Mobile Communication with Virtual Network Address Translation

Gong Su and Jason Nieh

Computer Science Department
Columbia University
New York, NY 10027

Abstract

Virtual Network Address Translation (VNAT) is a novel architecture that allows transparent migration of end-to-end live network connections associated with various computation units. Such computation units can be either a single process, or a group of processes of an application, or an entire host. 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 proxies. We have implemented a VNAT prototype with the Linux 2.4 kernel and demonstrated its functionality on a wide range of popular real-world network applications. Our performance results show that VNAT has essentially no overhead except when connections are migrated, in which case the overhead of our Linux prototype is less than 7 percent.

1 Introduction

Ubiquitous mobile computing is a coming reality, fueled in part by continuing advances in wireless transmission technologies and handheld computing devices. As computations are increasingly networked, 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 can not 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 [MB98-2, Perk01, Perk96, QYB97, SB00, ZD95]. However, previous approaches either require changes to network or transport layer protocols, or suffer from substantial performance penalties [ZM01], which limit their deployment.

To effectively support efficient transparent migration of end-to-end live network connections without any changes to existing network protocols, we introduce Virtual Network Address Translation (VNAT). VNAT is a novel mobile communication 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 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 a loadable kernel module in Linux 2.4. Our experience with VNAT shows that it works effectively with a wide range of popular real world applications. Our experimental results on an unoptimized VNAT prototype show that VNAT imposes almost no overhead except when connections are migrated, in which case the overhead of our prototype is between 2 to 7 percent for the applications tested.

This paper describes the VNAT architecture with a focus on the VNAT connection migration mechanism and is organized as follows. Section 2 surveys related work. Section 3 presents the main VNAT architecture concepts and constructs. Section 4 illustrates how VNAT can be used in a few example connection migration scenarios. Section 5 describes the implementation of our VNAT prototype in Linux 2.4. Section 6 shows experimental results that measures the performance overhead of our VNAT prototype. Finally, we present some concluding remarks.

2 Related Work

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 mechanisms. We discuss these approaches and also describe related work in process migration and network address translation.

MobileIP [Perk01, Perk96] is the best-known network layer mobile communication architecture. MobileIP 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. Therefore, MobileIP does not require any modification to existing protocols and applications above network layer. MobileIP 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". Unfortunately, MobileIP requires network layer protocol changes that are costly and make it very difficult to deploy.

To allow migration of individual end-to-end connections between two applications rather than just the entire host, [SB00] recently proposed a transport layer mobility architecture called Migrate. Since traditional transport protocols are not built with mobility in mind, Migrate introduces a new TCP option to support suspending and resuming TCP connections. [SAB01] further considers fine-grained failover of long-running connections across a collection of replica servers and uses [SB00] as its vehicle for migrating TCP connections. Migrate does not support migration of TCP connections for which both endpoints move simultaneously. Since Migrate is TCP-specific and requires transport layer protocol changes, its architecture also makes it difficult to deploy.

We note that MobileIP and Migrate also provide mechanisms for mobile host location. MobileIP uses the notion of home and foreign agents to provide mobile host location technologies. Migrate uses dynamic DNS updates [SB00]. Our work on VNAT focuses on the "tracking" aspect of connection mobility while being compatible with and taking advantage of existing mobile host location technologies, such as those used in MobileIP and Migrate.

MSOCKS [MB98-2] is a proxy-based mobility architecture based on the "TCP Splice" [MB98-1] technique. Essentially, a single TCP connection between a mobile client and a stationary server is "spliced" (transparent to both the client and the server) by a proxy in the middle into two separate TCP connections: one between the mobile client and the proxy and the other between the proxy and the stationary server. The proxy, acting as a "TCP connection switch", 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.

Higher layer approaches such as [QYB97, ZD95] modify the socket library to introduce another layer between the application and the transport protocol. This layer maintains a location-independent connection identification (usually the 5-tuple) invariant to applications and "switches" the invariant onto an appropriate real 5-tuple to maintain the TCP connection between two applications when one or both applications move. The invariant 5-tuple idea is similar to VNAT's connection virtualization idea. However, similar to the proxy in MSOCKS, the extra layer is tied to the transport protocol (i.e., TCP) and has to deal with TCP specific issues to maintain the semantics of TCP when the application moves. Due to the duplicated functions of the transport protocol, the extra layer creates substantial performance overhead as shown in [ZM01].

Much work has been done in the area of process migration [MDW99]. Kernel-level and user-level mechanisms have been previously developed that can migrate processes or groups of processes from one machine to another. Although there is some limited work in this area on supporting networked processes using a stub mechanism similar to the home agent idea used in MobileIP, the work on process migration mostly focuses on non-networked processes instead of communication mobility and is complementary to our work on VNAT.

VNAT make use of the well-known and widely used Network Address Translation (NAT) technology [SE01]. Traditional NAT is typically used to translate IP addresses from one realm to another. The purpose of NAT is to provide transparent routing solution to hosts using "private" IP addresses that cannot be routed on a "public" network. This basic functionality of NAT, although not directly intended for mobility, turns out to be a powerful mechanism for mobility solutions. As a matter of fact, [SH99] describes a variation of NAT called "twice NAT" (modifying both source and destination IP addresses) that can be used when a site changes its Internet service provider and elects to keep its (internal) addresses assigned by the first provider; and one can indeed consider the case as a very rudimentary form of mobility. VNAT capitalizes on this fundamental function of NAT and utilizes it to translate virtual endpoints to and from physical endpoints for end-to-end transport connections.

3 The 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, the reason it is impossible to keep end-to-end transport connections alive when one or both connection endpoints move is 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 in Figure 3-1. VNAT connection virtualization is the mechanism used to virtualize the 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 keeping alive connections during migration, locating the endpoints of a migrated connection when it is resumed, and securing the migrating connection. As discussed in Section 5, 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. We describe the function of these three components in more detail in the following sections.

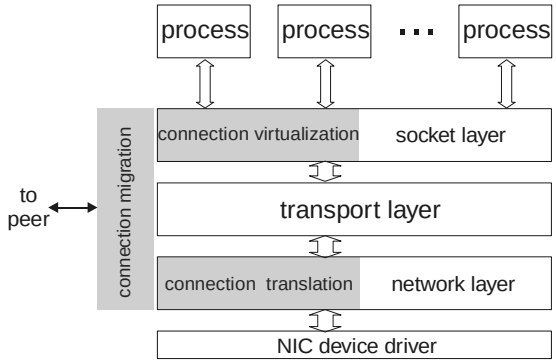

Figure 3-1: VNAT architecture overview

3.1 VNAT 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, the examples we use in this paper will leave 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.

Let us use an example to explain the virtual connection idea. Although VNAT is designed to be independent of any particular transport protocol, throughout the paper we will use TCP as the transport protocol to illustrate various functions of VNAT. Figure 3-2 illustrates how VNAT virtualizes a TCP connection. VNAT intercepts connection setup requests from the application to the transport protocol and replace the physical addresses supplied by the application with virtual addresses. For example, a server typically calls `bind` with the address `INADDR_ANY` to indicate that it's willing to accept incoming connections from any of the physical addresses assigned to the host. VNAT intercepts the `bind` call and replaces `INADDR_ANY` with a virtual address `2.2.2.2`. Similarly, a client typically calls `connect` with the physical address, `20.20.20.20`, of the server. VNAT intercepts the `connect` call and replaces `20.20.20.20` with the virtual address `2.2.2.2`. Note that `connect` usually does an "autobind" for the client. This is also handled by VNAT so that the client is bound to a virtual address `1.1.1.1` rather than the physical address `10.10.10.10`.

Without explaining how such a virtualized connection can actually be established across the physical network, which is described in Section 3.2, we can see the end result is that the TCP on both the client and the server will perceive a virtual connection `{1.1.1.1,2.2.2.2}` rather than a physical connection `{10.10.10.10,20.20.20.20}` (note we ignore the order of source and destination address pair in our discussion). This virtual connection identification will stay unchanged for the life of the connection no matter where the client or the server moves. For example, should the client later decide to move to another host with physical address `30.30.30.30`, the virtual connection perceived by both the client and the server will stay as `{1.1.1.1,2.2.2.2}` rather than change to `{30.30.30.30, 20.20.20.20}`.

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. As a matter of 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 and this is what we used in our example (e.g., `1.1.1.1` and `2.2.2.2`), careful selection of the virtual addresses can greatly simplify the system. Since both par-

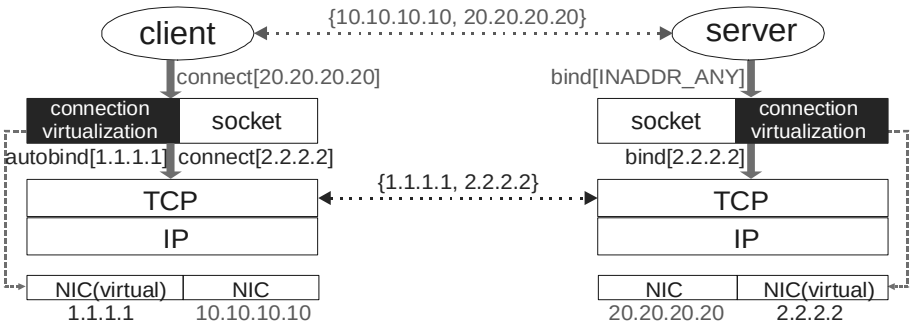

Figure 3-2: VNAT connection virtualization

ties 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 as in our example with virtual addresses `1.1.1.1` and `2.2.2.2`, 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.

This 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. Note that 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. This selection of virtual addresses also has benefits for connection translation, as discussed in Section 3.2.

3.2 VNAT connection translation

Once a TCP connection is virtualized, it's 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. Recall in Figure 3-2 that a packet with header `{1.1.1.1,2.2.2.2}` sent by a client using TCP is never going to go anywhere on the physical network; and the server using TCP is never going to receive a packet with header `{1.1.1.1,2.2.2.2}` from the physical network.

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 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.

We illustrate VNAT connection translation by continuing with our example from Figure 3-2. In Figure 3-3, it is clear that a packet with header `{1.1.1.1,2.2.2.2}` sent by the client TCP must be translated into a packet with header `{10.10.10.10, 20.20.20.20}` for it to reach the intended server. Similarly, a packet with header `{10.10.10.10,20.20.20.20}` must be translated back into a packet with header `{1.1.1.1,2.2.2.2}` for it to be accepted by the server TCP.

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.

3.3 VNAT 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 security protection key, determine the migration roles of the endpoints of a migrating connection, and activate a connection migration helper. To resume a suspended connection, VNAT locates the migrated endpoints, verifies the security protection key if it is available, and updates the appropriate virtual-physical endpoint mappings. The protocol messages used by VNAT connection migration to perform its various functions are collectively called the VNAT Connection Migration Protocol (VCMP). The various functions in the timeline of a typical connection migration are described in further detail in the following sections.

3.3.1 Suspend 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

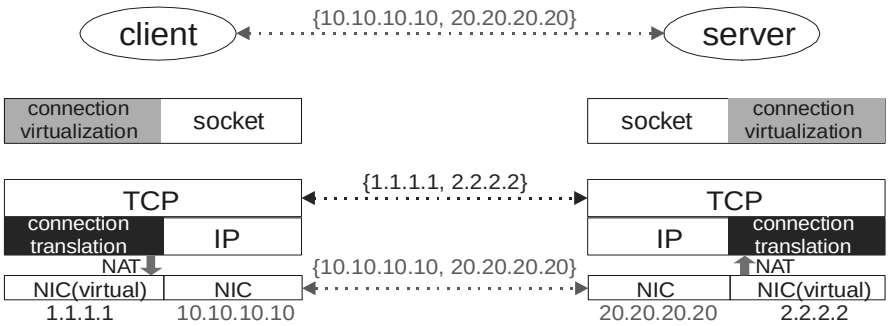

Figure 3-3: VNAT connection translation

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. We have in fact integrated VNAT with APM for moving suspended laptops and also built a process migration mechanism to operate with VNAT to enable migration of various computation units. However, a discussion of these systems is beyond the scope of this paper.

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. These optional functions are discussed further below.

3.3.1.1 Establish security 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.

This scenario is very similar to the security problem with "binding updates" in MobileIP where a mobile node informs its home agent or correspondent node about its new care-of address. MobileIPv6 mandates the use of IPsec authentication (IPsec AH) [KA98] for binding updates and binding acknowledgements. In the absence of IPsec AH, [OR01] has proposed a unilateral authentication protocol (CAM) for MobileIPv6 binding messages. Although VNAT can make use of these solutions, both IPsec AH and CAM (which is specifically designed for IPv6) are not yet widely deployed.

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. Note again that the key exchange only happens if a connection is to be migrated. VNAT is designed to use existing techniques, such as Diffe-Hellman [DH79], to establish the key. Diffe-Hellman is particularly suited for VNAT because the secret key can be established over a public network without any prior shared knowledge between the two endpoints. 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. It is assumed that the secret key is part of the connection state saved and transported by the migration mechanism used.

3.3.1.2 Determine migration roles

When a connection is suspended, VNAT can set up some state to differentiate between the roles played by the two endpoints after their migration. One role is called the *primary*, and the other the *secondary*. The purpose of this option is to minimize the protocol messages exchanged between the migrated endpoints for locating each other when they are resumed. This will become clear later on when we describe how migrated endpoints locate each other in Section 3.3.2.1.

When one endpoint is about to migrate and before it suspends, it sends a VNAT_SUSP_PRIMARY message to its peer claiming the role of the primary. If an endpoint receives a VNAT_SUSP_PRIMARY message before it sends its own, it replies with a VNAT_SUSP_SECONDARY message to accept the role of the secondary (Figure 3-4(a)). Otherwise, the VNAT_SUSP_PRIMARY messages sent by both endpoints would have crossed each other and a simple arbitration mechanism is used to decide which is the primary and which is the secondary. For example, the host with a "higher" IP address can be the primary (Figure 3-4(b)). In case of an arbitration taking place, there will be no further reply sent to each other by the two endpoints. Note that in the case of only one endpoint moves, the moving endpoint will *always* become the primary through message exchange (a).

3.3.1.3 Activate connection migration helper

When only one endpoint of a live network connection is suspended for migration, it may be desirable to provide additional functionality to preserve the connection while one endpoint is suspended. In order not to require application modification and not to tie the core VNAT architecture to any particular applica-

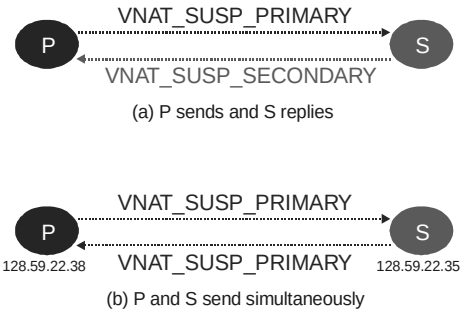

Figure 3-4: Determine migration roles

tion, VNAT uses the concept of a *connection migration helper* to address this issue.

A connection migration helper is a user-defined function that "hooks" into the VNAT system through a well-defined interface. Once activated for a virtual connection whose other end is suspended and being migrated, it is the responsibility of the helper function to monitor potential outgoing traffic on the virtual connection and to buffer and/or respond to the traffic in any application-specific manner. There are two types of outgoing traffic a helper function has to deal with:

(1) a "keepalive" message sent by the transport protocol independent of the application. This is relatively easy to deal with and can be done in an application-independent manner.
(2) a generic message sent by the application. This is more difficult and may require the helper to have detailed understanding of the application's semantics. Alternatively, a more generic helper may be used that essentially blocks the corresponding process associated with the live endpoint of the connection.

VNAT maintains a repository of helper programs. Helpers for known application protocols such as TELNET, FTP, SSH, and HTTP, etc., are shared by applications and must be installed by privileged users such as *root*. Unprivileged users can also submit their own custom helpers and request these helpers on the remote end for their connection migration. These custom helpers run in unprivileged mode and have access to only the connections their are intended for. With connection migration helpers, VNAT can achieve true migration of live connections without any data loss for the duration of the migration while keeping the core of the VNAT architecture independent of application specifics.

3.3.2 Resume a connection

Resuming a connection is the reverse of suspending a connection. If a connection endpoint is migrated by checkpointing 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 are resumed.

3.3.2.1 Locate migrated endpoints

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. When both endpoints of a connection migrate, however, there must be a mechanism for the two endpoints to inform each other their new location if neither endpoint is aware where the other party is migrating beforehand. There are several potential approaches for 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. VNAT combines both approaches to take advantage of existing connection states that is being migrated. However, we note that VNAT does not leave states behind when a connection is suspended. It only stores the new location state at the old locations when a connection is resumed.

Recall in Section 3.3.1.2 that as part of connection suspension procedure, the two endpoints will negotiate their roles as primary and secondary via VCMP. The procedure for locating each other carried out by the primary and the secondary host after migration and during restart is illustrated in Figure 3-5 and described below.

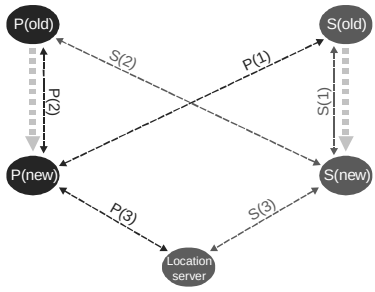

Figure 3-5: Locate migrated endpoints

The primary follows these procedure to locate the secondary:

P(1) Contact the old location of the secondary and query for the new location of the secondary. If successful, go to P(4); otherwise (the secondary has not resumed yet), attempt to store its own new location so it can be looked up when the secondary is resumed. If successful, the procedure finishes and the primary returns to suspended state (the length of the suspension is a policy decision that can be negotiated at the connection suspension time); otherwise (this happens when the entire secondary has moved),
P(2) Contact the old location of the primary, perform the same query and store operations as in P(1). If both query and store operations fail (this happens when the entire primary has moved),
P(3) Contact one or more predefined location servers, perform the same query and store operations as in P(1). If both query and store operations fail, the location of the secondary cannot be determined and the migration of the connection will be aborted.
P(4) Once the primary learns the new location of the secondary, it uses VCMP update message (described in Section 3.3.2.3) to communicate with the secondary and update the virtual-physical address mapping for the migrated virtual connection.

The secondary follows almost exactly the same procedure except the order of contacting the old locations are switched.

S(1) Contact the old location of the secondary (instead of the primary) and query for the new location of the primary. If suc-

cessful, goto S(4); otherwise, attempt to store its own new location so it can be looked up when the primary is resumed. If successful, the procedure finishes and the secondary returns to suspended state; otherwise,

S(2) Contact the old location of the primary and perform the same query and store operations as in S(1). If both query and store operations fail (this happens when the entire primary has moved),

S(3) Contact one or more predefined location servers, perform the same query and store operations as in S(1). If both query and store operations fail, the location of the primary cannot be determined and the migration of the connection will be aborted.

S(4) Once the secondary learns the new location of the primary, it uses VCMP update message (described in Section 3.3.2.3) to communicate with the primary and update the virtual-physical address mapping for the migrated virtual connection.

Readers can now see the purpose of differentiating the roles of the primary and the secondary when the migrated endpoints try to locate each other. It is used so that the two endpoints can "meet" each other at a common location more quickly rather than "cross" each other. For example, if the protocol didn't differentiate the two endpoints and simply stated that an endpoint should always go to its peer's old location first when resumed, we could see that the first try by both endpoints will always fail.

Recall from Section 3.3.1 that a connection may be migrated by simply unplug the network cable of the host and move the host to another place. In this case, VNAT on both ends will never have a chance to negotiate the roles. Or the user has simply turned off the option for negotiating the roles. Either way, when the connection is being resumed and there is no roles negotiated, VNAT on both ends will default to the primary role independent of each other. From our previous description, this is the same case as when the protocol does not differentiate the roles. It will not cause any additional message exchanges in the case of only one endpoint moves. For the case of both endpoints move, they will "cross" each other on the first try since they both assume the primary role and will contact each other's old location first. But eventually they will locate each other either at one of the old locations or at the location server.

The function of the location server is similar to that of home/foreign agent used in MobileIP and dynamic DNS update used in [SB00]. It maps an invariant virtual connection tuple (a pair of virtual addresses) into the current physical address (IP address and port number) of either the primary or the secondary of the virtual connection. In the case of MobileIP, an invariant home IP address of a mobile host is mapped into its current IP address; and in the case of [SB00], an invariant DNS name of a mobile host is mapped into its current IP address. We can use these mechanisms to provide the functionality needed by our location server; therefore avoid unnecessary re-engineering. Note also that consulting a location server, which usually requires manual configuration, is used as a last resort when attempts for the migrated endpoints to locate each other through their old locations have failed.

3.3.2.2 Verify security protection key

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. For example, when using Diffie-Hellman, one endpoint can simply encrypt the update request message with the secret key; the other endpoint can decrypt the request only if it possesses the same secret key. Recall that the secret key is part of the connection states saved and transported by the migration mechanism. 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.

3.3.2.3 Update virtual-physical endpoints mapping

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. The virtual-physical address mapping is updated by exchanging two simple messages, VNAT_UPD and VNAT_UPD_R.

(1) VNAT_UPD message

To resume a connection at a new location, the VNAT system running at the new location sends a VNAT_UPD message to notify the corresponding VNAT running on the remote peer to update its virtual-physical address mapping for a virtual connection. The format of the VNAT_UPD message is shown in Figure 3-6. The message contains the virtual addresses of both endpoints as well as the physical address of the new location. For implementation efficiency, the message format is aligned on a 32-bit address boundary.

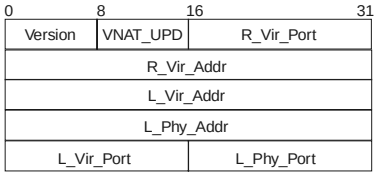

Figure 3-6: VNAT_UPD message

- Version: VCMP version
- Command: VNAT_UPD
- R_Vir_Port: old remote virtual port
- R_Vir_Addr: old remote virtual address
- L_Vir_Addr: old local virtual address
- L_Phy_Addr: new local physical address
- L_Vir_Port: old local virtual port
- L_Phy_Port: new local physical port

Upon receiving a VNAT_UPD message, VNAT searches its virtual-physical address mapping table for a virtual connection `{L_Vir_Addr:L_Vir_Port, R_Vir_Addr:R_Vir_Port}`. If the virtual connection is found, its remote physical address and physical port are updated by the L_Phy_Addr and L_Phy_Port fields supplied in the message, and connection translation NAT rules for the virtual connection is updated accordingly.

(2) VNAT_UPD_R message

The VNAT_UPD_R message is sent by VNAT in response to a VNAT_UPD message. It contains the new virtual-physical address mapping on the recipient side, if any, for a virtual connection identified by the sender's VNAT_UPD message. The format of the VNAT_UPD_R message is shown in Figure 3-7.

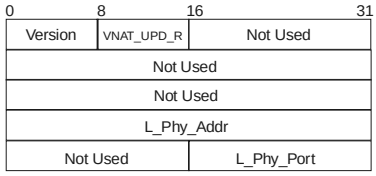

Figure 3-7: VNAT_UPD_R message

The fields in VNAT_UPD_R message are the same as those in VNAT_UPD message except the fields used for identifying the virtual connection are not used.

Upon receiving a VNAT_UPD_R message, VNAT updates the remote physical address and physical port of the migrated virtual connection returned in the L_Phys_Addr and L_Phy_Port fields, and updates the connection translation NAT rules for the virtual connection accordingly.

3.3.2.4 Deactivate connection migration helper

If a connection migration helper was activated when the migrated connection was suspended, VNAT notifies the helper that the connection has been restored. The helper can then perform any necessary operations to deactivate gracefully. This may be as simple as just unloading a kernel module or stopping a userspace program. It may also be as complex as playing back buffered data seamlessly, requiring an understanding of application semantics. Due to space constraints, a detailed discussion of the operation of connection migration helpers is beyond the scope of this paper.

3.4 Other architectural issues

We consider, in this subsection, certain architectural issues that, although orthorgonal to the VNAT architecture itself, are nevertheless important ones and worth mentioning.

3.4.1 Support connectionless protocols

Our discussion so far has implicitly concentrated on connection-oriented transport protocols using TCP as our example. The reason we focused on connection-oriented transport protocols is because today the most popular internet applications, such as remote login (e.g., TELNET, SSH), file transferring (e.g., FTP), web (e.g., HTTP), email (e.g., SMTP), news (e.g., NNTP), and text-based chat (e.g., IRC), etc., are all based on TCP. However, we believe it is important to understand the relative merit in how to support connectionless transport protocols due to a couple of reasons. First, many multimedia applications, which are becoming increasingly popular, use UDP based protocols such as RTP [SCFJ01]. Second, 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 provides such a mechanism as an option on a per application base. When the option is turned on, VNAT will hide from the application the fact that its location or its peer's location has changed; therefore enable transparent migration of such UDP based "connections". However, we would like to point out that in doing so we are also violating the conventional transport protocol behavior, which is to always tell applications the truth of the current host location. VNAT is committed to be compatible with existing networking protocols and therefore will *not* by default hide host location change from applications.

3.4.2 Resolve virtual addresses conflict

As connection endpoints migrate from host to host, so do the virtual identifications for the endpoints. This creates a situation where a virtual identification may be reused after it migrates to another host. For example, a process on host *C* at port *P1* connecting to a host *S* at port *P2* may be migrated to another host *C'*. Later, another process on *C* may reuse *P1* for its connection to *S* at *P2*. Another possibility is that the whole host *C* might be migrated to another place and later its original IP address is recycled by a new host and *P1* is reused by a process on the new host.

In either case, as long as the original connection made by *C* from *P1* to *S* at *P2* is still alive somewhere, no other process can make a connection to *S* at *P2* originating from *C* at *P1*. Note that this restriction is imposed by the transport protocol itself rather than VNAT. IP addresses and port numbers are considered as "hard" resources that cannot be shared by different connections. When a connection migrates, even though it might be physically disassociated with a host, as far as the transport protocol is concerned, the IP address and port number are still in use. Although it is desirable to allow reuse of these resources on a different physical host, it would require changes in the transport protocol itself.

There is also a situation when processes are indeed able to reuse a pair of virtual addresses still in use to make a new connection. This happens with both endpoints of a connection migrate. For example, {*C, P1*; *S, P2*} can be used for a virtual connection between *C* and *S* and the virtual connection is migrated to between *C'* and *S'*. Now the virtual address {*C, P1*; *S, P2*} can be reused for another connection between *C* and *S*. However, if either endpoint of the new connection were to migrate to *C'* or *S'*, either separately or simultaneously, there would be a conflict of the virtual address {*C, P1*; *S, P2*} at either *C'* or *S'* or both.

One solution for this problem is to disallow reuse of virtual identifications, i.e., a virtual identification can not be used until a previous connection that uses the same virtual identification has been closed. This approach however has a couple of serious drawbacks. First, it requires leaving state behind on the "original" host to keep track of which virtual identifications have migrated away but are still in use. Second, it prevents efficient use of virtual identifications; it is really harmless as long as two connections with the same virtual identification do not share a common host.

The approach adopted by VNAT is to allow reuse of virtual identifications but deal with conflicts as they occur. Because of the condition for the conflict to occur, we believe that conflicts occur rarely under normal circumstance. When a virtual identification conflict does happen during a connection migration, a policy can be set on a per-connection base to resolve the conflict by either

aborting the migration or preempting the existing virtual connection on the target host.

3.5 Incremental usability

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.

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 provides an optional location server which can be deployed when migrating both endpoints of a connection simultaneously. However, the location server is not necessary in the common case when migrating only one endpoint of a connection at a time. As discussed in Section 5, VNAT can be implemented as a loadable kernel module 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.

VNAT further facilitates incremental usability in terms of performance. The computational cost of additional functionality in VNAT is only paid for by those users and applications that use it. In particular, non-migrating connections do not require any connection translation or connection migration functionality, resulting in almost no extra VNAT overhead for such connections.

4 Example Migration Scenarios

Let's now put all the pieces from previous sections together and describe two typical scenarios and see how VNAT migrates a live network connection. The two scenarios we describe are migrating one endpoint of a connection and migrating both endpoints of a connection.

4.1 Migrate one endpoint

Migrating one end of the connection is probably the most common case of mobility today. For example, users connect their laptops at work, suspend their laptops when they get off work and then resume at home to continue working. Or business users travel around with their laptops connecting to and disconnecting from different networks all the time.

Assume a client on host `10.10.10.10` opens a TCP connection to a server on host `20.20.20.20`. As shown in Figure 4-1, 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, unlike in Figure 3-2 and Figure 3-3, we are using the initial physical addresses as the virtual addresses based on our discussion in Section 3.1. In this example, we assume the client migrates and it doesn't matter whether a process or the whole client host migrates.

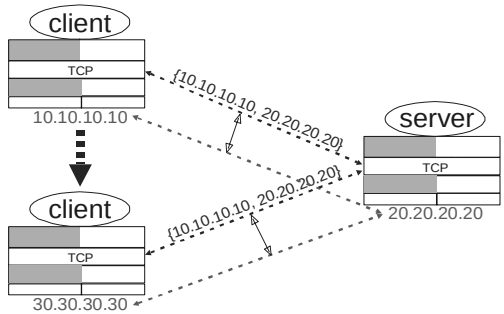

Figure 4-1: Migrate one endpoint

At the time of suspending the connection, the client will send a VNAT_SUSP_PRIMARY message and will receive a VNAT_SUSP_SECONDARY message and therefore claim the role of the primary. Secret key will be established between the client and the server and connection migration helper may be activated on the server for the migrating connection.

At the time of resuming the connection, the client VNAT at the new location will, being the primary, try to contact the "old" location of the server (rather than its own old location) following our VCMP. And the client will trivially locate the "new" server location. 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. And 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.

4.2 Migrate both endpoints

We now look at the scenarios when both endpoints of a connection migrate, which are a little bit more involving. Although we do not expect such cases happen frequently today, we envision that in the future when process migration technology has matured it would not be uncommon to see such cases. The example we consider here is when the whole client host migrates while only a process on the server migrates. For instance, this could occur when a client (laptop) is suspended and traveling and the process handling the connection on the server end is moved off to another machine due to maintenance or load balancing.

In this example, we assume for simplicity that the server will

claim the primary role at the time of suspending the connection since it has a "higher" IP address. We further assume that the migrated server process will be resumed first.

When the migrated server process is resumed, the server VNAT at the new location will, being the primary, try to contact the old location of the client following our VCMP. Since the whole client host has moved, the attempt will fail (step 1 in Figure 4-2). The server VNAT will then try to contact its own old location which will succeed since the server host didn't migrate, just the process (step 2). But when the server VNAT at the new location tries to find out the new location of the client from the server VNAT at the old location, it will fail because the client has not yet been resumed. However, the server VNAT at the new location will be able to store its new location on the server VNAT at the old location for the client VNAT to look it up when the client host is resumed. After the new location is stored, the migrated server process will continue to be suspended.

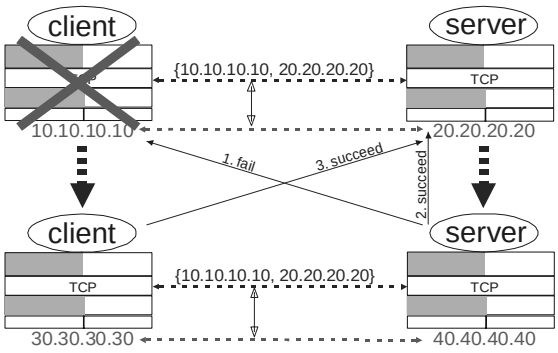

Figure 4-2: Migrate client host and server process

At a later time when the client host is resumed, the client VNAT will, being the secondary, try to contact the old location of the server. It will succeed and be able to retrieve the new location of the server process (step 3). Now both the client VNAT and the server VNAT at the new locations can verify their secret key and update their respective new physical addresses for the virtual connection `{10.10.10.10,20.20.20.20}`, which will be translated to and from the physical connection `{30.30.30.30, 40.40.40.40}`.

Because the original server host is still available after the server process has migrated, VNAT on both the client host and the new server host make use of the original server host to locate each other without having to go through a separate location server.

5 Implementation

We have implemented the VNAT system in the Linux operating system as a loadable kernel module. As a result, VNAT can be easily installed and used without modifying or recompiling the operating system kernel. The module can be loaded at any time and will commence virtualizing and translating connections as needed once it is loaded. Since none of the connections will have migrated before VNAT is installed, the virtual addresses used for those connections will be the same as the respective physical addresses, requiring no change in kernel state to virtualize those connections. As a result, VNAT can be used to provide connection mobility to connections that already exist even before VNAT is installed. All that is required is the creation of some VNAT internal state per connection, which can be easily obtained by reading the existing network kernel state for each connection.

In the following sections, we describe some of the implementation details of the connection virtualization, translation, and migration components of the VNAT system. Section 5.1 describes how system calls are intercepted to provide connection virtualization. Section 5.2 describes how VNAT uses the Linux netfilter system for connection translation. Section 5.3 describes how connection migration is supported by the VNAT daemon that runs on each system.

5.1 Intercept socket system calls

VNAT connection virtualization is implemented at the kernel socket layer by intercepting socket calls that open and close connections. All system calls on Linux goes through the entry routines in *arch/i386/kernel/entry.S*. These routines look up the system call number passed in a register and jump to the value stored in the system call table, essentially an array of function pointers. So the standard way of intercepting system calls on Linux is to write a kernel module that overwrites the relevant function pointer with a pointer to one's own code.

More specifically, we intercept three socket system calls: `accept`, `connect`, `close`. When a connection is being setup and before these calls reach the transport protocol, virtual addresses states are saved for the virtual-physical address mapping for the connection. VNAT saves a very small amount of state about the virtual connection so that if the virtual connection were to be suspended and migrated later, its physical mapping and other related OS states can be quickly looked up given its tuple. When the connection is closed, its associated address mapping states are cleaned up.

In addition, we also intercept the `getsockname` and `getpeername` calls, which may seem strange since these calls have nothing to do with opening and closing a connection. Recall in Section 3.4.1 we discussed VNAT's optional support for connectionless protocols which requires hiding physical host location from the application. And the `getsockname` and `getpeername` calls are what the applications use to find out the physical host location. In addition to supporting connectionless protocol, certain "strange" applications using connection-oriented protocol, notably FTP, explicitly check the connection endpoints (using `getsockname` and `getpeername` calls) for its protocol interaction. As a result, applications like FTP are either not willing or not prepared to be moved. VNAT also provides support for transparently migrating connections created by this type of application using the same optional mechanism it uses for supporting connectionless protocols. We note that applications like FTP are in the minority of existing network applications and can be identified and dealt with on a case by case base using this option. We emphasis again that the default behavior of VNAT is completely compatible with existing transport protocol behavior.

5.2 Instrument netfilter hooks

VNAT connection translation is entirely done through the *netfilter* system in the Linux 2.4 series kernel. Linux *netfilter* system is a packet filtering and mangling system [Russ01]. It instruments the IP protocol stack at well-defined points during the traversal of the stack by a packet. It provide hooks that invoke user-registered functions to process the packet at these well-define points.

For outgoing traffic, the VNAT system use the hooks NF_IP_LOCAL_OUT for destination address translation (DNAT) and NF_IP_POSTROUTING for source address translation (SNAT), respectively, to perform connection translation. For incoming traffic, the VNAT system uses the hooks NF_IP_PREROUTING for DNAT and NF_IP_LOCAL_IN for SNAT, respectively, to perform connection translation.

Using the same example as we used in Section 3.1, we will illustrate how the translation is done. When the client tries to send a packet, the TCP on the client side will construct a packet with source address `1.1.1.1` and destination address `2.2.2.2` since the connection has been virtualized. At the NF_IP_LOCAL_OUT hook, a DNAT is performed on the packet to translate `2.2.2.2` into `20.20.20.20`. This will allow correct routing functions to be performed. Once the routing decision for the packet has been made and before it is sent out to the appropriate interface, an SNAT is performed at the NF_IP_POSTROUTING hook to translate `1.1.1.1` into `10.10.10.10`. This is necessary for the reply packet to come back to the client. The process is illustrated in Figure 5-1. On the server, the reverse translation is done at the hooks

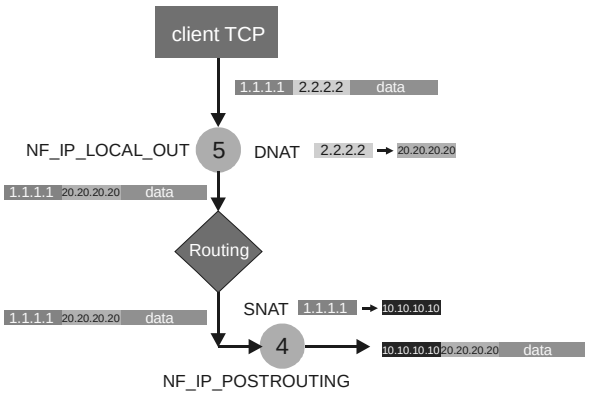

Figure 5-1: Client side connection translation

NF_IP_PREROUTING (DNAT) and NF_IP_LOCAL_IN (SNAT).

5.3 Automate migration tasks

All the VNAT connection migration related tasks are done by the VNAT daemon *vnatd* using VCMP without any manual intervention. *Vnatd* is a very simple kernel thread that listens on a well-known port (2031) and functions exactly the same as a normal server process except it runs entirely within the kernel address space for performance reasons.

When *vnatd* is notified with a suspension event of a virtual connection, it contacts the *vnatd* on the other end of the connection and carries out the routine tasks for suspending a connection. These tasks include establishing a secret key to protect the virtual-physical address mapping update for the migrating connection, negotiating migration roles to minimize the message exchanges for locating the peer's new location after migration, and activating connection migration helper to keep the connection alive during the migration.

After a virtual connection migrates, the *vnatd* will restore the connection when it is notified with a resumption event. After locating the peer's new location, *vnatd* on both sides will verify the secret key for the virtual connection established at suspension time, and exchange VNAT_UPD and VNAT_UPD_R messages to update their virtual-physical address mapping for the migrated virtual connection.

6 Experimental Results

We present some experimental data measuring the performance of our VNAT prototype implementation in Linux. We measured the performance overhead of VNAT in terms of throughput, latency, CPU utilization, and connection setup. We also measured the performance overhead associated with resuming a migrated network connection in a typical LAN environment. We have used VNAT with a suite of popular real world applications and discuss some of our experiences with the system.

To measure the performance overhead of VNAT, we compared the performance of three different system configurations: *Vanilla*, *Netfilter*, and VNAT. The *Vanilla* system is a stock Linux system without either netfilter or VNAT loaded into the kernel. The *Netfilter* system is a system with netfilter loaded into the kernel without any rules configured. The VNAT system is a system with both netfilter and VNAT loaded into the kernel. We measured the performance of VNAT for two cases, which we refer to as *VNAT1* and *VNAT2*. *VNAT1* represents a VNAT system with all connections not migrated and hence only performing connection virtualization. *VNAT2* represents a VNAT system with all connections migrated and hence incurs both connection virtualization and translation overhead.

Our experiments were conducted using two machines: an IBM ThinkPad 760 (TP760) with a 150Mhz Pentium CPU, 80 MB RAM, and a Linksys PCMPC100 10/100 Ethernet PC Card, and an IBM ThinkPad 770 (TP770) with a 266Mhz Pentium II CPU, 160 MB RAM, and a 3Com Fast EtherLink XL 3C575-TX 10/100 Ethernet PC Card. To ensure that we were accurately measuring the performance overheads of our systems as opposed to raw network link performance, we intentionally choose slow machines for our experiments so that the 100Mbps network link capacity could not be saturated easily.

We measured throughput, latency, and connection setup overhead using *netperf* [Jone96], a network performance benchmarking program. To minimize any discrepancies that might arise from using different tools, we used *netperf* as our primary measurement tool for the results presented here as it offers the types of measurements that were relevant in a single tool package. We ran the *netperf* client on the TP760 and the *netperf* server on the TP770. Three different types of *netperf* experiments were conducted: throughput, latency, and connection setup. The throughput experiment simply measures the throughput achieved when sending messages as fast as possible from client to server. The latency experiment measures the inverse of the transaction rate in which a transaction is simply the exchange of a request message and reply message of the same size. The connection setup experiment is the same as the latency experiment except that a new connection is used for every request/response transaction. This experiment simulates the interaction between a client and server in which many short-lived connections are opened and closed. In the throughput and latency experiment, we also measure the CPU utilization. For each type of experiment, eight different message sizes were used, ranging from 64 bytes to 8192 bytes and doubling in size, for a to-

tal of 24 different runs. Each experiment was run for 60 seconds with a given constant message size. Since these measurements focused on processing overheads on the end systems, we used a 100Mbps cross-cable between the machines to make sure that there were no other external factors such as hub contention or switching delay affecting the results.

We measured connection restoration overhead using a simple client and server program to represent typical TCP connections created by real world applications. The program opens a number of TCP connections and keeps these connections open so that VNAT can be used to migrate them. For this experiment, connections were migrated by simply changing the physical network location of the machine. No connection migration helper or security protection key was used. In these experiments, the TP760 was used as the client and the TP770 was used as the server. Since these measurements are impacted by network round-trip latencies, we conducted these measurements in a more realistic LAN environment by connecting the machines together through a 3Com OfficeConnect 3C16700 10Mbps hub. To suspend and resume connections, the connections were initially created over a separate 802.11b wireless LAN network using an Orinoco Gold PC Card in the client. The NIC card was then removed causing the connections to be suspended. The Linksys NIC card was then inserted into the client, connecting it in the new 10Mbps network test environment and moving the client to a new network location.

6.1 Throughput overhead

Figure 6-1 shows the throughput measurements for running the *netperf* throughput experiments. The results show that the *VNAT2* system has an overhead of around 9%-13% over the *Vanilla* system. However, as indicated by the results with the *Netfilter* system, about half of the overhead comes from just loading netfilter itself without any rules configured, which implies only about 4%-6% overhead is contributed by the *VNAT* system alone. The *VNAT1* system performs almost identically to the *Netfilter* system, indicating that connection virtualization requires almost no additional overhead.

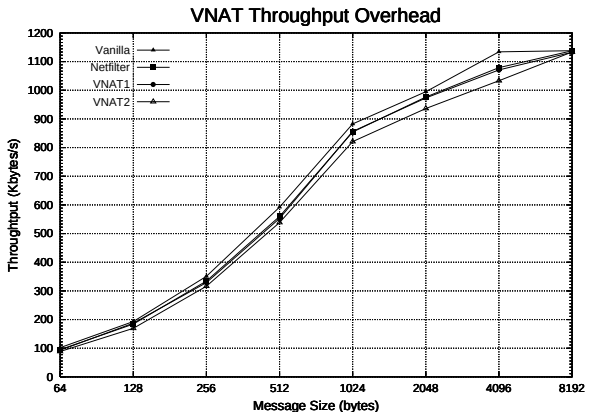

Figure 6-1: VNAT throughput overhead

Figure 6-2 shows the CPU utilization for the throughput experiment. When using the *VNAT2* system, the sender incurs about 8%-15% overhead while the receiver incurs about 29%-44% overhead compared to the *Vanilla* system. However, as indicated by the results with the *Netfilter* system, about half of the sender overhead and 94% of the receiver overhead are contributed by just loading netfilter. This implies that the overhead just due to VNAT is actually about 4%-7% for the sender and 2%-3% for the receiver. The *VNAT1* system CPU utilization is almost identical to the *Netfilter* system, again indicating that connection virtualization requires almost no additional overhead, as expected.

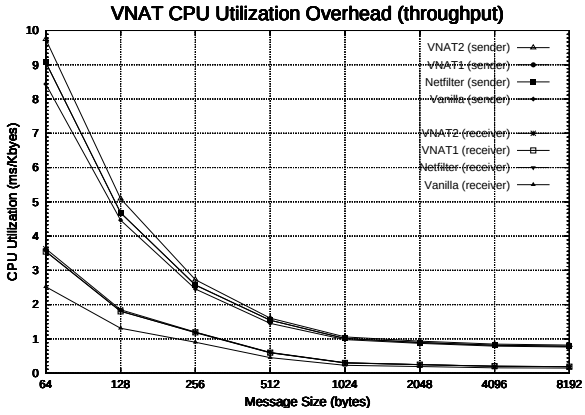

Figure 6-2: Throughput CPU utilization overhead

We argue that it is more fair to compare a VNAT system with a *Netfilter* system rather than a *Vanilla* system. Due to increased security concerns, major Linux distributions such as RedHat are now shipped with a default setup of "medium" firewall protection which requires netfilter to be loaded. As a result, we expect that an increasing number of Linux hosts will have netfilter loaded by default.

6.2 Latency overhead

Figure 6-3 shows the latency measurements for running the *netperf* latency experiments. The results show that the *VNAT2* system

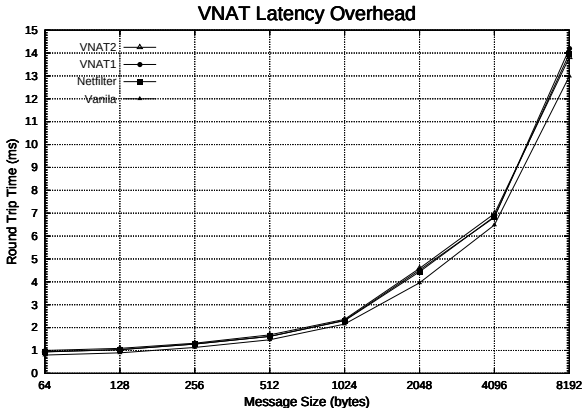

Figure 6-3: VNAT latency overhead

has an overhead of around 14%-20% over the *Vanilla* system. However, as indicated by the results with the *Netfilter* system, more than 70% of the overhead comes from just loading netfilter itself without any rules configured. So the *VNAT2* system alone effectively contributes about 4%-6% overhead, similar to the result of throughput measurement. We again notice that the *VNAT1* system performs identically to the *Netfilter* system, indicating that connection virtualization requires almost no additional overhead.

Figure 6-4 shows the CPU utilization for the latency experiment. When using the *VNAT2* system, the sender incurs about 7%-20%

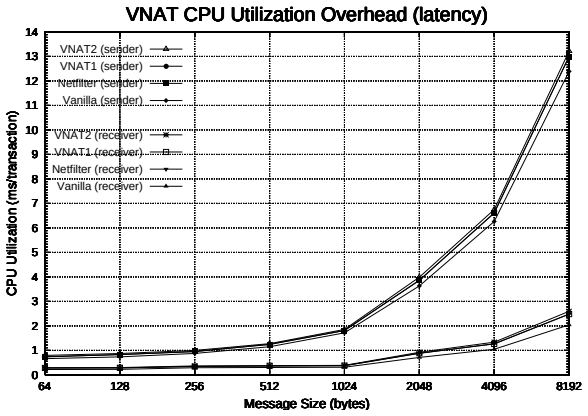

Figure 6-4: Latency CPU utilization overhead

overhead while the receiver incurs about 27%-34% overhead compared to the *Vanilla* system. However, as indicated by the results with the *Netfilter* system, about 87%-95% of the sender overhead and 72%-79% of the receiver overhead are contributed by just loading netfilter. This implies that the overhead just due to VNAT is actually about 2%-7% for the sender and 5%-6% for the receiver. Again, the *VNAT1* system incurs virtually no overhead over a *Netfilter* system.

6.3 Connection setup overhead

Figure 6-5 shows the latency measurements for running the *netperf* latency experiments with a new connection for each transaction to measure connection setup overhead. Since connection

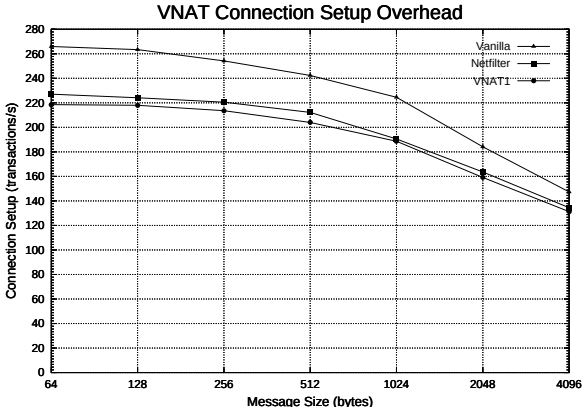

Figure 6-5: VNAT connection setup overhead

setup in VNAT occurs before migration, there is no translation overhead associated with connection setup, so VNAT results are only shown for the *VNAT1* system. The results show that *VNAT1* system has an overhead of about 11%-18% over the *Vanilla* system, of which about 80% is contributed by just loading netfilter. So the VNAT system alone contributes about 2%-3% of the overhead.

6.4 Connection restoration overhead

Table 1 shows the measurements for running the experiments to measure connection restoration overhead. No comparisons with the *Vanilla* or *Netfilter* systems are shown since they do not provide connection restoration functionality. The results show that on average it takes about 47 milliseconds (including the round trip delay) to restore a connection and the overhead stays fairly constant. Based on the VCMP interaction and its implementation using TCP in our prototype, we can infer that once the migrated endpoint(s) located each other, it would take two round trips and some local processing to restore the connection. The round trip time obviously depends on the actual physical network condition, while the bulk of the local processing involves searching a virtual connection given its "tuple". The results show that most of the time required to restore the connection is due to local processing as opposed to network latency. Depends on the particular implementation, the cost of searching for a virtual connection could range from $O(\log(n))$ with a binary search to $O(n)$ with a linear search, where n is the total number of virtual connections. Our current implementation uses a linear search and this is reflected in the results.

Total number of connections	Total restoration time (seconds)	Restoration time per connection (milliseconds)	Average round trip delay (milliseconds)
10	0.434	43.4	3.104
50	2.374	47.5	3.134
100	4.856	48.6	3.159
500	24.657	49.3	3.147

Table 1: VNAT connection restoration overhead

6.5 Migrate popular network applications

We tested the migration capability of our VNAT system with a suite of popular real world Linux applications, including but not limited to:

- telnet client and server (standalone and via xinetd)
- ftp client and server (standalone and via xinetd), both active and passive mode
- ssh client and server
- mozilla/netscape/opera and apache
- Ximian evolution and qpopper/sendmail
- slrn and innd
- VNC thin client and VNC server
- remote X client and X server

All the above applications worked over a virtualized connection right out of the box. We were able to migrate live connections created by all the above applications and the connections stayed alive as if nothing had happened. Among all the application we tested, FTP was the only one that we had to turn on the special option we mentioned in Section 5.1 in order to migrate its connections. We are glad to see that the majority of today's network applications behave as we have expected. Rather than relying on transport connection properties for their application logic, they use the transport protocol solely for the purpose of transporting data.

7 Conclusions

We have introduced in this paper 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 virtual-

ization, 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.

We have implemented a prototype of VNAT in the Linux operating system and we have shown it performs with very low overhead and works very well with a wide range of popular real world applications. Our results on an untuned prototype show that there is no noticeable overhead for connections that do not migrate, and a fairly constant and small 2%-7% overhead on top of standard Linux distributions with netfilter loaded for migrated connections. These results are due to the fact that VNAT connection virtualization only introduces very small overhead at connection setup time and connection translation only performs simple deterministic NAT functions.

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.

8 References

[DH79] W. Diffie and M. Hellman, *Privacy and Authentication*, Proc IEEE, **67**(3):397-429, March 1979.

[Jone96] R. Jones, *Netperf: a Network Performance Benchmark*, Information Networks Division, Hewlett-Packard Company, February 1996. http://www.netperf.org/netperf/NetperfPage.html

[KA98] S. Kent and R. Atkinson, *IP Authentication Header*, RFC2402, IETF, November 1998.

[MB98-1] D. Maltz and P. Bhagwat, *TCP splicing for application layer proxy performance*, IBM Research Report 21139 (Computer Science/Mathematics), IBM Research Division, March 1998.

[MB98-2] D. A. Maltz and P. Bhagwat, *MSOCKS: An Architecture for Transport Layer Mobility*, Proceedings of the IEEE INFOCOM'98, San Francisco, CA, 1998.

[MDW99] D. Milojicic, F. Douglis, and R. Wheeler, *Mobility*, ACM Press, 1999.

[OR01] G. O'Shea and M. Roe, *Child-proof Authentication for MIPv6 (CAM)*, ACM Computer Communication Review, **31**(2):4-8, 2001.

[Perk01] C. Perkings, *IP Mobility Support for IPv4, revised*, draft-ietf-mobileip-rfc2002-bis-08.txt, Internet Draft, September 2001.

[Perk96] C. Perkins, *IP Mobility Support*, RFC2002, IETF, October 1996.

[QYB97] X. Qu, J. X. Yu, and R. P. Brent, *A Mobile TCP Socket*, International Conference on Software Engineering (SE '97), San Francisco, CA, November 1997.

[Russ01] R. Russell, *Linux 2.4 Packet Filtering HOWTO*, Linux Netfilter Core Team, November 2001. http://netfilter.samba.org/

[SAB01] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, *Fine-Grained Failover Using Connection Migration*, Proceeding of the Third Annual USENIX Symposium on Internet Technologies and Systems (USITS), March 2001.

[SB00] A. C. Snoeren and H. Balakrishnan, *An End-to-End Approach to Host Mobility*, Proceedings of 6th International Conference on Mobile Computing and Networking (MobiCom'00), Boston, MA, August 2000.

[SCFJ01] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, draft-ietf-avt-rtp-new-11.txt, IETF, November 2001.

[SE01] P. Srisuresh and K. Egevang, *Traditional IP Network Address Translator (Traditional NAT)*, RFC3022, IETF, January 2001.

[SH99] P. Srisuresh and M. Holdrege, *IP Network Address Translator (NAT) Terminology and Considerations*, RFC2663, IETF, August 1999.

[ZD95] Y. Zhang and S. Dao, *A "Persistent Connection" Model for Mobile and Distributed Systems*, 4th International Conference on Computer Communications and Networks (ICCCN), Las Vegas, NV, September 1995.

[ZM01] V. C. Zandy and B. P. Miller, *Reliable Sockets*, Unpublished, June 2001.