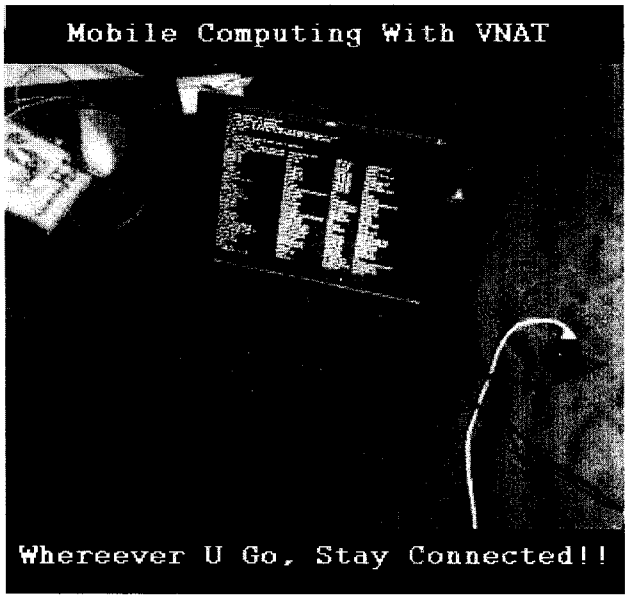
"Where ever you Go, Stay Connected."



REFERENCES

14.REFERENCES

- Douglas E Comer, "**Internetworking with TCP/IP**", **Volume I**, Prentice Hall of India Private Limited, March 2003, Fourth Edition.
- Douglas E Comer, "**Internetworking with TCP/IP", Volume II**, Prentice Hall of India Private Limited, March 2003, Third Edition.
- Andrew S Tanenbaum, "**Computer Networks**", Pearson Education, Inc., 2003 Fourth Edition.
- Richard W Stevens, "**UNIX Network Programming**", Prentice Hall of India Private Limited, October 2002, Second Edition.
- Maurice J Bach, "**The Design of the UNIX OS**", Prentice Hall of India Private Limited, December 2000, Third Edition.
- Brain W Kernighan, "**The Unix Programming Environment**", Prentice Hall of India Private Limited, December 1998, Second Edition.
- Neil Matthew, "**Beginning Linux Programming**", Wrox Press Ltd., December 1999, Second Edition.
- Robert L Ziegler, "**Linux Firewalls**", Techmedia Publications, 2000, First Edition.
- Dennis M Ritchie, "**The C Programming Language",** Prentice Hall of India Private Limited, June 1999, Second Edition.



APPENDIX

SOURCE CODE

```
/* VNAT Peer Service Module (vserver.c) */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

#include "virtualip.h"
#include "nat.h"
#include "tcptimer.h"

#define SUCCESS 0
#define ERROR   1

#define END_LINE 0x0
#define SERVER_PORT 1500
#define MAX_MSG 100

extern char vhostip[25];
char shostip[25];
char shostip_old[25];

int read_line();
int draw_line();
char *my_ip_address();
char *my_old_ip_address();

void *thread_function();

char message[] = "VNAT THREAD";

int sd, newSd, cliLen;

struct sockaddr_in cliAddr, servAddr;
char line[MAX_MSG];

static int rcv_ptr=0;
static char rcv_msg[MAX_MSG];
static int n;
int offset;
```

```
int main()
{
      int res;
      pthread_t a_thread;
      void *thread_result;

      res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);

      if (res != 0)
      {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
      }
      printf("Waiting for thread to finish...\n");

      /*-------------- Initialize SNAT Table (DA 1)----------------*/

      init_snat_table();
      set_timer(99999,9999,9999);

      /*-------------- Initialize SNAT Table (DA 1)----------------*/

      while(1)
      {
        /*-------------- Client Coding (DA 2) ----------------------*/

            /*
            Commands to send
                  1. suspend oldip
                  2. resume oldip newip
                  3. prepare dnat table

            Commands to process
                  1. open
                  2. close
                  3. prepare snat table
                  4. down
            */

            char cmnd[50],*cmnd_split[10],msg1[50],msg2[50],dip[50];
            int count=0,count_ref=0,choice=0,tmp;

            draw_line();
            printf("\n\t\t\t   Mobile Computing With VNAT");
            printf("\n\t\t\t       VNAT PEER SERVICE");
            draw_line();

            printf("\n\n>>> ");
            gets(cmnd);

            cmnd_split[count]=strtok(cmnd," ");
            count++;
            while((cmnd_split[count]=strtok(NULL," "))!= NULL) count++;
            count_ref=count;

            if(!strcmp(cmnd_split[0],"open")) choice = 1;
```

```
        else if(!strcmp(cmnd_split[0],"close")) choice = 2;
        else if(!strcmp(cmnd_split[0],"suspend")) choice = 3;
        else if(!strcmp(cmnd_split[0],"resume")) choice = 4;
        else if(!strcmp(cmnd_split[0],"down")) choice = 5;

        switch(choice)
        {
              case 1: /* open */

                    if((tmp = check_dnat(cmnd_split[1]))==1)
                    {
                          printf("    Info# alredy registered ip");
                          getchar();
                          continue;
                    }
                    add_dnat(cmnd_split[1],cmnd_split[1]);
                    dnat(cmnd_split[1],cmnd_split[1],1);

                    sprintf(dip,"hname=%s",cmnd_split[1]);
                    putenv(dip);
                    sprintf(msg1,"cmd1=%s","open");
                    putenv(msg1);
                    sprintf(msg2,"cmd2=%s",my_ip_address());
                    putenv(msg2);
                    system("tcs $hname $cmd1 $cmd2");

                    printf("    Done !");
                    getchar();
                    continue;
              case 2: /* close */
                    remove_dnat(cmnd_split[1]);
                    close_dnat(cmnd_split[1]);
                    break;
              case 3: /* suspend */
                    continue;
              case 4: /* resume */
                    {
                          FILE *address_file;
                          char msg1[50],msg2[50],msg3[50],dip[50];
                          char from_file_1[25],from_file_2[25];

                          address_file = fopen("dnat.df","r");

                          if(address_file==NULL)
                          {
                                printf("    Error opening file
dnat.df");
                                return;
                          }
                          while(!feof(address_file))
                          {

      fscanf(address_file,"%s",from_file_1);

      fscanf(address_file,"%s",from_file_2);

      sprintf(dip,"hname=%s",from_file_1);
```

```
                                    putenv(dip);
                                    sprintf(msg1,"cmd1=%s","resume");
                                    putenv(msg1);

      sprintf(msg2,"cmd2=%s",my_old_ip_address());
                                    putenv(msg2);

      sprintf(msg3,"cmd3=%s",my_ip_address());
                                    putenv(msg3);
                                    system("tcs $hname $cmd1 $cmd2
$cmd3");

                              }

                              printf("    Done !");
                              getchar();

                              fclose(address_file);
                        }
                        continue;
                  case 5: /* down */
                        reset_timer();
                        remove_snat();

                        {
                              FILE *address_file;
                              char msg1[50],msg2[50],dip[50];
                              char from_file_1[25],from_file_2[25];

                              address_file = fopen("dnat.df","r");

                              if(address_file==NULL)
                              {
                                    printf("    Error opening file
dnat.df");

                                    return;
                              }
                              while(!feof(address_file))
                              {

      fscanf(address_file,"%s",from_file_1);

      fscanf(address_file,"%s",from_file_2);
                                    dnat(from_file_1,from_file_2,2);

                              }

                              printf("    Done !");
                              getchar();

                              fclose(address_file);
                        }
                        continue;
                  default:
                        printf("    %s : command not
found",cmnd_split[0]);
                        continue;
```

```
        }


    /*------------- Client Coding (DA 2) --------------------------*/
    }
    res = pthread_join(a_thread, &thread_result);
    if (res != 0)
    {
        perror("Thread join failed");
          exit(EXIT_FAILURE);
      }
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    exit(EXIT_SUCCESS);
}

void *thread_function()
{
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd<0)
    {
        perror("cannot open socket ");
        return ERROR;
    }

    servAddr.sin_family = AF_INET;
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servAddr.sin_port = htons(SERVER_PORT);

    if(bind(sd, (struct sockaddr *) &servAddr, sizeof(servAddr))<0)
    {
        perror("cannot bind port ");
        return ERROR;
    }

    listen(sd,5);

    while(1)
    {
        int count_ref=0,choice=0;

        cliLen = sizeof(cliAddr);
        newSd = accept(sd, (struct sockaddr *) &cliAddr, &cliLen);
        if(newSd<0)
        {
            perror("cannot accept connection ");
            return ERROR;
        }

        memset(line,0x0,MAX_MSG);
        while(read_line(newSd,line)!=ERROR)
        {
          /*------------- Server Coding (DA 3)--------------------*/

            char *cmd[10];
            int count=0,tmp;
            cmd[count]=strtok(line," ");
            printf("\n%s",cmd[count]);
```

```
                count++;

                while((cmd[count]=strtok(NULL," "))!= NULL) count++;
                count_ref = count;
                memset(line,0x0,MAX_MSG);

                /*
                commands expected
                        1. open
                                -add dnat.df
                                -add_dnat
                        2. resume oldip newip
                */

                if(!strcmp(cmd[0],"open")) choice=1;
                else if(!strcmp(cmd[0],"resume")) choice=2;

                switch(choice)
                {
                        case 1:/* open*/
                                if((tmp = check_dnat(cmd[1]))==1) break;
                                add_dnat(cmd[1],cmd[1]);
                                dnat(cmd[1],cmd[1],1);
                                break;
                        case 2: /* resume */
                                dnat(cmd[1],cmd[1],2);
                                dnat(cmd[1],cmd[2],1);
                                close_dnat(cmd[1]);
                                add_dnat(cmd[1],cmd[2]);

                                break;
                        default:
                                break;
                }

                /*-------------- Server Coding (DA 3)----------------
--*/

        }

    }
}


int read_line(int newSd, char *line_to_return)
{

    offset=0;

    while(1)
    {
        if(rcv_ptr==0)
        {
                memset(rcv_msg,0x0,MAX_MSG);
                n = recv(newSd, rcv_msg, MAX_MSG, 0);
                if (n<0)
                {
```

```
                        perror(" cannot receive data ");
                        return ERROR;
                }
                else if (n==0)
                {
                        close(newSd);
                        return ERROR;
                }
        }

        while(*(rcv_msg+rcv_ptr)!=END_LINE && rcv_ptr<n)
        {
                memcpy(line_to_return+offset,rcv_msg+rcv_ptr,1);
                offset++;
                rcv_ptr++;
        }

        if(rcv_ptr==n-1)
        {
                *(line_to_return+offset)=END_LINE;
                rcv_ptr=0;
                return ++offset;
        }

        if(rcv_ptr <n-1)
        {
                *(line_to_return+offset)=END_LINE;
                rcv_ptr++;
                return ++offset;
        }

        if(rcv_ptr == n)
        {
                rcv_ptr = 0;
        }
    } /* while */
}


int init_snat_table()
{
    FILE *file;
    char shostip[25];

    system("hostname -i > myip.df");
    file = fopen("myip.df","r");
    fscanf(file,"%s",shostip);
    fclose(file);

    snat(shostip,1);
    return;

}


int remove_snat()
{
```

```
        FILE *file;
        char shostip[25];

        file = fopen("myip.df","r");
        fscanf(file,"%s",shostip);
        fclose(file);
          snat(shostip,2);
}

char *my_old_ip_address()
{
        FILE *file;

        file = fopen("myip.df","r");
        fscanf(file,"%s",shostip_old);
        fclose(file);
        return shostip_old;
}

int remove_dnat(char *ip)
{
        FILE *address_file;
        char from_file_1[25],from_file_2[25];
        address_file = fopen("dnat.df","r");

        if(address_file==NULL)
        {
              printf("    Error opening file dnat.df");
              return;
        }
        while(!feof(address_file))
        {
              fscanf(address_file,"%s",from_file_1);
              fscanf(address_file,"%s",from_file_2);
              if(!strcmp(from_file_1,ip))
                    dnat(from_file_1,from_file_2,2);
        }
        fclose(address_file);
        return;
}
char *my_ip_address()
{
        FILE *file;

        system("hostname -i > myip_cur.df");
        file = fopen("myip_cur.df","r");
        fscanf(file,"%s",shostip);
        fclose(file);
        return shostip;
}

int draw_line()
{
        int i=80;
        printf("\n");
        while(i--)
        {
```

```
            printf("-");
        }
}


/* ------------------------------------------------------------------*/

/* Client Module - Part of VNAT Peer (tcs.c) */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>

#define SERVER_PORT 1500
#define MAX_MSG 100

char msgdata[150];
int main (int argc, char *argv[])
{
      int sd, rc, i;
      struct sockaddr_in localAddr, servAddr;
      struct hostent *h;

      h = gethostbyname(argv[1]);
      if(h==NULL)
      {
          printf("    Error# unknown host '%s'\n",argv[1]);
          exit(1);
      }

      servAddr.sin_family = h->h_addrtype;
      memcpy((char *) &servAddr.sin_addr.s_addr, h->h_addr_list[0], h-
>h_length);
      servAddr.sin_port = htons(SERVER_PORT);

      sd = socket(AF_INET, SOCK_STREAM, 0);
      if(sd<0)
      {
            perror("    Error# cannot open socket ");
            exit(1);
      }

      localAddr.sin_family = AF_INET;
      localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
      localAddr.sin_port = htons(0);

      rc = bind(sd, (struct sockaddr *) &localAddr, sizeof(localAddr));
      if(rc<0)
      {
            printf("    Error# cannot bind port TCP %u\n",SERVER_PORT);
            perror("error ");
            exit(1);
```

```
	}

	rc = connect(sd, (struct sockaddr *) &servAddr,
sizeof(servAddr));
	if(rc<0)
	{
		perror("    Error# cannot connect ");
		exit(1);
	}
	strcpy(msgdata,"");
	/*------------------ DA -----------------*/
	for(i=2;i<argc;i++)
	{
		strcat(msgdata,argv[i]);
		strcat(msgdata," ");
	}
	rc = send(sd, msgdata, strlen(msgdata) + 1, 0);
	/*------------------ DA -----------------*/

	return 0;

}


/* ---------------------------------------------------------------------*/

/* Kernel Timer Tuner - Declared in tcptimer.h (tcptimer.c)*/

#include "tcptimer.h"
int set_timer(int kt,int kp,int ki)
{
	char env[25];
	sprintf(env,"ktime=%d",kt);
	putenv(env);
	system("sysctl -w net.ipv4.tcp_keepalive_time=$ktime >>
/dev/null");
	sprintf(env,"kprobes=%d",kp);
	putenv(env);
	system("sysctl -w net.ipv4.tcp_keepalive_probes=$kprobes >>
/dev/null");
	sprintf(env,"kintvl=%d",ki);
	putenv(env);
	system("sysctl -w net.ipv4.tcp_keepalive_intvl=$kintvl >>
/dev/null");
}

int reset_timer(void)
{
	system("sysctl -w net.ipv4.tcp_keepalive_time=7500 >>
/dev/null");
	system("sysctl -w net.ipv4.tcp_keepalive_probes=9 >> /dev/null");
	system("sysctl -w net.ipv4.tcp_keepalive_intvl=75 >> /dev/null");
}

/* ---------------------------------------------------------------------*/
```

```
	if(address_file == NULL || tmp_file == NULL)
	{
		printf("    Error opening file dnat.df/dnat.tmp !");
		return;
	}
	while(!feof(address_file))
	{
		fscanf(address_file,"%s",from_file_1);
		fscanf(address_file,"%s",from_file_2);
		if(strcmp(from_file_1,ip) && strcmp(from_file_1,"0000"))
		{
			fprintf(tmp_file,"%s\n",from_file_1);
			fprintf(tmp_file,"%s\n",from_file_2);
		}
		strcpy(from_file_1,"0000");
		strcpy(from_file_2,"1111");
	}
	fclose(address_file);
	fclose(tmp_file);
	system("rm -f dnat.df");
	system("mv dnat.tmp dnat.df");
	return;
}

/*-------------------------------------------------------------------*/

/* snat.c - Source Network Address Translation
 * declared in nat.h*/

void snat(char *ip,int choice)
{
	char env[50];
	switch(choice)
	{
		case 1:
			sprintf(env,"sip=%s",ip);
			putenv(env);
			system("iptables -t nat -A POSTROUTING -o eth0 -j
SNAT --to $sip");

			break;
		case 2:
			sprintf(env,"sip=%s",ip);
			putenv(env);
			system("iptables -t nat -D POSTROUTING -o eth0 -j
SNAT --to $sip");
			break;
		default:
			exit(0);
	}
}

/*-------------------------------------------------------------------*/

/* dnat.c - Destination Network Address Translation
 * declared in nat.h*/
```