# Remote Linux Client Howto

On this page we provide useful information for accessing build systems and other tools from an off-site/remote linux machine.

## Useful SSH Tricks

Use ~/.ssh/config for your configurations!

### Use Wildcards and Nicknames

you can apply the same config settings to multiple hosts, e.g:

```
Host ioc-* cpu-*
User                  laci
SendEnv               TERM=xterm
HostName              %h.slac.stanford.edu
ProxyJump             <gateway>
StrictHostKeyChecking no
UserKnownHostsFile    /dev/null
```

### How to Avoid "*WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!*"

If you contact a server that is frequently rebooted and has no permanent storage (embedded system) then it might recreated a new host key each time it boots. Avoid the annoying warning using `/dev/null` for `UserKnownHostsFile` as shown above.

### Connecting via "Multiple Hops"

Often a ssh server is not directly reachable from the internet. In this case you can use a ProxyCommand - which is an arbitrary shell command that connects its stdio to the ssh server. In particular, you can use an ssh connection to a host which does have a connection (directly or indirectly) to the targeted server as a proxy.  See above for an example. Note that this can be applied recursively, i.e., if the 'gateway' in the previous example is not directly reachable from the internet but 'firewall' is then you can use

```
Host gateway
User          myself
ProxyCommand   ssh -X firewall /usr/bin/nc %h %p
```

However, as Faisal has pointed out - in the most common use-case the `ProxyJump` directive is more convenient:

```
Host gateway
User          myself
ProxyJump      firewall
```

Now when you say

```
bash$ ssh cpu-b12-xyz
```

then ssh will transparently set up the multi-hop connection. Note that others options can be passed and work as expected, i.e., you can set up port forwarding to the target machine, e.g.,

```
bash$ ssh cpu-b12-xyz -L8000:localhost:9000
```

sets up an encrypted port forward from the external machine (where ssh runs), port 8000 to port 9000 on cpu-b12-xyz.

### Use '-t'

When you run a remote command via ssh as in

```
bash$ ssh cpu-b12-xyz lengthy_command
```

then it is important to know that '`lengthy_command`' is *not* associated with any (remote!) terminal.  This means that when you kill `ssh` on your local machine (Ctrl-C) then the remote command will keep executing (the remote command is not a child of the local '`ssh`' and cannot be notified of its death). The '-t' option forces the remote ssh server to allocate a pseudo-terminal and associates the `lengthy_command` process with that terminal and this will allow propagation of signals:

```
bash$ ssh -tt cpu-b12-xyz lengthy_command
```

If you kill this ssh on your local machine then it will cause the `lengthy_command` to receive a signal (via its controlling terminal) and terminate as well. Multiple 't's ensure a remote terminal is allocated even if there is no local terminal (e.g., if the ssh command is called from a daemonized script).

## *screen* is Your Friend!

When you run an interactive session from remote then this often carries a lot of context information (environment, running processes etc.). It can be very painful if you get disconnected and as a consequence lose all of this context and have long-running build processes killed. Use the '`screen`' utility. If you work on AFS then `screen` also keeps your tokens alive for you (until they expire, of course) - provided that you run a `pagsh:`

```
bash$ screen pagsh
```

You start authenticating yourself (`kinit -f; aklog`) and then go about your business. You can at any time disconnect from the screen (which continues running all of its sub-processes and keeps terminal history etc) by hitting "`<CTRL>-A D`". You can then safely log-out and when you come back resume your screen session (hopefully that vivado build has completed in the mean time)

```
bash$ screen -r
```

If there are multiple screen sessions running then you may have to specify which one to resume. The above also works if you have been interrupted (e.g., by losing connectivity) but if screen things the last connection is still alive then you might have to reattach forcefully

```
bash$ screen -Dr
```

## Remote GUI Access

While you can run X11 over ssh it is frustratingly slow if your connection is not super-fast. VNC is a much better solution (for use with a windows client see here). Plus, if you run your VNC server inside a screen/pagsh then you can disconnect and reconnect to your GUI without killing the GUI application 😉. Of course, the VNC connection can be tunneled with ssh. The TightVNC `vncviewer` which is available under linux can be instructed to do so.

### Inside Machine

On the 'inside' you typically start a vncserver inside a screen/pagsh session (see above). Use a screen geometry that fits your remote viewer (client)

```
bash$ vncserver -geometry 1920x1200

New 'somemachine:1 (strauman)' desktop is somemachine:1
```

Once the server is up it prints its 'X11 display' information (e.g., which you use to set up the `DISPLAY` environment variable.

```
bash$ export DISPLAY=:1
```

After this step you can start GUI applications, e.g., (ruckus)

```
bash$ make gui
```

### Remote Client

On linux you can conveniently use the 'via' option to connect via ssh (this can be a multihop nickname as described above).

```
bash$ vncviewer -via somemachine     :1
```

where ':1' is the X11 display matching the info given by the server (note the blank space between the machine name which is an argument to -via and the display argument). If – for some reason – you use a different kind of tunnel or connection then the viewer might believe it is directly connected to the server. In this case you probably want to explicitly specify an efficient encoding - otherwise your experience will be frustratingly slow...

## Remote Access to JTAG / USB

Access to JTAG is commonly required for reprogramming the FPGA (some platforms) and for access to embedded ILA (logic-analyzer) cores.

### USB over IPs

If the system is equipped with a USB<->JTAG bridge and the vivado hw_server is not an option (e.g., because no linux x86 PC is in proximity) then linux' *us bip* feature is an option. This approach requires two linux computers. A remote system with physical access to the USB port (this can be e.g., a small portable device such as a raspberry) and a host with access to the hw_server. These two systems must be connected via TCP (but ssh tunneling etc. is possible).

*usbip* implements a virtual USB device on the local host where the remote device seems to be locally attached, i.e., it shows up on the USB bus of the local host where hw_server can find it. In a nutshell:

- On the remote machine: `usbip bind -b id`
- On the local machine: `usbip -r <remote_ip> -b <remote_id>`

You need root access on both machines for these operations. It is also noteworthy that if the remote USB device is disconnected (e.g., because of a FPGA power-cycle) it is necessary to recreate the virtual USB device on the local host. Consult the *usbip* documentation for details.

### XVC

It is also possible to drive JTAG with a firmware core and use the Xilinx XVC protocol to remotely access JTAG. The SLAC surf library provides the necess ary components (a firmware block and a software XVC server which must be run on a linux box with connectivity to the firmware). This is a purely networked solution and no hardware JTAG nor USB are required.

***Note***: when operating over a slow connection then I get better response when I start a Xilinx hw_server on some machine that is close to the xvcSrv (rather than tunneling XVC from the remote machine):

1. 
```
Start xvcSrv on a machine with fast connectivity to the FPGA

  xvcSrv -t <target_ip>[:<udp_port>]

A pre-compiled binary is installed here

  /afs/slac/g/reseng/xvcSrv/bin/<architecture>/
or here
  /afs/slac/g/lcls/package/xvcSrv/<architecture>/
```
2. Start `hw_server` on the same or a close-by machine (no special arguments necessary); if you run Vivado on-site then vivado takes care of this step and you may skip step 3.
3. From remote (e.g., laptop) you might have to use an `ssh` tunnel to get to the `hw_server`. You can in fact use `ssh` to directly launch the server and tunnel the connection (assuming `hw_server` is on the PATH):
```
  ssh -L 3121:localhost:3121 gateway_machine  hw_server
```
4. On the remote machine launch Vivado and connect to the default target (since the `ssh` tunnel was opened on local port 3121 Vivado will find it)
5. In the Vivado hardware manager connect to the hw target:
```
  % open_hw_target -xvc_url  <machine_where_xvcSrv_runs>:2542
```
6. That's it. Vivado should find the new target. You will need to load the debug probe (.ltx) file and possibly "refresh" the target.
7. The initialization then takes quite a while (from my home it takes ~30s for the GUI to populate in the hardware manager) but further interaction (including display of waveforms) is acceptable.