



OpenSSH

Teil 2: SSH-Tunnels

Johannes Franken
<jfranken@jfranken.de>

Auf dieser Seite erkläre ich, wie man ssh oder andere Protokolle über OpenSSH tunnelt und welche Vorteile sich dadurch ergeben.

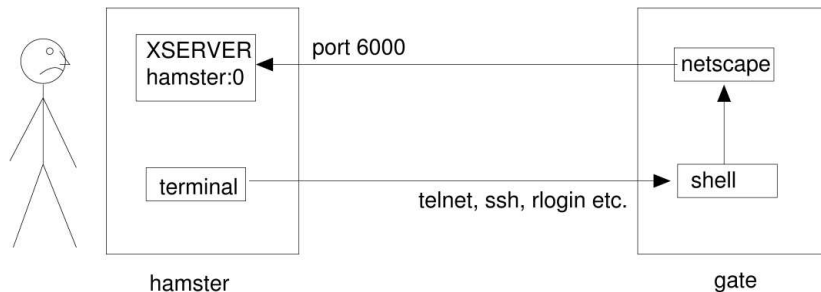
Inhalt

1. [X11-forwarding](#)
2. [Pipes](#)
 - a) [imap \(fetchmail, mutt\)](#)
 - b) [smtp \(exim\)](#)
 - c) [rsync](#)
 - d) [scp, sftp](#)
 - e) [uucp](#)
 - f) [cvs](#)
3. [Port forwarding](#)
 - a) [Local port forwarding](#)
 - b) [Remote port forwarding](#)
 - c) [Tunnels ineinander stecken](#)
4. [ppp over ssh](#)
5. [Umgang mit Netz-Timeouts](#)
 - a) [Keepalives](#)
 - b) [autossh](#)
6. [Weiterführendes](#)

X11-forwarding

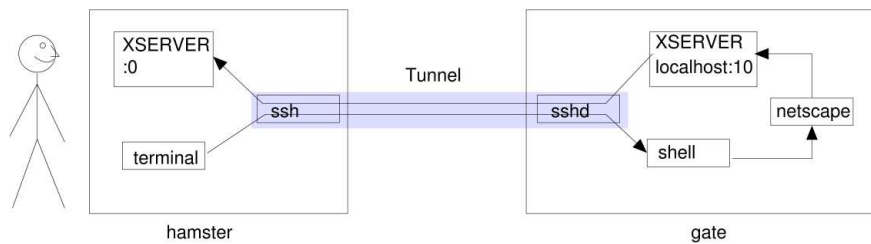
Unter Unix ist es üblich, Anwendungen ("X-Clients") auf anderen Rechnern aufzurufen und am eigenen Bildschirm ("X-Server") zu bedienen. Es gibt zwei Methoden des Verbindungsaufbaus:

- ohne 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: $
```

- mit 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: $
```

Hierzu muß auf gate die **libX.so** und **xauth** installiert sein und **X11Forwarding yes** in **/etc/ssh/sshd_config** stehen.

Alternativ zu dem **-x**-Parameter kann man dem ssh den Parameter **-o ForwardX11=yes** übergeben oder in der **~/.ssh/config**-Datei **ForwardX11 yes** eintragen.

Bewertung:

Kriterium	ohne X11-Tunnel	mit X11-Tunnel
Verschlüsselung	- Die Kommunikation läuft unverschlüsselt über's Netz. Ein anderer Netzteilnehmer könnte z.B. alle Tastendrücker und eingegebenen Passwörter mitschneiden.	+ Die Kommunikation erfolgt verschlüsselt. Der Zeitverlust für die Verschlüsselung wird durch die Kompression mehr als wett gemacht.
X11-security	- Der X-Server muß von aussen per tcp-port 6000 erreichbar sein, was weitere Gefahren bietet, z.B. unbefugtes Bildschirmauslesen	+ Der X-Server darf mit -nolisten tcp aufgerufen sein, was vor unberechtigten Zugriffen anderer Netzteilnehmer schützt.
Firewall/NAT	- Diese Methode funktioniert nicht mehr, sobald eine Firewall dazwischen steht.	+ Kein Problem mit Firewalls, solange sie ssh durchlassen.

Die Verwendung von X11-Tunnels über ssh bringt also klare Vorteile.

Pipes

Wenn man ssh einen Befehl übergibt und nicht den `-t`-Parameter setzt, leitet ssh stdin/stdout/stderr des Befehls unverändert an die aufrufende Shell weiter. Auf diese Weise kann man ssh in Pipes einbauen:

- Die folgende Befehlsfolge zeigt den Füllstand der Rootpartition des Rechners `gate` an:

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

- Die folgende Befehlsfolge kopiert das Verzeichnis `mydir` in das `/tmp`-Verzeichnis des Rechners `gate`

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

imap (fetchmail, mutt)

Imap ist ein Protokoll zum Übertragen von Mails. Leider sendet es die Mails dabei unverschlüsselt über's Netz. Wer Shellzugriff auf seinen Mailserver hat, sollte es über ssh tunneln, was die Übertragung wesentlich sicherer (Verschlüsselung, Publickey-Authentifikation) und schneller (Kompression) macht. Am einfachsten geht das, indem der Mailuseragent auf dem Mailserver einen `imapd` im `PreAuth`-Modus aufruft und sich über stdin/stdout der ssh mit diesem unterhält:

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

Beispiele zur Konfiguration einiger Mailuseragents:

- **fetchmail**: Alle Mail an die Domain `jfranken.de` landet bei meinem Provider (`our-isp.org`), von wo fetchmail sie regelmäßig über imap auf meinen lokalen Mailserver (`gate.jfranken.de`) abholt. Mit der folgenden `~/fetchmailrc` tunnelt fetchmail die imap-Kommunikation über 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",
  smtp host gate,
  fetchall
```

- **mutt**: Meine Mail liegt also auf `gate`, und ich greife normalerweise lokal mit mutt über imap darauf zu. Falls ich dies von einem anderen Rechner versuche (z.B. mit dem Notebook über's Internet), tunnelt mutt die imap-Kommunikation zu `gate` über ssh. Folgende Zeilen in der `~/muttrc` ermöglichen das:

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

smtp (exim)

Einige Domains nehmen keine Mails von Dialin-Systemen entgegen. Die folgende `/etc/exim/exim.conf` bringt exim (Version 3.35) dazu, die Mails an diese Domains über eine ssh-Verbindung zu `johannes.our-isp.org` zu leiten, der über eine feste IP-Adresse verfügt:

```
ssh:
  driver = pipe
  bsmtp = all
  bsmtp_hello = 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$
               /*lista.sourceforge.net$ /*bigpond.com$ /*aol.com$
               ...
```

rsync

rsync ist ein geniales Tool zum inkrementellen Spiegeln von Verzeichnissen, z.B. über verschiedene lokale Festplatten, nfs oder smbfs. Wenn man es mit dem Parameter `-e ssh` aufruft, tunnelt es sämtliche Kommunikation über eine ssh-Pipe, gerne auch über's Internet. Mit folgendem Aufruf übertrage ich die Webseiten auf meinen Webserver:

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

Mehr zum Thema:

siehe: [rsync web pages](#)

scp, sftp

Tools zum Kopieren von Dateien über ssh.

Mehr zum Thema:

siehe: [scp\(1\)](#), [sftp\(1\)](#) manpages.

uucp

uucp dient dem Bereitstellen, Abholen und und Weiterverarbeiten von Dateien. Die traditionelle Anwendung besteht in der Verteilung von emails und usenet-news. Da die Anmeldung unverschlüsselt über's Netz geht, empfehle ich, die gesamte Kommunikation über ssh zu tunneln. Hierzu richte man dem user **uucp** auf dem antwortenden System für jeden möglichen Anrufer eine Zeile in der `~uucp/.ssh/authorized_keys` ein:

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

Auf dem anrufenden System konfiguriere ich einen speziellen Modemport, der eben alles über **ssh** piped:

```
/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 ist ein Programm zur Versionsverwaltung beliebiger Dateien. Zusätzlich löst es die Konflikte, die dadurch entstehen, dass mehrere User gleichzeitig an den selben Dateien Änderungen vornehmen. Zum Abgleich greift **cv**s entweder über das Filesystem (also lokal, nfs, samba etc.) auf ein gemeinsames Verzeichnis zu, oder kommuniziert mit einem CVS-Server, den man beispielsweise über ssh ansprechen kann. Zur Einrichtung eines CVS-Servers empfehle ich, auf dem Repository-Rechner einen User **cv**s anzulegen, in dessen Homedir das Repository liegt und ihm in seiner `~/.ssh/authorized_keys` für jeden Benutzer eine Zeile anzulegen, die einen CVS-Server-Prozess startet:

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

Um den CVS-Server über ssh anzusprechen, geben Benutzer zuerst folgende Befehle ein:

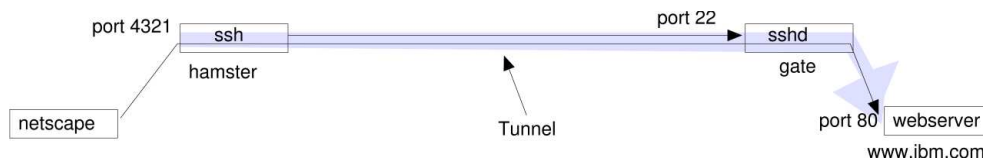
```
$ export CVS_RSH=ssh  
$ export CVSROOT=:ext:cvs@host:/export/home/cvs
```

und können dann wie gewohnt mit cvs arbeiten:

```
$ cvs co module
```

Port forwarding

Local port forwarding



Wenn ich auf hamster `ssh -g -L 4321:www.ibm.com:80 gate` aufrufe, baut ssh eine Verbindung zu `gate` auf, lauscht währenddessen auf Port 4321 und reicht alle dort ankommenden TCP-Verbindungen an den `sshd` an `gate` weiter, der sie an Port 80 von `www.ibm.com` weiterleitet. Der Rückweg funktioniert entsprechend. Ich habe einen Tunnel von `hamster:4321` auf `www.ibm.com:80` gelegt.

Im Browser würde `http://hamster:4321` also genauso aussehen wie die Webseite von IBM.

Man muss auf `hamster` Rootrechte haben, wenn man einen lokalen Port <1024 öffnen möchte.

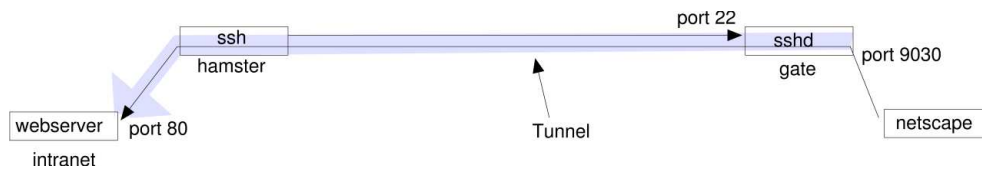
Wenn man den `-g` Parameter weglässt, können Clients den Port nur über die IP-Adresse `127.0.0.1` oder entsprechende Aliase (z.B. `localhost`) erreichen, müssen also auf dem selben Rechner wie der ssh-Client laufen, oder aus einem anderen Tunnel dort umsteigen.

Sollen die Benutzer auf dem `sshd`-Rechner keinen Shellzugriff, sondern nur Portforwarding können, kann man zunächst in der `/etc/ssh/sshd_config` die Option `PasswordAuthentication=no` setzen und dann für jeden betreffenden Publickey in der `~/.ssh/authorized_keys` am Zeilenanfang einen Aufruf wie `command="/bin/cat"` oder `command="sleep 2000d"` einfügen.

Wer zusätzlich einen keealive einbauen möchte, damit die Natting-Tabelle der Firewall im Leerlauf die Tunnels nicht kappt, schreibt an den Beginn der Publickeyzeilen jeweils:
`command="while :;do date;sleep 10;done"`

Um die möglichen Portforwarding-Ziele zu beschränken, kann man eine Liste von `permitopen`-Optionen vor den jeweiligen Publickeys einfügen, z.B.:
`permitopen="192.168.42.5:80",permitopen="127.0.0.1:8080"`

Remote port forwarding



Wenn ich auf hamster `ssh -R 9030:intranet:80 gate` eingebe, nimmt der `sshd` auf `gate` die Verbindung entgegen, lauscht auf Port 9030 und reicht alle dort ankommenden Verbindungen an den `ssh`-Client auf `hamster` weiter, der sie auf den Port 80 von `intranet` weiterleitet. Der Rückweg funktioniert entsprechend. Ich habe einen Tunnel von `gate:9030` nach `intranet:80` gelegt.

Wer im Browser `http://gate:9030` aufruft, landet auf dem Intranetserver.

Man muss auf `gate` Rootrechte haben, wenn man remote Port <1024 öffnen möchte. Wer den Rootzugriff per `ssh` nicht erlauben möchte, kann den `ssh`-Tunnel auf einen hohen Port (z.B. 9030) legen und mit `xinetd`, `netcat` oder Firewallregeln auf Port 80 umleiten.

Dies leisten folgende `/etc/inetd.conf`:

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

oder `/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
}
```

oder ein Firewallscript:

```
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
```

Der `sshd` bindet die remote Tunnels in seiner Standardkonfiguration an das loopback-Interface; er nimmt also nur Verbindungen von localhost entgegen. Wer möchte, dass seine Tunnels auf allen Netzinterfaces erreichbar sind, trage in der `/etc/ssh/sshd_config` auf `gate` die Zeile `GatewayPorts yes` ein oder leite den Port wie oben beschrieben mit `ssh` oder `xinetd` um.

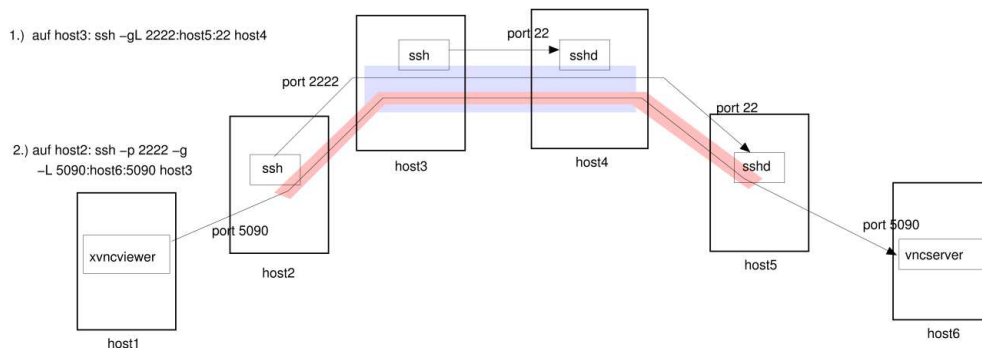
Tunnels ineinander stecken

Mit dem `-p` Parameter weise ich den `ssh`-Client an, den `sshd` auf einem anderen Zielport als 22 anzusprechen. Das ist z.B. sinnvoll, wenn ich eine ssh-Verbindung über einen bereits bestehenden Tunnel aufbauen möchte.

Mit jedem weiteren Tunnel erhöhe ich die Reichweite um bis zu zwei Hosts:

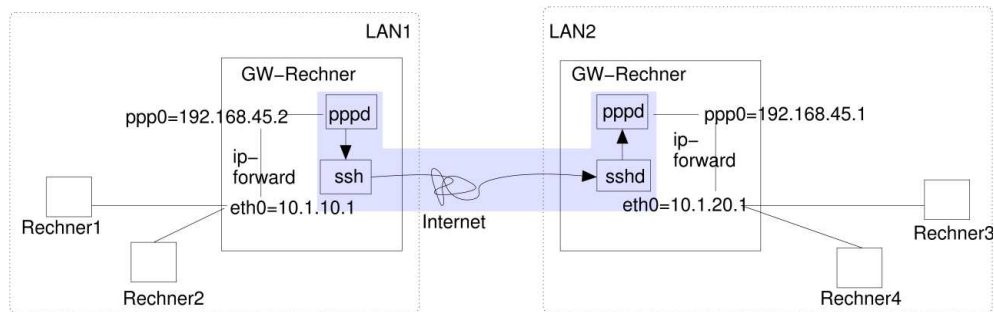
Zahl der Tunnels	Max. hops
0	1
1	3
2	5
3	7

Die folgende Abbildung zeigt, wie man mit zwei ineinander gesteckten Tunnels bereits fünf hosts beteiligen kann, etwa um eine VNC-Verbindung durch drei Firewalls zu tunneln, die VNC sonst nicht durchließen:



ppp over ssh

Das Point-to-point-Protokoll beschreibt Verbindungsaufbau und IP-Kommunikation zwischen zwei virtuellen Netzinterfaces. Mit etwas Aufwand kann man es auch über ssh tunneln, und so beliebige IP-Pakete über eine ssh-Verbindung routen.



Konfiguration des Servers:

1. Installation ppp (z.B. Version 2.4.1.uus-4 aus Debian GNU/Linux 3.0)
2. Dateirechte prüfen:

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

3. Einen Benutzer anlegen, der zum Aufruf des pppd berechtigt ist:

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

4. PAP-Passwort und IP-Adressbereich vergeben:

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

5. In der `~pppuser/.ssh/authorized_keys` den RSA-keys IP-Adressen zuordnen (Eine Zeile pro Publickey):

```
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. Die `/etc/ppp/ip-up.d/*`-Scripts von allen störenden Routing- und Firewall-Initialisierungen bereinigen, welche die Distribution für die Internetwahl vorgesehen hatte. Wenn auf dem Rechner weitere PPP-Verbindungen (z.B. DSL-Einwahl ins Internet) erforderlich sind, können sich die Scripts an der `$LINKNAME`-Variablen orientieren, die den Wert "pppoverssh" hat.

Konfiguration des Clients:

1. Installation ppp (z.B. Version 2.4.1.uus-4 aus Debian GNU/Linux 3.0)
2. Sicherstellen, dass der User den pppd setsuid aufrufen kann:

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

3. Einen neuen Provider anlegen:

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

4. Das PAP-Passwort des Servers eintragen:

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

5. `/etc/ip-up.d/*` ggf. anpassen (z.B. defaultroute auf `$PPP_IFACE` umsetzen)

So sieht der Verbindungsaufbau aus, wenn alles geklappt hat:

```
jfranken@hamster:~ $ /usr/sbin/pppd call ssh
Using interface ppp0
Connect: ppp0 <--> /dev/tty4
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
```

Mehr zum Thema pppd-Optionen:
siehe: [pppd\(8\)](#) manpage.

Umgang mit Netz-Timeouts

Keepalives

Man kann konfigurieren, ob und wie `ssh` und `sshd` Verbindungsabbrüche feststellen sollen:

Seite	Option	Auswirkung
ssh	<code>ProtocolKeepAlives=<i>n</i></code>	ssh sendet nach der Authentifizierung alle <i>n</i> Sekunden ein 32 Byte langes Leerpaket an den sshd. Das interessiert den sshd nicht weiter, aber der TCP-Stack des Servers muss ja TCP-Pakete regelmäßig mit ACKs beantworten. Wenn der TCP-Stack des Clients kein ACK auf dieses oder ein späteres Paket erhält, retransmitted er eine Zeit lang und signalisiert dann dem ssh ein Connection-Timeout, worauf dieser sich beendet. Linux 2.4 sendet 15 Retransmits und benötigt dafür etwa 14 Minuten . Die Zahl der Retransmits ist in <code>/proc/sys/net/ipv4/tcp_retries2</code> und <code>/etc/sysctl.conf</code> konfigurierbar. TCP wartet zwischen den Retransmits 3,6,12,24,48,60,60,60,... Sekunden auf Antwort.
ssh,sshd	<code>KeepAlive=(<i>yes/no</i>)</code>	Der Prozess setzt beim Öffnen der TCP-Verbindung die Keepalive-Socketoption. Wenn der TCP-Stack darauf länger (z.B. 2 Stunden) nichts empfängt, sendet er nochmal ein altes Segment, zu dem er bereits ein ACK erhalten hat. Wenn die Gegenseite dadurch zum Wiederholen ihres letzten ACKs provoziert wird und dieses (z.B. innerhalb von 75 Sekunden) eintrifft, ist die Verbindung OK. Sonst wiederholt der TCP-Stack den Test einige (z.B. 9) mal und signalisiert dann dem Prozess ein Connection-Timeout. Die genannten Werte sind bei Linux 2.4 in <code>/proc/sys/net/ipv4/tcp_keepalive_intvl</code> , <code>/proc/sys/net/ipv4/tcp_keepalive_probes</code> und <code>/proc/sys/net/ipv4/tcp_keepalive_time</code> bzw. in <code>/etc/sysctl.conf</code> konfigurierbar. Mit den o.g. Standardwerten dauert die Diagnose der verlorenen Verbindung insg. 2h11'15" !
sshd	<code>ClientAliveInterval=<i>s</i></code> <code>ClientAliveCountMax=<i>n</i></code>	Wenn sshd <i>s</i> Sekunden nichts empfangen hat, bittet er den ssh-Client im Abstand von <i>s</i> Sekunden um ein Lebenszeichen und beendet die Verbindung nach <i>n</i> erfolglosen Versuchen.

Feststellungen:

- Da es keinen `ServerAliveInterval`-Parameter für den Client gibt, bleibt der Client bei einigen Netzproblemen (z.B. NAT-Timeouts) mindestens eine Viertelstunde hängen, was für die getunnelten Verbindungen besonders lästig ist.
- Wenn man `ProtocolKeepAlives=0`, `KeepAlive=no` und `ClientAliveInterval=0` setzt und nach der Authentifizierung nichts mehr sendet, kann man die Netzverbindung trennen und beliebig (z.B. Jahre) später fortsetzen.

autossh

autossh ist ein C-Programm von Carson Harding <harding@motd.ca> (siehe <http://www.harding.motd.ca/autossh/>). Es löst das Problem hängender Tunnels, indem es

1. den ssh-Client startet ,
2. sich dabei mit zwei Portforwardings eine Testschleife einrichtet ,
3. regelmäßig die Verbindung über die Testschleife prüft und

