SSH2

Updated Apr 25, 2012 by mkobetic

This is an experimental implementation of SSH2 protocol.

Here's how to use it as a server:

```
        | listener socket server configuration |
        "This is just to set up a TCP socket connection, nothing to do with SSH2"
        listener := SocketAccessor family: SocketAccessor AF_INET type: SocketAccessor SOCK_STREAM.
        listener soReuseaddr: true.
        listener bindTo: (IPSocketAddress hostAddress: IPSocketAddress thisHost port: 2222).
        [ socket := listener listenFor: 1; accept ] ensure: [ listener close ].
        "Now we have a socket and can set up an SSH2 connection on it, here playing the server side"
        configuration := SSH2Configuration new keys: SSH2KeysTest sampleKeys.
        server := configuration newServerConnectionOn: socket.
        "This is just to have all SSH messages echoed to transcript"
        server when: SSH2Announcement do: [ :m | Transcript cr; print: m ].
"       server when: SSH2TransportMessage, SSH2ChannelSetupMessage, CHANNEL_CLOSE do: [ :m | Transcript cr; print: m ].
"       [       "Server normally doesn't do much beyond accepting the client handshake and then waiting for a disconnec
                Everything is initiated by the client side and handled by background threads handling any established c
                server accept; waitForDisconnect
        ] ensure: [ server close. socket close. configuration release ]
```

after starting it, hit it with an ssh client as follows:

```
        ssh -p 2222 localhost 3 + 4
```

to evaluate single smalltalk expression, or

```
        scp -P 2222 <filename> localhost:
```

to upload a file (will be saved in the image directory), or just

```
        ssh -p 2222 localhost
```

to run a simple REPL loop in the smalltalk image (use Ctrl-D to end it, backspace should work too).

You'll have to rerun the code above after each ssh interaction, as it handles only single connection (as written above), and each ssh invocation will require a new connection.

If you have local sshd running, and are able to use it from your current account using the ssh/scp commands, the following will connect to it as a client.

```
        | home user keys socket client keys configuration |
        "We need to load your personal keys from your $HOME/.ssh directory.
        We need valid keys to successfully authenticate with the server."
        home := '$(HOME)' asLogicalFileSpecification asFilename.
        user := home tail.
        keys := SSH2Keys fromUser: home.
        "Create a socket"
        socket := SocketAccessor newTCPclientToHost: 'localhost' port: 22.
        "Set up an SSH client connection on it."
        configuration := SSH2Configuration new keys: keys.
        client := configuration newClientConnectionOn: socket.
        "This is just so that all SSH messages are echoed into the Transcript"
        client when: SSH2Announcement do: [ :m | Transcript cr; print: m ].
"       client when: SSH2TransportMessage, SSH2ChannelSetupMessage, CHANNEL_CLOSE do: [ :m | Transcript cr; print: m ].
"       [       "A client has to connect as particular user (using the preconfigured keys)"
                client connect: user.
                "A channel service can provide an interactive session or a tunnel.
                You can ask for as many sessions, tunnels as you want, each will get its own channel multiplexed over th
                "You can ask for as many sessions, tunnels as you want, each will get its own channel multiplexed over a
                session := client session.
                "Given a session you can execute a command, or upload/download a file or directory, etc..."
                [       session exec: 'ls -l'.
                ] ensure: [ session close ].
 "              [       [ session put: 'visual.im visual.cha' to: '/dev/shm/' ] timeToRun
                ] ensure: [ session close ]
 "      ] ensure: [ client close. socket close. configuration release ]
```

It is also possible to set environment variables, invoke subsystems and get the exitStatus after command execution.

Tunneling is also available. A client can open a tunnel to a server using #tunnelTo:port:

```
        tunnel := client tunnelTo: 'localhost' port: 4444
```

Note that, unlike the command line ssh clients, this doesn't set up a local listening socket. Instead the tunnel is used as an internal stream source/destination that can be used inside the image via the usual #reading/#writing messages. Like sessions, tunnels should be closed with #close message. On the server side the remote end of the tunnel is also just a stream source/destination, no sockets are being connected. The port argument above only applies if the server is not an Xtreams based server. For that reason there's also a short version of the method #tunnelTo: as well. However something has to be waiting for the incoming tunnel on the server side, otherwise it will be rejected. Synchronization with the waiting process is done via named tunnel queues, and the queue name is the argument of the tunnelTo: message. If the queue with given name exists the tunnel request will be queued up in it, otherwise it will be rejected. The waiting process simply waits for the next tunnel request coming from the queue. The queues are managed with #queueAt:, #openQueueAt: and #closeQueueAt: messages. Here's a sample showing both sides.

```
        "server side"
        queue := connection openQueueAt: 'test'.
        [       tunnel := queue next.
                (tunnel writing encoding: #ascii) write: 'Hello'; close
        ] ensure: [ connection closeQueueAt: 'test' ]

        "client side"
        tunnel := connection tunnelTo: 'test'.
        [ tunnel reading rest ] ensure: [ tunnel close ]
```

The queue model allows establishing multiple tunnels to the same target (concurrently or sequentially) as long as something keeps watching the queue. The tunnels described so far are initiated strictly from the client side (called "local" or "direct" tunnels in SSH). SSH also provides the notion of "reverse" or "forwarded" tunnels, where the client first requests forwarding of a particular queue from the server and then processes on the server side can initiate tunnels from the remote side. Conversely, something has to watch the local queue on the client side and handle the incoming tunnels.

```
        "client side"
        queue := connection openQueueAt: 'test'.
        [       tunnel := queue next.
                (tunnel writing encoding: #ascii) write: 'Hello'; close
        ] ensure: [ connection closeQueueAt: 'test' ]

        "server side"
        tunnel := connection tunnelTo: 'test'.
        [ tunnel reading rest ] ensure: [ tunnel close ]
```

The pattern looks exactly the same as in the previous case except the roles are reversed, the client sets up a queue and the server requests a tunnel.

Setting up a reverse tunnel queue with normal SSH server requires sending address and port where there address is meant to identify which network interfaces should the server listen on and port identifies the listening port number on the server side. The usual values used for address would be an empty string (any interface), 'localhost' (any loopback interface), etc.

```
                queue := client openQueueAt: '' port: 4444.
                [       tunnel := queue next.
                        [ tunnel reading rest ] ensure: [ tunnel close ]
                ] ensure: [ client closeQueue: queue ]
```

DONE

- Transport Protocol (RFC 4253)
  - basic transport seems to work including encryption, macs and compression (compression not tested with OpenSSH yet)
  - the fixed DH key exchange works as well
- Authentication Protocol (RFC 4252)
  - publickey authentication
  - password authentication
- Connection Protocol (RFC 4254)

  - channel/session opening
  - channel closing
  - data transfer
  - channel requests
  - command invocation (exec)
  - shell invocation (shell)
  - variable passing (env)
  - exit-status support (SSH2Session>>exitStatus)
  - extended data reception (SSH2Channel>>stderr:)
  - port forwarding (client - tunnelTo:(port:); server - open|closeQueueAt:)
  - reverse port forwarding (client - accept|cancelQueueAt:; server - tunnelTo:(port:)
- SCP
  - basic file and directory upload/download works (in either direction) and tested against OpenSSH
  - multiple source arguments are fully supported
  - sending/receiving APIs allow plugging in custom behavior for streaming to/from arbitrary streams, not just files
- Public Key Fingerprints (RFC 4716)
  - SSH2HostKey>>fingerprint

TODO

- Transport Protocol (RFC 4253)
  - key re-exchange
  - need more connection failure tests
  - fill in more of the cryptographic algorithm mappings
- Authentication Protocol (RFC 4252)
  - partial success
- Connection Protocol (RFC 4254)
  - extended data sending
  - signal passing (signal)
  - exit signal passing (exit-signal)
- SCP
  - atime/mtime is ignored and not sent (-p)
  - -r is currently implied in the default behavior unless overridden by a custom one

LIMITATIONS

- Transport Protocol (RFC 4253)
  - no guessed key support in key exchange (KEXINIT.first_kex_packet_follow = TRUE)
  - np pgp key support
- Authentication Protocol (RFC 4252)
  - probably won't do hostbased authentication
  - authentication specific messages in general, but SSH_MSG_USERAUTH_PASSWD_CHANGEREQ specifically
- Connection Protocol (RFC 4254)
  - no support for auto-assigned reverse forwarded ports (tcpip-forward port: 0); requires contextual parsing of the confirmation
  - won't do X11 forwarding
  - ignoring terminal requests (window-change, xon-xoff)

ISSUES

!!!! USER AND HOST NOT VALIDATED !!! in the default setup. The keys and signatures are all validated, but the client needs to check that a host key that checked out belongs to the server it's trying to connect and the server needs to verify that the user key that checked out belongs to the user that's trying to connect. For that one has to set the SSH2ClientConnection>>#keyValidator: and on the server side either SSH2ServerConnection>>#keyValidator: (for 'publickey' authentication) or SSH2ServerConnection>>#passwordValidator: (for 'password' authentication) accordingly. When these are not set, the code will currently skip the check, which effectively means the check passes. Yes that is bad for security, and I wouldn't ship an official release like that, but at this point I'd rather not raise too many entry barriers for people who just want to play with this. We're not pretending that this is ready to be deployed. A reasonable client side default is to attempt to read the $HOME/.ssh/known_hosts file and match the host key to that. The server side is tricky though since servers normally run in sufficiently privileged mode and read the $HOME/.ssh/authorized_keys file of the connecting user. That's why there are the pluggable validator blocks, which gives a lot of flexibility, but they must be configured accordingly.

Releasing external configuration resources: This isn't really an issue, just something to keep in mind. SSH2Configuration (class)>>random/keys, may hold onto external resources and may therefore require releasing them when not needed or re-acquiring them (e.g. on image startup if the global class side parameters are employed). The SSH2Configuration (class)>>release methods should be useful for that, but it's the application responsibility to decide when they should be called.

Enter a comment: Hint: You can use Wiki Syntax.

Submit