Securing BDL Applications Over the Internet with SSH - HOWTO

Morgan Ho, MoHo@4js.com.au

Draft Revision 0.4.a, Created 11dec2000, Last update 16jan2001

This document introduces SSH (secure shell) and how it can be used to run secure Four Js BDL applications over the Internet including a case study that covers an implementation using OpenSSH.

Table of Contents

1 Introduction

SSH is a secure shell replacement for **rsh** (or **rcmd**). SSH also provides port forwarding capabilities which allows other socket connections (eg. Windows or Java client) to be tunelled through an established SSH connection.

For more information about SSH, see http://www.ssh.org and FAQs at http://www.employees.org/~satch/ssh/faq/ssh-faq.html.

This document describes how to implement SSH to run secure BDL applications over the Internet. We will also examine a simple case study using open source implementations of SSH that support both SSH1 and SSH2 protocols.

2 Requirements

The requirements are pretty straight forward - you need an ssh daemon (**sshd**) on the server side to listen for requests and **ssh** on the client side.

Other client tools include **scp** (the secure analogy for **rcp**).

The following case study is based on OpenSSH http://www.openssh.com and OpenSSL http://www.openssl.org implementations.

The Linux versions used were from RedHat http://www.redhat.com.

There are a few Windows NT versions around. The one we used included SSH daemon for NT as well and can be found at:

http://marvin.criadvantage.com/caspian/Software/SSHD-NT/default.php
or
http://www.certaintysolutions.com/tech-advice/ssh_on_nt.html

which are Windows ports of OpenSSH using Cygnus GNU libraries (http://www.cygnus.com). Note this is a character mode implementation.

If you just want a minimal SSH client, try this lite package which is a simple subset of one of Windows OpenSSH ports:

http://www.4js.com.au/ssh/openssh-lite.zip

There are also many other commercial and shareware implementations, with varying features and levels of conformity. The main things to look for are (and beware, most are only **stelnet** clients):

- Protocol support (SSH1/SSH2)
- Range of ciphers
- Port redirection (ie. tunnelling)
- Compression (for performance)

Here are some others with GUI front-ends that we have tested:

MindTerm (Java based http://www.mindbright.se)
SSH Communications (http://www.ssh.com)
F-Secure SSH Client (http://www.f-secure.com)

Note: SSH1 protocol relied on encryption algorithms originally patented by RSA Data Security which expired in Sept 2000, which ends the US export restrictions. However there are many SSH product sites yet to acknowledge this and you may experience
some difficulty downloading from these anachronistic sites. You can either wait (I'm sure they'll get their act together) or use http://www.anonymous.com to work around this.

As an aside, this was an exceptionally good (and free) SSH and telnet terminal emulator (though it doesn't have all the features we need for tunnelling, it was a very good replacement for our previous commercial emulator).

PuTTY http://www.chiark.greenend.org.uk/~sgtatham/putty/

3 Server Setup

The server was an Intel based PC running RedHat 6.2 at kernel 2.2.16-3. The following RPMs were installed:

```
rpm -i openssl-0.9.6-1.i386.rpm
rpm -i openssl-misc-0.9.6-1.i386.rpm
rpm -i openssh-2.1.1p1-1.i386.rpm
rpm -i openssh-server-2.1.1p1-1.i386.rpm
rpm -i openssh-clients-2.1.1p1-1.i386.rpm
rpm -i openssh-askpass-2.1.1p1-1.i386.rpm
```

These are either available on your RedHat CD or can be downloaded from **http://www.redhat.com**. The server package will automatically generate a server key for you.

Then you need to allow access to sshd by adding the following entry to your **/etc/hosts.allow** file:

```
sshd : ALL
```

If you don't wish to provide global access you can replace **ALL** with a list of IP or domain name patterns.

Now, if **sshd** wasn't started up automatically (which it should do next time you reboot), then you can start it with:

```
/etc/rc.d/init.d/sshd start
```

4 Testing the Server

If you installed all of the above RPMs, you would have also installed the SSH client. You can test the server from the same server by running:

```
ssh server [-l user]
```

where *server* is the name or IP address of your server and *user* is the login id (default is your current login id).

By default, you will be using standard login/password authentication. You can also improve security with other options such as implementing RSA (SSH1) or DSA (SSH2) public keys as well as optional passphrases (if you don't use passphrases, you won't need to enter them – but if someone steals your private certificate...).

To use certificates, you need a tool like **ssh-keygen** to generate a *private* and *public* key, which can also incoporate your *passphrase* (typically a sentence rather than a password for added security). The public key is stored on the server (typically in your home directory's **.ssh** directory in file **identity.pub** (RSA) or **id_dsa.pub** (DSA).

You will need to examine your security requirements to determine the most appropriate authentication strategy to use. See the SSH FAQs for more information to help you decide.

For our example, we'll use the default password authentication.

The first time you connect, you will be warned that the server cannot be authenticated and the server's RSA/DSA fingerprint will be provided for you to verify (for the ultra paranoid). If you trust it, it will be added your list of known hosts. This is a safeguard against spoofing.

```
$ ssh server -l fred
The authenticity of host 'server.domain.com' can't be
established.
RSA key fingerprint is
af:33:8d:c5:8d:43:75:88:27:60:12:dd:ed:23:57:9b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'server.domain.com,192.0.0.1' (RSA)
to
the list of known hosts.
fred@server.domain.com's password:
$
```

After you answer "**yes**", you will be logged into a shell.

That's it.

5 Setting Up A Windows Client

Most of the GUI SSH clients use their own self-extracting installers.

The *openssh-lite* is quite straight forward to install.

1. Create a directory **`%SystemDrive%:\etc`**. The **passwd** file must reside on your system drive (typically C:).

2. Install the contents of the zip archive into this directory.

3. For command syntax

```
ssh -help
```

4. To login to your server

```
ssh server -l user
```

where *`user`* is your loginid on *`server`*. The first time you run this, it will create a **.ssh** directory to store **known_hosts** and a random number seed used for permuting subsequent ciphers. Using the lite version (ie. without the rest of the Cygnus GNU baggage), it will fail to find a shell and exit.

Note that Windows does not allow you to create a directory **.ssh**, so allow this initial step to create it for you.

Just run it again to actually connect.

```
C:\etc> ssh server1
setsockopt IPTOS_LOWDELAY: Invalid argument
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'server1' added to the list of known hosts.
Creating random seed file ~/.ssh/random_seed.  This may take a while.
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
execv /bin/sh failed: No such file or directory
Write failed: errno ESHUTDOWN triggered

C:\etc> ssh server2 -l fred
setsockopt IPTOS_LOWDELAY: Invalid argument
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'server2' added to the list of known hosts.
fred@server2's password:
setsockopt IPTOS_LOWDELAY: Invalid argument
$
```

5. If you wish to avoid using the **-l *loginid*** arguments, modify the entry in **passwd** to your login name on Windows.

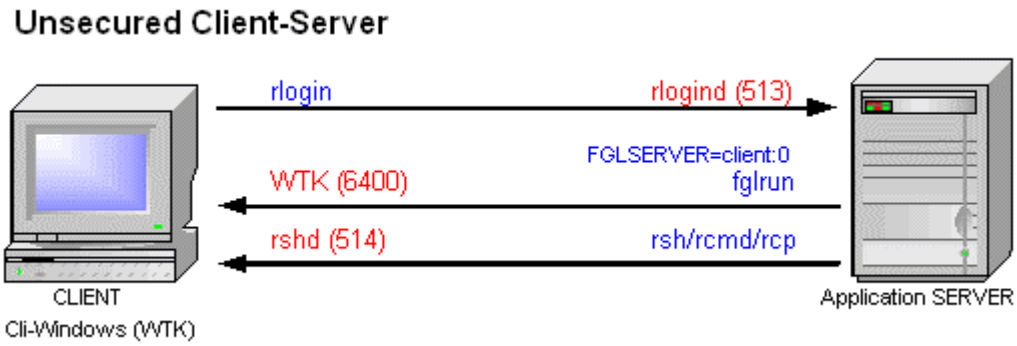
While OpenSSH may not be as elegant as the GUI offerings, it works quite well and as you have access to the source code, you can encapsulate it with your own GUI front end.

This Windows port is a little dated and does not support SSH2 or show the server fingerprint on initial connection, but works adequately enough. If you really need the latest, have a hunt on the web or grab the sources and the Cygus toolkit and go for it.

6 Tunnelling

One of the most powerful features of SSH is the ability to establish tunnels through the encrypted connection between the client and the server.

Let's look at how a typical Windows WTK client might start an application on a server.



1. The client starts the WTK display server and listens on the default port **6400**. If the option is enabled for WTK, the client will also start an **rshd** (remote shell daemon) on port **514**.

2. The client connects to the **rlogind** port (**513**) on the server for a nominated user login.

3. Assuming the client is not trusted (ie. not defined in **/etc/hosts.equiv** or **.rhosts**), and that the client's IP address passes any other security checks (eg. **tcpd** wrapper), the *server* will typically prompt for the user's *password*.

4. The client will respond by sending the user's password, which is sent in **clear text** which can be intercepted by anyone on the Internet.

5. If the password is correct, a shell is spawned.

6. The client monitors the standard output stream from the shell and considers itself "logged in" when it sees the expect sequence ***LOGIN_OK*** (typically a string like "Last login ..." – refer to *Four Js Windows Client User Guide*).

7. Initial commands, if defined, are then sent to the remote shell session to start the application. Typically this consists of setting the **FGLSERVER** environment variable and **fglrun** to run a BDL application.

8. The fglrun process examines the **FGLSERVER** (eg. **client:0**) environment variable and connects to port **6400** on the client. If the session offset is something other than **:0**, then this value would be added to the base port of **6400**. Hence if we had **FGLSERVER=client:10**, then the port used would be **6410**.

9. If this is the first application, an initialisation occurs (loading of **fgl2c.tcl**) and displays any initial content from the application.

Clearly in the above example, we are publicly exposing:

- The password
- All traffic between client and server
- A publicly accessible (and hackable) login port

SSH can be used to secure all data between the client and server and provides improved authentication to guard against unauthorised intrusion.

A ***tunnel*** is a logical network connection between two hosts, where one or more protocols are encapsulated within another. If the tunnelling protocol is SSH in this case, then we can encapsulate other protocols such as Four Js, FTP, smtp, etc. securely between the two hosts.

In SSH, tunnelling is implemented with by redirecting ports, also known as **port forwarding**. Thus it is possible to listen on a port on the remote host and redirect this through the tunnel to a port on the local host or vice versa. Depending on the direction, the tunnel may be referred to as **inbound** or **outbound**.

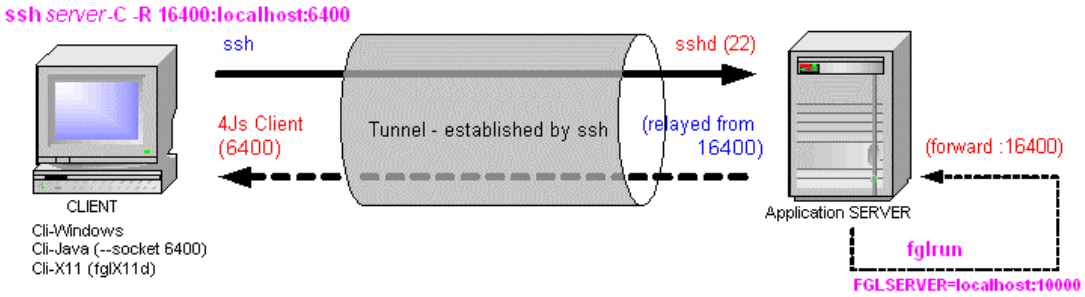
Of course this would normally impose an overhead, but SSH allows for *compression* which can be used to compensate.

Assuming we can make an SSH connection to the server from the client, we would need to establish an **inbound tunnel** from the server to the client on port **6400**. On the server, we can map an arbitrarily unused port, say **16400** to effectively be our proxy port to the client.

So from the application server's perspective, instead of setting **FGLSERVER=*client:0***, we would set this to **FGLSERVER=*localhost:10000***. When the application runs, it would add **10000** to base port **6400** and connect to port **16400** on the local server, which then redirects through the tunnel to port **6400** on the client.

This is how we can use SSH to login to the server and start an application.

SSH Tunnelling - Basic



1. The client starts the display server and listens on the default port **6400**.

2. SSH connects to **sshd** (port **22**) on the server for the nominated user.

3. After some protocol handshaking (negotiate protocols and ciphers, exchange of keys, random numbers, etc.) a secure connection is established.

4. The server will require some form of *authentication*. This can vary as SSH allows for several authentication schemes including *public key*, *passphrases* and *Kerberos*. The default and simplest is traditional password (which will be encrypted at this stage).

5. Passing authentication, a shell is spawned and an inbound tunnel is established mapping port **16400** on the server to port **6400** on the client.

6. If a command was specified, it is executed by the remote shell. The environment variable **FGLSERVER** is typically set to **localhost:10000**.

7. The **fglrun** process examines the **FGLSERVER** (**localhost:10000**) environment variable and connects to port **16400** on the server, which redirects through the tunnel to port **6400** on the client.

8. If this is the first application, an initialisation occurs (loading of **fgl2c.tcl**) and displays any initial content from the application.

NOTE: A unique and unused port number must be allocated for each secure client. For example,

User 1 FGLSERVER=localhost:10000
User 2 FGLSERVER=localhost:10001
User 3 FGLSERVER=localhost:10002
etc.

The policy for allocation of server port numbers can be automated (eg. counting number of Four Js locks) or manually allocated for each user (ie. each user is allocated a unique number).

7 Starting A Secure Application

The first step is to start an application manually.

1. Start your Windows, Java or X11 client

2. Login to the server

```
ssh -C -R 16400:localhost:6400 server -l user
```

what this does is:

```
-R 16400:localhost:6400
```

will establish a remote tunnel that listens on port **16400** on the remote server and redirects this through the tunnel to port **6400** on the **localhost**.

`-C` provides compression
`-l` user is the user to login as

3. Authenticate
Enter your password, passphrase, etc.

4. Set FGLSERVER
Once you are logged in, set your **FGLSERVER** environment variable:

```
FGLSERVER=localhost:10000 ; export FGLSERVER
```

5. Start your application
eg.

```
. $FGLDIR/envcomp
cd $FGLDIR/demo
fglrun ia
```

8 Launching A Secure Application

You can also pass an initial command to the server to launch your application. In this example, **start.sh** is a script that sets up the environment and starts your application.

```
ssh -C -R 16400:localhost:6400 server →
      "export FGLSERVER=localhost:10000 ; exec start.sh"
```

9 Performance

We have generally found that using SSH with compression did not adversely affect performance over the Internet. It was about the same as running the application unsecured.

10 Other Four Js Clients

You can also use SSH with other Four Js clients (in fact any other protocol). We have tested this with:

- Cli-ASCII (of course)
- Cli-Windows
- Cli-Java (in direct application mode - no web server)
- Cli-X11 client

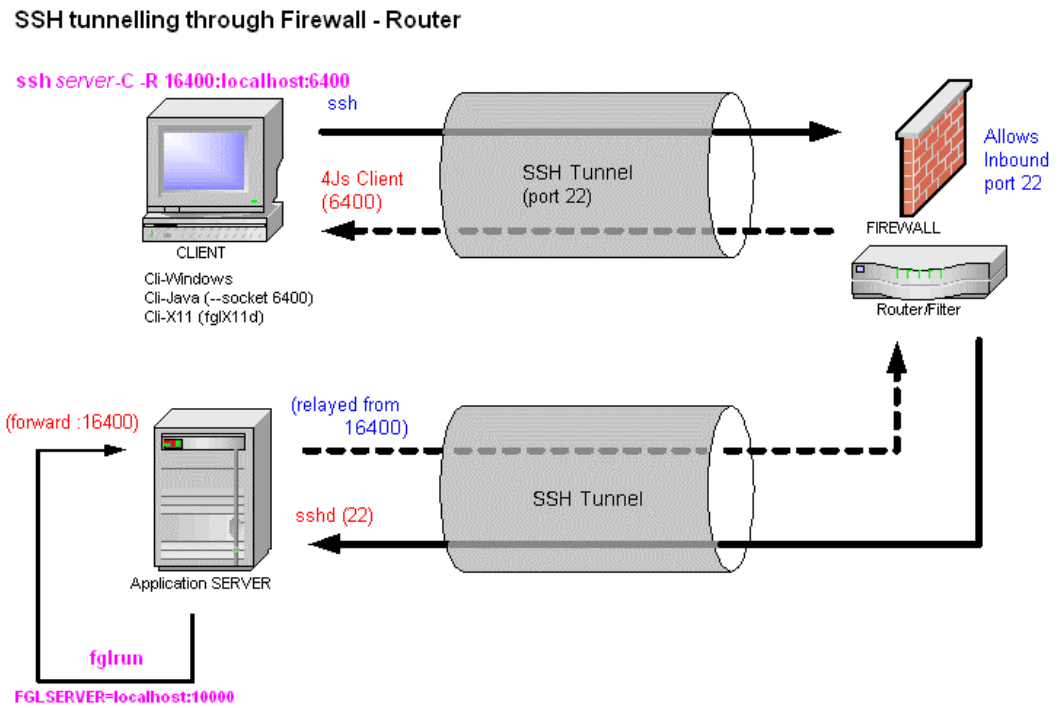
Note: Even though SSH supports X11 redirection, using a remote X display server is very slow. Ideally you should have the **fglX11d** daemon running on the client, as the Four Js protocol is much more efficient than X11 display protocol.

11 Firewalls

SSH with its port forwarding capability can be used to pass through firewalls of varying configurations. While it is not possible to cover the permutations, we will look at some typical firewall configurations. Although you may need to make some configuration changes to your firewall, with SSH, only port 22 is required as everything else can be tunnelled through the SSH connection.

A simple firewall consists of a router with packet filtering capabilities. The router needs to be configured to allow an inbound connection to port 22 to the application server.

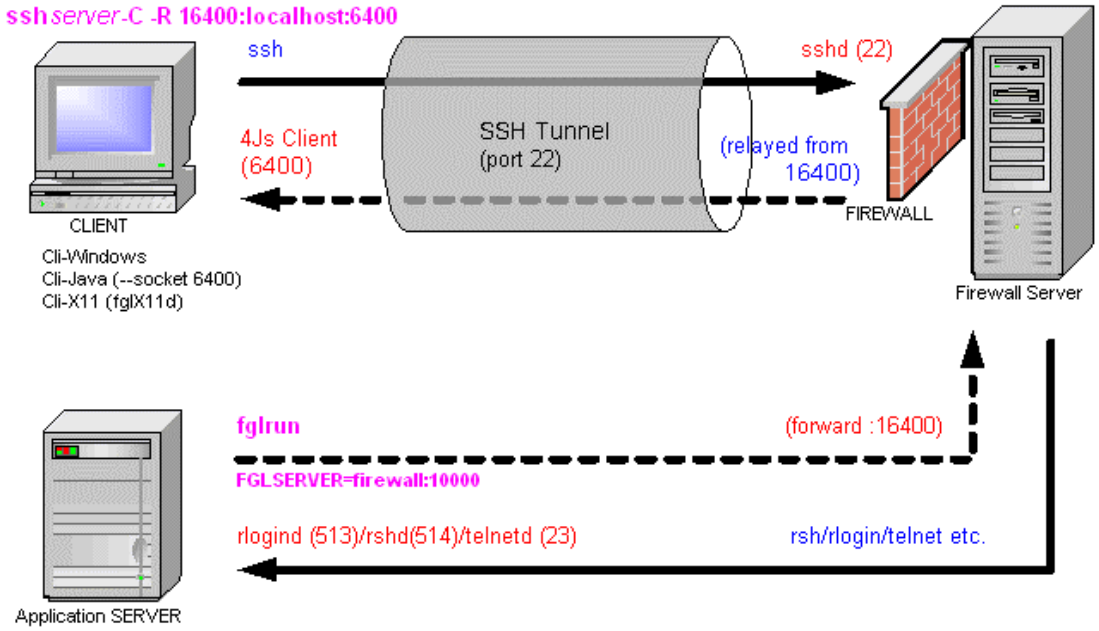
Note that IP addresses may be translated through the router, as long as we can connect to port 22 on the server.



Once we have the SSH connection to the server, the tunnel is established and the application processes (**fglrun**) can then connect back to the client on port **6400** by forwarding to port **16400** on **localhost**, which is then redirected through the tunnel to the client.

In the next configuration, the firewall is a server that has an **sshd** daemon running.

SSH tunnelling through Firewall (Server)



The client in this case connects with SSH to the firewall server and establishes a secure tunnel. After logging in to the firewall, the application can be remotely executed or is logged in to the application server to launch the application. **FGLSERVER** is set to ***firewall*:10000** in this case to redirect client traffic back through the tunnel.

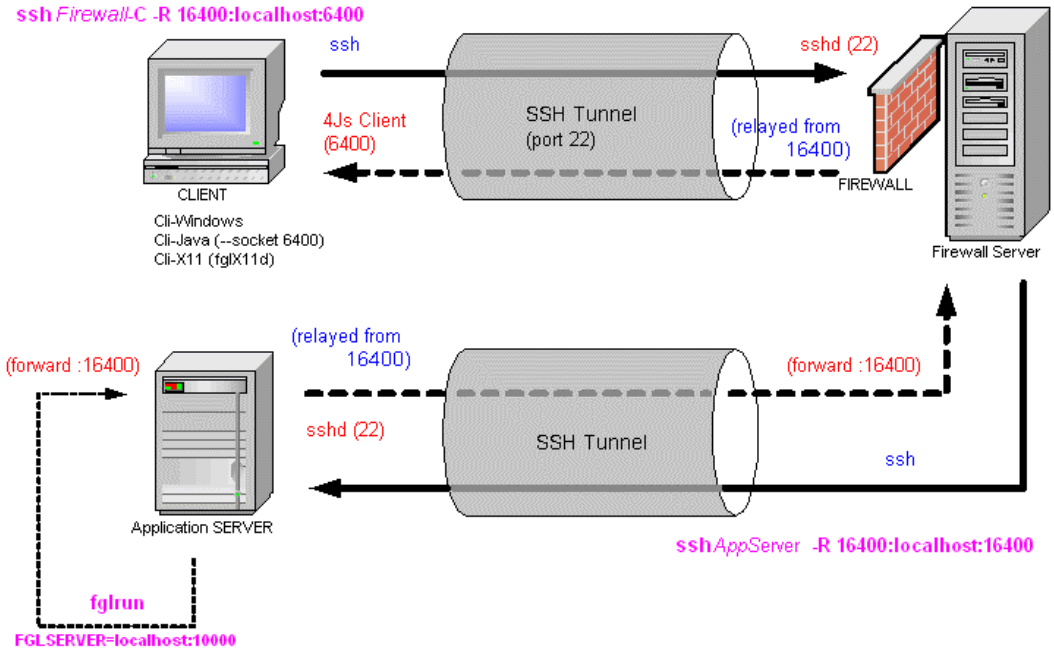
Note: By default, **sshd** on a *multi-homed* firewall server (ie. a server/router with more than one IP interface such as LAN card or PPP, but excluding localhost) will establish the listening port 16400 on **localhost**. This is a security measure to ensure that only applications on the firewall server can connect to 16400 and hence, the client. You can override this by setting **GatewayPorts** to **yes** in **/etc/ssh/sshd_config**. Depending on the configuration of your firewall server, you should probably **deny access** to this port 16400 from the public internet (typically through **ipchains** or **ipfwadm** if this is not done already) and only allow access from the **intranet**.

From a security perspective however, it is also not a good idea to allow remote execution (with trusted hosts, ie. without passwords) from a firewall server.

This configuration is also useful where *IP masquerading* is used or you only have one valid IP address (ie. the firewall).

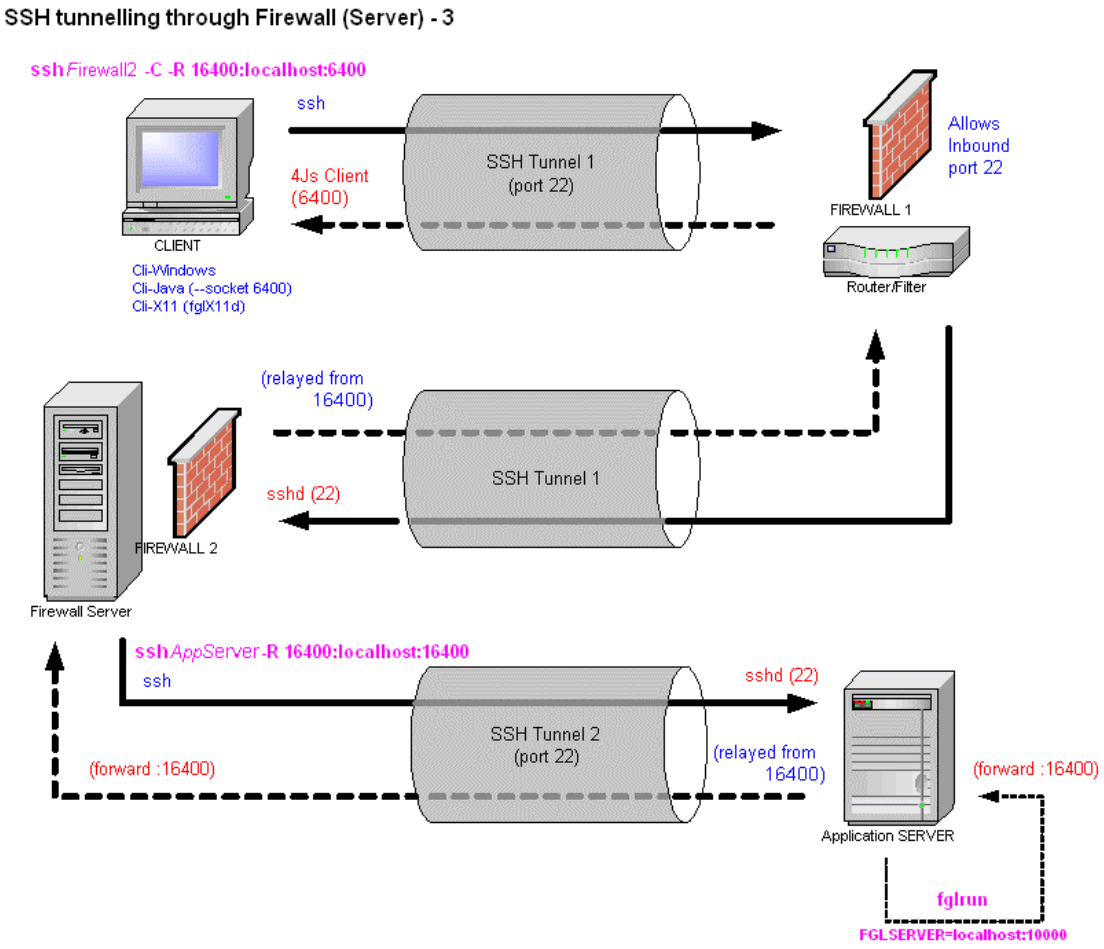
The next configuration sets up two tunnels, the first to the firewall, and the second between the firewall and the application server.

SSH tunnelling through Firewall (Server) - 2



This provides a better security by keeping the traffic encrypted all the way to the application server. Unlike the previous configuration, **GatewayPorts** can remain set to **no**. Note that the second **ssh** from the fireall to the application server uses port forward mapping of **16400** on the *application server* to **16400** on the *firewall*. Also note that no *compression* is required if running on the intranet as this is assumed to be at LAN speed.

The last configuration consists of two firewalls with a possible DMZ between the two firewalls.



There are many other possible firewall configurations, but with SSH's port forwarding capabilities, you should be able to create a configuration that will allow you to work through a firewall securely.

12 Limitations

12.1 Remote Shell Daemon on Windows client (WTK and Java)

The Windows client provides an optional **rshd** daemon that can be used to transfer files between client and server as well as to remotely execute client side applications. Although there is an **sshd** daemon that could be used, the ones we found only seem to work on Windows NT/2000. We have not checked for any other **sshd** ports to Windows at this stage (although, the source is there ...).

If you rely on this, you may need to re-engineer some part of your application to initiate file copies, etc from the client. On the other hand, from a security perspective, it is probably better to have a cogniscent user pull files from the server (ie. the user is always aware of when files are added or modified on their system).