NAT - Network Address Translation

Introduction

Network Address Translation generally involves *"re-writing the source and/or destination addresses of IP packets as they pass through a router or firewall"* (from http://en.wikipedia.org/wiki/Network_Address_Translation)

This tutorial should explain what Network Address Translation is about, what to use it for and how to configure it under Linux (or more generally Unix-derivates). This introduction does not claim to be complete or covering all details, its main purpose is to provide the reader a feeling for what is possible and meaningful in modern computer networks and what is not.

First of all the structure of an IP-packet will be considered. After a short overview of the possibilities of the (Linux-)kernel I will jump right into the main area of application of NAT, namely the connection of a private subnet to the internet using a router (in our case a linux machine with iptables). After that I would like to present some further possibilities like redirection or how to circumvent restrictive proxies. However, I do not claim my presented solutions to be the most intelligent, most powerful or the most ingenious ones, they shall rather be understood as a proof of concept.

Packets within a network

Before we start to manipulate packets we have to take at look at their main attributes. I will restrict my considerations to IP-packets using TCP/UDP for the transport layer since these are the most common ones.

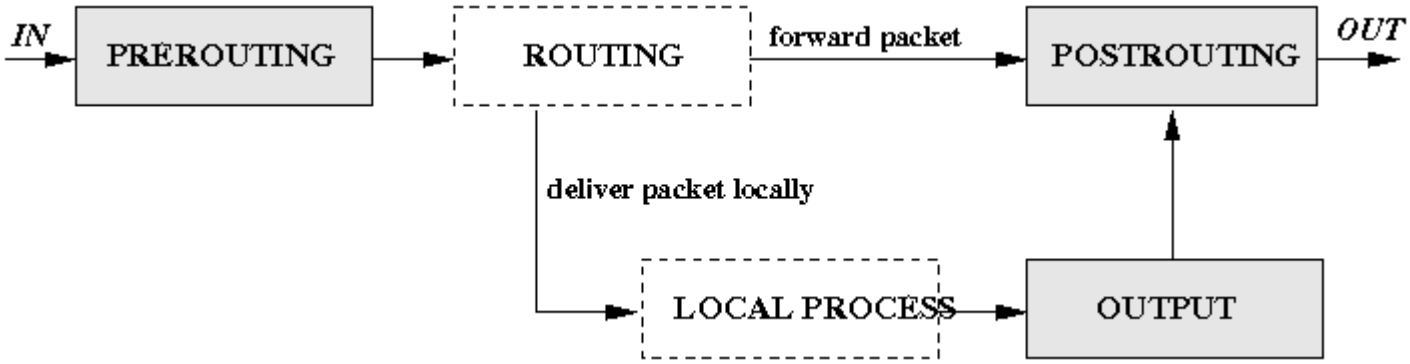
A detailled description of the structure of IP-packets can be found at various locations, e.g. IPv4 at Wikipedia. Our main focus will be on the two fields **Source Address** and **Destination Address** because they are containing - nomen est omen - the IP addresses of the source and the destination respectively.

Once an IP packet is received the receiver has to assign the data to a process, which is the role of the transport layer, in our case TCP and UDP. Once again further details can be found at various locations, e.g. TCP at Wikipedia and UDP at Wikipedia and the linked resources there. For our purposes it is sufficient to know that each (networking) processes uses its own port number(s). For example a http-Server uses port number 80, SSH uses port 22 and so on. The combination of IP-address and port number is called socket and is unique. Therefore connections are uniquely defined by their endpoints (=sockets), a connection sends data from the clients socket to the server socket and vice versa, for example from the socket with IP 123.123.123.123, port 65432 to the socket with IP 112.112.112.112, Port 80 as it may occur for a browser on 123.123.123.123 that connects to a http-server on 112.112.112.112. Server processes are usually using standardised ports, so called 'well known ports', c.f. well known ports at IANA or well known ports at Wikipedia. A client usually uses a port from the upper port range (larger than 1023).

Linux and Netfilter

The Linux kernel usually posesses a packet filter framework called **netfilter** (Project home: netfilter.org). This framework enables a Linux machine with an appropriate number of network cards (interfaces) to become a router capable of NAT. We will use the command utility

'iptables' to create complex rules for modification and filtering of packets. The important rules regarding NAT are - not very surprising - found in the 'nat'-table. This table has three predefinded chains: **PREROUTING**, **OUTPUT** und **POSTROUTING**.



The chains **PREROUTING** und **POSTROUTING** are the most important ones. As the name implies, the PREROUTING chain is responsible for packets that just arrived at the network interface. So far no routing decision has taken place, therefore it is not yet known whether the packet would be interpreted locally or whether it would be forwarded to another machine located at another network interface. After the packet has passed the PREROUTING chain the routing decision is made. In case that the local machine is the recipient, the packet will be directed to the corresponding process and we do not have to worry about NAT anymore. In case that the recipient is located in a (sub-)net located at a different network interface, the packet will be forwarded to that interface, provided that the machine is configured to do so. Just before our forwarded packet leaves the machine it passes the POSTROUTING chain and then leaves through the network interface. For locally generated packets there is a small difference: Instead of passing through the PREROUTING chain it passes the OUTPUT chain and then moves on to the POSTROUTING chain.

Before we start with our packet manipulations we have to enable the required features. To get all the needed functionality the commands (without '$>' at the beginning, lines starting with '#' are comments)

```
# IMPORTANT: Activate IP-forwarding in the kernel!

# Disabled by default!
$> echo "1" > /proc/sys/net/ipv4/ip_forward

# Load various modules. Usually they are already loaded
# (especially for newer kernels), in that case
# the following commands are not needed.

# Load iptables module:
$> modprobe ip_tables

# activate connection tracking
# (connection's status are taken into account)
$> modprobe ip_conntrack

# Special features for IRC:
$> modprobe ip_conntrack_irc

# Special features for FTP:
$> modprobe ip_conntrack_ftp
```

should be sufficient. In case of error messages it is likely that you do not have the routing features compiled into your kernel and should for example have a look

at http://www.comptechdoc.org/os/linux/usersguide/linux_ugipmasq.html.

Example: Connect a private subnet to the internet using NAT

On the one hand we know how IP packets look like, on the other hand we are ready to manipulate packets under Linux (and other Unix derivates). Therefore we are ready for our first application! The most popular question regarding NAT seems to be the one about sharing an internet connection for computers within a private subnet. For this reason I want to start with this particular scenario.

An analogon: Several subtenants without own postal addresses

First we should consider the following accurate analogon which is hopefully much easier to understand: Let us assume the following situation: there is a landlord with several subtenants. The postman has no idea about the subtenants und would reject every letter that is directly addressed to one of the subtenants. The landlord has several pigeon holes that can be used for addressing. The subtenants have the possibility to place their letters in a postbox at the landlord's office who will then take the letters to the post office. The question now is: How can all the subtenants fully participate at any kind of mail correspondence (i.e. send and receive letters)?

One solution for this given problem is the following: The landlord takes the letters sent by the subtenants, assigns each subtenant a pigeon hole and then replaces the subtenants address (which is in some sense invalid since the postman would reject any answers) by the landlord's own address including the pigeon hole number. The receipient of such a letter will then send the reply back to the landlord including the pigeon hole number and then the landlord could easily hand over (after he has replaced his address by the subtenant's address so that the subtenant does not recognise this 'cheat') the letter to the matched subtenant. This solution is optimal in the sense that it is fully transparent for the subtenants, none of them would ever notice the postman not being capable of sending letters directly to the subtenants!

From the subtenant problem to the computer world

NAT just works similar to the subtenant problem mentioned above. Every subtenant family represents an IP address in the local net, every subtenant family member represents a port number, the landlords represents a router and the recipient acts as an arbitrary computer in the internet. Consequently a socket can be seen as a combination of address and pigeon hole or subtenant family und a member of that family. Let us recapitulate: The process of communication is as follows:

- The subtenants have to put their letters into the postbox at the landlord's office
- The landlord replaces the sender's address by his own including the pigeon hole number
- Once he gets a reply the landlord has to replace his address (including the pigeon hole number) with the corresponding subtenant's address.

Actually the situation in the local net is nearly the same:

- All computers within the local net ('clients') send their packets with the receipient's socket to the router (this is actually realised by setting the router as standard gateway at the client, the delivery is then handled using Ethernet or any other lower level protocol).
- The router replaces the sender's socket by an own, unused socket.
- Replies to this specific socket will be forwarded to the appropriate computer in the local net, replacing the receipients address (the router's socket) by the clients socket.

We will presume that the standard gateway is set properly at each client. All that is left is to configure the router. Fortunately the netfilter framework automatically adds to each rule its inverse rule, therefore we only have to set one explicit rule. Usually the decision for one of these two rules is made by taking the one with the lower level of undetermination. For example, the rule 'Replace

the sender's address for all packets from the local subnet' is much easier than 'if a client has sent something to a server, then replace the receipient in the server's response by something'. As a rule of thumb can be used that the rule that is executed first is the one that is set explicitly in the kernel.

How to set rules

All we want to have is the following: packets arriving from the local net with a receipient's IP address somewhere in the internet have to be modified such that the sender's address is equal to the router's address. For further command examples let us assume that the first interface 'eth0' is connected to the local net and that the router is connected to the internet via the second interface 'eth1'. The command for a shared internet connection then simply is:

```
# Connect a LAN to the internet
$> iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

This command can be explained in the following way:

iptables:	the command line utility for configuring the kernel
-t nat	select table "nat" for configuration of NAT rules.
-A POSTROUTING	Append a rule to the POSTROUTING chain (-A stands for "append").
-o eth1	this rule is valid for packets that leave on the second network interface (-o stands for "output")
-j MASQUERADE	the action that should take place is to 'masquerade' packets, i.e. replacing the sender's address by the router's address.

Some further comments on the above command instruction: packets generated by the router itself are masked as well because they pass the POSTROUTING chain too! (see illustration further above) However, since the kernel tries to keep the source ports unchanged and processes running on the router acquire free ports only, locally generated packets usually remain unchanged, although the rule is executed. The output interface can be of any type, ISDN or DSL interfaces are also possible (often 'ppp0' or 'ippp0'). A quick overview of all available network interfaces gives

```
# Display available network interfaces
$> ifconfig
```

Disadvantages of using NAT

Local computers can access the internet, but there are still some restrictions left. A computer located in the internet is not able to establish a connection to a local computer, all he can do is address (a port of) the router and hope for the best. Usually the addressed port is currently not used and hence the packet will be rejected. Even if the port is currently used by a local machine the packet might be forwarded but will then usually be rejected since the computer is already communicating with a different computer. Establishing connections from the internet to one of the local computers is therefore nearly impossible. For regular services it is possible to statically map ports on the router to sockets in the local net, for example one can configure the router to forward packets arriving at port 80 to a HTTP-server located in the local net. Very often this is needed for playing online games, especially if you want to host games.

A closer look at iptables

Now that we have mastered our first challenge it is time to have a closer look (or two) at the possibilites of *iptables*. An iptables-call has the following pattern:

```
# Abstract structure of an iptables instruction:
iptables [-t table] command [match pattern] [action]
```

For NAT we always have to choose the *nat*-table. A command might need further options, for example a pattern and an action to perform in case the pattern matches.

Choosing a table

All our commands regarding NAT will start like this:

```
# Choosing the nat-table
# (further arguments abbreviated by [...])
iptables -t nat [...]
```

This selects the *nat*-table. There are two other tables, namely *mangle* und *filter*, but those are not used for NAT and therefore I mention them for completeness only. Since the default table is *filter* we have to select the *nat* table every time again.

Commands

The most important commands are the following: (further patterns and actions again abbreviated with [...])

```
# In the following "chain" represents
# one of the chains PREROUTING, OUTPUT and POSTROUTING

# add a rule:
$> iptables -t nat -A chain [...]

# list rules:
$> iptables -t nat -L

# remove user-defined chain with index 'myindex':
$> iptables -t nat -D chain myindex

# Remove all rules in chain 'chain':

$> iptables -t nat -F chain
```

For a full listing of all possible commands I recommend the manual pages of iptables. To view them, simply type

```
# manual pages of iptables
$> man iptables
```

and quit by typing 'q'.

Choosing match patterns

To manipulate specific packets we have to use appropriate match patterns, therefore there a numerous options to specify them. I will present the most popular ones to clarify their usage. All

available match patterns can be found in the manual pages of *iptables*.

```
# actions to be taken on matched packets
# will be abbreviated by '[...]'.
# Depending on the match pattern the appropriate chain is selected.

# TCP packets from 192.168.1.2:
$> iptables -t nat -A POSTROUTING -p tcp -s 192.168.1.2 [...]

# UDP packets to 192.168.1.2:
$> iptables -t nat -A POSTROUTING -p udp -d 192.168.1.2 [...]

# all packets from 192.168.x.x arriving at eth0:
$> iptables -t nat -A PREROUTING -s 192.168.0.0/16 -i eth0 [...]

# all packets except TCP packets and except packets from 192.168.1.2:
$> iptables -t nat -A PREROUTING -p ! tcp -s ! 192.168.1.2 [...]

# packets leaving at eth1:
$> iptables -t nat -A POSTROUTING -o eth1 [...]

# TCP packets from 192.168.1.2, port 12345 to 12356
# to 123.123.123.123, Port 22
# (a backslash indicates contination at the next line)
$> iptables -t nat -A POSTROUTING -p tcp -s 192.168.1.2 \
   --sport 12345:12356 -d 123.123.123.123 --dport 22 [...]
```

For most of the switches there exists a long form, e.g. *--source* instead of *-s*. Using them makes the whole instruction longer but more readable, especially if you are new to iptables.

Actions for matched packets

We are already able to select desired packets, all that is left is an appropriate action. For the nat-table only the actions *SNAT*, *MASQUERADE*, *DNAT* and *REDIRECT*, all of them with preceeding '*-j*', are meaningful. Their exact meaning will be explained in the subsequent section.

```
# In the following the table selection, the command and the match pattern
# will be abbreviated using [...]

# Source-NAT: Change sender to 123.123.123.123
$> iptables [...] -j SNAT --to-source 123.123.123.123

# Mask: Change sender to outgoing network interface
$> iptables [...] -j MASQUERADE

# Destination-NAT: Change receipient to 123.123.123.123, port 22
$> iptables [...] -j DNAT --to-destination 123.123.123.123:22

# Redirect to local port 8080
$> iptables [...] -j REDIRECT --to-ports 8080
```

Explaination of possible actions

Now most of the options of iptables are explained and it is time to have a closer look at the four possible actions:

Source-NAT (SNAT) - Change sender statically

In our previous example of connecting a local net to the internet we already used Source NAT (short: SNAT). As the name implies the sender's address is changed statically. The reason for choosing MASQUERADE in the previous example anyway has the following reason: For SNAT one has to specify the new source-IP explicitly. For routers with a static IP address SNAT is the best choice because it is faster than MASQUERADE which has to check the current IP address of the outgoing network interface at every packet. Since SNAT is only meaningful for packets leaving the router it is used within the POSTROUTING chain only.

```
# Options for SNAT (abstract of manual page)
--to-source <ipaddr>[-<ipaddr>][:port-port]
```

MASQUERADE - Change sender to router's IP-Adress

Using the MASQUERADE target every packet receives the IP of the router's outgoing interface. The advantage over SNAT is that dynamically assigned IP addresses from the provider do not affect the rule, there is no need to adopt the rule. For ordinary SNAT you would have to change the rule every time the IP of the outgoing interface changes. As for SNAT, MASQUERADE is meaningful within the POSTROUTING-chain only. Unlike SNAT, MASQUERADE does not offer further options.

Destination-NAT (DNAT) - Changing the receipient

If you want to change the receipient of a packet, Destination NAT (DNAT) is your choice! DNAT can be used for servers running behind a firewall. Obviously the receipient has to be changed before any routing decisions are made, therefore DNAT is meaningful within the PREROUTING chain and the OUTPUT chain (for locally generated packets) only.

```
# Options for DNAT (abstract of manual page)
--to-destination <ipaddr>[-<ipaddr>][:port-port]
```

REDIRECT - Redirect packets to local machine

A special case of DNAT is REDIRECT. Packets are redirected to a local port of the router, enabling for example transparent proxying. As for DNAT, REDIRECT acts within the PREROUTING and the OUTPUT chain respectively.

```
# Options for REDIRECT (abstract of manual page)
--to-ports <port>[-<port>]
```

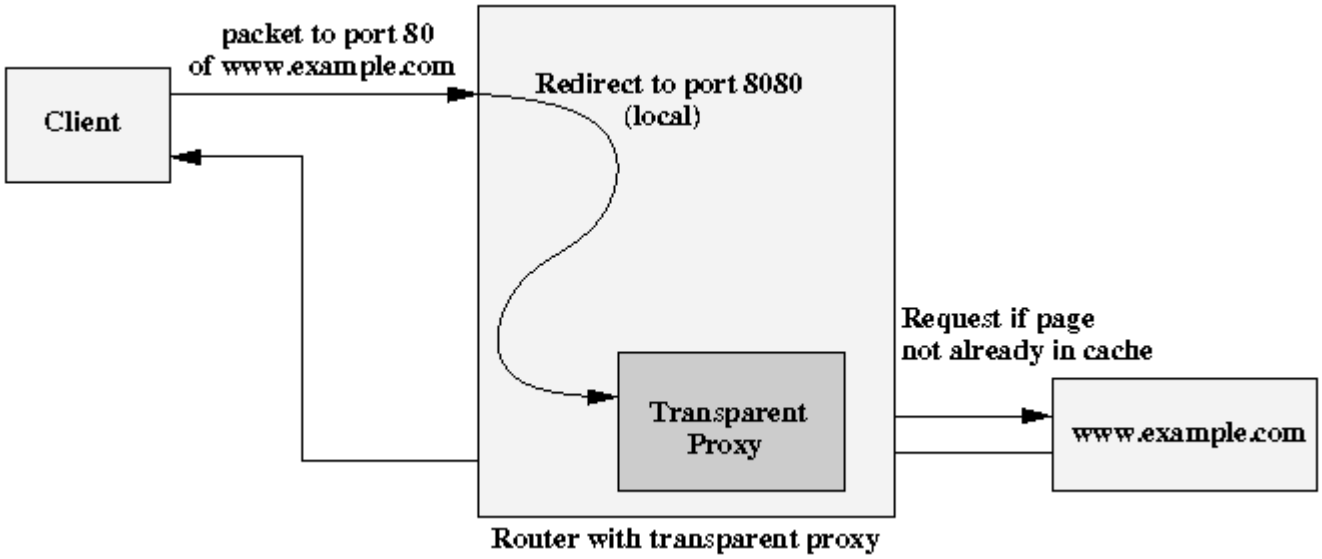
Applications

Probably you have been confused by the cryptic instructions used in the first example (connecting a local net to the internet). Now, after some explanations, the instructions are hopefully a little bit clearer than before. The question might have changed from 'How can someone construct such cryptic instructions' to 'Okay, and what can I do with all these cryptic instructions?'. This section tries to give you some ideas on how to use NAT. The range of applications is moreless unlimited, however I will try to cover the most spread use cases.

Transparent Proxying

Let us assume that we have a local net connected to the internet using NAT. To keep the traffic low we would like to run a HTTP-proxy on port 8080 of the local network interface handling all of the http-traffic.

The first solution that comes in mind is to 'motivate' each user (i.e. do it on your own) to configure their browser to use the proxy server and afterwards block all outgoing traffic to port 80. This might be a satisfactory solution for small networks but does not scale for large networks, because you would have to take care on every single client! (At least you avoid the disadvantages of transparent proxying...)



With NAT we have another possibility: All incoming packets going to port 80 will be redirected to port 8080. The command is:

```
# Transparent proxying:
# (local net at eth0, proxy server at port 8080)
$> iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 \
   -j REDIRECT --to-ports 8080
```

Of course a HTTP-Proxy at port 8080 needs to be up and running. Maybe some special configurations (or even a special compilation) are needed for your proxy server in order to support transparent proxying. Disadvantages of transparent proxying are the higher CPU load (especially for really large networks) and some problems with old or very simple browsers.

Help! I am behind a restrictive firewall!

Before we start I have to place a warning:

Everybody has to check on his or her own, whether the following steps violate any existing usage conditions, BEFORE he or she uses one of the presented techniques! Usage of the following commands are at your own risk; I am not responsible for damages or fines that result from an inappropriate use of the following commands and techniques!

Although you may not expect it, but NAT can even help you in such a case! Let us assume that only a few ports can be reached from your local network. First of all one has to find these open ports. One wide spread utility to use is nmap: (please scan your own computers only, scanning unknown computers can be interpreted as a first step to intrusion!)

```
# Scan a machine:
# (Replace www.example.com by an appropriate machine)
$> nmap www.example.com
```

The output will display various ports, most of them will be in state 'closed' (no service at that port), others will be in state 'filtered' (no connection to that port), perhaps some are in state 'open' (service running). Let us assume all ports below 5000 are closed, except port 80, but there are ports starting at 5000 that can be reached. To get a connection to any port below 5000 to an arbitrary machine one needs a (Linux-) machine that is located outside the firewall (no matter where as long as it can be reached and is not itself restricted by a firewall), that can be accessed and that supports NAT (iptables).

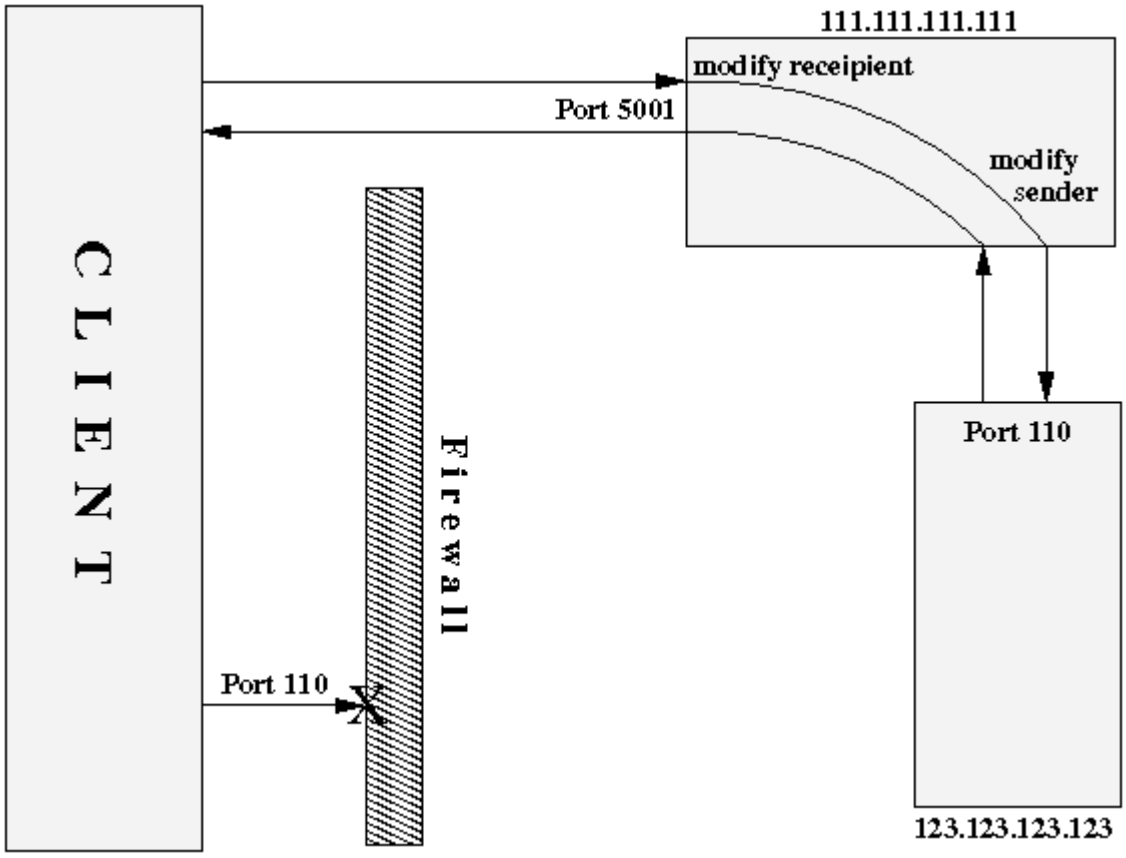
First of all we have to gain access to that machine (suppose IP 111.111.111.111) outside the firewall. We use any workstation outside the firewall to establish a SSH connection to 111.111.111.111. Then we issue the command

```
# Redirect SSH from port 5000 to port 22:
$> iptables -t nat -A PREROUTING -p tcp --dport 5000 -j REDIRECT --to-ports 22
```

Now we can return to our machine behind the restrictive firewall and can access 111.111.111.111 via SSH on port 5000. Alternatively you can configure your SSH-daemon to run on port 5000. However, now you are able to configure the remote machine appropriately. To connect to port 110 (POP3) on machine 123.123.123.123, issue the command

```
# redirect port 5001 to port 110 (POP3) at 123.123.123.123:
$> iptables -t nat -A PREROUTING -p tcp --dport 5001 \
   -j DNAT --to-destination 123.123.123.123:110

# Change sender to redirecting machine:
$> iptables -t nat -A POSTROUTING -p tcp --dport 110 \
   -j MASQUERADE
```



Instead of the last MASQUERADE command SNAT is possible as well (*-j SNAT --to-source 111.111.111.111*), but since we are the only user of that connection this does not matter, therefore I chose MASQUERADE because it is shorter and easier to use.

We can connect to all other ports in exactly the same manner as long as we have enough open ports. Even secure connections (like IMAPS) can be redirected in that way, warnings regarding the security certificate have to be ignored (as long as we know what we are doing!). For the POP3-connection from above you still have to configure your mail client accordingly, your server for mails would then be 111.111.111.111, port 5001.

If our 'provider' runs a proxy for connections to port 80 (i.e. the provider uses a transparent proxy) and monitors the content and we do not agree with that, then NAT can help us once again. Assuming that proxy avoidance is not prohibited, we can set up a proxy (configured for transparent proxying) at 111.111.111.111, port 5002. After that you enter on your **local** machine the following command:

```
# redirect http-Traffic going to Port 80 to 111.111.111.111:5002:
$> iptables -t nat -A OUTPUT -p tcp --dport 80 \
   -j DNAT --to-destination 111.111.111.111:5002
```

In that way you have successfully circumvented the provider's proxy! (*In my view it is very fancy to circumvent a transparent proxy using a transparent proxy! :-)*) Alternatively you can again configure your browser by hand to use your new proxy, but this may still lead to some troubles with other programs. However, configuring your browser by hand would at least avoid the disadvantages of transparent proxying.

To close this application I want to recapitulate our steps: First we find an open port for a SSH-connection. Then we statically redirect the other open ports to the desired destinations (usually one is not using that many non-HTTP-servers). If one manages to do this redirection dynamically, then

two open TCP ports (one for SSH) and one open UDP port are sufficient to connect to nearly every port at every machine, the only disadvantage then would be that you can have only one connection at a time (per protocol).

Running a Server behind a NAT-router

For servers running behind a NAT-router additional steps are needed since at first you cannot connect from outside to the server. Let us assume that we have a HTTP-server with IP 192.168.1.2 and our router has the IP address 192.168.1.1 and is connected to the internet over its second network interface with IP 123.123.123.123. To reach the HTTP-server from outside, type

```
# redirect http traffic to 192.168.1.2:
$> iptables -t nat -A PREROUTING -p tcp -i eth1 --dport 80 -j DNAT --to 192.168.1.2
```

and you are done. Now you are able to access the HTTP-server from outside using the IP 123.123.123.123.

Related articles

Similar topics can be found under:

- http://iptables-tutorial.frozentux.net/iptables-tutorial.html : Very comprehensive source of informations about iptables.
- http://www.faqs.org/docs/Linux-mini/TransparentProxy.html : Dealing extensively with transparent proxying.
- http://www.barryodonovan.com/publications/lg/108/ : Further abilities of the netfilter Framework by use of extensions.