OpenSSH
Section 2: SSH-tunnelling

Johannes Franken
<jfranken@jfranken.de>

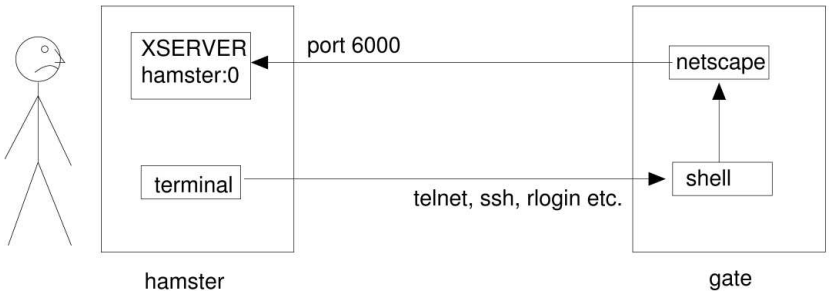
On this page I show you, how to tunnel ssh or other protocols over OpenSSH and what advantages you can obtain doing so.

Contents

X11-forwarding

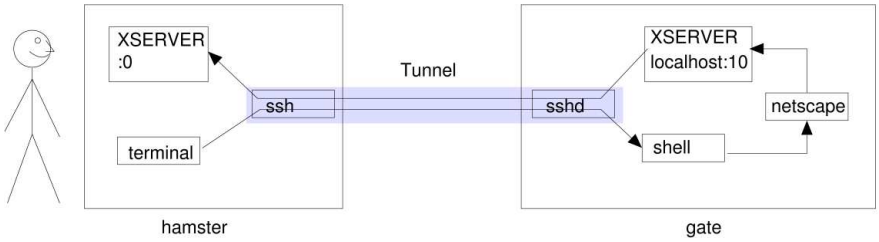
Under unix it's common to start applications ("X-clients") on distant computers and to teleguide them from the local screen ("X-server"). There're two different methods to setup such a connection:

- **not using a X11-tunnel**




```
jfranken@hamster: $ xhost gate
gate being added to access control list
jfranken@hamster: $ ssh gate
jfranken@gate: $ export DISPLAY=hamster:0
jfranken@gate: $ netscape &
[1] 17101
jfranken@gate: $
```

- **using a X11-tunnel**




```
jfranken@hamster: $ export DISPLAY=:0
jfranken@hamster: $ ssh -X gate
jfranken@gate: $ echo $DISPLAY
localhost:10.0
jfranken@gate: $ netscape &
[1] 17153
jfranken@gate: $
```

For this to work, on gate **`libX.so`** and **`xauth`** must be installed and **`/etc/ssh/sshd_config`** contain **`X11Forwarding yes`**.

Alternative to the **`-X`**-parameter you could call ssh with **`-o ForwardX11=yes`** or append **`ForwardX11 yes`** to your **`~/.ssh/config`** file.

Weighting:

criterion	not using a X11-tunnel	using a X11-tunnel
Encryption	**-** All communication goes over the wire in cleart-ext. For example, some network participant could catch all keys you press within your xterm and see any passwords you use.	**+** Communication is encrypted. The time exposure for encrypting is retrieved with ease by the compression.
X11-security	**-** The X-server must accept connections on tcp-port 6000, which implies a bunch of security disasters, e.g. unauthorized access to your screen.	**+** The x-server may be run with **`-nolisten tcp`**, which protects it agains unauthorized access from other network participants.
Firewall/NAT	**-** This method will not work with a firewall between ssh server and client.	**+** No problem with firewalls, as long a they let ssh pass trough.

Using X11-tunnels over ssh gives you a lot of advantages.

Pipes

If you call ssh with a command to execute, not using the **-t**-parameter, ssh will redirect that command's stdin/stdout/stderr to the shell it was called from. This way you can easily build ssh in pipes:

- The following command will show you the filling grade of the root partition at the host **gate**:

```
$ ssh gate df | awk '/\/$/ {print $5}'
64%
$
```

- The following command will copy the **mydir** directory into the **/tmp**-directory on the computer **gate**

```
$ tar cf - mydir/ | ssh gate 'cd /tmp && tar xpvf -'
```

imap (fetchmail, mutt)

Imap is a protocol for transferring mails. Unfortunately it transfers all mails other the net unencryptedly. If you have shell access to your mailserver, you should tunnel imap over ssh, which makes the transferring much safer (encryption, publickey authentication) and faster (compression). The easiest way is to have your mailuseragent call an imapd in preauth-mode on the mailserver and talk to it over ssh's stdin/stdout:

```
jfranken@hamster $ ssh gate imapd
* PREAUTH [231.36.30.64] IMAP4rev1 v12.264 server ready
```

Example configurations for some Mailuseragents:

- **fetchmail**: Any mail to the domain **jfranken.de** is received at my provider (**our-isp.org**), from where fetchmail picks it up at regular intervals and feeds it to my local mailserver (**gate.jfranken.de**). Due to the following **.fetchmailrc** fetchmail will tunnel the imap-protocol over ssh:

```
poll johannes.our-isp.org
        with options proto imap,
        preauth ssh,
        plugin "ssh -x -C jfranken@%h /usr/local/bin/imapd Maildir 2>/dev/null",
        smtphost gate,
        fetchall
```

- **mutt**: So my mail lays on **gate**, and I normally access it ifrom a local mutt, using imap. When I need to access my mail over the Internet, mutt will tunnel all imap comminucations over ssh. The following lines in **~/.muttrc** make this possible:

```
set tunnel="imapd || ssh -qC jfranken@gate.jfranken.de imapd"
set folder="{gate}~/Mail"
```

smtp (exim)

Some domains refuse receiving mails from dial-up systems. The following **`/etc/exim/exim.conf`** makes exim (version 3.35) route mails for such domains over a ssh-connection to **`johannes.our-isp.org`**, who has a fixed IP address:

```
ssh:
  driver = pipe

  bsmtp = all
  bsmtp_helo = true
  use_crlf = true
  prefix = ""
  suffix = ""

  command = ssh johannes.our-isp.org netcat -w 1 localhost smtp
  user = jfranken

  timeout = 300s
  temp_errors = 1
  return_fail_output = true

t_online:
  driver = domainlist
  transport = ssh
  route_list = "^t-online\.(de|at|com)|t-dialin\.net$ ;^mail.com$                    ;^lists.sourceforge.net$          ;^bigpond.com$                  ;^aol.com$              ...
```

rsync

rsync is an ingenious tool for incremental mirroring of directories, e.g. over several local hard drives, nfs or smbfs. When called with **`-e ssh`**, it will tunnel any communications over ssh, gladly through the Internet. I use the following commandline to upload my web pages to my webserver:

```
$ rsync --delete -a -e ssh ./ jfranken@www.jfranken.de:public_html/
```

More about:
see: rsync web pages

scp, sftp

More about:
see: scp(1), sftp(1) manpages.

uucp

uucp is a protocol for making files avaliable and picking them up. It's traditionally used for the submission of email and usenet-news. Since the authentification is done without encryption, I recommend tunnelling uucp over ssh. To allow this, just add one line for each uucp-user to the **~uucp/.ssh/authorized_keys** on the answering system:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,
command="/usr/sbin/uucico -l" ssh-rsa AAAAB3NzaC1yc2 ...
```

and configure some special modemport on the the calling system, which actually sets up an ssh connection and pipes everything through it:

```
/etc/uucp/sys:
        system YOURPROVIDER
        call-login *
        call-password *
        time any
        chat " \d\d\r\c ogin: \d\L word: \P"
        chat-timeout 30
        protocol i
        port ssh

/etc/uucp/port:
        port ssh
        type pipe
        command /usr/bin/ssh -qi ~/.ssh/id_rsa.uucp uucp@YOURPROVIDER
        reliable true
        protocol etyig
```

cvs

cvs is a versioning system for arbitrary files. It even solves the conflicts as well, which occour, whenever multiple users are trying to edit the same set of files. Synchronisation is done using either a shared directory on a filesystem (local, nfs, samba, etc) or a CVS-server, which can be connected to ssh. If you want ot setup a CVS-server, I recommend creating a **cvs** account on your repository server, which homes to the repository directory. Then put one line like this to its **~/.ssh/authorized_keys** for each user, which will actually start the cvs server process:

```
no-port-forwarding,no-X11-forwarding,command="/usr/bin/cvs server" ssh-rsa AAAAB3NzaC1yc2...
```

To tune their cvs to talk to the the cvs-server over ssh, users only need to enter the following commands:

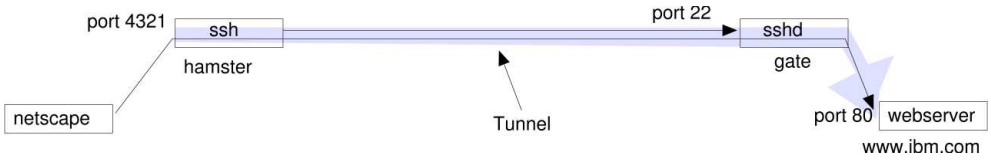
```
$ export CVS_RSH=ssh
$ export CVSROOT=:ext:cvs@host:/export/home/cvs
```

Then they can run cvs like usual, e.g.

```
$ cvs co module
```

Port forwarding

Local port forwarding



When I run **`ssh -g -L 4321:www.ibm.com:80 gate`** on hamster, ssh will intiate a session with gate, listen on port **`4321`** and handle any tcp connection attempts on that port to the sshd on **`gate`**, which will pass them to port **`80`** on **`www.ibm.com`**. The way back works vice versa. I actually setup a tunnel from **`hamster:4321`** to **`www.ibm.com:80`**.

In my webbrowser the website **`http://hamster:4321`** would look pretty much like IBM's.

I need root access on **`hamster`** to create a listening port <1024.

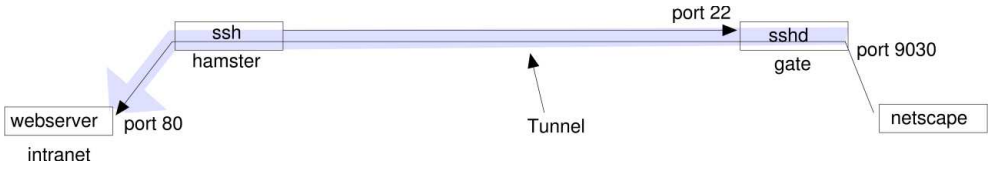
If I leave away the **`-g`** parameter, ssh will only listen on **`127.0.0.1`** (alias **`localhost`**), so the clients would have to be local for this, or change from another tunnel locally.

If the users on the sshd-host must not be authorized for shell access, but need to do portforwarding, you should set **`PasswordAuthentication=no`** in your **`/etc/ssh/sshd_config`** and then insert something like **`command="/bin/cat"`** or **`command="/bin/sleep 2000d"`** at the beginning of each public-key line in **`~/.ssh/authorized_keys`**.

If you're looking for a keepalive function, which keeps your firewall's natting tables from dropping the tunnels when idle, put the following option to the beginning of each public-key line:
`command="while :;do date;sleep 10;done"`

To restrict the accessible hosts and ports for forwarding, add some **`permitopen`**-options before the respective public keys. For example:
`permitopen="192.168.42.5:80",permitopen="127.0.0.1:8080"`

Remote port forwarding



When I type **`ssh -R 9030:intranet:80 gate`** on hamster, the **`sshd`** at **`gate`** will accept the connection, start listening on port **`9030`** and pass any packets on that port back to the **`ssh`**-client on **`hamster`**, which will forward them to port **`80`** at **`intranet`**. The way back works vice versa. I actually setup a tunnel from **`gate:9030`** to **`intranet:80`**.

So, people browsing to **`http://gate:9030`** will see your intranet server.

I need root access on **`gate`**, if you want to open a remote port <1024. If root-logins via ssh are forbidden, I can let the ssh-tunnel end on a high port (e.g. 9030) first, and have **`xinetd`**, **`netcat`** or firewall rules redirect it to the actual (low) port.

Try this **`/etc/inetd.conf`**:

```
80  stream tcp nowait nobody /bin/nc /bin/nc -w 3 localhost 9030
```

or **`/etc/xinetd.d/intranet`**:

```
service intranet
{
        type            = UNLISTED
        flags           = REUSE
        socket_type     = stream
        protocol        = tcp
        user            = root
        wait            = no
        instances       = UNLIMITED
        port            = 80
        redirect        = localhost 9030
        disable         = no
}
```

or this firewall script:

```
echo 1 > /proc/sys/net/ipv4/ip_forward # turns on forwarding
iptables -F -t nat # Flush existing translation tables
iptables -t nat -A PREROUTING -p tcp --dport 9030 -j DNAT --to localhost:80
iptables -t nat -A POSTROUTING -j MASQUERADE
```

As of its default configuration, **`sshd`** binds remote tunnels to the loopback interface, making them listen to requests from localhost only. If you want your tunnels to work for your other network interfaces as well, either add **`GatewayPorts yes`** to your **`/etc/ssh/sshd_config`** or redirect the port locally using ssh or xinetd as described above.

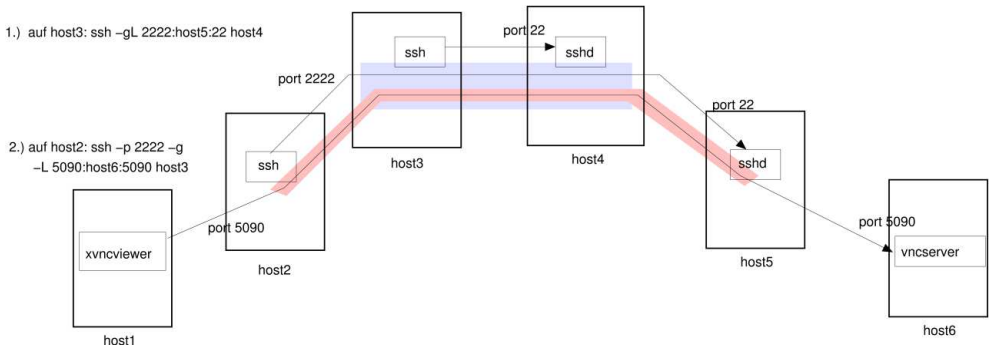
Telescoping tunnels

You can tell your **ssh**-client to connect to a port other than 22 by using the **-p** parameter. This comes in handy if you want to setup a ssh connection through a tunnel that you've already installed (e.g. in another ssh session).

With each tunnel you increase your reach up to two hosts:

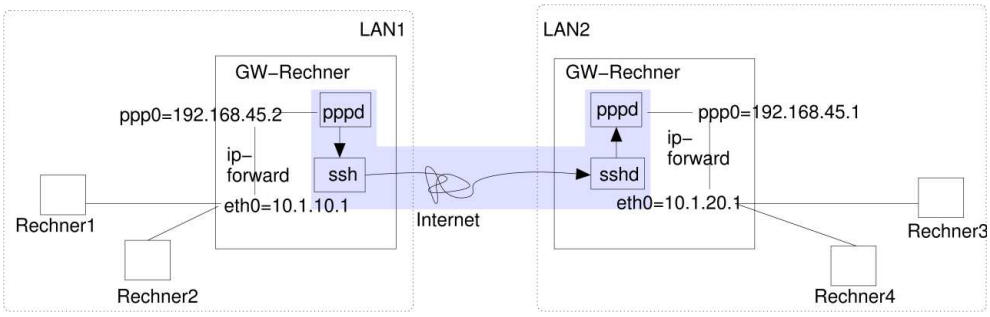
Number of tunnels	Max. hops
0	1
1	3
2	5
3	7

The following outline shows you how to involve five hosts with two telescoped tunnels, in order to setup a vnc session over three firewalls that don't like vnc.:



ppp over ssh

The point-to-point-protocol describes the connection establishment and IP communications between two virtual network interfaces. With a five minutes effort you can tunnel it over ssh, allowing you to route arbitrary IP packets over a ssh-connection.



Configuring the server:

1. Install ppp (e.g. version 2.4.1.uus-4 from Debian GNU/Linux 3.0)
2. Check the file attributes:

```
$ ls -l /usr/sbin/pppd
-rwsr-xr-- 1 root  dip   230604 10. Dez 2001  /usr/sbin/pppd*
```

3. Create a dedicated user account and authorize it to execute pppd:

```
$ adduser --group dip pppuser
```

4. Select a PAP-password and IP range:

```
$ echo 'pppuser * geheim *' >> /etc/ppp/pap-secrets
```

5. Assign IP-addresses to RSA-keys in **~pppuser/.ssh/authorized_keys** (one line per key)

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,command="/usr/sbin/pppd remotename pppuser refuse-chap refuse-mschap refuse-mschap-v2 refuse-eap require-pap 192.168.45.1:192.168.45.2 notty de...
```

6. Remove any unwanted routing- and fireall-initialisations from those **/etc/ppp/ip-up.d/*** scripts, as provided by your distribution for dialing into the Internet. If there are other PPP-connections configured on this system (e.g. for dialing into the Internet), the scripts can make use of the the $LINKNAME-variable, which has the value "pppoverssh".

Configuring the client:

1. Install ppp (e.g. version 2.4.1.uus-4 from Debian GNU/Linux 3.0)
2. Make sure you can access pppd and that it's setuid root:

```
$ ls -l /usr/sbin/pppd
-rwsr-xr-- 1 root  dip   230604 10. Dez 2001  /usr/sbin/pppd*
$ usermod -G dip jfranken
```

3. Create a new provider:

```
$ cat >/etc/ppp/peers/ssh <<EOF
pty 'ssh -e none pppuser@SERVER-HOSTNAME false'
user pppuser
nodetach
linkname pppoverssh
# debug
EOF
```

4. Store the server's PAP-password:

```
$ echo 'pppuser * geheim' >> /etc/ppp/pap-secrets
```

5. Tweak **`/etc/ip-up.d/*`** as neccessary (e.g. setting the defaultroute to **`$PPP_IFACE`**)

It should look like this, if you've done everything right:

```
jfranken@hamster:~ $ /usr/sbin/pppd call ssh
Using interface ppp0
Connect: ppp0 <--> /dev/ttyp4
Remote message: Login ok
kernel does not support PPP filtering
Deflate (15) compression enabled
Cannot determine ethernet address for proxy ARP
local  IP address 192.168.45.2
remote IP address 192.168.45.1
```

More about pppd-options:
see: pppd(8) manpage.

Handling network-timeouts

Keepalives

You can configure, if and how **ssh** and **sshd** should detect network aborts:

Side	Option	Effect
ssh	**ProtocolKeepAlives=***n*	After authentication, ssh sends a 32 byte empty packet to the sshd every *n* seconds. sshd does not care about this, but the server's TCP stack must send back an ACK for that packet. If the client's TCP stack does not receive an ACK for this or a later packet, it will retransmit for some time and then signal a connection-timeout to ssh, causing ssh to exit. Linux 2.4 sends 15 retransmits within **14 minutes**. You can configure the number of retransmits in **/proc/sys/net/ipv4/tcp_retries2** and **/etc/sysctl.conf**. TCP will retransmit in intervals of 3,6,12,24,48,60,60,60,... seconds.
ssh,sshd	**KeepAlive=***(yes|no)*	When opening the TCP-connection, the process will set the keepalive-socketoption, causing the TCP-stack to resend an old (already ACKed) segment when it does not receive data for some time (e.g. 2 hours). If this packet provokes the opposite side to repeat its last ACK and this ACK arrives within a timeout (e.g. 75 seconds), the connection is assumed to be alive. Otherwise the TCP-stack will repeat testing some (e.g. 9) times and then signal a connection-timeout to the process. With Linux 2.4 you can configure these times at **/proc/sys/net/ipv4/tcp_keepalive_intvl**, **/proc/sys/net/ipv4/tcp_keepalive_probes** and **/proc/sys/net/ipv4/tcp_keepalive_time** or permanently set them in **/etc/sysctl.conf**. With the prementioned default values the diagnosis of a lost connection takes **2h11'15"** altogether!
sshd	**ClientAliveInterval=***s* **ClientAliveCountMax=***n*	If sshd has not received any data for *s* seconds, it asks ssh to show a sign of life in intervals of *s* seconds, and drops the connection after *n* unsuccessfull challenges.

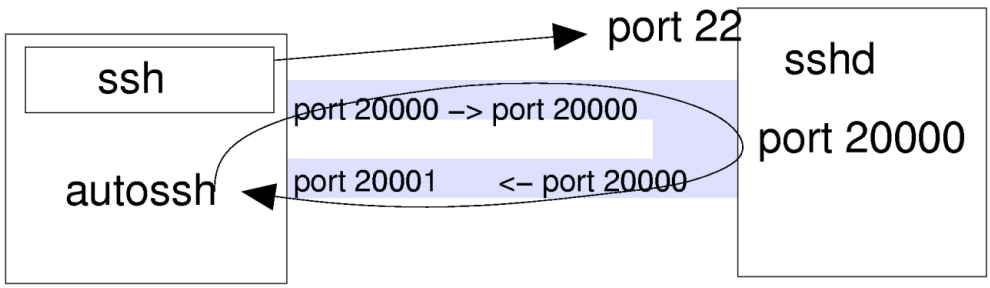
Discoveries:

- Since there is no **ServerAliveInterval** for the client, the client will hang for at least 15 minutes after certain network problems (e.g. NAT-timeouts), which is particularly annoying to any tunnels.
- If you set **ProtocolKeepAlives=0**, **KeepAlive=no** and **ClientAliveInterval=0**, you can take down the network connection and resume it at any time, e.g. after years.

autossh

autossh is a C-program by Carson Harding <harding@motd.ca> (see http://www.harding.motd.ca/autossh/). It solves the problem of hanging tunnels by

1. starting the ssh-client,
2. with a testloop through two portforwardings,
3. testing the connection over the testloop regularly and
4. stop and restart the ssh-client on problems.



With the following call I make my IMAP-server locally available. autossh checks the ssh-connection every 15 seconds after the first 30 seconds:

```
$ export AUTOSSH_GATETIME=30
$ export AUTOSSH_POLL=15
$ autossh -M 20000 -g -N -C -L 143:localhost:143 gate.jfranken.de
```

If you frequently setup tunnels, you might want to define a bash-alias to **ssh**:

```
$ alias ssh=':& a=$! ; port=$(( $a%45536 +20000 ))
  AUTOSSH_GATETIME=30 AUTOSSH_POLL=15 autossh -M $port'
$ ssh -g -N -C -L 143:localhost:143 gate.jfranken.de
[1] 6418
```

This writes the next PID to $a, calculates a value between 20000 and 65553 by adding 20000 or -25536 to $a, and then passes the result as monitoring-port to autossh.

Further readings

- Part 3: Breaking firewalls
- The Secure Shell TM Frequently Asked Questions
- OpenSSH Project Page