SSH Server Technical Reference

Interniche Legacy Document

Version 1.00

Date: 11-May-2017 15:39

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

Table of Contents

- Introduction 4
  - System Requirements 4
  - Terminology 5
  - Terms and Conventions 6
- Product Overview 7
- SSH Security Concepts 8
  - The Public/Private Key Pair 8
  - Digital Signatures 8
  - Known Hosts File 9
  - Diffie-Hellman Key Exchange 9
  - Client Authentication 9
- SSH Architecture and Protocols 10
  - The SSH Communication Sequence 10
  - Event Sequence of an SSH Communication 11
  - Transport Layer 11
  - Authentication Layer 12
  - Connection Layer 12
  - Implemented for Embedded Devices 14
- Key Management 15
  - Managing the Host Key 15
  - Managing User Keys 15
  - Generating PEM files for use with userutil 16
    - Private-key Authentication. 16
    - Public-key Authentication 17
  - Managing the USERINFO file 18
- SCP - Secure Copy 19
  - Example: Copying a file to the server 19
  - Example: Copying a file from the server 19
- Building the SSH Server 20
  - Source file list 21
  - Init-time Configuration 22
    - Tunable SSH Parameters 22
      - Global variables modifiable at run time 22
    - Memory Allocation 24
  - Cipher Support 25
  - Basic and Full Implementation feature sets 25
- SSH Menu Commands 26
  - ssh cfgfwds 27
  - ssh config 28
  - ssh memstat 29
  - ssh netstat 30
  - ssh nosecurity 31

Usage Examples 32
- Local vs. Remote Port Forwarding 32
- Understanding the Diagrams 34
  - Local vs. Remote Port Forwarding 34
  - Security for Off-Host Connections 34
- Usage examples using Putty 35
  - General Putty Configuration 35
    - Session top screen 35
    - Session Logging screen 35
    - Connection top screen 35
    - Connection Data Screen 35
    - Connection SSH screen 35
    - Connection SSH Auth screen 36
  - Direct Commands 36
  - Port Forwarding 36

Testing Examples 37
- Testing an internal command over SSH using putty 37
- Testing SCP SOURCE using pscp 37
- Testing SCP SINK using pscp 37
- Local Port Forwarding of Telnet 38
  - Addresses in Examples 38
  - Port Numbers in Examples 38
    - SSH->Connection->Tunnels Screen 39
  - Off-Host Local Port Forwarding 39
- Remote Port Forwarding of Telnet 40
  - SSH->Connection->Tunnels Screen 40
- Port Forwarding Between NicheStacks 41
  - Local Port Forwarding between NicheStacks 41
    - SSH->Connection->Tunnels Screen 41
  - Remote Port Forwarding between NicheStacks 42
    - SSH->Connection->Tunnels Screen 42

1 Introduction

This technical reference is provided with the InterNiche SSH server. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can understand the architecture of the SSH Server and each of its modules. A conceptual understanding of what SSH does and familiarity with networking concepts is also helpful.

If you will be using InterNiche SSH Server with an InterNiche TCP/IP stack and OS, and are targeting a hardware platform supported by InterNiche, then there is very little work involved. You simply need to select which SSH Server options you want in you product (see Building the SSH Server) and compile to get the SSH Server working on your development target.

This document also covers the basics of moving the InterNiche SSH Server to a non-InterNiche IP stack. However this is considerably more complex than using InterNiche's IP. The porting process can take anywhere from a few hours to several days.

1.1 System Requirements

- TCP/IP implementation supporting BSD Sockets
- Dedicated encryption hardware or application processor with sufficient power to encrypt and decrypt data using 3DES and AES128
- Development environment supporting ANSI-C

1.2 Terminology

Term	Definition
Blob	Binary Large Object, a term used for the binary representation of cryptographic key, digest, signature, etc.
Client	The side of a TCP connection which makes the Connection Request. It may refer to an SSH Client, a client application or both. The specific meaning is determined by the SSH Client. Common SSH Clients are "putty.exe" and "/usr/bin/ssh"
Forward TCPIP	Sometimes used for Remote Port Forwarding. Once Remote Port Forwarding is established, the SSH Server forwards TCP/IP connection requests and data back to the SSH client
KEX	Abbreviation for Key Exchange, as in KEXINIT: key exchange initialization
Local Port Forwarding	From the perspective of the SSH Client, the Application Client connects to SSH using a port which is LOCAL to the SSH Client's IP address. With Local Port Forwarding the SSH Client listens for connection requests on a user specified port that is local to its IP address and forwards those requests across SSH to a remote application server.
Off Host	An application that connects to an SSH Server or Client that has a different IP address
PEM	A format used for representing security certificates and certificate requests.
Port Forwarding	The routing of data belonging to another application through an SSH tunnel. Executing a menu command on a server is not port forwarding.
Putty	PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform. It is open source software that is available with source code and is developed and supported by a group of volunteers.
Remote Port Forwarding	From the perspective of the SSH Client, the Application Client connects to SSH using a port at the server's IP address. With Remote Port Forwarding the SSH Client instructs the SSH Server to listen on one of its ports and forward any connection requests back across SSH to the SSH Client, which then forwards it to an application server on its side of the SSH tunnel.
SCP	Secure Copy program. An integrated service in the iNiche Server for transferring files using syntax similar to the Unix cp command.
Server	The side of a TCP connection which LISTENS for a connection request. This may refer to the SSH Server, an Application Server or both. The specific meaning is determined by an SSH Client.
SSH	Secure Shell, version 2.0 as specified in: • RFC 4251 - The Secure Shell (SSH) Protocol Architecture • RFC 4252 - The Secure Shell (SSH) Authentication Protocol • RFC 4253 - The Secure Shell (SSH) Transport Layer Protocol • RFC 4254 - The Secure Shell (SSH) Connection Protocol

1.3 Terms and Conventions

In this document, the term "SSH", when used without other qualification, means version 2.0 of the SSH protocol code as ported to an embedded system. "System" refers to your embedded target system. A "user" or "porting engineer" usually refers to the engineer who is porting the the InterNiche SSH Server. An "end user" refers to the person who ultimately ends up using the "user's" product.

Names of files, C structures and C routines are displayed as follows: c_routine()

Source from C programs is displayed in these boxes:

```
/* C source file - yet another hello program. */
main(){
   printf("hello world.\n");
}
```

2 Product Overview

InterNiche SSH is a portable, low footprint server that runs as an application on InterNiche's IPv4 and/or IPv6 stacks and includes the Authentication, Transport Layer and SSH Connection Layer protocols. The protocol layers exist simultaneously with each layer supporting multiple simultaneous sessions.

The SSH protocol has a client/server architecture. An SSH server program, typically installed and run by a system administrator, accepts or rejects incoming connections to its host computer. Users then run SSH client programs, typically on other computers, to make requests of the SSH server, such as "Please log me in," "Please send me a file," or "Please execute this command." All communications between clients and servers are securely encrypted and protected from modification.

Although SSH stands for Secure Shell, it is not a true shell in the sense of the Unix Bourne shell and C shell. It is not a command interpreter, nor does it provide wildcard expansion, command history, and so forth. Rather, InterNiche's SSH Server creates a secure socket connection which can be used for executing menu commands or for tunneling data between the clients and servers of other applications.

InterNiche's SSH implements the server-side of the protocol suite, allowing the data transmission between two network devices across a secure connection, with each connection capable of supporting multiple channels. The various encryption methods used in SSH protocol implementation gives confidentiality and integrity of data over an insecure network.

InterNiche's SSH Server implements version "2.0" of the protocol. It supports tunneling, forwarded and direct TCP/IP. It normally listens for connections on port 22, which has been assigned for its use by the IANA. The SSH client and SSH server negotiate as which security protocols they will run, selecting from lists of protocols that SSH protocol specifications consider as either required, recommended, or optional.

Security policies are set up by administrators either via compile-time or dynamic parameters. SSH is largely (but not quite) transparent to applications using it. Most protocols and network applications programs can run over SSH without change, though the port numbers and IP addresses used in the command may be different from normal. Programs which change port-numbers during their operation will not run over SSH without modification.

3 SSH Security Concepts

If a group of sites that wish to communicate share a private key, they can use this to secure their communications. However, securely distributing a private key to every site with which you wish to communicate is often difficult. Also, a key loses security when it is widely used over a period of time, and redistributing new keys on a periodic basis is even more difficult.

SSH uses a combination of security mechanisms to overcome these problems.

3.1 The Public/Private Key Pair

A public key is derived from a private key. Site "A" has a private key that it keeps secret, and a public key that it can freely distribute. The first property of a public/private key pair is that if a message is encrypted with the public key, then only someone with the private key can decrypt it. Remote site "B" can use A's public key to encrypt messages it sends to A. Only someone with A's private key can decrypt the message. If B also has a private key and gives its public key to A, then A can use B's public key to encrypt messages that only B can read.

3.2 Digital Signatures

There is still a problem with exchanging keys over the web. When B receives a public key, how can it be sure that the key really came from A?

The second property of a public/private key pair is the reverse of the first property. Only someone with the private key can encrypt messages that can be decrypted with its public key. Each SSH Server has a unique Host key based on a public/private key pair. If a client can use the Server's public key to successfully decrypt identity information sent by the server, it can be sure that only someone with the server's private key could have encrypted the message.

However, if the protocol were this simple, then once a third party saw an encrypted identity message sent by the server, it could simply duplicate that message and masquerade as the server. To avoid this problem, the SSH key exchange starts with the client sending a challenge to the server to prove its identity. The challenge contains information that can not be predicted in advance and that is unique to that challenge. This information is used as the SessionID for all future messages in this session. In the server's response, it must use its private key to encrypt the session ID as well as its identity information. This prevents a third party from simply duplicating a previous response from the server.

3.3 Known Hosts File

An SSH client has a Known Hosts file that contains the public keys of all SSH servers that it knows about. A server's response to the clients challenge includes its public key, and this should match the key in the client's Known Hosts File. The SSH specification does not define how the public keys are put in this file. Ideally, before their first communication, the client should have received the Server's public key by some secure administrative means. In practice, this is often not the case, and a client user may simply accept the public key supplied by a server the first time they communicate. This new key is added to the Known Hosts file, and from then on, the public key supplied by the server should match the one in the file. If no one was masquerading as the server the first time, then this method will work for all future communications. Obviously, this is a security risk.

3.4 Diffie-Hellman Key Exchange

The unfortunate third property of the public key mechanism is it is extremely computer-intensive. It is simply not practical to use a public/private key pair to encrypt anything more than a few hundred bytes. In SSH, public keys are used to exchange signatures and to determine a mechanism for creating the private keys that will be used to encrypt the data messages for the session. Once the server's identity has been proven, the Diffie-Hellman Key Exchange is used to exchange a shared master Secret and a mechanism that both sides will use to generate an identical set of ephemeral keys to be used to encrypt the session's data messages.

3.5 Client Authentication

Normally, after key exchange, the server will require the users on the client to be authenticated before they have permission to do anything on the server. Authentication can be done with a password, a key blob, or both. Typically, the client will pass the user's name and public key to the server. The server matches this with the user's entry in its user file. Note that in SSH, typically before the two sides connect, both sides should already have the other's public key in a table. The distribution of public keys is much safer and easier than the distribution of private keys.

4 SSH Architecture and Protocols

The SSH-2 protocol is implemented in three distinct layers:

Layer	Description
Transport	This is a secure, low level transport protocol layer. This layer negotiates security protocols and performs both the initial and re-key exchanges. It sits "between" the SSH Authentication and TCP protocols, enabling the higher layers to send data securely with the negotiated algorithms. It provides a single, secure, full-duplex stream between the SSH client and an authenticated SSH Server. The Transport Layer authenticates the server to the client.
Authentication	This layer authenticates the SSH client to the SSH server and relies upon the Transport layer to provide integrity and confidentiality protection. It implements a number of techniques used for client authentication (public key, password, and so on). Individual mechanisms specified in the authentication protocol use the session id provided by the transport protocol and depend on the security and integrity guarantees of the transport protocol.
Connection	This layer establishes and maintains application connections. It is the "highest" layer in the SSH stack and provides for remote execution of commands and forwarded TCP/IP connections. It also implements "channels", which allow a single SSH connection to host multiple channels simultaneously, each transferring data in both directions.

4.1 The SSH Communication Sequence

The following sequence of events helps protect the integrity of SSH communication between two hosts:

1. A cryptographic handshake is performed so that the client can verify that it is communicating with the correct server.
2. After the initial negotiation, the Transport layer encrypts all messages on the connection and it uses a Message Authentication Code (MAC) to ensure message integrity.
3. The client authenticates itself to the server.
4. The remote client interacts with the remote host over the encrypted connection.

4.2 Event Sequence of an SSH Communication

The following series of events help protect the integrity of SSH communication between two hosts.

1. A cryptographic handshake is made so that the client can verify that it is communicating with the correct server.
2. After the initial negotiation, the transport layer encrypts all messages on the connection and it uses a Message Authentication algorithm (MAC) to ensure message integrity
3. The client authenticates itself to the server.
4. The remote client interacts with the remote host over the encrypted connection.

4.3 Transport Layer

The primary role of the transport layer is to facilitate safe and secure communication between the two hosts at the time of authentication and during subsequent communication. It accomplishes this by handling the encryption and decryption of data and by providing integrity protection of data packets as they are sent and received.

Once an SSH client contacts a server, key information is exchanged so that the two systems can correctly construct the transport layer. The following steps occur during this exchange:

- Keys are exchanged.
- The public key encryption algorithm is determined.
- The symmetric encryption algorithm is determined.
- The message authentication algorithm is determined.
- The hash algorithm is determined.

During the key exchange, the server identifies itself to the client with a unique host key. If the client has never communicated with this particular server before, the server's host key is unknown to the client and it does not immediately connect. Instead, the user is notified and asked to verify the new host key. The user must accept the key or the connection is rejected. In subsequent connections, the server's host key is checked against the saved version on the client, providing confidence that the client is indeed communicating with the intended server. If, in the future, the host key no longer matches, the user must remove the client's saved version before a connection can occur.

Warning:

An attacker can masquerade as an SSH server during the initial contact because the local system does not know the difference between the intended server and a false one set up by an attacker. To help prevent this, verify the integrity of a new SSH server by contacting the server administrator before connecting for the first time, or in the event of a host key mismatch.

SSH is designed to work with almost any kind of public key algorithm or encoding format. After an initial key exchange creates a hash value used for exchanges and a shared secret value, the two systems immediately begin calculating new keys and algorithms to protect authentication and future data sent over the connection.

After a certain amount of data has been transmitted using a given key and algorithm (the exact amount depends on the SSH implementation), another key exchange occurs, generating another set of hash values and a new shared secret value. Even if an attacker is able to determine the hash and shared secret value, this information is only useful for a limited period of time.

4.4 Authentication Layer

Once the transport layer has constructed a secure tunnel to pass information between the two systems, the server informs the client of the authentication methods it supports, such as using a private key-encoded signature or entering a password. The client then tries to authenticate itself to the server using one of these supported methods.

SSH servers and clients can be configured to allow different types of authentication, giving each side the optimal amount of control. The server can decide which encryption methods it supports based on its security model, and the client can choose which order to use the available authentication methods in.

4.5 Connection Layer

After a successful authentication over the SSH transport layer, multiple channels may be opened by using multiplexing, with each channel handling communication for different interactive or forwarded sessions.

Both clients and servers can create a new channel. Each channel is assigned a different number at each end of the connection. When the client attempts to open a new channel, the client sends the channel number along with the request. This information is stored by the server and is used to direct communication to that channel. This is done so that different types of session do not affect one another and so that when a given session ends, its channel can be closed without disrupting the primary SSH connection.

Channels also support flow control, allowing them to send and receive data in an orderly fashion.

The client and server negotiate the characteristics of each channel automatically, depending on the type of service the client requests and the way the user is connected to the network. This gives great flexibility in handling different types of remote connection without having to change the basic infrastructure of the protocol.

Note: When an application such as Telnet is forwarded over a secure SSH channel, it is unaware of the authentication used to create that channel. For this reason the application often requires its own username and password.

4.6 Implemented for Embedded Devices

To keep the code-size to a minimum InterNiche SSH differs from many non-embedded implementations in that it does not implement PTYs nor X-11 forwarding for fully-interactive user sessions. Instead, it supports ability to pass single command strings (and the associated output) directly to the menu system. Similarly, instead of supporting an array of services our implementation implements a single service (SCP) and focuses on port forwarding to route secure traffic between network applications.

For detailed examples of port forwarding, see Usage examples using Putty.

5 Key Management

5.1 Managing the Host Key

The SSH Server requires DSA and RSA hostkey files in order to provide host authentication with the Key Exchange. These files will be created at an administrative site. They must be included with the NicheStack installation, but they can be replaced via SCP.

If USE_FULL_IMPL is defined, then NicheStack expects two standard PEM files to be located in the current directory: RSAhostkey.pem and DSAhostkey.PEM. In the minimum implementation, only DSAhostkey.pem is required. The files are created using the following OpenSSL commands:

```
openssl genrsa -out RSAhostkey.pem 1024
openssl dsaparam -out dsaparam.pem 1024
openssl gendsa -out DSAhostkey.pem dsaparam.pem
```

5.2 Managing User Keys

The "`userinfo`" file contains authorization information for each user who is to be given access via the SSH server. Besides the user's name, each entry in userinfo may contain a password and/or either a public or a private key blob. A provided desktop utility, "`userutil`", must be used to add or delete entries in userinfo. The file `userinfodefs.h` contains userinfo definitions that are used by both userutil and the SSH Server. Both projects must reference the same file or an identical copy of the file.

Normally, the userinfo file will be created and maintained at an administrative location. The initial version can be included with the rest of the NicheStack software at installation time. Changes to userinfo could be made by changing it at an administrative location and then using scp to replace it, along with the associated outkeyfiles, on the field system(s).

Userinfo has one entry for each username in the following format:

```
USERNAME,PASSWORD,OUTKEYFILE
```

The `username` is required. The other fields may be empty. Fields within the entry are ASCII strings delimited by a comma. Each entry ends with a newline. Note that the two commas which delimit the fields are required even if the password or keyfile fields are empty.

An `outkeyfile` is a binary file created by userutil from a standard SSL public/private key file (a PEM file) that was specified as input by the "`userutil -a`" command. The `outkeyfile` contains information extracted from the input file and formatted in a manner that enables the iNiche SSH Server to read and use the information without having to make SSL library calls. Each unique `outkeyfile` entry is a reference to a separate file within the file system that is readable by NicheStack. All outkeyfiles referenced by userinfo should be loaded into the NicheStack file system along with the userinfo file.

When a SSH client requests a connection with a NicheStack system, the SSH Server's Authentication Layer will read the userinfo file searching for an entry associated with the username. If the username is not found, the request will be denied. Otherwise, if the entry contains the name of a keyblob file, the file will be read and used to validate the user. If a password is required, it will send a response requesting the password.

A typical SSH client provides only the username with its first authorization request and **it is perfectly normal for the SSH server to send one or two authorization failure messages** before the SSH client provides all of the required user authentication data.

InterNiche has been designed so that the porting engineer can provide routines to encrypt any or all of the fields within the userinfo file. Before the userutil writes a field into the file, it calls one of the "xxxENCRYPT" macros, and whenever the SSH Server reads a field from the userinfo file, it calls one of the "xxxDECRYPT" macros. These macros are defined in h/userfiledefs.h.

As shipped, the encrypt macros all call the stub function ufield_encrypt() and the decrypt macros all call the stub function ufield_decrypt(). If "UFILE_ENCRYPT" is not defined, that all of the macros will define to nothing. If "UFILE_ENCRYPT" is defined, then for testing purposes, the functions will simply copy the input buffer to the output buffer. It is up to the porting engineer to provide actual encryption and decryption functions.

Note: Functions that read and write the userinfo file expect the fields to be comma separated ASCII strings. When encryption is used, the final step should be to convert the encrypted output to Base64.

5.3 Generating PEM files for use with userutil

When keys are used, one input to userutil is a public or private key in the standard PEM file format. These input PEM files can be generated with any of the standard SSL programs: OpenSSL on Windows (genrsa or gendsa), Putty (puttygen), linux/unix (ssh-keygen), etc.

Private-key Authentication.

Note: Putty is designed to generate keys only for public key authentication. However, Puttygen can be told to load a private key generated by another SSL program

The following is an example of how to generate a privite-key PEM file using OpenSSL on Windows. It assumes that the commands are in your command search path:

RSA (1 step):

```
openssl genrsa -out privkey.pem 1024
```

DSA (2 steps)

```
openssl dsaparam -out dsaparam.pem 1024
openssl gendsa -out privkey.pem dsaparam.pem
```

The following is an example of how to load a private-key PEM file into a PuTTY client:

1. Type puttygen
2. Select LOAD (private key)
3. Browse to and open the saved private key file
4. Click on Save private key.

Public-key Authentication

For public-key authentication, the SSH client must generate the public key PEM file that will be used as input for userutil.

The following is an example of how to obtain a public key from PuTTY (version 0.60):

1. Type `puttygen`
2. In the "Parameter" section, select the type of key to be generated, either "SSH-2RSA" or "SSH-2 DSA".
3. Click on the box labeled `generate`.
4. Click on the `Save private key` button in a directory where it can be used by PuTTY
5. In the "Conversions" drop-down menu, select `export OpenSSH Key`, and select the directory that contains the userutil.
6. In the directory with `userutil`, type:

   ```
   openssl { rsa | dsa } -pubout -in <privkey.pem> -out <pubkey.pem>
   ```

7. The `pubkey.pem` file is now ready to be used by `userutil.exe`. The `privkey.pem` file may be discarded. Of course, PuTTY needs the `<privatekey>.ppk` file saved in Step 4.

5.4 Managing the USERINFO file

Name

`userutil - a desktop utility`

Syntax

```
userutil -a username [-p password] [-k inkeyfile ]userutil -d username [-f]
userutil -v [ -u username ]
```

Description

Add, delete or display `userinfo` information.

Parameters

-a	Add `username` to the `userinfo` file.
-p	Optional password to be associated `username`
-k	Optional input key file to be associated `username`
-d	Delete the `userinfo` entry for `username`.
-f	Delete the outkeyfile associated with `username`.
-v	Dump the `userinfo` file.
-u	Restrict the `-v` operation to `username`.

Notes:

- The first two characters of the name of the input key file are used to determine whether the file contains a public or private key.
  - If the file contains a public key, the name must begin with "`pu`".
  - If the file contains a private key, the name must begin with "`pr`".
- The `-a`, `-d` and `-v` options are mutually exclusive
- To change a field within a user entry, delete the old entry and add it again with the new information.
- Care should be taken to ensure that each unique key file has a unique name.
- The same input key file may be specified for multiple users. A warning will be issued if the resulting output name matches the name of an existing key file. However, the entry will be made and it will point to the existing outkeyfile. If this was not what was intended, then delete the new entry and add it again with a unique name for the input key file.

6 SCP - Secure Copy

The SCP server is an integrated service provided by the NicheStack SSH Server. It responds to file transfer requests from SCP clients such as PuTTY or OpenSSH. Client requests are received on the same port on which the SSH Server listens. The SSH Server handles authentication and encryption, while SCP performs the actual file transfer.

SCP file transfer is typically initiated from a command line. The syntax is similar to the syntax of the Unix `cp` command:

6.1 Example: Copying a file to the server

```
scp [Path/]srcfilename  user@ServerAddr:[Path/]destfilename
```

6.2 Example: Copying a file from the server

```
scp user@ServerAddr:[Path/]srcfilename [Path/]destfilename
```

ServerAddr can be an IPv4 or IPv6 address or a resolvable system name. When VFS is not mapped to a local file system, then no pathname is used between the ServerAddr and the destination file name, because the NicheStack implementation of VFS uses a flat file system.

When the destination of the transfer is the local directory, the "." can be used as the destfilename.

7 Building the SSH Server

The following `#defines` should be present in `ipport.h_h` file to compile SSH and SCP into the InterNiche stack. Options which are inappropriate for your particular configuration should be moved to a "not used" section of the file.

```
#define USE_SSH           1   /* Include SSH Server in build */
#define USE_SCP           1   /* Include SCP service in build */
#define SSH_FULL_IMPL     1   /* Include minimal or "full" feature set */
#define SSH_DEBUG         1   /* Prints SSH debugging messages */
#define SSH_ALLOW_NOSECURITY  1 /* Debug only. Allow SSH with no security */
#define SSH_MENUS         1 /* Include SSH menus */
#define SSH_DEBUG_DUMP    1  /* Print first 32 bytes of data messages */
```

The following #defines are located in `ssh/ssh_common.h`

```
#define SSH_BANNER        1   /* Whether or not the SSH Client displays a message once the
                                              transport-level connection has been made */
#define UFILE_ENCRYPT     1   /* Is the userinfo file encrypted   *
```

There are a number of defines related to the InterNiche CryptoEngine module included with the SSH product. At product delivery, these defines will be set to reasonable values. See the *CryptoEngine Technical Reference* for information on fine-tuning these defines for your product.

7.1 Source file list

The following is the list of SSH source files and short description of the files,

ssh_main.c	Implements the SSH initialization and clean up functions. It also contains the SSH "main loop."
ssh_trans.c	Implements the SSH transport layer.
ssh_auth_fsm.c	Implements the SSH Authentication layer.
ssh_connection_message.c	Implements the SSH connection layer.
ssh_connection_utils.c	Implements the SSH connection layer utility functions.
ssh_utils.c	Implements the general utility functions used throughout the SSH server.
ssh_menu.c	Implements the SSH server menus.
scp.c	Contains all routines for handling scp SOURCE and SINK commands.
ssh_mod.c	Contains the SSH task and module arrays and the SSH module prep, init, start, and close functions
ssh_nt.c	Implements the SSH server menus

7.2 Init-time Configuration

Tunable SSH Parameters

Global variables modifiable at run time

The following variables are stored in the global variable: `struct sshcfg ssh_globs`. All these variable are given an initial compile time value, which can be modified at run time.

Variable	Default Define	Definition
ssh_port	SSH_DFT_PORT	The SSH Server listens on this port. Default = 22.
ssh max auth tries	SSH_DFT_MAX_AUTH_TRIES	The number of consecutive failed authentication attempts allowed for a user name. Default = 20.
ssh_auth_idle_time	SSH_DFT_AUTH_IDLE_TIME	This is the total time in ticks within which a client must complete its authentication or it will be disconnected by the server. Default = 120 * TPS (2 minutes) Use INFINITE_DELAY for no idle timeout.
ssh_conn_idle_tmo	SSH_DFT_CONN_IDLE_TMO	Maximum time in ticks that a connection may be idle before the SSH Server will send a disconnect message and free connection resources. Default = 1200 * TPS (20 minutes). Use INFINITE_DELAY for no idle timeout.
ssh_none_cipher	SSH_DFT_NONE_CIPHER	Allow client to request the NULL cipher Default = 0. Parameter is ignored unless SSH_ALLOW_NOSECURITY set in ipport.h.
ssh_none_mac	SSH_DFT_NONE_MAC	Allow client to request the a NULL MAC algorithm. Default = 0. Parameter is ignored unless SSH_ALLOW_NOSECURITY set in ipport.h
ssh_allow_pwd_with_none_cipher	SSH_DFT_PWD_WITH_NONE_CIPHER	Allow client to send a password when the NULL cipher is used. Default = 0. Parameter is ignored unless SSH_ALLOW_NOSECURITY set in ipport.h

ssh_allowed_auth	SSH_DFT_ALLOWED_AUTH	The SSH Server supports 4 authentication schemes defined as follows:
		`#define NONE_AUTH      0x01`
		`#define KEY_AUTH       0x02`
		`#define PASS_AUTH      0x04`
		`#define PASS_KEY_AUTH  0x08`
		The variable ssh_allowed_auth is a mask of the authentication schemes the SSH Server is allowed to use. It overrides the values for user entries in the userinfo file. That is, if only the KEY_AUTH bit is set, then, the user cannot login without passing a key, even if only a password is specified by his user entry. The default = PASS_AUTH | KEY_AUTH, which means the user can use either password or key authentication, depending on the user's entry in the userinfo file. The value for ssh_allowed_auth is set at compile time and cannot be changed at run time, unless SSH_ALLOW_NOSECURITY is set in ipport.h. A value of 0x0F means that the authentication required of a user depends entirely on the user's entry in the userinfo file.

ssh_locfwd_port_start	SSH_DFT_LOCAL_FWD_PORT_START	Lowest port number that may be used for local port forwarding. Default = 0 (no restriction)
ssh_locfwd_port_end	SSH_DFT_LOCAL_FWD_PORT_END	Largest port number that may be used for local port forwarding. Default = 0 (no restriction)
ssh_rmtfwd_port_start	SSH_DFT_REMOTE_FWD_PORT_START	Lowest port number that may be used for remote port forwarding. Default = 0 (no restriction)
ssh_rmtfwd_port_end	SSH_DFT_REMOTE_FWD_PORT_END	Largest port number that may be used for remote port forwarding. Default = 0 (no restriction)
ssh_debug	SSH_DFT_DEBUG	Enables/disables printing of SSH debug messages. Ignored unless SSH_DEBUG is defined in ipport.h

Memory Allocation

At "init time" InterNiche SSH creates several memory pools for its operation, the sizes of which can be limited by changing "define" values within the source code. The SSH Server maintains a separate memory pool for each structure listed in the following table.

```
/* Requires one for each active connection */
struct ssh_client_connection
struct transport_algo
struct kex_options
struct channelIdPool

/* Requires less than, but possibly nearly one per active connection */
struct ssh_upperlayer_sockets
struct ssh_subsystem_callbacks
struct scp_command

/* Requires one to many per active connection */
struct ssh_channel
struct tcp_forward
struct ssh_msgbuffer
```

The following should be considered when determining the size of each memory pool:

- Each client connection requires one struct ssh_client_connection.
- Each active connection requires a `struct transport_algo`, a `struct key_options`, and a `struct channelIdPool`.
- There will be one `ssh_upperlayer_socket` structure for each application layer connection. None are required for client connections with remote port forwarding, as long as the server is simply listening for an incoming application connection request.
- Normally the client will open a separate connection with each scp command, so you will not need more `scp_command` structures than `ssh_client_connection` structures.
- One ssh_subsystem_callbacks structure is used for each scp connection.
- Each client connection may have multiple channels. Depending on expected usage, you may need three or more `ssh_channel` structures for each `ssh_client_connection`.
- tcp_forward structures are associated with channels, so there could be as many required as there are channels.
- You will need multiple `ssh_msgbuffer` structures for each channel that may be sending or receiving messages as the same time. The total number required can be reduced if you do not expect all open channels to be sending messages at the same time. However, if the SSH server is sending messages faster than TCP can handle them (e.g., the target's window is closed), then some `ssh_msgbuffer`s will be used to buffer outgoing data.

7.3 Cipher Support

In addition to the `3DES-CBC` and `AES128-CBC` encryption ciphers, InterNiche also provides the "`none`" cipher. It is important to recognize that this should be used **ONLY** when debugging or when benchmarking alternate ciphers as it will cause all data to be sent as "clear text".

The file `ssh_trans.c`, contains the lists of algorithms offered by the SSH server for selection by the SSH client. These lists are:

- kex_algos_list
- hostkey_algos_list
- cipher_algos_list
- mac_algos_list
- compress_algos_list

Including support for any of the other ciphers allowed by the RFCs involves making changes to `ssh_trans.c`, which should be done with care and in consultation with HCC Technical Support.

7.4 Basic and Full Implementation feature sets

Module	Function	Basic Implementation	Full Implementation
SSH-USERAUTH	Authentication	None, Publickey	Also includes Password and host-based
SSH-CONNECT	Connection Protocol Channel	Local Port Forwarding, Remote Port Forwarding, Menu commands	(same)
SSH-TRANS	Encryption cipher	None, 3des-cbc	Also includes aes128-cbc, aes192-cbc
SSH-TRANS	Message Authentication Code(MAC) Algorithm	None, hmac-sha1	Also includes hmac-sha1-96
SSH-TRANS	public key and/or certificate formats	ssh-dss	Also includes ssh-rsa
SSH-TRANS	Compression Algorithm	None	(same)
SSH-TRANS	Key Exchange Method	diffie-hellman-group1-sha1	Also includes diffie-hellman-group14-sha1
SCP	Secure Copy service	Compile time option	(same)

8 SSH Menu Commands

8.1 ssh cfgfwds

Command Name

`ssh cfgfwds - Configure or display SSH forwarding parameters`

Syntax

```
ssh cfgfwds [-p <port>] [-q <port>] [-r <port>] [-s <port>]
```

Parameters

(none)	Command without arguments displays the current state of ssh forwarding parameters
`-p`	lowest port number to be used for local port forwarding.
`-q`	highest port number to be used for local port forwarding
`-r`	lowest port number to be used for remote port forwarding.
`-s`	highest port number to be used for remote port forwarding

Description

This command is used to configure or display SSH forwarding parameters

Location

This command is provided by the `SSH server` module when `USE_SSH` and `SSH_MENUS` are defined.

8.2 ssh config

Command Name

`ssh config - display or modify SSH server configuration parameters`

Syntax

```
ssh config [-a <maxattemps>] [-t <maxidle>] [-v <"on" | "off">]
```

Parameters

(none)	Command without arguments displays the current state of ssh configuration parameters
`-a`	Integer: Maximum allowed authorization attempts.
`-t`	Integer: Maximum allowed idle time during authorizations.
`-v`	String: Turn ssh debug "on" or "off"

Description

This command displays or sets SSH configuration parameters

Location

This command is provided by the `ssh` module when `USE_SSH` and `SSH_MENUS` are defined.

8.3 ssh memstat

Command Name

`ssh memstat - Display SSH server memory usage`

Syntax

`ssh memstat`

Parameters

This command takes no arguements

Description

This command is used to to display SSH server memory usage

Notes/Status

SSH obtains memory from pre-configured SSH buffer pools. The memstat display shows:

- The name of the memory pool.
- The size of each element in bytes
- The maximum number of elements that may be allocated in this pool
- The lowest number that were available at any time.
- The number of currently allocated elements in the pool (whether used or not).

Location

This command is provided by the `SSH server` module when `USE_SSH` and `SSH_MENUS` are defined.

8.4 ssh netstat

Command Name

`ssh netstat - displays SSH Server statistics and status`

Syntax

```
ssh netstat [-i <client ID>] [-l ]
```

Parameters

(none)	Command without arguments displays statistics for all connections
`-i`	Integer: Display statics for specified Client ID (Obtained with ssh netstat -l).
`-l`	List active connections

Description

This command is used to display SSH Server statistics and status

Location

This command is provided by the `SSH server` module when `USE_SSH` and `SSH_MENUS` are defined.

8.5 ssh nosecurity

Command Name

```
ssh nosecurity - Allow/Disallow NULL authorization and security parameters
```

Syntax

```
ssh nosecurity [-a <"yes"|"no">] [-c <"yes"|"no">] [-c <"yes"|"no">] [-p <"
yes"|"no">]
```

Parameters

(none)	Command without arguments displays the current state of ssh NULL authorization and security parameters
-a	Allows/disallows access without authorization.
-c	Allows/disallows connections that do not use encryption.
-m	Allows/disallows connections that do not use MAC integrity protection
-p	Allows/disallows the use of passwords on connections without encryption.

Description

This command is intended for use during development and debugging. It enables/disables ssh server configuration parameters for permitting the use of NULL authorization and security

Notes/Status

- This command is only available if `ssh_globs.ssh_nosecurity_allowed` was set at compile time
- Key files specified in the usertable cannot be used with the NULL cipher. If `NONE_AUTH` is allowed, key_files will be ignored. Otherwise an error will be returned if the entry for the user specifies a key file.
- Passwords specified in the usertable cannot be used with the NULL cipher unless `ssh_globs.ssh_allow_pwd_with_none_cipher` is set. If that parameter is set and `NONE_AUTH` is allowed, passwords in the usertable will be ignored. Otherwise an error will be returned if the entry for the user specifies a password.

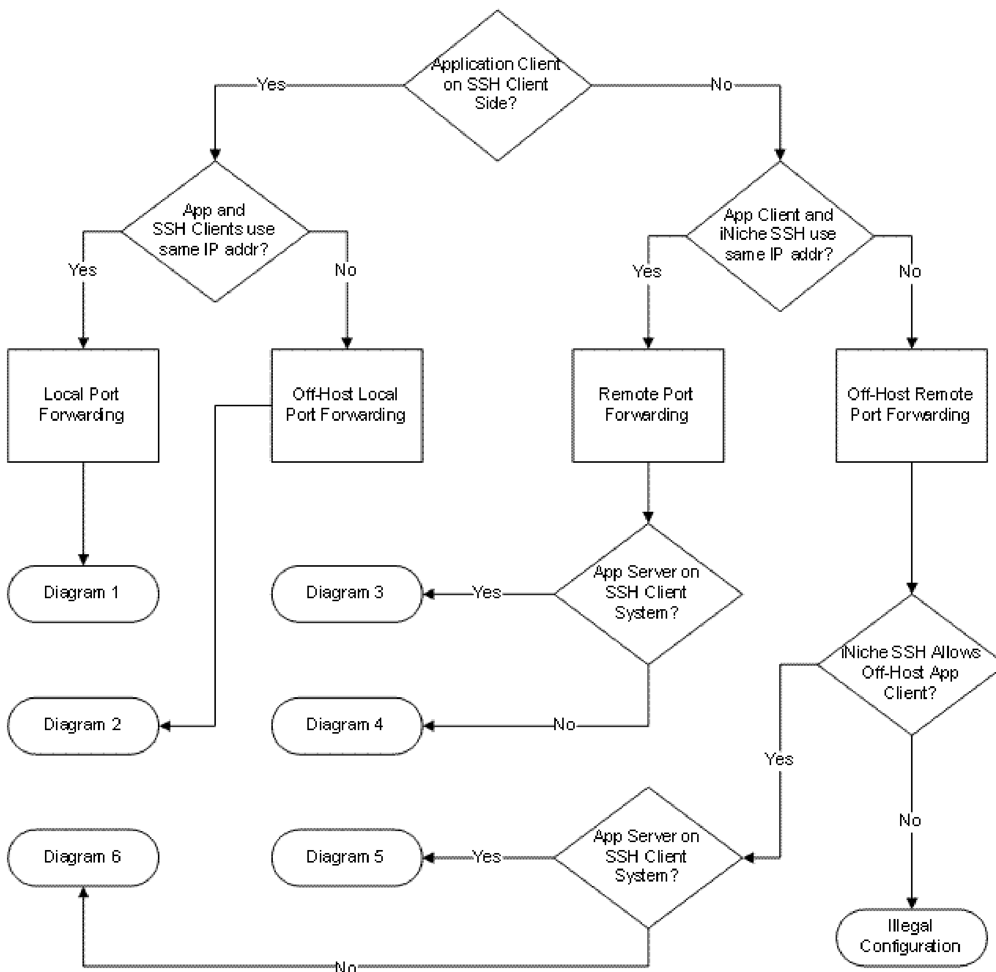
Location

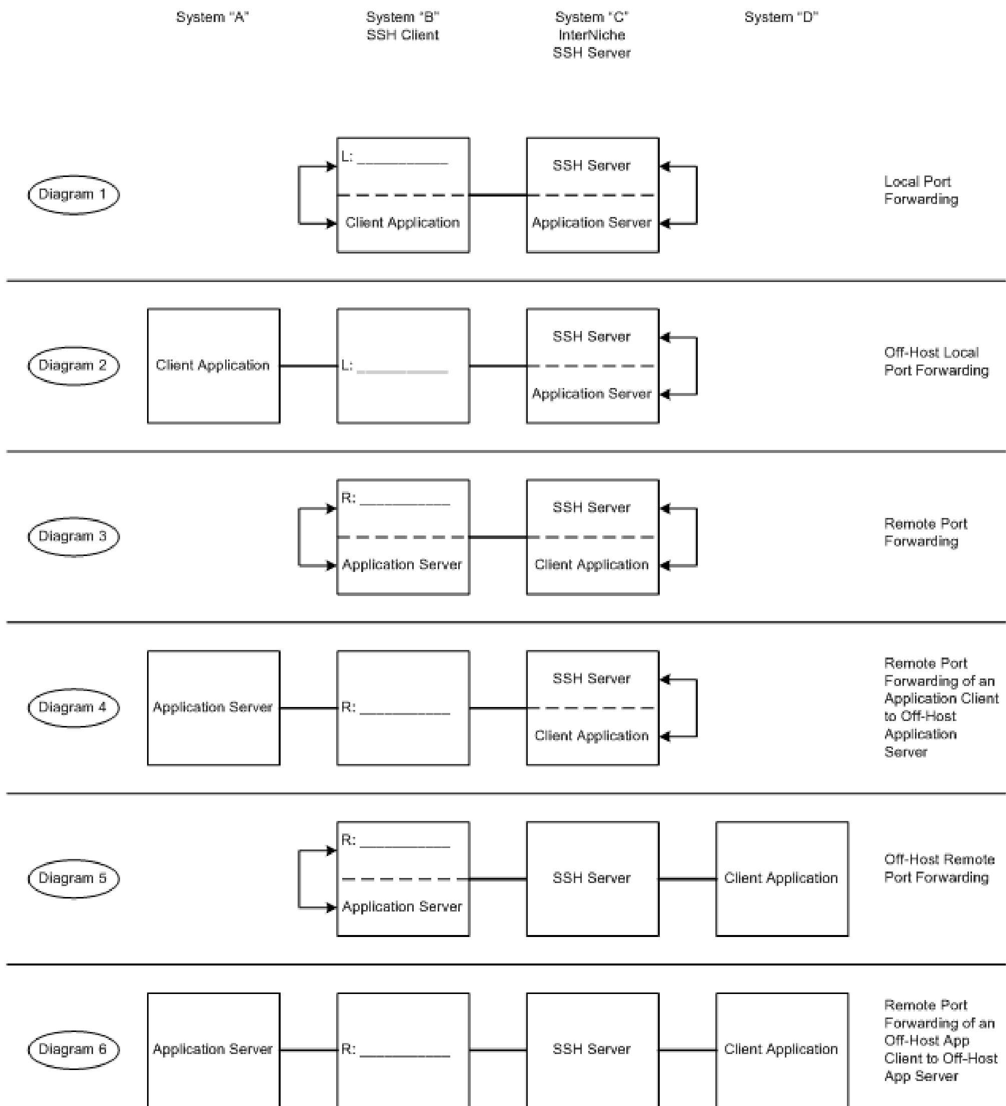
This command is provided by the `SSH server` module when `USE_SSH` and `SSH_MENUS` are defined.

9 Usage Examples

This section gives detailed examples for how to use the NicheStack SSH Server. NicheStack does not provide an SSH client. The SSH client will normally be some standard SSH client. On Windows, Putty is by far the most common SSH client, and it will be used as the SSH client in all examples in this document. Other applications can use NicheStack SSH by connecting either directly to a NicheStack server or indirectly via port forwarding.

9.1 Local vs. Remote Port Forwarding




System "A"
System "B"
SSH Client
System "C"
InterNiche
SSH Server
System "D"
Diagram 1
L: __________
Client Application
SSH Server
Application Server
Local Port Forwarding
Diagram 2
Client Application
L: __________
SSH Server
Application Server
Off-Host Local Port Forwarding
Diagram 3
R: __________
Application Server
SSH Server
Client Application
Remote Port Forwarding
Diagram 4
Application Server
R: __________
SSH Server
Client Application
Remote Port Forwarding of an Application Client to Off-Host Application Server
Diagram 5
R: __________
Application Server
SSH Server
Client Application
Off-Host Remote Port Forwarding
Diagram 6
Application Server
R: __________
SSH Server
Client Application
Remote Port Forwarding of an Off-Host App Client to Off-Host App Server

9.2 Understanding the Diagrams

Local vs. Remote Port Forwarding

Port forwarding comes in two basic types: local port forwarding and remote port forwarding. It is the application client's port that is being forwarded, and it is the SSH client (e.g., Putty) that is setting up the forwarding. Therefore, the terms "local" and "remote" here are refer to the application client's relation to the SSH client (e.g., Putty). Is the client port (and client application) local or remote in relation to the SSH client (e.g., Putty.)?

Security for Off-Host Connections

Off-host means that the application client or server is not at the same IP address as the SSH client or server at its end of the SSH connection. For example, the SSH server is at one end of the SSH connection. The application client or server that is directly connected to it may either be at the same IP address or it could be "off-host" on a different CPU.

Putty provides no security for this Off-Host connection. This connection requires some form of administrative security. A simple example would be a locally administrated network protected by a firewall. The application client may be allowed, by administrative action, to reside on any computer within the local network, but the firewall prevents a computer outside of the network from making the connection.

9.3 Usage examples using Putty

Note: PuTTY is used here as an example SSH client because of its well-organized graphical interface. It is not endorsed or supported by InterNiche. Putty is only one of several clients proven to interoperate with InterNiche SSH Server. In October 2014 PuTTY was available at http://www.chiark.greenend.org.uk /~sgtatham/putty/ .

General Putty Configuration

The following Putty configuration commands should be set for all of the usage examples.

Session top screen

Host Name	Enter the Host Name or IP address of the NicheStack SSH Server. Enter the port number used by the NicheStack SSH Server to listen for incoming connection requests. Normally this will be Port 22, unless the SSH Server has been specifically configured to listen on a different port.
Connection type	Always select SSH.
Saved Sessions	Once you have fully configured Putty to perform a specific function, you can save this configuration by coming back to this screen, entering a name for the configuration, and clicking on the Save button. Thereafter you can load the entire configuration by coming to this screen, selecting the name of the configuration and clicking on the Load button

Session Logging screen

Fill this out as you desire for logging. NicheStack imposes no requirements on this.

Connection top screen

Use the Putty documentation to determine how you want to set the parameters on this screen. NicheStack imposes no requires for this configuration.

Connection Data Screen

In order to avoid having to enter the user name for each session, you can save the user name here.

The Terminal details are not relevant unless you are connecting to the server via a Serial line.

NicheStack does not support the receipt of environmental variables from the client.

Connection SSH screen

Because the NicheStack SSH server does not support shells or PTY sessions, you MUST always check the box that says "Don't start a shell or command at all.

Preferred SSH protocol version: Leave the default with version 2 is selected. NicheStack does not support the Version 1 SSH protocol.

Encryption cipher selection policy: Nothing needs to be done here. However the SSH Server only supports the ciphers listed in Basic and Full Implementation feature sets.

Connection SSH Auth screen

The checkboxes can be left with their default values. If the user specified in the "Connection Data Screen" will use Key Authentication, then for the box "Private key file for authentication" browse to the `<privatekey>.ppk` file that PuTTY should use.

Note: Even though you selected a private key, PuTTY will send the associated public key for authentication.

Direct Commands

Although the NicheStack SSH Server does not support PTYs, shells, or interactive commands, it is possible to run a single specific NicheStack menu command from Putty, e.g., netstat, queues, a user-written command that executes several other commands, etc. However, this method is cumbersome. Normally, configuring Putty to tunnel a Telnet session over SSH will produce more desirable results.

In order to send a single NicheStack menu command from Putty, configure Putty as described above and enter the command in the Remote command box for the Connections SSH screen. Any output from the command will appear in the NicheStack console window or wherever console output is directed. NicheStack will close the connection once the command has been executed.

Port Forwarding

Non-secure protocols such as Telnet or HTTP can be tunneled over SSH using Port Forwarding techniques. Port forwarding across SSH requires no code change to either the application client or the application server. However, the command arguments used for the application client are different. The arguments must include a pre-selected port number, "CPort" (see Port Numbers in Examples below), and the IP address of the SSH client or Server that will forward this port. The application client code is not aware of the SSH connection and it is not given the address of the actual application server. Because the application client is not aware of the authentication used to make the connection, you will also need to coplete whatever authentication is required by the aapplication.

10 Testing Examples

10.1 Testing an internal command over SSH using putty

1. Run the SSH server.
2. Run putty and load the IP and port of the SSH server.
3. In putty configuration, user SSH, de-select the check box that says "Don't start a sheel or a command at all" and enter an internal command in "Remote command" text box (say "iface").
4. Start the putty session and continue with authentication.
5. Once authentication is complete, we see the output of the "iface" command on the putty screen.

10.2 Testing SCP SOURCE using pscp

1. Run the SSH server.
2. Ensure that the pscp command can be found in your shell's search path.
3. Run pscp using the following command: `pscp user@host:sourcefile targetdir` where:

user	valid username for SSH server
host	IP and port of SSH server
sourcefile	file whose contents have to sent to the client.
targetdir	The directory pathn where the file will be written at the client end.

10.3 Testing SCP SINK using pscp

1. Run the SSH server.
2. Run pscp using the following command: `pscp sourcefile user@host:targetfile` where:

user	valid username for SSH server
host	IP and port of SSH server
sourcefile	pathname of the file whose contents are to be copied to the server side.
target file	name of the file that is written at the server end.

10.4 Local Port Forwarding of Telnet

Addresses in Examples

In the examples below, addresses in the syntax for the SSH client and in the application client commands are specified as:

localhost

A Windows pseudo name for the IP address of the local CPU

Addr_of_S

The IP address of the NicheStack SSH Server

Addr_of_P

The IP address of the SSH client (e.g., Putty)

Cpu C

The IP address of the client application when different from Addr_of_P.

CPU AS

The IP address of the Application Server when different from Addr_of_S.

An address can be an IPv4 address, an IPv6 address, or a resolvable system name (unless the application client or server does not support name resolution).

Port Numbers in Examples

CPort

The SSH client (local forwarding) or the SSH server (remote forwarding) listens for application connections on this user selected port number.

This port can be any unused port, but it must be known in advance and used in both the Putty configuration and in the command arguments invoking the client application.

"PXX", where XX is a number. These specify "well known port numbers" designated for the common application servers, e.g. 22 for SSH, 23 for Telnet.

In this example (See Figure 1), the SSH client (e.g.,Putty) will be configured listen for a connection request on specific port, "CPort", pre-selected by the uers. When the SSH client receives a connection request on CPort, it will make a secure connection to the NicheStack SSH Server at Addr_of_S, and it will tell the SSH Server to forward the connection request to its port 23, the "well-known port" on which a Telnet Server normally listens . If the connection request succeeds, the subsequent data will be passed on both directions on that connection.

Figure 1. Local Port Forwarding of Telnet Client

Putty is first configured as described above in the General Putty Configuration section. In addition, we need to enter values into the Connection->SSH->Tunnels screen.

SSH->Connection->Tunnels Screen

- Port forwarding check boxes: For security reasons, leave these blank.
  Note: checking one or both of these boxes will not prevent a Telnet client on the local host from making a connection, but it would also allow off-host clients to make a connection (See the next example).
- Source port: Enter an unused port number. Normally this will be a value above 4096.
- Destination: Addr_of_S, followed by the port number to which it should forward the connection request.
- Selection Buttons: Local
- Click on the Add button. The source and destination information will be written into the empty box.

Off-Host Local Port Forwarding

Figure 2 shows an example of local port forwarding where the application client does not have the same IP address as the SSH Client (e.g., Putty). (It may or may not be on a different CPU).

The only differences in configuration between this case and the one above are:

- The box labeled "Local ports accept connections from other hosts." must be checked
- The arguments to the telnet client must use Putty's IP address rather than the argument "Localhost"

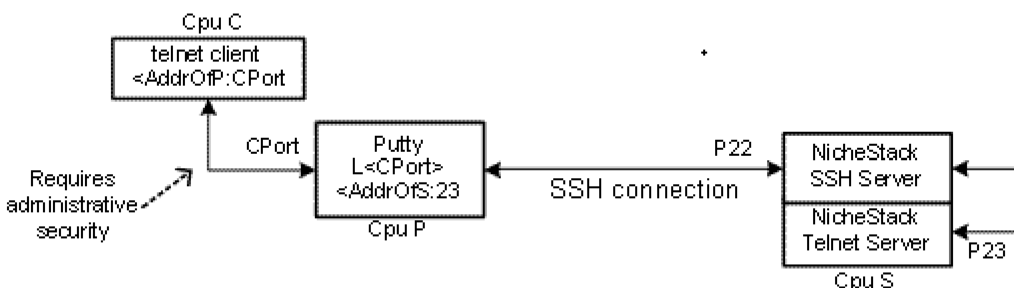


Figure 2. Off-host Local Port Forwarding of Telnet Client

Note: If the box "Local ports accept connections from other hosts" is checked, then the connection to the Telnet server can be initiated from any IP address that can reach CPort on Putty. This can be a major security issue (see Security for Off-Host Connections).

10.5 Remote Port Forwarding of Telnet

In the example shown in Figure 3, Putty is configured to tell NicheStack to listen on CPort and forward any connection requests to the SSH client (e.g.,Putty). The SSH client is also either configured to forward that connection to the Telnet server (listening on port 23), either on the local or at the specified IP address. This is call "remote" port forwarding because the application client's port that is being forwarded is remote to Putty.

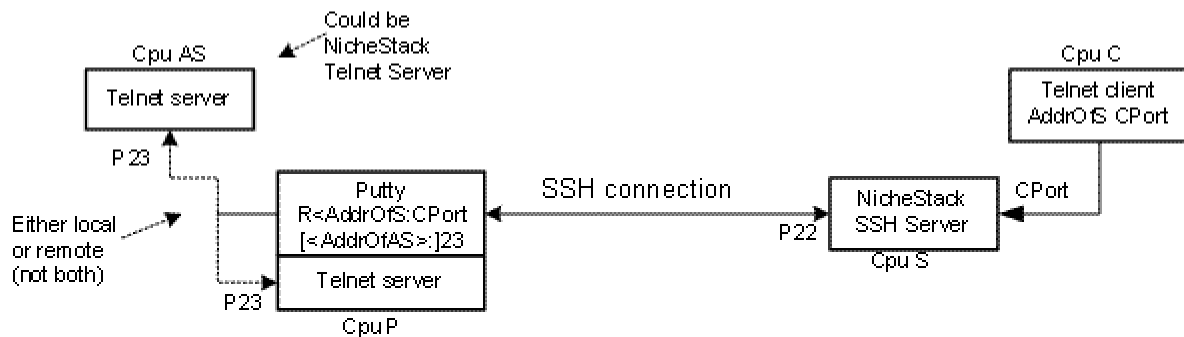


Figure 3. Remote Port Forwarding of Telnet Connection

In this particular example, the Telnet client must be Off-host because NicheStack does not contain an SSH client

SSH->Connection->Tunnels Screen

Port forwarding check boxes:

- Check the box "Remote ports do the same"
- Source port: Enter Addr_of_S, a ':' and CPort. (See Introduction for Port Forwarding section).
- Destination: If the Telnet server is on the local host, just enter its Well-known port number (23). If the Telnet server is located at another IP address, then enter its IP address, a ':', and the port number (23).
- Selection Buttons: Remote
- Click on the Add button. The source and destination information will be written into the empty box.

10.6 Port Forwarding Between NicheStacks

If an application that runs on NicheStack has both client and server capabilities, then it is possible to connect the client from one NicheStack to the server on another NicheStack via a secure SSH tunnel. However, the security is only between the SSH client (e.g., Putty) and the SSH Server. NicheStack does not have SSH Client capability; therefore, the NicheStack client will always be located at a different IP address from the SSH Client (e.g., Putty). See Security for Off-Host Connections.

Local Port Forwarding between NicheStacks

Figure 4 shows Local Port Forwarding of a NicheStack application client to the NicheStack application server. The NicheStack with the application client will have a different IP address from Putty, but it could be running either on the same CPU or on a different CPU. The configuration for both the SSH client and the application client will be the same in either case. See Security for Off-Host Connections concerning security for the connection between the NicheStack with trafgen and the SSH Client (e.g., Putty).

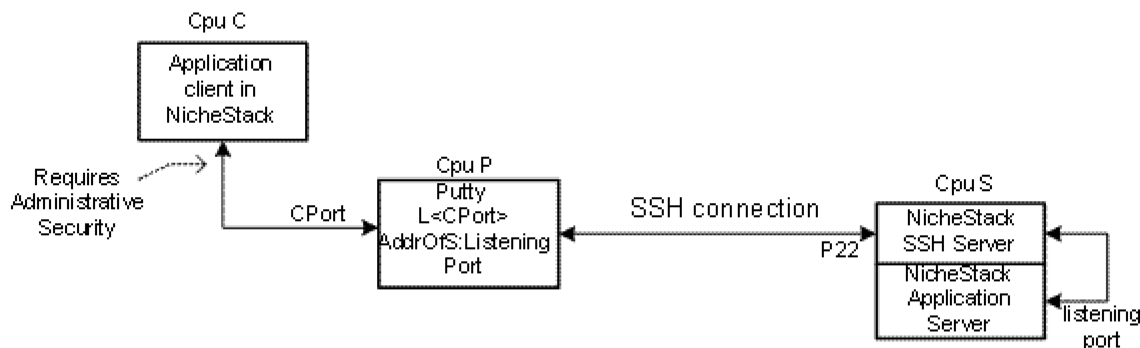


Figure 4. Local Port Forwarding of a NicheStack Client

Putty is first configured as described above in the General Putty Configuration section. In addition, we need to enter values into the Connection->SSH->Tunnels screen

SSH->Connection->Tunnels Screen

- The box labeled "Local ports accept connections from other hosts." must be checked
- Check the box "Remote ports do the same"
- Source port: Enter "CPort", a pre-selected application client port number.
- Destination: IP address of the SSH Server, followed by the Well-known port number to which it should forward the connection request.
- Selection Buttons: Local
- Click on the Add button. The source and destination information will be written into the empty box.

Note: With Local Port Forwarding, the application server cannot be off-host. There is no way to configure the SSH Client (e.g., Putty) so that it will tell the SSH Server to forward the connection to an Off-host application server.

Remote Port Forwarding between NicheStacks

In Remote Port Forwarding, the application client port (CPort), which is being forwarded, is remote to the SSH client that sets up the connection. Because NicheStack does not contain an SSH client, the application server must be running in a NicheStack that is Off-host in relation to the SSH client (e.g., Putty)-that is, it must be at a different IP address, although it could be on the same CPU. The NicheStack application client can either be in the same NicheStack as the NicheStack SSH server or it could be within a different NicheStack running on Cpu C.

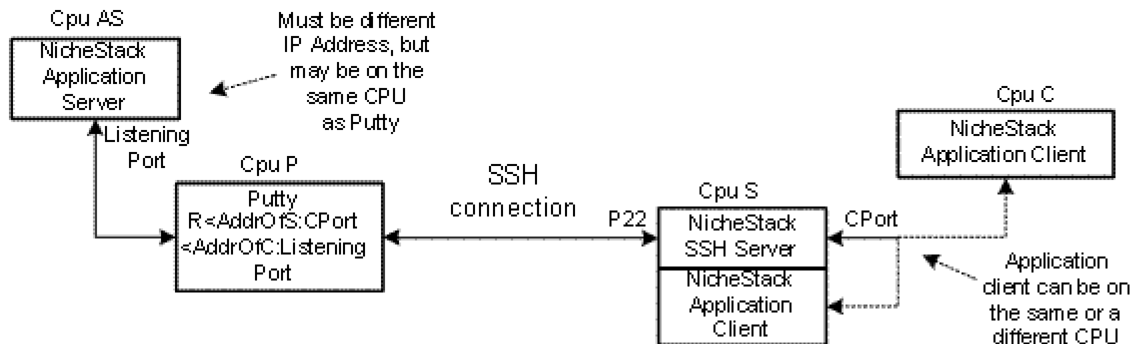


Figure 5. Remote Port Forwarding of a NicheStack Client

Putty is first configured as described above in the General Putty Configuration section. In addition, we need to enter values into the Connection->SSH->Tunnels screen

SSH->Connection->Tunnels Screen

Port forwarding check boxes:

- Check the box "Local ports accept connections from other hosts."
- Check the box "Remote ports do the same", if the NicheStack containing the application client is running on Cpu C.
- Source port: Enter <Addr_of_S>, a ':', and "CPort", a pre-selected application client port number.
- Destination: Enter <AddrOfC>, a ':', and the port number on which the application server listens.
- Selection Buttons: Remote
- Click on the Add button. The source and destination information will be written into the empty box.