Diplomarbeit

IP Address Translation

URL: http://www.csn.tu-chemnitz.de/HyperNews/get/linux-ip-nat.html

Michael Hasenstein, 1997

Contents

1 Introduction

The Internet has grown immensely over the last few years. Today one estimates that several thousand new hosts get newly connected every day. This vast growth rate has caused considerable problems, due to the fact that the Internet's infrastructure and its protocols were designed decades ago, when only a few still countable number of hosts used it and it was unforeseeable that there would ever be such an immense need for connectivity. As a result, the Internet's transport protocol IPv4 does not provide enough unique addresses for all the new hosts on the internet. The number of distinct IPs is still large enough, but because of routing issues they can only be given away in relatively big chunks.

One consequence was that people started developing a new Internet protocol, known today as IPv6 or 'IP next generation', that should overcome limitations imposed by IPv4. Among other improvements they considerably increased the address space so that it will hopefully last for the next few decades. However, since developing a new protocol which in addition is of such major importance as the IP protocol takes some years and the migration can also not be done in a day it was clear that we would still have to live with IPv4 for a couple of years. That meant to find solutions to the problem of scarce address space, since this was the most pressing one. One such solution is to use private internal addresses on ones own network and make connections to the internet through proxies, so that no direct IP-connectivity is needed. This is of course only possible if there are proxies for the application/protocol that is being used, but only a single IP address is needed for an entire network.

A more general solution is to convert those private, internal addresses to official addresses when crossing the border to the internet. Since the number of hosts that communicate over the internet at a given time is considerably lower than the total number of hosts, that will save address space, because only those hosts currently communicating will dynamically get an official address assigned by a NAT-router. This is (mostly) application independent since it happens on the IP protocol layer, where no application specific information is stored. Ideally, the translation will be entirely invisible to the applications.

There are many products available today that do NAT. The reason that I choose that topic is because I wanted to try to find a more general approach. Most of the products I have seen, including the system that I selected for an example implementation of my ideas, Linux, only implement $m:1$ IP-translation ($m>1$), some also support $m:n$ translation with $m=n$ (static NAT) or $m \neq n$ (dynamic NAT) and $m,n>1$. I will try to enhance this by introducing such things like a virtual IP address space for the kernel and even find new applications for NAT-techniques.

In addition, working on and with IP address translation one quickly finds that this technique can do much more than helping to solve the address space problem. I will discuss address translation in detail in the following sections. A part of this work is the example implementation of the techniques discussed using a more and more widely used system, Linux (kernel version 2), as a base.

2 History

IP address translation is a relatively new technology. The first papers on the subject were written in the early 90s. NAT was introduced as a short term solution for the address space problem and a complementary technology to CIDR. To understand why the NAT idea was born we have to look back at the situation at the beginning of the decade and some technologies that have been introduced in order to solve the most pressing problems of those years, IP address depletion and scaling in routing. There are three approaches: CIDR, private IPs and NAT.

2.1 Classless InterDomain Routing (CIDR)

In the early 90s it became apparent that the number of free IP addresses would soon be depleted. The total number of IPs was large enough (and still is), but because of routing issues — routing tables can not grow infinitely due to memory and timing problems — they could only be used in blocks. There are three classes of IPs: class A, B and C addresses.

	7	15	23	31
Class A	0 Netz-ID	Host-ID		
Class B	1 0 Netz-ID		Host-ID	
Class C	1 1 0 Netz-ID			Host-ID
Class D	1 1 1 0 (multicast)			
Class E	1 1 1 1 (experimental)			

For each block in a class one entry in a routers routing table was necessary. Class A, allowing more than 16 million hosts, is much too large for most purposes, besides that only a few class A networks are available. Class C networks on the other hand (254 hosts) are too small. Class B allows for some ten thousand addresses, a good number for medium sized organizations, but in 1992 already half of the available class B address space was in use, with an Internet growing at more than 100% annually. That is why many newly connected organizations ended up with several class C networks, because there were many left of them, which in turn caused a routing table overflow on some devices, because one entry for every single class A, B or C network, respectively, was needed.

This is the point where CIDR comes into the game (See [2]). CIDR makes it possible to have just one routing entry in a router for a whole block of class C networks. It introduces some rules how to build these blocks – you can't use arbitrary networks. Note that the problem of scaling in routing mainly relates to Internet backbone routing, since the backbone routers have to know all networks on the Internet. Within an organization you can use any routing strategy, whatever you like best. Now that we have built blocks of class C addresses we give them to the Internet providers who in turn give them to their customers, but the latter does not matter. The goal of CIDR was to reduce routing entries in the backbone routers, which began to overflow due to the huge number of entries needed for class C networks (up to about 2 million). After implementing CIDR that number decreased significantly, allowing some more time for developing long term solutions (especially IPv6).

A problem with CIDR is when a customer changes the provider but wants to keep the IP addresses: The old provider still announces the route to the entire block while the new provider must announce a route to the extra net -> there are two routes for that net, the CIDR route and the single route. One possible solution is to use the most specific route, another one is NAT. The first one has the disadvantage of needing a new entry in a backbone router, which CIDR should have prevented. This can be avoided by using NAT, so that the customer keeps the addresses of the first provider for internal use but uses address translation to translate them into addresses of the new provider when communicating over the Internet.

2.2 Internal IP addresses

With the proliferation of TCP/IP technology even outside the Internet more and more enterprises began reserving IP address space for sole internal communication. So far there was only one global IP pool out of which all addresses were taken, and everyone needing IPs got globally unique addresses. This was unnecessary in most cases since the majority of enterprises that suddenly needed IP addresses used them only internally, and even when they connected their enterprises networks to the Internet they did not need unique addresses for all their hosts, since for reasons of security and others (e.g. caching web traffic) no direct IP connectivity was allowed between internal enterprise computers and hosts on the Internet. It was therefore just a question of time that special IP addresses out of the global pool were reserved for internal IP networks, as described in [3].

Now everyone can use one of the reserved class A, B or C networks for their internal communication. These addresses can't be used on the Internet, of course, since they will not get routed. Advantages are that no reservation has to be made in order to get address space, and everybody can pick the addresses best suited for a purpose, e.g. now everyone can use one of the rare class B networks, which makes internal routing easier than having lots of different class C networks.

There are disadvantages, too, but they are by far outweighed by the address space saved. One such disadvantage is that in an ever changing environment nobody knows if networks, that are independently administrated today and have chosen the same address space out of the reserved pool, will be directly connected in the future. This may be the case within enterprises, where before the network age many smaller networks existed independently, or it may even concern different companies that have to merge their networks for some reason. Again, network address translation could be of help in this case.

2.3 IP address translation

CIDR served as a short term solution for the routing table problem, and therefore also for the problem of address depletion, because now the many class C networks were available for use. To further ease the situation with IP addresses address space was reserved for pure internal use, simultaneously IPs were only given away for those who wanted to connect computers to the Internet.

As an additional measure some people proposed to reuse IP addresses [1]. The idea was that only a small percentage of hosts communicated across network boundaries at a time, so only those hosts would need a globally unique IP. Of course you can't change the system's IP each time your computer wants to establish a connection with another computer outside your network, so it was proposed to let a special device, a so called NAT-router, assign a global IP to a connection dynamically. Since the process should be transparent for both endsystems, assigning an IP meant to exchange the local IP numbers in the IP packets with the global IPs. That means you only need a relatively small number of global IPs and only that many hosts can communicate across the borders of your network simultaneously.

Disadvantages are that your hosts are not reachable from the outside (which may also be an advantage), that the number of simultaneous connections is limited or that the process might not be completely transparent due to the fact that there are protocols like FTP, that transmit their IP to the other host.

A special form of this approach to NAT is to have just one official address and to use just this address for all communication. To allow more than one host to communicate at a time not just the IP, but also the TCP port numbers are replaced, using a different port number for each connection. The number of simultaneous connections is limited only by the number of ports available for the outgoing connections. That Linux implements this form of NAT (called masquerading in Linux) is one of the reasons that this is being widely used today.

All the above ideas have been developed as short term solutions to overcome the most pressing problems caused by the growth of the Internet. They are all meant to be abandoned as soon as the new Internet transport protocol, IPv6, is available and the migration to it has been finished. I think, however, that some of the ideas will and should survive longer. CIDR can be found in IPv6 in a similar form, since it is obvious anyway. Private addresses may be useful under certain circumstances even in the future, e.g. it is not always possible or even desirable to ask a central organization for address space, even if there is enough, possibly because you need it now and for purely internal use. IP address translation, at last, can do much more than what its inventors intended it to do, as I am going to show next.

3 NAT and Networks

3.1 Introduction

When network address translation was invented it was a mere hack to circumvent IP shortage. Meanwhile it has proven to be useful in completely different fields nobody had thought of at the beginning, and there are probably many more useful applications that have not been found yet. In that context I want to try to explain the role NAT currently has and that it might gain in the future, proving that it is more than a short term solution and that it will stay with us for much longer, especially when we look at the current state of the IPv6 implementation. Experiments done by some people have shown that the IPv6 protocol itself does not cause many problems so migration could be swift, but lots of applications cause problems and it is therefore likely that IPv4 will be the major Internet- and Intranet-protocol for longer than expected.

Before we begin explaining NAT's role in todays and future networks I want to show in what different areas NAT is being used today. The explanations will be made from a technological point of view, i.e. I will not try to give advice on how a special kind of NAT should be used. The following sections are just an overview. The details that have to be thought of when implementing NAT or examining implications of using NAT are laid out in the chapter thereafter.

I have divided the overview into two parts. I call the first one *classic NAT*, meaning that this is the original NAT as invented in the early nineties which is covered by RFC 1631, mainly meant to save IP address space on the Internet. The second part introduces more recent forms of NAT-usage that do not serve the original purpose but opened up additional fields.

3.2 Classic NAT Techniques

Speaking about NAT we must know that address translation can be done statically or dynamically. In the first case the assignment of NAT-IPs to original IPs is unambiguous, in the latter case it is not. In static NAT a certain fixed original IP is always translated to the same NAT IP at all times, and no other IP gets translated to the same NAT-IP, while in dynamic NAT the NAT IP depends on various runtime conditions and may be a completely different one for each single connection.

In the following sections m, n are defined as follows:

m: number of IPs that need to be translated (original IPs)

n: number of IPs available for translation (NAT IPs)

3.2.1 Static Network Address Translation

$m : n$-Translation, $m, n \geq 1$ and $m = n$ $(m, n \,\epsilon\, N)$

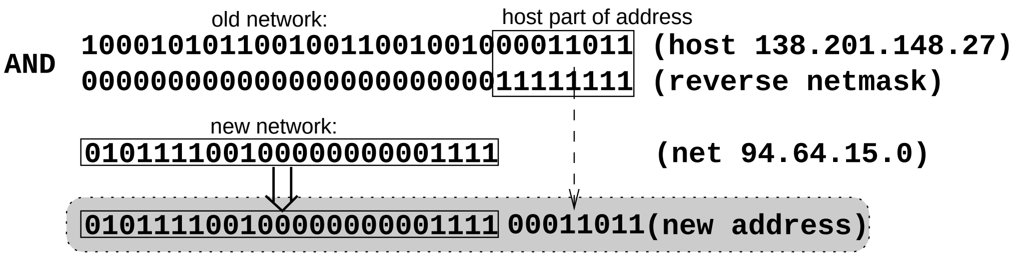
With static address translation we can translate between IP networks that have the same size (contain the same number of IPs). A special case is when both networks contain just one IP, i.e. the netmask is 255.255.255.255. This NAT strategy is easy to implement, since the entire translation process can be written as one line containing a few simple logic transformations:

new-address = new-network OR (old-address AND (NOT netmask))

In addition, no information about the state of connections that are being translated needs to be kept, looking at each IP packet individually is sufficient. Connections from outside the network to inside hosts are no problem, they just appear to have a different IP than on the inside, so static NAT is (almost) completely transparent.

Example:

- NAT rule: translate all IPs in network 138.201.148 to IPs in network 94.64.15, netmask is 255.255.255.0 for both
- now 138.201.148.27 is translated to 94.64.15.27, and so on



3.2.2 Dynamic Address Translation

$m : n$-Translation, $m \geq 1$ and $m \geq n$ $(m, n \,\epsilon\, N)$

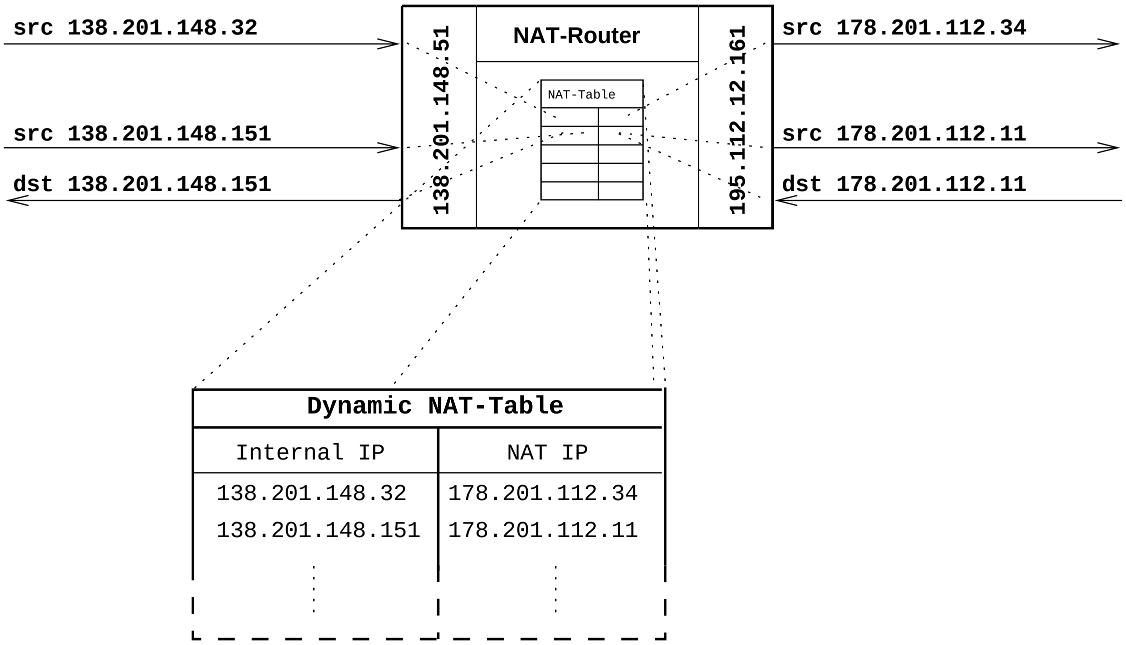
Dynamic address translation is necessary when the number of IPs to translate does not equal the number of IPs to translate to, or they are equal but for some reason it is not desirable to have a static mapping. The number of hosts communicating is generally limited by the number of NAT IPs available. When all NAT IPs are being used then no other connections can be translated and must therefore be rejected by the NAT router, for example by sending back 'host unreachable'. Dynamic NAT is more complex than static NAT, since we must keep track of communicating hosts and possibly even of connections which requires looking at TCP information in packets.

As mentioned above, dynamic NAT may also be useful when there are enough NAT IPs, i.e. when $m = n$. Some people use this as a security measure: it is impossible for someone outside a network to get useful IP numbers to connect to of hosts behind a NAT router doing dynamic address translation by looking at connections that take place, since next time the same host may connect using a completely different IP. In this special case even having more NAT IPs than IPs to be translated ($m < n$) may make some sense.

Connections from outside are only possible when the host that shall be reached still has a NAT-IP assigned, i.e. if it still has an entry in the dynamic NAT table, where the NAT router keeps track of which internal IP is mapped to which NAT IP. For instance, non-passive FTP sessions, where the server attempts to establish the data-channel, are no problem (for protocol specific problems see Section 23), since when the server sends its packets to the FTP-client there is already an entry for the client in the NAT-table, and it is extremely likely it still contains the same client-IP to NAT-IP mapping that were there when the client started the FTP-control channel, unless the FTP session has been idle for longer than the timeout of the entry.
However, if an outsider wants to establish a connection to a certain host on the inside at an arbitrary time there are two possibilities: the inside host does not have an entry in the NAT-table and is therefore unreachable, or it has an entry, but which NAT-IP must be used is unknown, except, of course, the IP to connect to is known because the internal host is communicating with the outside. In the latter case, however, only the NAT-IP is known but not the internal IP of the host, and this knowledge is valid only while the communication of the internal host takes place plus the timeout of the entry in the NAT routers table.

Example:

- NAT rule: dynamically translate all IPs in (class B) network 138.201 to IPs to IPs in (class C) network 278.201.112
- each new connection from the inside gets assigned an IP from the pool of class C addresses, as long as there are unused addresses left
- if a mapping already exists for the internal host this one is used instead
- as long as the mapping exists the internal host can be reached via the IP that has been (temporarily) assigned to it



Masquerading (NAPT)

$m : n$-Translation, $m \geq 1$ and $n = 1$ $(m, n \,\epsilon\, N)$

A very special case of dynamic NAT is $m : 1$-translation, a.k.a. masquerading which became famous under that name because Linux can do it. It is probably the kind of NAT-technique that is used most often these days. Here many IP numbers are hidden behind a single one. In contrast to the original dynamic NAT this does not mean there can be only one connection at a time. In masquerading an almost arbitrary number of connections is multiplexed using TCP port information. The number of simultaneous connections is limited only by the number of TCP-ports available.

A special problem of masquerading is that some services on certain hosts only accept connections coming from privileged ports in order to ensure that it does not come from an ordinary user. The assumption that only the superuser can access those ports is not valid, since on DOS or Windows machines everybody can use them, nethertheless, some programs rely on this and cannot be used over a masqueraded connection. The Linux implementation uses no privileged ports for masquerading to avoid interfering with 'regular' connections to these ports. Masquerading usually uses ports in the upper range, in Linux this range starts at port 61000 and ends at 61000+4096, which is the default and can easily be changed by editing linux/include/net/ip_masq.h. This also shows that the Linux implementation by default only allows 4096 concurrent connections. To allow masqueraded connections on ports outside of such a port range requires keeping and managing even more information about the state of connections. Linux, for example, simply treats all packets with *destination IP = local IP* and *destination port is inside the range used for masquerading*, as packets that have to be demasqueraded, i.e. they are answers to packets that have been masqueraded on their way out.

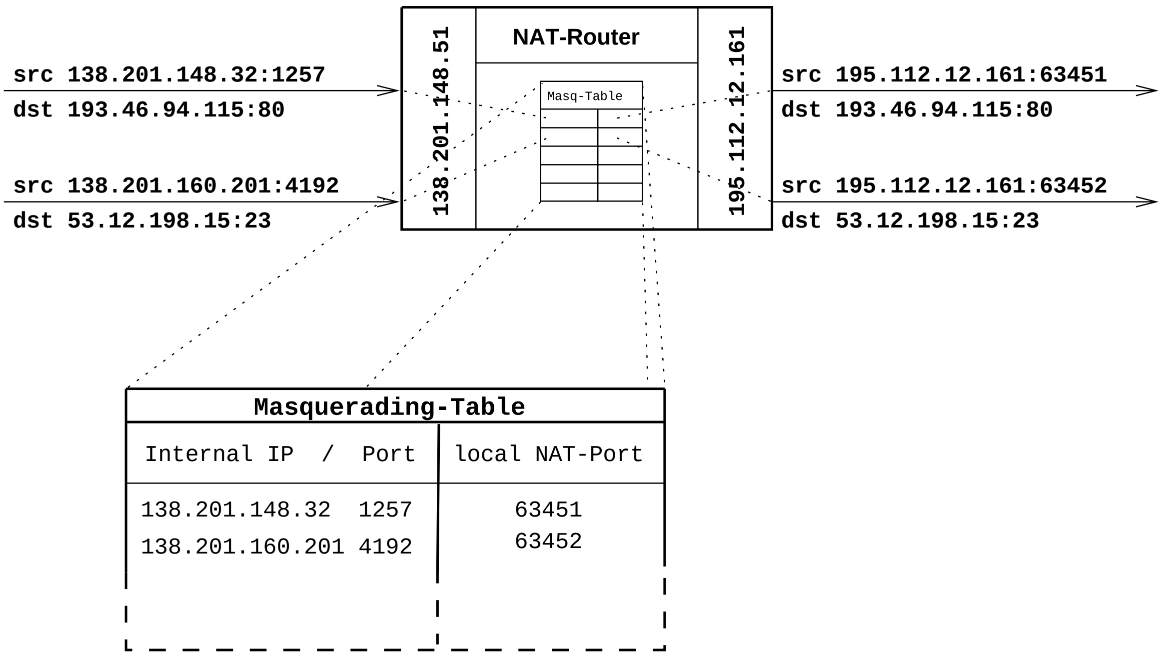
Incoming connections are impossible with masquerading, since even when a host has an entry in the masquerading table of the NAT device this entry is only valid for the connection being active. Even ICMP-replies that belong to connections *(host/port unreachable)* do not get through to the sender automatically but must be filtered and relayed by the NAT-routers software.
While it is true that incoming connections are impossible we can take additional measures to enable them, but they are not part of the masquerading code. We could, for an example, set up the NAT-device so that it relays all connections coming in from the outside to the telnet-port to a host on the inside. However, since we have just one IP that is visible outside for enabling incoming connections for the same service but for different hosts on the inside we must listen on different ports on the NAT-device, one for each service and internal IP. Since most applications listen on well-known ports that cannot be easily (and transparently!) changed, this is quite inconvenient and often no option, especially not for public services. The only solution is to have as many (external) IPs as the number of services that shall be provided. An external IP can still be shared by different services, and then be remapped to different internal IPs using NAT, but that is not part of masquerading, then.

Example:

- NAT rule: masquerade the internal network 138.201 using the NAT routers own address

- for each outgoing packet the source IP is replaced by the routers (external) IP, and the source port is exchanged against an unused port from the range reserved exclusively for masquerading on the router

- if the destination IP of an incoming packet is the local router IP and the destination port is inside the range of ports used for masquerading on the router, the NAT router checks its masquerading table if the packet belongs to a masqueraded session; if this is the case, the destination IP and port of the internal host is inserted and the packet is sent to the internal host



The greatest advantage of masquerading for many people is that they only need one official IP-address but the entire internal network can still directly access the Internet. This is so important because IP addresses have become quite expensive. As long as there are application level gateways we do not need any IPs or any kind of NAT and one IP is still enough, but for some protocols, e.g. all UDP based services, there is just no gateway so direct IP connectivity is necessary.

At the time of this writing there existed an Internet Draft (which I should not reference here, since it is just a draft) from the same people who wrote RFC 1631 (NAT). It explains masquerading, that they call Network Address Port Translation (NAPT), in great depth. There is no IETF-paper (none that I could find, at least) on more recent forms of NAT like the ones introduced in the following chapters, although there are (commercial) implementations of them. It seems like for the IETF NAT only exists for helping to solve the classical Internet address space shortage problems described above.

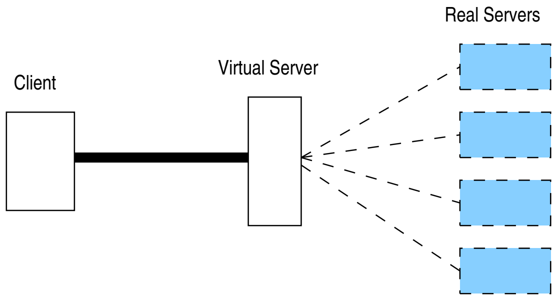
3.3 Other NAT Techniques

While classic NAT technologies serving the purpose of saving valuable address space have been known for a long time (when measured in Internet-time) other uses for this technology have been found, that are independent of the problems NAT was proposed to help to solve. That means, NAT will not be obsoleted by the long term solutions (like IPv6) developed to solve all of the problems discussed at the beginning of this paper, since completely new fields of application have been found.

Some of these new technologies are introduced below. I write 'some', because I am sure more are to follow that nobody even dreams of today. Not that NAT is vitally important, we could probably live without it somehow, but that is true for many more things developed or invented in the past $X * 1.000$ years. It just can make life easier sometimes. It can also make it harder — again, this is true not only for NAT, since everything can be used for good *and* bad.

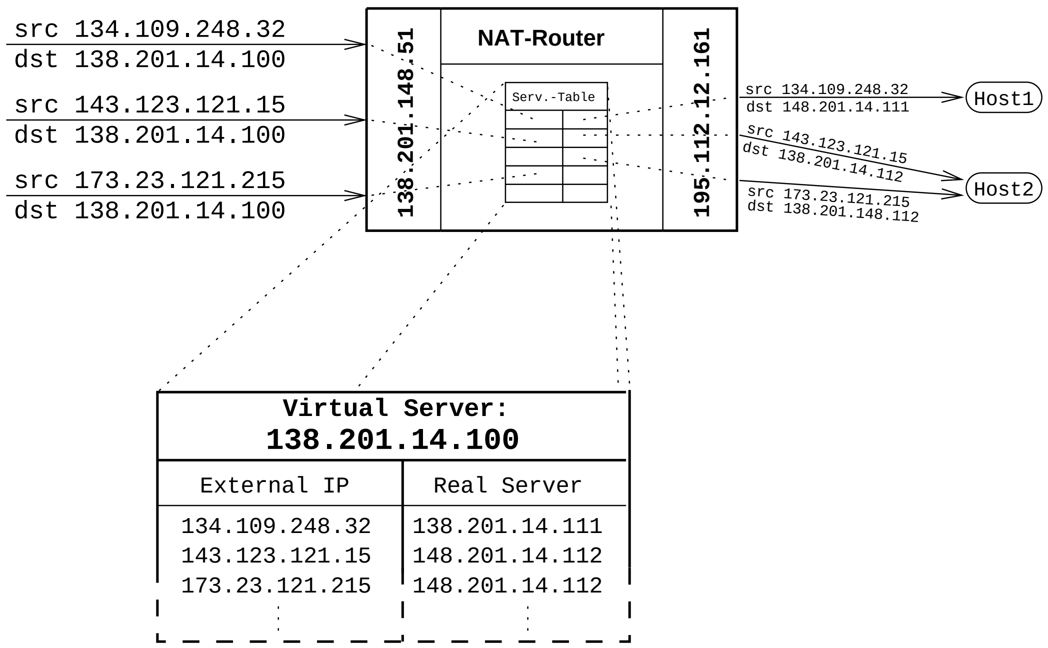
3.3.1 Virtual Servers (Load Balancing)

Since the notion of a *virtual server* gets used in a variety of ways, here it means to have an IP that represents the virtual server, for which there is no real device to which this IP belongs to, and several other IPs, for each of which there is a real device. When a host connects to the virtual IP the NAT device exchanges the destination address, which is the virtual IP, against an IP of one of the real devices. Depending on the algorithm that is used to select the (real) server IP, virtual servers can serve various purposes.



Example:

- Nat rule: create a virtual server using IP 138.201.14.100
- use the two hosts with the IPs 138.201.14.111 and 148.201.14.112, respectively, as (real) servers for the virtual server
- now connections from outside to the virtual server are remapped by the NAT-router to use one of the two hosts
- which of the two hosts available is actually used for a new connection (that is not yet in the virtual server table) depends on the algorithm used, whereby any algorithm one can think of can be plugged in



Load Balancing

The algorithm used to determine which real IP to use may, for instance, check the load on each of these servers (e.g. by counting the packets/second that flow through the NAT device to the servers) and select the IP of the server with the lightest load, thereby achieving a relatively equal distribution of the traffic on the virtual IP over the servers. The number of algorithms that might be used here is uncountable, but virtually all of them will be compromises, since the notion *load* cannot be clearly and uniquely correct defined. We could, for example, run a daemon on each of the servers that informs the NAT-router about the load on that machine — however the machine's load maybe defined — and remap new connections to the system where this number is the lowest. This requires communication between the hosts and the NAT-router, so we might prefer to use data available on the router anyway, such as the number of connections that are currently being remapped to a host, or we might use data that is not naturally available on the router but that can easily be collected, such as the average number of bytes or packets per second, that a host currently handles.

The critical element here will always be the algorithm used to achieve equal load distribution. The more accurately we try to measure the load, the more data we need to handle on the NAT-router and the harder it gets to collect the data in the first place. This is somehow similar to Heisenberg's Uncertainty-Principle in Quantum theory, so we must find a way to minimize

the tradeoff between what it costs to determine the load and what possible use we can get out of that knowledge.
Even when we assume we could find a way to accurately determine the load (based on an ultimate definition of what *load* actually is) practice does not honour our efforts: Since an IP packet has a minimum size (like a quantum in physics) and, in addition, we can only select to which host we want to send it when a new connection is opened, we will never be able to achieve an infinitely equal load distribution. Of course, the above is not of any practical interest, but it certainly is interesting. It does have a practical impact in so far that it shows us when it is useless to refine the algorithms any further.

There are numerous other approaches to load balancing, most of them on a higher (user) level. One example is described in RFC 1794 (DNS Support for Load Balancing). Here the DNS controls the load of machines by giving away the IP of the least busy machine when queried. Since DNS-queries will be cached by subsequent DNS-servers the control is severely limited, but it will work quite well if there are many queries and when they come from a lot of different clients. However, even if load balancing may work under certain circumstances this approach will not help when one of the servers fails and is no longer available, since even if this particular IP is no longer given in queries, it still is in many caches.
Another example is the famous cache program *squid*, which uses complicated algorithms to find out where to get an object from[7]. This solution is no general solution but limited to this particular program. With NAT on the other hand we can do load distribution for a much larger variety of services, as long as they are based on IP. Squid serves a different purpose so a comparison does not work, I used it as an example where the intelligence to do load balancing and to collect the data is implemented in all the programs involved and not in an independent central authority.

Backup Systems

Virtual servers can also be used to achieve a higher availability of a service, since the service provided by the virtual server is available as long as any one of the real servers is able to provide it. When the probability of a failure of a single server is p, the probability that a virtual server formed by n real servers fails can be calculated as follows:

$p_1, \ldots, p_n$: probability of a failure of server n; $n \epsilon N$ is number of servers that provide the virtual service

p_{NAT} : probability of a failure of the NAT-router, which fails independently of the other devices

p_{virt} : probability of a failure of the virtual server, when the individual servers fail independently of one another

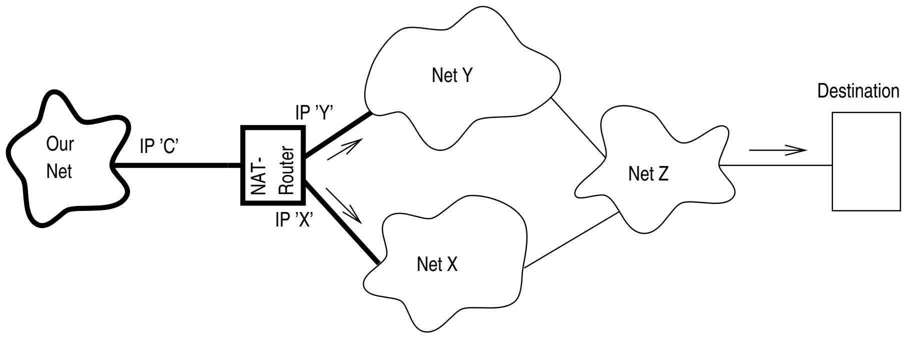
$$p_{virt} = 1 - \left(\left(1 - \prod_{i=1}^{n} p_i\right) * (1 - p_{NAT})\right)$$

Of course, the setup used above for load balancing must be enhanced in order to make changes to the list of servers used by the NAT-router to remap connections to as soon as one of the real servers is no longer available. This, however, does not belong in the NAT-code but can better be controlled in higher layers, even from shell-scripts. There must than be a mechanism to remove servers from the virtual server table. Since there must be an interface to build the virtual server table in the first place anyway it is not hard to add features, so that IPs can be added and removed from a virtual server during run time. With this setup we have combined the two functions load balancing and high availability, using virtual servers, and it is even transparent to all hosts, users and programs using the virtual service.

3.3.2 Multiple Routes per Destination

We have seen above that we can use NAT to distribute load over several hosts and achieve a higher availability of host based services. Can we use NAT to do the same for networks? Yes, we can. Above we have introduced virtual hosts that represent several real hosts, we can also create a virtual network connection that consists of several real wires which has the same advantages and disadvantages as the virtual server technique.

How can we do this with NAT? Imagine, we had two Internet providers. Two, because we do not want to rely on the network of just one of them in case of a failure of their networks. Every host that needs Internet connectivity needs a unique IP, so we buy one IP for each of them from each provider. When our hosts want to use provider one they use this provider's IP as local IP; when they want to use provider two they use the IP given by this one as local IP. Every host with an IP of both providers can now use either one to send its packets to the same destination.
Now we already see where we are going. The setup described has the potential to solve the problem, we could do load distribution by letting some hosts use provider one and others provider two, and we have a higher availability of the connection to the Internet, since it is more unlikely that both providers have a major breakdown than it is for one of them (how we calculate the probability has been illustrated above). However, as it is easy to imagine we would have a very hard time trying to do load balancing when each host decides on its own where it sends its packets, not to mention how hard it would be to convince a network application to use one or the other local IP. This calls for a central authority to do the decision which host should use which provider, and this authority will, of course, be a special NAT-router.



Using NAT, our local computers need just one IP, since it is no longer up to them to decide which provider (and therefore which IP) to use. If we had a favorite provider, we could use this providers IPs for our hosts, but we can also use internal IPs. Now, when an internal hosts wants to establish a new connection with a destination on the Internet, it just sends its packets to its default router, which is the NAT-router (in the end, there might be other routers involved), and the source IP is the hosts local (internal) IP. The NAT-router, because it knows all connections, decides which provider will route this connection, replaces the source hosts (internal) address with one of the provider chosen and sends it out to this providers router. Since the source address is an address of this providers network, the answers will also come in that way. The host where the packets originated never gets to know which provider had been chosen by the NAT-router, so this process is transparent.

We can use the same algorithms as for virtual servers, so we can do load balancing and we have the high availability feature. The essential difference to the virtual server implementation is that we have to interfere with the routing process. In the above example we actually have two default routes, for example.

3.4 Problems Common to All Techniques

The very special thing with all kinds of NAT is that the five-tuple that uniquely identifies a connection: *protocol*, *source IP and port*, *destination IP and port* that is the same on the source, the destination and on routers in between, is different for all three entities as soon as NAT is active on the router. Special, since we now suddenly have three different such five-tuples where each identifies the same connection on a different section of the route: section one is from the source to the NAT router, section two is from the NAT router to the destination and, at last, section three is inside the NAT router that has to know both the other two sections. We could also say only the NAT router knows what is really going on, which also means the NAT device has to store lots of information about the connection it translates

which 'regular' routers do not need to do.
This is something they have in common with firewalls: because they both do not just relay packets from one side to another but also control the data flows they must know as much about every connection as each network device knows about its own connections, i.e. they must keep state information. It is obvious that this requires a significant overhead compared to simply routing packets.

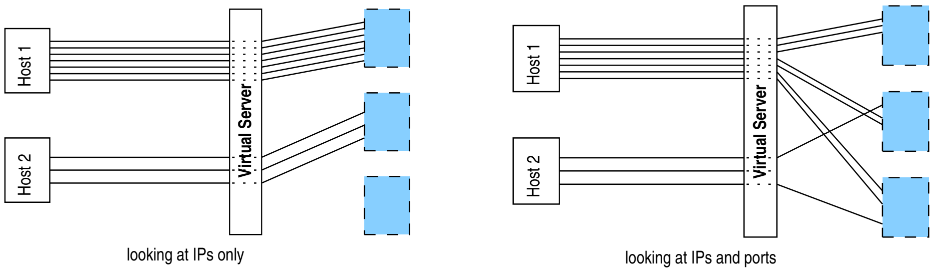
I must not forget to say that if NAT is being used all packets must go through the NAT-router, i.e. there must not be any alternative routes a packet could take, circumventing the address translation. The reason is obvious, but due to their nature as tools for organizing private networks NAT routers are mostly placed on the borderlines of internal (leaf) networks this should be no problem.

3.4.1 Keeping State Information

Except for static NAT we need to store and manage dynamic information about the clients currently using the system as a router and possibly about their connections. In addition, this state information must eventually time-out, so that hosts that have long been switched off or stopped transmitting packets for some other reason are deleted from our list eventually so that the NAT IP assigned to this host can be reused. This timeout is also the reason why looking at TCP-headers is recommended: The timeout can be short for TCP-connections that have just been closed and must be high for TCP connections that are still established, since for example many telnet-sessions may hang around for a very long time without exchanging any packets. In this case, if we have sufficient NAT-IPs left we do not need to let these connections time out, but we might want to assume they are dead when there are lots of new connection requests that need NAT-IPs from the pool.

On the other hand, if we do not keep state information but only look at the IPs that need a NAT-IP assigned, it is much simpler to implement NAT and it will in many cases work as well as the more complicated solution above. Under light load, i.e. when there are always enough unused NAT-IPs left, we will not notice much difference between both variants, except for in the telnet-session (and related programs, e.g. ssh) case. Only when there are not many NAT-IPs left keeping state information is recognizably of advantage, since we are able to exactly identify connections that have just been closed and can reassign their associated NAT-IPs immediately without waiting for a timeout. That keeping track of the state of individual connections adds to overall security if it is used by firewall code is another issue that has nothing to do with NAT.

There is another case where NAT should not just look at the IPs but at the connections individually, that is when we are using virtual servers or virtual network routes for load distribution and there are a significant number of connections coming from only a few IPs, maybe because the IPs belong to big servers that open many connections. In this case when only the IP is examined load distribution will not fully work, because the traffic generated by an individual IP can than not be divided any further. When we look at the (TCP,UDP) ports in addition to the IPs, we can distribute the load more equally by remapping individual connections rather than individual IPs, as the following picture illustrates.



3.4.2 Fragmentation

Closely related with the above problem of keeping state information about TCP and possibly UDP connections is the problem of IP fragments. It is often desirable to base the decision whether a packet should be translated not just on the IP addresses, but also on the TCP/UDP addresses, the ports. This way telnet packets can be treated different compared to http packets. An example for a use of it is to have just one (virtual) IP or DNS name for all services, which will then be remapped to different hosts providing the service. Such services might even be provided a virtual host. A firewall with application level gateways does the same to a certain degree, but it can only do this for services where a gateway exists and mostly these gateways are not transparent. Besides, this is just an example for why it might be of use to look at the ports of packets for NAT.
The problem is that as soon as a packet has been fragmented the NAT router cannot tell the port except of the first fragment containing the TCP header. That is why we must also keep state information about fragments. We must store all data of the first fragment including its TCP or UDP port, of course, in order to be able to know the port the other fragments are going to. Even this measure might not be sufficient since IP does not guarantee that packets arrive with the correct sequence, i.e. the third fragment of a fragmented packet might pass through the NAT router first, before the first fragment still carrying the port information. In this case all we can do is to hold back all non-first fragments of a fragmented packet until the first one arrives, so that we know if we have to translate the packet or not. See also

page 30.

When we have to do that much just to be able to use the TCP or UDP addresses for our to-NAT-or-not decision the idea to use NAT to also rewrite these addresses is almost obvious. This way we have not just IP address translation but also UDP/TCP address translation. It may be less significant but it certainly is a useful extension. An example for its use are virtual servers: Let us assume we want to create a virtual webserver. Let us further assume that the real webserver daemons running on different machines listen on different ports for some reason. When we do not rewrite the destination port in packets to the virtual server (and insert the original port on answer packets) this setup is impossible, then all webservers must listen on the same port where the virtual server provides its web service.

3.4.3 Protocol Specific Issues

Network Address Translation is not always as transparent a process as it should be. Everything would be fine if IP was the only protocol carrying IP address information. There are some protocols that send IPs as part of the data they transmit, and if this is a translated IP that is sent to a receiver behind our NAT-router we are going to have a problem. This means the receiver will have a problem, since he cannot reach the host the IP transmitted has been meant to address. The only way to solve this issue is to look at the data transmitted by certain protocols known to include IP information, which of course means additional overhead and complication.

Some examples for such protocols:

FTP [8]
: The FTP-commands PORT and the response to a PASV both send an IP and a port to the other party. For FTP to work over a translated connection we have to replace the IP in the message. The most complicated part of this is that IP and port are transmitted as their(decimal) ASCII representation, i.e. each single number of the decimal representation of the IP is a byte in the packet. For this reason the IP does not have a fixed length in such FTP-packets. When we now try to replace the IP by another one that has more (less) digits in its decimal representation the packet gets larger (smaller). This in turn makes it necessary to adjust the TCP-sequence numbers accordingly, so that we have to keep some more information about these connections in order to adjust the numbers in each subsequent packet of that connection. This is not just a problem of FTP, but of all protocols where exchanging the IP changes the packets length.

ICMP [?]

Some ICMP messages, depending on the type of the message, include a part of the header of the original packet that caused this ICMP message to be generated, including the entire IP header of that packet. If the packet had been translated, this header will contain the NAT-IP rather than the real local IP of the host that gets this ICMP message. Depending on if and how this extra header information is used this may present a problem or not.

DNS

Obviously, this service is a major problem if nameservice for internal IPs shall be provided outside the NAT-domain. A solution would be to have two DNS-services, one with internal IPs for internal address resolution and an external one with NAT-IPs. Of course, the IPs resolved by the external server should not be part of a pool of addresses for dynamic NAT, as one can easily see. Since NAT-routers are mostly placed on the borderlines of networks separating internal and external DNS data makes sense anyway and is widely used for security reasons. If the far more complicated approach of rewriting all DNS data relayed by the NAT router is used I would suggest an application level gateway rather than an implementation inside NAT, because DNS is well suited for being “gatewayed” and we should only put code into the kernel that is really necessary there.

BOOTP

BOOTP should be no problem most times, because it is very unlikely that this protocol ever has to cross the border of a NATed network.

Routing-Protocols (RIP, EGP,...)

I do not need to explain why routing protocols are a problem. That there are a lot of different routing protocols doesnot make it any easier. The three solution are, again,

- not to use these protocols (static routing only), which may be a good option anyway since mostly there are few connections from our network to the outside, and this is the place where NAT routers make the most sense
- to use an application level gateway
- to rewrite the packets.

4 Virtualizing the Network

When we look back a couple of years at how computers evolved we will find that in the beginning of the development of a new resource it was used and addressed directly. A great example is a computers memory. For quite a long time the processor and each program had direct access to every single RAM cell by using each cell's hardware address directly. A program always knew where in memory its code had been stored and there were no differences between cells storing data and cells storing code. I am talking about computers that can be described as 'PCs', others already used new concepts that we can find in today's devices. Over time, direct access became a disadvantage, systems and application programmers did no longer want to take care of where exactly data was stored and that there were no conflicts with other programs or even within a program. So a new layer hiding physical memory from the software was introduced. The microprocessors now had units that translated virtual addresses used by the software to physical addresses. Advantages were that now several programs could use the same addresses to address their data, and the memory management unit translated them to different physical addresses. Also, memory that did not even exist could be simulated, using for instance the hard disc as a media where portions of RAM could be stored temporarily.
Today we can see many similarities when we look at the development of networks. Until a short time ago most computers were not networked. Existing networks were relatively small and clearly laid out, the administrators almost knew each single node personally. We almost always use a host's IP (or its name, which is mapped 1:1 to one or more IPs) to access certain services or programs on that computer, just as we used to access certain regions in memory by their hardware addresses in order to access a certain piece of code (e.g. calling an subroutine) or data. Today we still use memory addresses for this (for instance pointers in C[++]), but they have a completely different meaning. They are *virtual* addresses rather than physical ones, i.e. we do not know where the data we want to access is physically stored.
In the future a similar process will happen to network addresses: we will no longer address single hosts (cells) directly with their Internet-address (hardware address), but we will use a virtual address (that does not need to be numeric) in order to access a certain service, regardless of what IP (physical address) the host providing the service has. This process has already begun, an example is the Domain Name Service. However, the mapping done by DNS is much less powerful than the one used for virtual memory, so we can assume there is still a lot of potential. In addition, we are still at the beginning of the networking age and we still think in absolute terms when we design new networks. There are concepts, like virtual private networks, virtual LANs or virtual servers, where we can see some of the new

directions. Today nobody tries to find or even use the physical address of a memory cell, where some piece of code or data is stored (except for operating systems designers, but that is what they do, providing us with mechanisms that hide those details), but we still almost always use the physical IPs of network nodes, even DNS is hardly more than a mapping name to IP. We could imagine a mechanism just like virtual memory, where we can (virtually) address code, data or services on a network without knowing or even being able to find out what physical address (IP) is associated with the virtual network address. This, however, is beyond this document, all I wanted was to show where NAT may find or has found its place in the entire scheme. NAT will not be the general solution of course, indeed it has been a hack introduced to circumvent limitations of the current system, which is marked by not providing much virtualization of the network yet. NAT is, on the other hand, a useful tool to achieve virtualization on a local scale, as long as more general solutions have not been developed. It is probable that NAT will even contribute to such a general solution (as has become the virtual memory system for virtualizing memory), either by being part of it or by gaining new insights and experiences that will help to understand the virtualization process better, or both.

NAT is being widely used already, although only for purposes described at the beginning, to help with IPv4 address space limitations and it is very successful at that. This is also another reason why I believe IPv6 will not come that fast, the most pressing problem of the IP-addresses has been partly and temporarily solved and many people can live with it. On the other hand recompiling each application for IPv6 is hardly manageable in large networking environments where the administrators have a lot to do anyway.

5 Example Implementation

5.1 Examining the Premises

I implemented everything inside the Linux kernel because:

- The firewalling code, which is similar in that it works on the same objects, IP packets and their data, has already been integrated into the kernel so I thought it appropriate to do the same with NAT.
- Using a kernel interface to manipulate each single packet in user space takes a lot of time compared to direct kernel level access. The entire packet (not just the header, see Section 3.4.3) needs to be copied twice, we cannot simply pass a pointer.

Some protocol specific things like rewriting DNS data could be handled by user level daemons, using Linux' local redirect function, for example.

In order to make the implementation as flexible as possible and to avoid interfering with the rest of the kernel whereever possible I have not integrated NAT into an existing framework inside the kernel, like the one used for registering new firewall modules. This would have been a cleaner approach, but this part of the (2.0) Linux kernel has some inconsistencies. Masquerading may work well or not, but the way it is implemented in the kernel does not encourage others to do the same with their code. The firewall code contains a somehow generic interface to register new firewall modules, but much is missing. For example, masquerading needs to keep state information about all connections, the firewall should do the same (which it does not) and NAT in general needs state information. So the thought to have this information collected, stored and managed by the same generic routines, made available for still other future kernel enhancements, has come to me. After thinking about it for a while I decided that it would be too great a task to solve in the time available. It would have involved not just designing this generic routines and data structures, but also rewriting large parts of the firewall- and masquerading code. Implementing a better firewall with state information alone is a task currently tackled by a group of people on the Internet, not to mention that changing the now working masquerading code, which is quite complex, is far more than just a month's work. Not to mention the famous saying "Never touch a running system", and that was what I wanted, to have a running (NAT)system, not just framework. This reveals my attitude towards the different software development models: I prefer RAD because I want to have a running system to experiment with rather than playing with concepts that may work or not. That does not mean I started coding immediately. I discussed many aspects of NAT for more than two months, thought it over for another month or two and **then** I wrote the first line of code. There is a point were mere thoughts do not get us much further, where experiment must start. This is the same as in physics — experimentalists against theorists — and as history shows we began becoming as successful as never before in human history when both directions were taken. I thought I had to say this because some people already have asked me why I did not implement it this or that way, to integrate it with firewalling or the like – in general, why did I add a new layer. The simple answer is, simply because I wanted to have a result and did not want to think about anything else than NAT more than necessary. I thought it would be necessary to see if and how NAT works in the first place, to see the implications and the restrictions, before we can go the next step to integrate it with other features. It is just like in physics, like looking for the great theory of everything: for the great picture we have to know the components first.

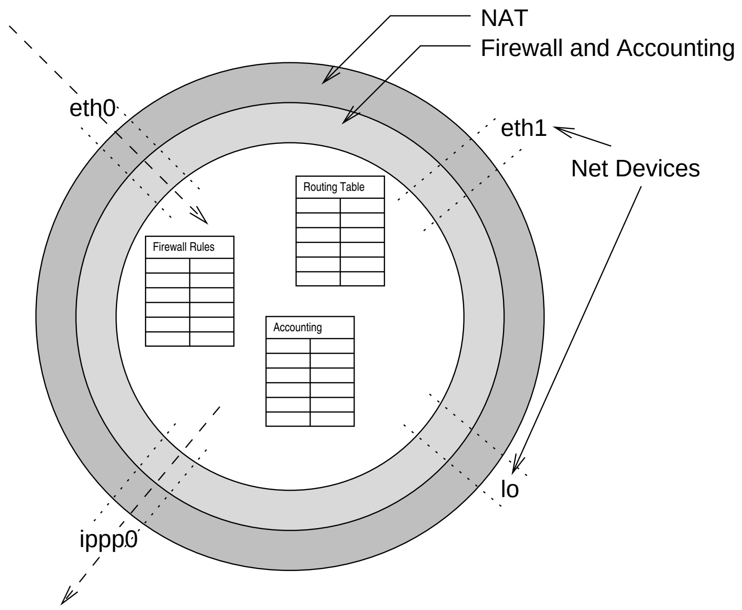
An interesting issue when doing kernel programming is debugging. No code will work exactly as expected, as every programmer knows. Although I had read about a tool which should allow me some basic debugging on kernel code I did not trust it and used break points and the reset button on the computer instead. It worked quite well, especially after getting used to kernel programming. The first time I encountered a NULL-pointer reference I spent almost a week hunting the bug, by the end of the project finding even more subtle such NULL-references did not take me longer than ten minutes. This is interesting, because I did not really know more about the kernel and about C than when I started, at least nothing that could be useful for finding this kind of bug, so it was my feeling that had developed rather than my knowledge that guided me. This side note shall just show what a powerful ally the subconscious mind is, which stores and processes lots of unbelievably fuzzy data, something the artificial intelligence people have been working on for years.

I started using the new 2.1.X-kernel but soon felt it would be better to use a stable kernel. I could never be sure if a kernel panic was caused by my code or by experimental code contributed by others who worked on different parts of the system. Going back to 2.0 I realized that 2.1 kernel code was significantly cleaner in some places, but despite that I chose stability of the development system over new features and cleaner implementation. In 2.1 even when the kernel worked well I could be sure that there was always some tool that did not work very well with this new kernel.

5.2 The Core NAT Implementation

As discussed above NAT should be a layer encapsulating the kernel. Therefore all other network kernel functions, and thereby all user level programs, would not be able to see the real (IP-) world but only the addresses made visible by NAT. Whether these are real IPs or translated IPs does not matter, they all came from the NAT layer. This makes it possible to have the kernel (and all programs) live in a virtual address space that does not exist in the real world. Only NAT will know the reality. An example of a setup that can be supported by this implementation that a "regular" NAT that does translation inside the kernel cannot do is when you have networks that use the same IP address space and want to communicate with one another. I do not know any such setup and never heard of one, but it can be imagined there are two independently administrated networks that once have been built using the same RFC 1918 address space for both. We can connect them to a different interface of the NAT-router, set up rules to translate the IPs depending on which interface the packets arrived on, and the two networks can exchange IP packets.

To achieve independence and flexibility the changes made to the kernel itself are minimal. Most of them do some initialization on kernel startup or are hooks where the real NAT module can register itself with the kernel. The only real NAT code in the kernel are calls to a function in the module that examines each IP packet. These calls take place right when an IP packet comes in and right before an IP packet is going to be transmitted, and are done only if the NAT module has been inserted into the kernel, i.e. without the module the system will behave just as if it had been compiled without NAT support.



NAT-Rules can be bound to the inbound or the outbound direction. If that would not be possible NAT could not be laid around the entire kernel and we could only translate incoming or outgoing packets, loosing flexibility.

I reused large portions of the packet matching code of Linux' firewalling. Here the above discussion about why it has been implemented the way it is and not integrated with other parts of the kernel could be continued, because one might rightly argue that packet matching is the same for firewalling and for NAT. Again, to cite myself, I wanted to implement and test NAT in the time I had for the project, and finding a good and general solution for integrating parts of the kernel was too big a task, especially since

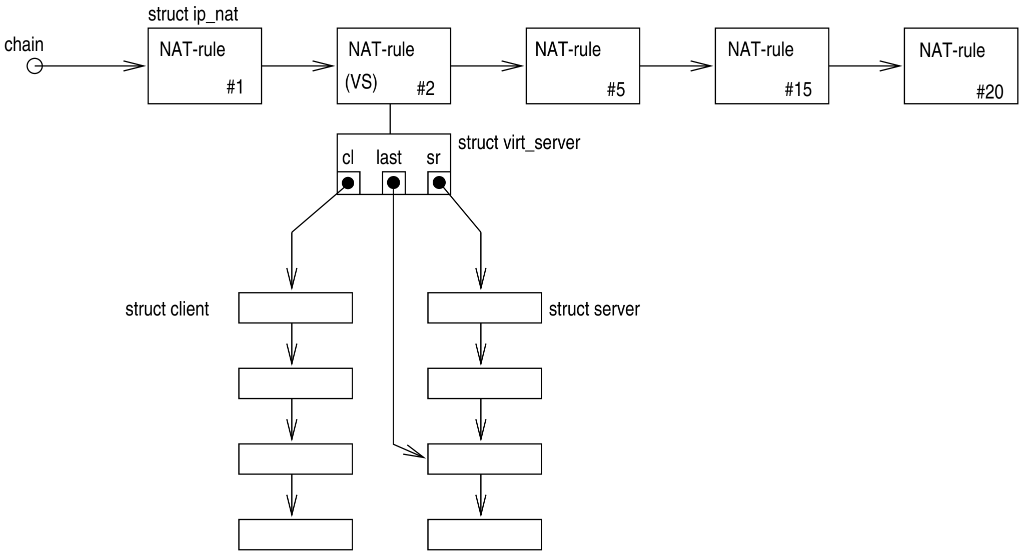
- the packet matching code is not identical, there are many special cases that need to be considered for NAT
- implementing NAT as a further firewall in the firewall chain would have made NAT less flexible. Even if it could still fulfill all real-world tasks that way that would not have been a good way for my experiments.

Above I mentioned we need to keep information about fragments if we want to do port dependent NAT. Linux has the ability to defragment all packets it routes. This is even more than we need since it's only necessary when we need the port information for NAT. This way we do not need to keep fragment information. However, it does not completely work: since NAT has been laid around the kernel encircling everything it also encircles the defragmenting code, i.e. NAT for incoming packets is called before defragmentation can be done. That is why port dependent translation can only be done reliably for outgoing packets. Changing the defragmenting code so that it gets incoming packets before NAT does should be easy, but I did not bother to do it.

The data structure used to store the various address translation rules is a list. Each rule specifies some criteria a packet has to match in order to be translated using the rule's NAT-IPs. These criteria are source and destination IP and mask, source and destination port, protocol (UDP,ICMP,TCP) and the interface where the packet has arrived on or is going to be sent through. It is also checked if the packet matches the reverse of the current rule in order to enable bidirectional rules. If any of the data used for packet matching is left empty every packet matches this criteria. Because order is important there was no choice but to use a linear list. There are also skip-rules which, when matched by the packet's IP header, cause the NAT function to skip this packet. Each rule has a unique number so that new rules can be inserted anywhere into the chain. This is another example where NAT is more flexible than the current firewalling code that could have been used, and in order to still have this flexibility when integrating NAT into existing code I would have had to rewrite the other code as well, which would have been too much work.
Since I wanted the implementation to be as flexible as possible I had to find

a way to allow storing information needed for such different kinds of NAT as static NAT, dynamic NAT and virtual servers all in the same structure, i.e. the NAT rules should look equal whatever kind of NAT they represented. Therefore the the structure that stores exactly one rule contains all the information needed for packet matching, some additional information like flags, packet and byte counters and the rule's identification number. The various information that is different in respect to data types and amount of data for each kind of NAT gets stored in dynamically allocated memory and the rule only has a pointer to the start of the area where this information lives.

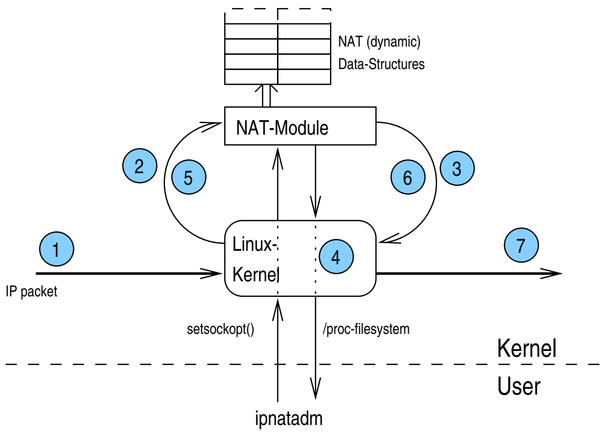
The following example shows some static NAT rules where all the necessary information is included in the rules and a virtual server rule (the second rule). The virtual server rule needs additional dynamic data, which are a list of all the IPs where all requests to the virtual server should be redirected to and a (more dynamic) list of clients and to what server they have been connected to. The list of clients may become quite large so an appropriate data structure is necessary to ensure a minimum overhead for searching this list. A hash or a (balanced) binary tree would work. I have used a linear list, in contradiction to my own proposal. The reason was because this implementation served for experiments and I thought it easier to track a linear list in case something went wrong and I had to debug the code. It can easily replaced by a more sophisticated data structure allowing much faster search algorithms.



Another puzzle to solve was how to get the NAT rules specified by the user into the NAT module, which is part of the kernel. Linux has several interfaces for user- to kernel-communication. One that is special to networking is the call to the function *setsockopt()*. The Linux firewalling administration tool *ipfwadm* written by Jos Vos (jos@xos.nl) uses this interface for sending data

to the firewall code. Since I have used *ipfwadm* as a basis for my NAT administration tool *ipnatadm*, I have reused the idea and modified it only slightly in order to allow for changes in the structure and size of the data exchanged without having to recompile the kernel each time, so that all checks for validity of the data are done by the module and the kernel just passes all data received via *setsockopt()* without looking at it.
The function *setsockopt()* provides one-way communication only. We want to get some data back, however, since we are a curious species who always want to know what is going on behind the scenes. The Linux kernel implements a great feature for this, the proc-filesystem. This is not really a filesystem although it looks just like that to the user. You will notice that all files under /proc have zero size, but if you try, for example, a *cat /proc/cpuinfo* you will get some output. What happens is that when you access a file of the /proc-hierarchy inside the kernel a function is called — which is a different one for each file under /proc — which produces some output that is given to the user space program as the contents of the file. It is also possible to write to some of these files, thereby sending data to a kernel level function, but I have not used this feature but chose *setsockopt()* instead for the only reason that the example program I used did so and I did not want to spend time writing completely different code. Currently there are two files starting with ip_nat_* in /proc/net/, showing information for core NAT and for virtual servers, when the module has been inserted into the running kernel. They list the contents of the dynamic data structures for the NAT service they represent, such as the NAT rules or what real servers belong to a virtual server rule.

Below is a graph that shows how the module interacts with the kernel and the user:



When an IP packet arrives (1) the kernel calls the NAT module (2) giving it a pointer to where the packet has been stored in kernel memory. The

NAT module examines the packet and does address translation if it matches any NAT rule. It then returns (3) the packet to the kernel which in turn continues as usual (4), doing routing or delivering it locally to a process. The same happens to all outgoing packets (5/6), which are packets we route and locally generated packets, just before they are transmitted and just before any ARP is done. The packet is then given back to the kernel for further processing, which means the device driver sends it out on the wire (7).
The user can influence the process by using *ipnatadm* to send instructions and data to the module via *setsockopt()*-calls, such as new NAT rules or instructions for deleting a rule or the like. They can view the contents of the dynamic data structures where the module stores the NAT rules and dynamic information collected while running directly by viewing the contents of the NAT-files in the /proc-filesystem or by using *ipnatadm*, which also uses these files but rewrites the lines into a more human readable format.

5.3 Static Address Translation

The standard translation function used by all other NAT functions (dynamic, virtual server,...) does static translation. It gets a pointer to the buffer holding the IP packet and the new source and destination addresses that shall be inserted, including a network mask. This mask is 255.255.255.255 when the function is called by the dynamic NAT functions, since only with static NAT can entire networks be translated using the same parameters for this function. All others have no 1:1 mapping and have to keep track of the real IP to NAT-IP mapping.
Included is the ability to rewrite source and destination UDP and TCP ports, which enhances this NAT implementation further. However, this function must be used with care. Since we do not keep state information about every connection, we cannot determine the correct port for an answer packet which has a replaced port. If we kept state information we would simply look up the connection the packet belongs to and would then know the correct ports. For this reason no bidirectional rules can be used for port rewriting. We always need two rules, one for the inbound and one for the outbound direction, each containing exactly one port the packet has to match in order to be translated. If the port specified is a source or a destination port depends on what port we want to rewrite. Most of the time this will be a destination port, I guess.
The port issue shows how important keeping state information is for NAT to really be flexible.

5.4 Dynamic Address Translation

Dynamic address translation has not been implemented. The reasons are

- that I think masquerading, which already works well in Linux, is a better choice for most purposes,
- and that the other, non-traditional uses for NAT like virtual servers and virtual routes are far more interesting for a sample implementation of NAT.

Despite that I have integrated hooks into the code so that dynamic NAT can easily be added. As for all non-static NAT variants, we have to keep dynamic information about what real IP has been mapped to what NAT-IP. The implementation gets not much harder when exceptions shall be allowed, where certain real IPs shall always be remapped to the same NAT-IP, so that incoming connections to these IPs are possible. All we have to do is to create an entry in the table where we store the current mappings that has no timeout, that means is valid forever. The other dynamic mappings eventually time out and get deleted from the table, so that they can be reused for another real IP.

5.5 Virtual Servers

Static NAT does not need to keep any dynamic data about current IP mappings, but for the virtual server function this is necessary. The implication is the standard NAT structure is not enough so that it must be enhanced in order to be able to store all the dynamic information and the data about real servers that answer packets for this virtual server. A virtual server is represented by exactly one NAT rule in the chain of rules, but since it is a dynamic rule (using dynamic data) the pointer reserved for such rules points to a structure that holds virtual server specific data. Also, the fields containing NAT-IPs and NAT-ports are meaningless for all dynamic rules, since the information which IP will be used for the translation is not static but needs to be gained from the dynamic data gathered so far using some algorithm. A virtual server is one virtual IP, so we store this IP in the field where we try to match the destination IP of incoming packets. In the virtual server case this will always be a full IP and not a network, but of course it would work just the same (not exactly, though, because in answer packets back to the client we need to substitute the source IP: the virtual servers IP for the real servers IP). See the figure on page 31 for how a chain of NAT rules containing a virtual server rule looks like.

I do not store complete connection state information, but only the IPs of clients using the virtual server. I have already covered this topic in section 3.4.1 above.

5.6 Virtual Routes

I have not implemented this function. Virtual routes are almost the same as virtual servers, the functions and the implementations are similar. Similar does not mean it is the same, where I have to change the destination address for the virtual server function I have to change the route and the source address here, but technologically it is the same, so I did not expect any surprises and left this out. Studying virtual servers revealed enough information for being prepared for an eventual implementation of virtual routes. Furthermore, as I already mentioned, I tried to avoid changes to Linux kernel code whenever possible. Especially the routing code has been radically redesigned in the 2.1.X-kernel series, so I thought it not to be of much use if I changed 2.0-kernel routing code.

6 Using NAT

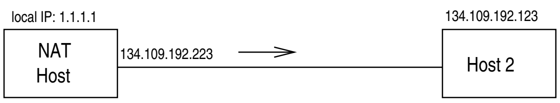
6.1 Static Address Translation

I have already lined out possible fields of application for static NAT, so I just want to give some examples of how to use my implementation for doing static NAT. Different from Linux firewall rules that many people know by now I have introduced a unique id (an integer) for each rule, so that deleting a certain rule and inserting rules at an arbitrary position in the chain is easy. If the id specified for a new rule that shall be inserted has already been taken by another rule this other rule's id (and that of following rules, if necessary) is incremented by one so that the administrator does not need to leave space in advance, as it was the case with line numbers in BASIC-programs, where it was custom to increment the line number by ten in case lines had to be inserted in between.

6.1.1 Changing localhosts IP

Here I only want to show how my implementation translates not just forwarded packets, so that packets destined for or originating from localhost will be treated equally. This is a result of the design of this implementation that makes NAT an additional layer around the kernel's network functions, see the figure on page 29.

We have two hosts, one of them is a Linux PC using the NAT module. Its local IP that is used to configure the network interface is 1.1.1.1, but on the network we want to appear as134.109.192.223 to the other host (IP 134.109.192.123).



Assuming the network (including routes!) has been configured already on both hosts I only mention the additional steps necessary to translate the local 1.1.1.1 address:

- Tell the NAT module to translate the IP 1.1.1.1 to 134.109.192.223 in outgoing packets and to do the reverse for incoming packets:
  Using one bidirectional rule:
  `ipnatadm -O -i -b -S 1.1.1.1/32 -M 134.109.192.223/32`
  Or using two rules (equivalent to the above rule):
  `ipnatadm -O -i -S 1.1.1.1/32 -M 134.109.192.223/32`
  `ipnatadm -I -i -D 134.109.192.223/32 -N 1.1.1.1`

The rules can be read like this:

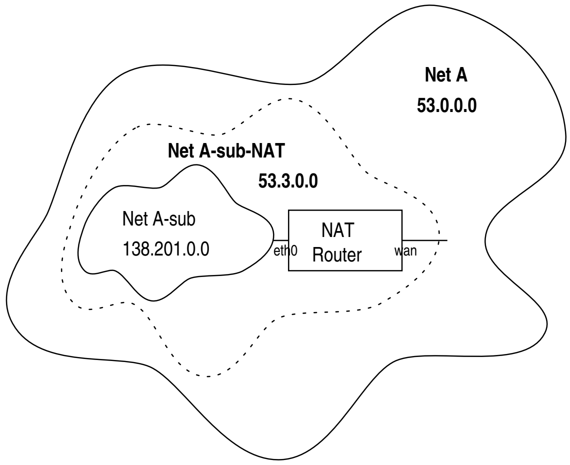
- insert (*-i*) a bidirectional rule (*-b*) into the chain of rules:
  If the packet will be sent (*-O*) through any interface (no *-W*), carrying any protocol (TCP,UDP,...) (no *-P*), the source IP (*-S*) is 1.1.1.1 and the port does not matter, replace the source IP (*-M*) by 134.109.192.223.
  Since it is a bidirectional rule it will also match incoming packets (opposite of *-O*), if the destination IP is 134.109.192.223. For matching packets this destination IP will be replaced by 1.1.1.1. The NAT module knows it is the opposite direction because the NAT-function was called from the IP packet receiving kernel function and the rule has been bound to the out-direction.
- The two alternative rules do exactly the same, but here we don't rely on the mechanism for bidirectional rules but do the translation manually.

Now host 2 can communicate with the NAT host using the IP 134.109.192.223, using 1.1.1.1 won't work even if a route for this address is inserted into host 2's routing table. Note that the implementation does not translate IPs inside the packets, so for example non-passive FTP from the NAT host to host 2 cannot work (wrong PORT command, it still contains IP 1.1.1.1 but the packet comes from host 134.109.192.223 from host 2's point of view).

6.1.2 Translating a Network

This is *the* classic case of NAT usage. In our example we have a big world-wide corporate network using class A addresses 53.0.0.0. There is a small network 138.201.0.0 that belongs to a department of the same company using the 53.0.0.0 network, but since they thought they would never need any connectivity with others or that there would be other solutions (like IPv6) before they needed it the administrators of that department did not want to go through the bureaucracy of the company to get a class B subnet out of the class A 53.0.0.0 addresses assigned, instead they choose some arbitrary addresses (not even knowing about RFC 1918 and private IPs). As we now know (it is always easy to say "I always knew it!") the Internet has grown so fast and with it the Intranets that it was just a question of (a short) time until the department's administrators saw their mistake. However, even if it is easy to get some 53.0.0.0-addresses from the company the department just cannot change their addresses now, since they have (production-)connections to many customers that rely on the connectivity 24 hours a day, 7 days a week and are of course not willing to accept any change saying 'it's not our business', and they are right to say so.

The solution, or let us say 'a solution' for there are many (as always!), is get the 53.0.0.0-addresses anyway and use static NAT on the physical network connection to the corporate Intranet.



There are various ways to use the NAT-module in this case. If we choose to bind the translation on the internal interface to our 138.201-network we need to have a route to this interface for network 53.3.0.0, the one we have obtained from our company, but if we translate on the interface to the company 53.0.0.0-network we need to have a route to 138.201.0.0. We can also do separate translations for packets from our net to the 53.-network and for packets coming in from the 53-network, because since NAT is a layer around the kernel there are always two points in the NAT layer a packet has to pass. It does not hurt to do this, but it is not nice, I would prefer to bind the entire translation for incoming and outgoing packets to one point.

The bidirectional NAT-rule we need to do all translations on the interface 'wan' is:

```
ipnatadm -O -i -b -W wan -S 138.201.0.0/16 -M 53.3.0.0/16
```

or, also possible and equivalent,

```
ipnatadm -I -i -b -W wan -D 53.3.0.0/16 -N 138.201.0.0/16
```

Again, we could also use non-bidirectional rules where we have to take care for packets in the opposite direction by specifying another rule accordingly. When we omit *-b* in the above two rules we have the pair of rules needed when no bidirectional rules would be possible with the implementation. As we can see *-b* simplifies writing NAT rules a lot.

6.1.3 Translating Ports

Support for port translation is very basic because here we really needed to keep some state information. The problem is to insert the original port into packets that are answers for packets we translated. Unless we keep that information if we do the forward translation we are unable to do the translation for the return packets, since we have absolutely no way to determine the port the client may have used. This is why bidirectional rules are completely impossible to use with this implementation, and doing the backward translation 'manually' by specifying an extra rule for it is not generic. Of course, I can specify a rule like

```
ipnatadm -O -W eth1 -i -D webserver/32 80 \
-N temp-replacement/32 8888
```

This will work, since we know exactly the IP we have to insert in return packets: it is port 80. So the rule for the return packets will be

```
ipnatadm -I -W eth1 -i -S temp-replacement/32 8888 \
-M webserver/32 80
```

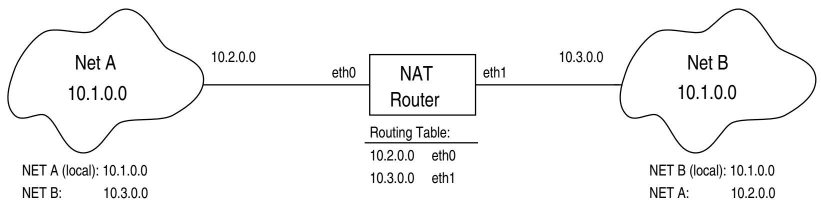
This will take care that the clients connecting to the webserver see the expected source address and port in the packets they get back, which must be from the IP and port they sent their packets to. In this example we have also done IP address translation, not just port translation. Port translation alone makes less sense than IP address translation, but it may still sometimes be useful.

6.1.4 Two Networks using the same Address space

Simple Case

Why I call this the *simple case* you will understand after you have looked the the *complicated case* below. Here is the situation: there are two networks that have existed independently of one another for a long time, and nobody had ever dreamed that one day there could be a need to connect these networks together. Incidentally the administrators of each network have chosen the same addresses out of the private address pool as described in RFC 1918, so both networks now use 10.1.0.0 addresses. For some unknown reason that really does not matter for this example both networks now need direct IP connectivity. The obvious solution is that one of the networks must change its addresses, or if we do not want the other administrator to have an advantage both have to change their addresses. However, in real (corporate) environments changing the addresses of an entire network often is painful process that needs lots of work and lots of talking and conferences, especially when more than one administration domain is involved. Sometimes parts of the hosts even cannot change their addresses at all because the customers using them don't want it (see the example above). So many administrators would welcome a fast and working solution like NAT. Of course, there are others who will say NAT is a dirty solution, but everybody has free choice and I am only writing this for those who consider using NAT. In the end, if it works it works, dirty or not, and often such quick solutions have lasted for years. One could now argue that this was bad because it prevented better solutions to be developed or used, but there are always examples for either side in this discussion and I really don't want to start one here. That is why I just assume the administrators of our two networks have had a meeting and have decided to use NAT after considering some alternatives, and since they know their situation best I will not try to convince them of another way, I simply tell them how to do what they want with my implementation (assuming everything would work and it would be out of the ALPHA and even the BETA-state by than).

We have the two networks, both using now and for the foreseeable future 10.1.0.0 addresses for the internal hosts. Network B will be addressed from network A using 10.3.0.0 addresses, and network A will be addressed from network B using 10.2.0.0 addresses.



There are other combinations of rules possible than the following ones, but I give the bidirectional rules needed to convert Net B addresses on the interface eth1 of the NAT router, and to convert Net A addresses on interface eth0, that means both incoming and outgoing packets for a network are translated on the same interface. Packets coming from the network to the router destined for the other network are translated when they come in, and return packets from the other network are translated just before they are sent out on the same interface.

These are the rules, one for either network:

```
ipnatadm -I -b -W eth0 -S 10.1.0.0/16 -M 10.2.0.0/16
ipnatadm -I -b -W eth1 -S 10.1.0.0/16 -M 10.3.0.0/16
```

Now, when a host in Net A contacts a host in Net B, its IP (the source IP of the packet) is converted to a 10.2.0.0-address, so that it appears to have come from that network for the host in Net B. Net B sends its response to this 10.2.0.0-address, which will be routed to interface eth0 on the

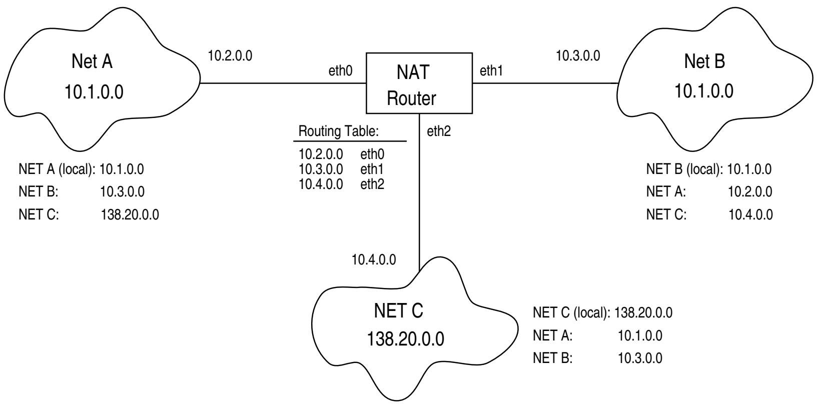
NAT router (see the routing table in the figure). Just before this answer packet is sent out to the host on Net A its destination address is changed to the 10.1.0.0-address of the host in Net A, so that this host recognizes the packet as an answer for the packet it had sent earlier. The router's kernel only sees the 10.2.0.0- and the 10.3.0.0-addresses but never the local 10.1.0.0-addresses, since the NAT-layer hides them from the kernel's network functions. The router's routing therefore works on virtual IPs that belong to no real host on any of the two networks the router serves. Here is the point to mention a bug in the implementation. It is not really a bug but missing corporation between the module and the kernel. What happens is that the NAT router will issue ARP requests for those virtual addressed. Everything works fine, though, it is just that there are some senseless ARP packets. I have not investigated the problem further after it became clear that first of all everything works and second, non-trivial changes to the kernel would be necessary. The latter conflicts with my intention to not interfere with the other kernel code unless it was absolutely essential for the module to work at all. This is even more important when we consider that large parts of the networking code have been rewritten in the 2.1 kernel series and I don't know how this new kernel version acts in this ARP case. It is not simply finding the function call causing ARP resolution before a packet is sent and placing my NAT function right before it, thereby preventing virtual IPs from being used in such requests, because this I have done already, NAT would otherwise not work at all in this virtual IP case. It seems to be connected to Linux' routing code that creates an entry in an rt(route)-cache, which also contains fields for address resolution information. Especially the routing code has been radically redesigned in the 2.1-series, so I don't see much sense in messing around in the 2.0 kernel code in this case.

Complicated Case

This is a variation of the above example. A similar case was introduced and long discussed when the group of people who started this whole NAT implementation thing discussed the aspects of NAT and what features we would like to have and so on. Unfortunately I was the only one left to do the implementation after months of heated Email discussions, although I really don't want to blame anyone for this; for me it serves as a (University) project I have to do anyway while the others already used lots of their spare time during the discussions.

The situation is basically the same as above, we only add something: There is a third network, using 138.20.0.0 addresses in this example, let's call it Net C. The tricky thing is Net A and Net C have already been connected using the router that now is our NAT-router and all the people in both networks have become used to the other network's IPs so that they are hardwired not just in the brains of the people but also in lots of code, e.g. in firewall rules in subnetwork-firewalls (where using DNS is a bad idea since DNS can be spoofed), or in /etc/hosts files. To summarize, whatever the reasons may be, Net C wants to continue talking to Net A using the 10.1.0.0 addresses and Net A people want to say 138.20.x.y to Net C-hosts.

This does not sound that complicated and it indeed is not, but I want to use it to show two different ways to solve the problem, one is using the packet matching code and the other one is to use a completely virtual address space. We have already had a completely virtual address space in the example above, but in this example if Net C connects to Net A using Net A's real IPs this is simple routing, I will make it more complicated for the example's sake. Note that it will be unnecessarily complicated for the real world, but this is an example used for demonstration.



Again, when we look at the routing table in the figure above there is not a single real world IP in it. This time the situation is different from the first example, though, since Net A needs to use the real IPs of Net C while in the above (simple) example all cross-network communication was done using virtual IPs.

At first the rules to create the virtual address space for the NAT router:

```
ipnatadm -I -b -W eth0 -S 10.1.0.0/16 -M 10.2.0.0/16
ipnatadm -I -b -W eth1 -S 10.1.0.0/16 -M 10.3.0.0/16
ipnatadm -I -b -W eth2 -S 138.201.0.0/16 -M 10.4.0.0/16
```

And now the rules needed for the 'specials' in this setup:

Net A wants to address Net C using 138.20.0.0, so convert this destination address to Net C's virtual addresses for routing:

```
ipnatadm -I -b -W eth0 -D 138.20.0.0/16 -N 10.4.0.0/16
```

Net C wants to see Net A as 10.1.0.0:

```
ipnatadm -I -b -W eth2 -D 10.1.0.0/16 -N 10.2.0.0/16
```

Now

- Net A connects to Net C using 138.20.0.0, to Net B using 10.3.0.0
- Net B connects to Net A using 10.2.0.0, to Net C using 10.4.0.0
- Net C connects to Net A using 10.1.0.0, to Net B using 10.3.0.0

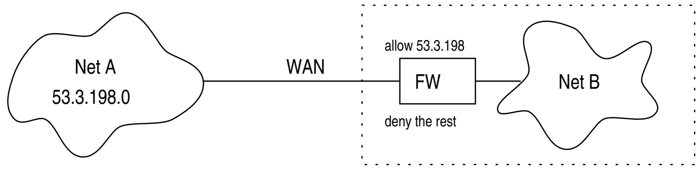
The number of rules doubles when we don't use bidirectional rules, which would make it even clearer what translations happen when and where, but would be more difficult to read. As long as the code used for bidirectional rules works as expected we can just rely on it to make our life and specifying NAT rules easier, because one thing is clear for me after writing both lots of firewall- and NAT-rules: debugging less than ten NAT-rules is more difficult than debugging 100 firewall rules. *tcpdump* was the tool I used most often.

6.2 Dynamic Address Translation

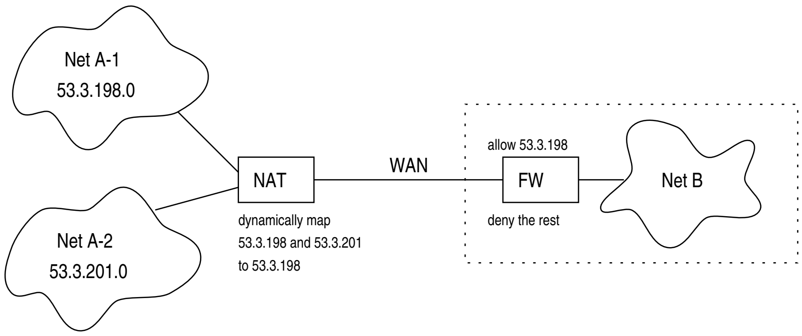
My example implementation does not include dynamic NAT. There are stubs for it, though, so including it is easy. Originally I wanted to implement static and dynamic NAT, but when I found virtual servers I changed my plans and abandoned dynamic NAT in favor for virtual servers. The Linux kernel already contains the special kind of dynamic NAT Linux'ers call masquerading, anyway.

RFC 1631, which describes dynamic NAT in detail, also tells us about possible uses. Another example for a possible use besides the ones described in the RFC is the following example. I did not completely invent the setup, I had the idea because in the company where I worked when I wrote NAT I encountered a similar situation, so nobody can say it is completely artificial and just a product of my imagination and there will never be such a situation.

Imagine the following setup: There are two departments, each one with their own private network (with some connections to the outside). For some reason they work together on a project and therefore connect their networks. However, department B is concerned about security and purchases a firewall, so that department A's access to the network B can be controlled. The procedure department A has to follow in order to get department B's firewall administrator to change or add rules is relatively complicated and slow, one reason being that nobody at department B has much experience with firewalls.



Now, after some time has passed and everything has worked well (more or less) department A decides it needs to hire more employees and therefore to increase its network. Since the class C network (network A-1) they have used so far does not contain many more free IPs a new class C network (network A-2) is used. The employees in that new network also want to access department B's servers in network B, but the firewall only allows network A-1 through. In addition, department B's firewall administrator is on vacation and the others don't dare touch the firewall. Luckily department A employs a bright administrator who knows NAT. He installs a NAT-router and establishes a dynamic NAT rule on it, mapping both network A-1 and A-2 dynamically to network A-1 addresses, thereby cheating the firewall.

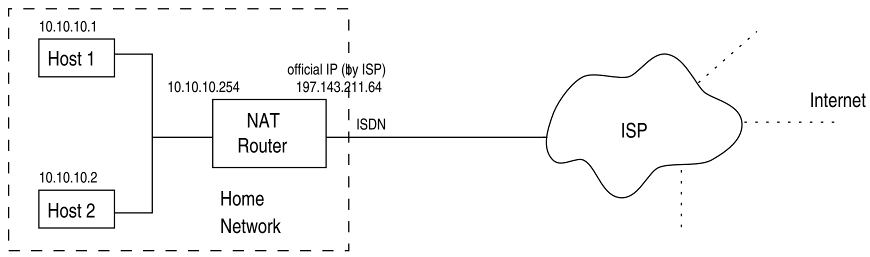


This setup is indeed a bit unusual, but it also is a real live example. Maybe it is unlikely someone else will be in the same situation and it is also possible to find other solutions, but especially the latter that is not a good argument because it is *always* applicable. The purpose of this example is just that, to give an example, not to tell anyone what to do.

Another not so obvious example would be a redirector. Dynamic NAT could for instance be used to redirect all packets for any IP/port 'something fixed' to a single IP. Another way to achieve this with Linux is using the local redirect feature and have a user space program do this. The advantage here is that this redirector program also gets to know the original destination, which is essential for using this feature to redirect all port 80 connections to a local web cache, because the web cache must be told what IP to connect to.

Masquerading

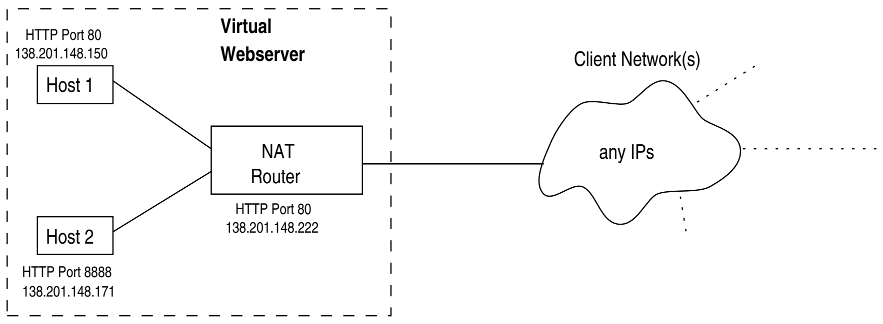
To complete this dynamic NAT section one example for masquerading as it is widely used these days. We have a small office- or home-network consisting of two hosts and a Linux-server. The Linux server possibly provides print- and fileservices, and it also serves as a NAT-router to the Internet for the other hosts. All internal IPs are translated using the official IP given by the ISP, this IP is the Linux router's IP on the network interface to the ISP.



Linux masquerading is extremely popular and many application specific modules have been written, that (among other things) take care of translating IPs transmitted in the data part of IP packets. Unfortunately, I don't know of any serious efforts to combine the various NAT-parts that have been developed for Linux, like masquerading or some very basic NAT in the routing code of 2.1-kernels, and, not to forget, my implementation.

6.3 Virtual Hosts

As an example for a virtual server I have chosen a virtual webserver, due to its popularity and the simplicity of the protocol (compared with FTP, for instance).



The following rule, which gets the unique rule id (-Y) 10, inserts a virtual server rule. The virtual server IP is 138.201.148.222, the algorithm used to select a real server is byte-counter, i.e. the server that has delivered less bytes than the other(s) is used:

```
ipnatadm -I -i -Y 10 -D 138.201.148.222/32 80 -X byte
```

Let us now add (*-a*) the real servers to the virtual server rule identified by the unique id (*-Y*) ten in the chain of NAT rules:

```
ipnatadm -I -a -Y 10 -D 138.201.148.171/32 8888
ipnatadm -I -a -Y 10 -D 138.201.148.150/32 -w 2
```

The first server's http-daemon listens on port 8888 for some reason we don't care about now. The second server is much bigger than the first server, so we assign it a weight of 2, so that it gets used twice as much as server one.

If one of the servers fails it can easily be removed from the virtual server rule, so that the other server(s) continue alone:

```
ipnatadm -d -Y 10 -D 138.201.148.171/32 8888
```

This command deletes the first server from the rule so that the virtual server now consists of just one real server. Here we get to another missing feature, which can easily be added but illustrates the long way we still have to go for a reliable implementation: When the server one has been repaired we want to take it back online, so we issue the command to append (*-a*) its IP to the virtual server rule. What happens now is that the byte-counter for this server gets initialized with 0, and the algorithm used to determine the real server to use for new connections will use only the new IP for a while since the other server(s), which has(have) been serving alone for some time have a much higher byte-count. The same happens when a virtual server is online continuously for a very long time, when the byte counter produces an overflow, but this is less likely with an 'unsigned long' counter.

6.4 Virtual Routes

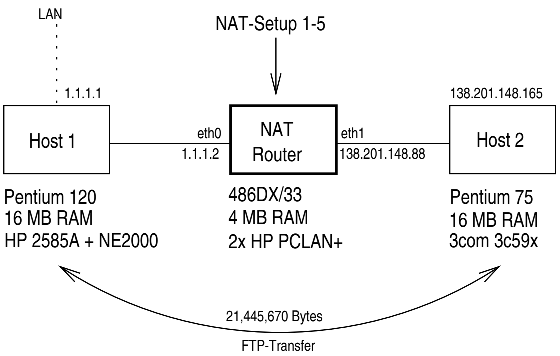
I have not implemented this feature although for me it is the most interesting one together with virtual servers, mostly because the idea occurred to me relatively late, when I was already working on virtual servers. I also do not know of any other implementation of it.

6.5 Performance

For testing the performance I used the following setup:

- three Linux-PCs (Host 1 and 2 + the NAT-router)
- Host 1 <-> NAT-router: 10Base2 (Host 1 has an additional LAN-connection)

- Host 2 <-> NAT-router: 10BaseT (Crossover)
- transfer (FTP) a ca. 20 MB file from Host 1 to Host 2 through the NAT-router and vice versa
- measure the time using the command '*time*'



The procedure was not meant to produce highly accurate numbers, it should just give an impression of the delays we can expect when using NAT. The conditions in networks are generally so complex I did not see much sense in trying to be more accurate. The numbers obtained are good enough for getting a feeling for what it means to have a NAT router, and that is all they are for.

I measured five different setups:

1. The NAT module has not been inserted into the kernel.
2. The NAT module has been inserted into the kernel, but there are no NAT rules (the chain is empty).
3. There is exactly one (bidirectional) rule in the chain:
   ```
   ipnatadm -I -i -b -S 1.1.1.1/32 -M 1.1.1.3/32
   ```

   The result is the NAT router and Host 2 see Host 1's IP as 1.1.1.3, and not 1.1.1.1.
4. There are four NAT rules in the chain:
   ```
   ipnatadm -I -i -W eth0 -S 1.1.1.1/32 -M 55.55.55.55/32
   ipnatadm -O -i -W eth0 -D 55.55.55.55/32 -N 1.1.1.1/32

   ipnatadm -I -i -W eth1 -D 1.1.1.1/32 -N 55.55.55.55/32
   ipnatadm -O -i -W eth1 -S 55.55.55.55/32 -M 1.1.1.1/32
   route add -net 55.55.55.0 netmask 255.255.255.0 eth0
   ```

The result of the first two rules is essentially the same as above, the difference is that instead of making one of the two rules bidirectional we specify everything 'manually', also, we convert Host 1's IP to 55.55.55.55 and not to 1.1.1.3, but that does not make any difference. The other two rules reverse the first two rules, so that Host 2 still sees Host 1 as having the address 1.1.1.1. This setup shall test the worst case, where a packet has to be translated twice, when the NAT router receives it and just before it is going to be sent out again.
Note that we need to establish a route for network 55.55.55.0, since Host 1's IP will be one of that network when the routing decision has to be made.

5. The same as above, but there are 46 (garbage) rules right before the four 'real' rules, so that the algorithm has to scan through all the 46 rules first (which do not match any packet), making this a 50-rules test.
   Unlike firewall rules, in reality there will almost never be that many NAT rules, because mostly one (static or dynamic) rule per network is sufficient.
   ```
   ipnatadm -I -i -b -S 123.123.123.0/24 -N 42.42.42.0/24
   .
   . (this one [garbage]rule repeated 46 times)
   .
   ipnatadm -I -i -W eth0 -S 1.1.1.1/32 -M 55.55.55.55/32
   ipnatadm -O -i -W eth0 -D 55.55.55.55/32 -N 1.1.1.1/32

   ipnatadm -I -i -W eth1 -D 1.1.1.1/32 -N 55.55.55.55/32
   ipnatadm -O -i -W eth1 -S 55.55.55.55/32 -M 1.1.1.1/32
   route add -net 55.55.55.0 netmask 255.255.255.0 eth0
   ```

The following tables show the numbers I obtained. There are two rows for each setup. The upper one shows the numbers when I transferred the 20 MB file from Host 2 to Host 1, the lower row is for the transfer from Host 1 to Host 2. I always completed one direction before I started taking numbers for the other one. Swap space was unused on both hosts, there where only the basic processes running.

Table 1: raw numbers from '*time*'-command

#	t1	t2	t3	t4	t5	t6	average
	seconds						
1	43.43	43.27	39.05	36.94	37.88	37.90	39.75
	40.39	41.31	39.66	37.29	34.95	34.80	38.07
2	40.57	38.86	36.17	37.70	35.94	37.08	37.72
	40.24	41.06	40.10	35.68	35.22	34.42	37.79
3	45.70	44.64	42.25	38.84	39.86	38.79	41.68
	40.54	43.03	39.43	37.42	35.85	35.44	38.62
4	49.27	48.96	41.85	39.97	39.20	39.94	43.20
	41.57	42.19	40.73	36.44	35.79	36.25	38.83
5	52.90	44.45	43.21	44.88	45.11	48.35	46.48
	45.82	45.80	42.39	40.75	41.02	41.93	42.95

Table 2: numbers calculated from Table 1

#	t1	t2	t3	t4	t5	t6	average
	Kbytes/sec (ca.)						
1	482	484	536	567	553	553	527
	519	507	528	562	599	602	550
2	516	539	579	556	583	565	555
	520	510	522	587	595	608	554
3	458	469	496	539	525	540	502
	517	487	531	560	584	591	542
4	425	428	500	524	534	524	485
	504	496	514	575	585	578	539
5	396	471	485	467	464	433	451
	457	457	494	514	510	499	487

It is interesting to note that even on the tiny private and closed network used for the tests there is a wide variation in the numbers. However, the general direction is clear. All test transfers started slow and became gradually faster, here we see TCP's slow-start algorithm at work. I have intentionally chosen a setup where the hosts and not the network are the bottleneck, because otherwise obviously the numbers would be worthless. I did no tests with virtual servers, because first I did not expect any surprises, i.e. completely different numbers, and second, I have implemented the structure that stores the dynamic client data as a list, which should be replaced by a more sophisticated structure like hashes or a binary tree anyway. The list, however, was the choice in order to not complicate my very first virtual server implementation unnecessarily. It made bug tracking easier. Third, testing the algorithm used for selecting a real server to map a client to is not useful since this choice is done just once.

7 Bibliography

References

[1] K. Egevang, P. Francis, RFC 1631 “The IP Network Address Translator (NAT)” (1994).

[2] Fuller, V., Li, T., and J. Yu, RFC 1519, "Classless Inter-Domain Routing (CIDR) an Address Assignment and Aggregation Strategy" (September 1993)

[3] IETF Network Working Group, RFC 1918, "Address Allocation for Private Internets" (1996)

[4] Linux Kernel Hacker Guide, http://www.redhat.com:8080/HyperNews/get/khg.html

[5] Load Distribution for Firewall-1, Check Point Software Technologies Ltd., seen at the 1997 CeBIT

[6] K. Washburn, J. Evans, “TCP/IP” (Addison Wesley, 1994)

[7] The Squid Internet Object Cache, http://squid.nlanr.net/

[8] J. Postel, J. Reynolds, RFC 959 “FILE TRANSFER PROTOCOL” (1985)

[9] J. Postel, RFC 792 “INTERNET CONTROL MESSAGE PROTOCOL” (1981)