# Selecting an IoServiceFactoryFactory

As part of the their initialization, both client and server code require the specification of a IoServiceFactoryFactory that is used to initialize network connections.

SshServer server = ...create server instance...  
 server.setIoServiceFactoryFactory(new MyIoServiceFactoryFactory());  
  
 SshClient client = ... create client instance ...  
 client.setIoServiceFactoryFactory(new MyIoServiceFactoryFactory());

If not set explicitly during the client/server setup code, then a factory is automatically detected and selected when the client/server is #start()-ed. The used IoServiceFactoryFactory is a **singleton** that is lazy created the 1st time DefaultIoServiceFactoryFactory#create is invoked. The selection process is as follows:

* The org.apache.sshd.common.io.IoServiceFactoryFactory system property is examined for a factory specification. The specification can be either a **fully-qualified** class name or one of the BuiltinIoServiceFactoryFactories values.
* If no specific factory is specified, then the [ServiceLoader#load](https://docs.oracle.com/javase/tutorial/ext/basics/spi.html#register-service-providers) mechanism is used to detect and instantiate any registered services in any META-INF\services\org.apache.sshd.common.io.IoServiceFactoryFactory location in the classpath. If **exactly one** implementation was instantiated, then it is used. If several such implementations are found then an exception is thrown.
* Otherwise, the built-in Nio2ServiceFactoryFactory is used.

**Note:** the default command line scripts for SSH/SCP/SFTP client/server are set up to use NIO2 as their default provider, unless overridden via the -io command line option. The org.apache.sshd.common.io.IoServiceFactoryFactory system property does not apply for the command line wrappers since they look only for the -io option and use it to initialize the client/server **explicitly** before starting the client/server. Therefore, the default selection process described in this section does not apply for them.

# Advanced configuration and interaction

## Properties and inheritance model

The code's behavior is highly customizable not only via non-default implementations of interfaces but also as far as the **parameters** that govern its behavior - e.g., timeouts, min./max. values, allocated memory size, etc... All the customization related code flow implements a **hierarchical** PropertyResolver inheritance model where the "closest" entity is consulted first, and then its "owner", and so on until the required value is found. If the entire hierarchy yielded no specific result, then some pre-configured default is used. E.g., if a channel requires some parameter in order to decide how to behave, then the following configuration hierarchy is consulted:

* The channel-specific configuration
* The "owning" session configuration
* The "owning" client/server instance configuration
* The system properties - **Note:** any configuration value required by the code can be provided via a system property bearing the org.apache.sshd.config prefix - see SyspropsMapWrapper for the implementation details.

The easiest way to configure a target instance (client/server/session/channel) is via one of the (many) available PropertyResolverUtils updateProperty methods:

PropertyResolverUtils.updateProperty(client, "prop1", 5L);  
 PropertyResolverUtils.updateProperty(server, "prop2", someInteger);  
 PropertyResolverUtils.updateProperty(session, "prop3", "hello world");  
 PropertyResolverUtils.updateProperty(channel, "prop4", false);

**Note**: the updateProperty method(s) accept **any** Object so care must be taken to provide the expected type. However, at least for **primitive** values, the various getXXXProperty methods automatically convert compatible types:

PropertyResolverUtils.updateProperty(client, "prop1", 7365L);  
  
 // all will yield 7365 converted to the relevant type  
 Long value = PropertyResolverUtils.getLongProperty(client, "prop1");  
 Integer value = PropertyResolverUtils.getLongProperty(client, "prop1");

including strings

PropertyResolverUtils.updateProperty(client, "prop1", "7365");  
  
 // all will yield 7365  
 Long value = PropertyResolverUtils.getLongProperty(client, "prop1");  
 Integer value = PropertyResolverUtils.getLongProperty(client, "prop1");

### Using the inheritance model for fine-grained/targeted configuration

As previously mentioned, this hierarchical lookup model is not limited to "simple" configuration values (strings, integers, etc.), but used also for **interfaces/implementations** such as cipher/MAC/compression/authentication/etc. factories - the exception being that the system properties are not consulted in such a case. This code behavior provides highly customizable fine-grained/targeted control of the code's behavior - e.g., one could impose usage of specific ciphers/authentication methods/etc. or present different public key "identities"/welcome banner behavior/etc., based on address, username or whatever other decision parameter is deemed relevant by the user's code. This can be done on **both** sides of the connection - client or server. E.g., the client could present different keys based on the server's address/identity string/welcome banner, or the server could accept only specific types of authentication methods based on the client's address/username/etc... This can be done in conjunction with the usage of the various EventListener-s provided by the code (see below).

One of the code locations where this behavior can be leveraged is when the server provides **file-based** services (SCP, SFTP) in order to provide a different/limited view of the available files based on the username - see the section dealing with FileSystemFactory-ies.

## Welcome banner configuration

According to [RFC 4252 - section 5.4](https://tools.ietf.org/html/rfc4252#section-5.4) the server may send a welcome banner message during the authentication process. Both the message contents and the phase at which it is sent can be configured/customized.

### Welcome banner content customization

The welcome banner contents are controlled by the ServerAuthenticationManager.WELCOME\_BANNER configuration key - there are several possible values for this key:

* A simple string - in which case its contents are the welcome banner.
* A file [URI](https://docs.oracle.com/javase/8/docs/api/java/net/URI.html) - or a string starting with "file:/" followed by the file path - see below.
* A [URL](https://docs.oracle.com/javase/8/docs/api/java/net/URL.html) - or a string containing "://" - in which case the [URL#openStream()](https://docs.oracle.com/javase/8/docs/api/java/net/URL.html#openStream) method is invoked and its contents are read.
* A [File](https://docs.oracle.com/javase/8/docs/api/java/io/File.html) or a [Path](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html) - in this case, the file's contents are **re-loaded** every time it is required and sent as the banner contents.
* The special value ServerAuthenticationManager.AUTO\_WELCOME\_BANNER\_VALUE which generates a combined "random art" of all the server's keys as described in Perrig A. and Song D.-s article [Hash Visualization: a New Technique to improve Real-World Security](http://sparrow.ece.cmu.edu/~adrian/projects/validation/validation.pdf) - *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*
* One can also override the ServerUserAuthService#resolveWelcomeBanner method and use whatever other content customization one sees fit.

**Note:**

1. If any of the sources yields an empty string or is missing (in the case of a resource) then no welcome banner message is sent.
2. If the banner is loaded from a file or URL resource, then one can configure the [Charset](https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html) used to convert the file's contents into a string via the ServerAuthenticationManager.WELCOME\_BANNER\_CHARSET configuration key (default=UTF-8).
3. In this context, see also the ServerAuthenticationManager.WELCOME\_BANNER\_LANGUAGE configuration key - which provides control over the declared language tag, although most clients seem to ignore it.

### Welcome banner sending phase

According to [RFC 4252 - section 5.4](https://tools.ietf.org/html/rfc4252#section-5.4):

The SSH server may send an SSH\_MSG\_USERAUTH\_BANNER message at any time after this authentication protocol starts and before authentication is successful.

The code contains a WelcomeBannerPhase enumeration that can be used to configure via the ServerAuthenticationManager.WELCOME\_BANNER\_PHASE configuration key the authentication phase at which the welcome banner is sent (see also the ServerAuthenticationManager.DEFAULT\_BANNER\_PHASE value). In this context, note that if the NEVER phase is configured, no banner will be sent even if one has been configured via one of the methods mentioned previously.

## HostConfigEntryResolver

This interface provides the ability to intervene during the connection and authentication phases and "re-write" the user's original parameters. The DefaultConfigFileHostEntryResolver instance used to set up the default client instance follows the [SSH config file](https://www.digitalocean.com/community/tutorials/how-to-configure-custom-connection-options-for-your-ssh-client) standards, but the interface can be replaced so as to implement whatever proprietary logic is required.

SshClient client = SshClient.setupDefaultClient();  
 client.setHostConfigEntryResolver(new MyHostConfigEntryResolver());  
 client.start();  
  
 /\*  
 \* The resolver might decide to connect to some host2/port2 using user2 and password2  
 \* (or maybe using some key instead of the password).  
 \*/  
 try (ClientSession session = client.connect(user1, host1, port1).verify(...timeout...).getSession()) {  
 session.addPasswordIdentity(...password1...);  
 session.auth().verify(...timeout...);  
 }

### SSH Jumps

The SSH client can be configured to use [SSH proxy jumps](https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Proxies_and_Jump_Hosts). A *jump host* (also known as a *jump server*) is an intermediary host or an SSH gateway to a remote network, through which a connection can be made to another host in a dissimilar security zone, for example a demilitarized zone (DMZ). It bridges two dissimilar security zones and offers controlled access between them.

Starting from SSHD version 2.6.0, the *ProxyJump* host configuration entry is honored when using the SshClient to connect to a host. The SshClient built by default reads the ~/.ssh/config file. The various CLI clients also honor the -J command line option to specify one or more jumps.

In order to manually configure jumps, you need to build a HostConfigEntry with a proxyJump and use it to connect to the server:

ConnectFuture future = client.connect(new HostConfigEntry(  
 "", host, port, user,  
 proxyUser + "@" + proxyHost + ":" + proxyPort));

The configuration options specified in the configuration file for the jump hosts are also honored.

## SshConfigFileReader

Can be used to read various standard SSH [client](http://linux.die.net/man/5/ssh_config) or [server](http://manpages.ubuntu.com/manpages/precise/en/man5/sshd_config.5.html) configuration files and initialize the client/server respectively. Including (among other things), bind address, ciphers, signature, MAC(s), KEX protocols, compression, welcome banner, etc..

### Reserved messages

The implementation can be used to intercept and process the [SSH\_MSG\_IGNORE](https://tools.ietf.org/html/rfc4253#section-11.2), [SSH\_MSG\_DEBUG](https://tools.ietf.org/html/rfc4253#section-11.3) and [SSH\_MSG\_UNIMPLEMENTED](https://tools.ietf.org/html/rfc4253#section-11.4) messages. The handler can be registered on either side - server or client, as well as on the session. A special [patch](https://issues.apache.org/jira/browse/SSHD-699) has been introduced that automatically ignores such messages if they are malformed - i.e., they never reach the handler.

#### SSH message stream "stuffing" and keys re-exchange

[RFC 4253 - section 9](https://tools.ietf.org/html/rfc4253#section-9) recommends re-exchanging keys every once in a while based on the amount of traffic and the selected cipher - the matter is further clarified in [RFC 4251 - section 9.3.2](https://tools.ietf.org/html/rfc4251#section-9.3.2). These recommendations are mirrored in the code via the FactoryManager related REKEY\_TIME\_LIMIT, REKEY\_PACKETS\_LIMIT and REKEY\_BLOCKS\_LIMIT configuration properties that can be used to configure said behavior - please be sure to read the relevant *Javadoc* as well as the aforementioned RFC section(s) when manipulating them. This behavior can also be controlled programmatically by overriding the AbstractSession#isRekeyRequired() method.

As an added security mechanism [RFC 4251 - section 9.3.1](https://tools.ietf.org/html/rfc4251#section-9.3.1) recommends adding "spurious" [SSH\_MSG\_IGNORE](https://tools.ietf.org/html/rfc4253#section-11.2) messages. This functionality is mirrored in the FactoryManager related IGNORE\_MESSAGE\_FREQUENCY, IGNORE\_MESSAGE\_VARIANCE and IGNORE\_MESSAGE\_SIZE configuration properties that can be used to configure said behavior - please be sure to read the relevant *Javadoc* as well as the aforementioned RFC section when manipulating them. This behavior can also be controlled programmatically by overriding the AbstractSession#resolveIgnoreBufferDataLength() method.

### RequestHandler(s)

The code supports both [global](https://tools.ietf.org/html/rfc4254#section-4) and [channel-specific](https://tools.ietf.org/html/rfc4254#section-5.4) requests via the registration of RequestHandler(s). The global handlers are derived from ConnectionServiceRequestHandler(s) whereas the channel-specific ones are derived from ChannelRequestHandler(s). In order to add a handler one need only register the correct implementation and handle the request when it is detected. For global request handlers this is done by registering them on the server:

// NOTE: the following code can be employed on BOTH client and server - the example is for the server  
 SshServer server = SshServer.setUpDefaultServer();  
 List<RequestHandler<ConnectionService>> oldGlobals = server.getGlobalRequestHandlers();  
 // Create a copy in case current one is null/empty/un-modifiable  
 List<RequestHandler<ConnectionService>> newGlobals = new ArrayList<>();  
 if (GenericUtils.size(oldGlobals) > 0) {  
 newGlobals.addAll(oldGLobals);  
 }  
 newGlobals.add(new MyGlobalRequestHandler());  
 server.setGlobalRequestHandlers(newGlobals);

For channel-specific requests, one uses the channel's add/removeRequestHandler method to manage its handlers. The way request handlers are invoked when a global/channel-specific request is received is as follows:

* All currently registered handlers' process method is invoked with the request type string parameter (among others). The implementation should examine the request parameters and decide whether it is able to process it.
* If the handler returns Result.Unsupported then the next registered handler is invoked. In other words, processing stops at the **first** handler that returned a valid response. Thus the importance of the List<RequestHandler<...>> that defines the **order** in which the handlers are invoked. **Note**: while it is possible to register multiple handlers for the same request and rely on their order, it is highly recommended to avoid this situation as it makes debugging the code and diagnosing problems much more difficult.
* If no handler reported a valid result value then a failure message is sent back to the peer. Otherwise, the returned result is translated into the appropriate success/failure response (if the sender asked for a response). In this context, the handler may choose to build and send the response within its own code, in which case it should return the Result.Replied value indicating that it has done so.

public class MySpecialChannelRequestHandler implements ChannelRequestHandler {  
 ...  
  
 @Override  
 public Result process(Channel channel, String request, boolean wantReply, Buffer buffer) throws Exception {  
 if (!"my-special-request".equals(request)) {  
 return Result.Unsupported; // Not mine - maybe someone else can handle it  
 }  
  
 ...handle the request - can read more parameters from the message buffer...  
  
 return Result.ReplySuccess/Failure/Replied; // signal processing result  
 }  
 }

#### Default registered handlers

* exit-signal, exit-status - As described in [RFC4254 section 6.10](https://tools.ietf.org/html/rfc4254#section-6.10)
* \*@putty.projects.tartarus.org - As described in [Appendix F: SSH-2 names specified for PuTTY](http://tartarus.org/~simon/putty-snapshots/htmldoc/AppendixF.html)
* hostkeys-prove-00@openssh.com, hostkeys-00@openssh.com - As described in [OpenSSH protocol - section 2.5](https://github.com/openssh/openssh-portable/blob/master/PROTOCOL)
* tcpip-forward, cancel-tcpip-forward - As described in [RFC4254 section 7](https://tools.ietf.org/html/rfc4254#section-7)
* keepalive@\* - Used by many client implementations (including this one) to "ping" the server and keep/make sure the connection is still alive. In this context, the SSHD code allows the user to configure both the frequency and content of the heartbeat request (including whether to send this request at all) via the ClientFactoryManager-s HEARTBEAT\_INTERVAL, HEARTBEAT\_REQUEST and DEFAULT\_KEEP\_ALIVE\_HEARTBEAT\_STRING configuration properties.
* no-more-sessions@\* - As described in [OpenSSH protocol section 2.2](https://github.com/openssh/openssh-portable/blob/master/PROTOCOL). In this context, the code consults the ServerFactoryManagder.MAX\_CONCURRENT\_SESSIONS server-side configuration property in order to decide whether to accept a successfully authenticated session.

## PROXY / SSLH protocol hooks

The code contains [support for "wrapper" protocols](https://issues.apache.org/jira/browse/SSHD-656) such as [PROXY](http://www.haproxy.org/download/1.6/doc/proxy-protocol.txt) or [sslh](http://www.rutschle.net/tech/sslh.shtml). The idea is that one can register either a ClientProxyConnector or ServerProxyAcceptor and intercept the 1st packet being sent/received (respectively) **before** it reaches the SSHD code. This gives the programmer the capability to write a front-end that routes outgoing/incoming packets:

* SshClient/ClientSesssion#setClientProxyConnector - sets a proxy that intercepts the 1st packet before being relayed to the server
* SshServer/ServerSession#setServerProxyAcceptor - sets a proxy that intercepts the 1st incoming packet before being processed by the server