# SCP

Both client-side and server-side SCP are supported. Starting from version 2.0, the SCP related code is located in the sshd-scp module, so you need to add this additional dependency to your maven project:

<dependency>  
 <groupId>org.apache.sshd</groupId>  
 <artifactId>sshd-scp</artifactId>  
 <version>...same as sshd-core...</version>  
 </dependency>

## ScpTransferEventListener

Callback to inform about SCP related events. ScpTransferEventListener(s) can be registered on *both* client and server side:

// Server side  
 ScpCommandFactory factory = new ScpCommandFactory(...with/out delegate..);  
 factory.addEventListener(new MyServerSideScpTransferEventListener());  
 sshd.setCommandFactory(factory);  
  
 // Client side  
 try (ClientSession session = client.connect(user, host, port)  
 .verify(...timeout...)  
 .getSession()) {  
 session.addPasswordIdentity(password);  
 session.auth().verify(...timeout...);  
  
 ScpClientCreator creator = ... obtain an instance ...  
 ScpClient client = creator.createScpClient(session, new MySuperDuperListener());  
  
 ...scp.upload/download...  
 }

## Client-side SCP

In order to obtain an ScpClient one needs to use an ScpClientCreator:

try (ClientSession session = ... obtain an instance ...) {  
 ScpClientCreator creator = ... obtain an instance ...  
 ScpClient client = creator.createScpClient(session);  
 ... use client ...  
}

A default ScpClientCreator instance is provided as part of the module - see ScpClientCreator.instance()

If the intended use of the client instance is "one-shot" - i.e., the client session should be closed when the SCP client instance is closed, then it is possible to obtain a special wrapper that implements this functionality:

// The underlying session will also be closed when the client is  
try (CloseableScpClient client = createScpClient(...)) {  
 ... use client ...  
}  
  
CloseableScpClient createScpClient(...) {  
 ClientSession session = ... obtain an instance ...;  
 ScpClientCreator creator = ... obtain an instance ...  
 ScpClient client = creator.createScpClient(session);  
 return CloseableScpClient.singleSessionInstance(client);  
}

The ScpClientCreator can also be used to attach a default ScpTransferEventListener that will be automatically add to **all** created SCP client instances through that creator - unless specifically overridden:

ClientSession session = ... obtain an instance ...  
ScpClientCreator creator = ... obtain an instance ...  
creator.setScpTransferEventListener(new MySuperDuperListener());  
  
ScpClient client1 = creator.createScpClient(session); // <<== automatically uses MySuperDuperListener  
ScpClient client2 = creator.createScpClient(session, new SomeOtherListener()); // <<== uses SomeOtherListener instead of MySuperDuperListener

## ScpFileOpener(s)

As part of the ScpClientCreator, the SCP module also uses a ScpFileOpener instance in order to access the local files. The default implementation simply opens an [InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) or [OutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html) on the requested local path. However, the user may replace it and intercept the calls - e.g., for logging, monitoring transfer progess, wrapping/filtering the streams, etc... The user may attach a default opener that will be automatically attached to **all** clients created unless specifically overridden:

/\*\*  
 \* Example of using a non-default opener for monitoring and reporting on transfer progress  
 \*/  
public class ScpTransferProgressMonitor extends DefaultScpFileOpener {  
 public static final ScpTransferProgressMonitor MONITOR = new ScpTransferProgressMonitor();  
  
 public ScpTransferProgressMonitor() {  
 super();  
 }  
  
 @Override  
 public InputStream openRead(  
 Session session, Path file, long size, Set<PosixFilePermission> permissions, OpenOption... options)  
 throws IOException {  
 return new MyProgressReportingInputStream(super.openRead(session, file, size, permissions, options), size /\* how much is expected \*/);  
 }  
  
 @Override  
 public OutputStream openWrite(  
 Session session, Path file, long size, Set<PosixFilePermission> permissions, OpenOption... options)  
 throws IOException {  
 return new MyProgressReportingOutputStream(super.openWrite(session, file, size, permissions, options), size /\* how much is expected \*/);  
 }  
}  
  
ClientSession session = ... obtain an instance ...  
ScpClientCreator creator = ... obtain an instance ...  
creator.setScpFileOpener(ScpTransferProgressMonitor.INSTANCE);  
  
ScpClient client1 = creator.createScpClient(session); // <<== automatically uses ScpTransferProgressMonitor  
ScpClient client2 = creator.createScpClient(session, new SomeOtherOpener()); // <<== uses SomeOtherOpener instead of ScpTransferProgressMonitor

**Note(s):**

* Due to SCP protocol limitations one cannot change the **size** of the input/output since it is passed as part of the command **before** the file opener is invoked - so there are a few limitations on what one can do within this interface implementation.
* By default, SCP synchronizes the local copied file data with the file system using the [Java SYNC open option](https://docs.oracle.com/javase/8/docs/api/java/nio/file/StandardOpenOption.html#SYNC). This behavior can be controlled by setting the scp-auto-sync-on-write (a.k.a. ScpFileOpener#PROP\_AUTO\_SYNC\_FILE\_ON\_WRITE) property to *false* or overriding the DefaultScpFileOpener#resolveOpenOptions, or even overriding the ScpFileOpener#openWrite method altogether.
* Patterns used in ScpFileOpener#getMatchingFilesToSend are matched using case sensitivity derived from the O/S as detected by the internal OsUtils. If a different behavior is required, then one needs to replace the default opener with one that uses a different sensitivity via DirectoryScanner#setCaseSensitive call (or executes the pattern matching in another way).
  + Windows - case insensitive
  + Unix - case sensitive

## Server-side SCP

Setting up SCP support on the server side is straightforward - simply initialize a ScpCommandFactory and set it as the **primary** command factory. If support for commands other than SCP is also required then provide the extra commands factory as a **delegate** of the ScpCommandFactory. The SCP factory will intercept the SCP command and execute it, while propagating all other commands to the delegate. If no delegate configured then the non-SCP command is deemed as having failed (same as if it were rejected by the delegate).

ScpCommandFactory factory = new ScpCommandFactory.Builder()  
 .withDelegate(new MyCommandDelegate())  
 .build();  
  
SshServer sshd = ...create an instance...  
sshd.setCommandFactory(factory);

The ScpCommandFactory allows users to attach an ScpFileOpener and/or ScpTransferEventListener having the same behavior as the client - i.e., monitoring and intervention on the accessed local files. Furthermore, the factory can also be configured with a custom executor service for executing the requested copy commands as well as controlling the internal buffer sizes used to copy files.

## The SCP "shell"

Some SCP clients (e.g. [WinSCP](https://winscp.net/)) open a shell connection even if configured to use pure SCP in order to retrieve information about the remote server's files and potentially navigate through them. In other words, SCP is only used as the **transfer** protocol, but the application relies on "out-of-band" information (shell in this case) in order to provide the user with the available files list on the remote server.

Due to various considerations, some users might not want or be able to provide a full blown shell interface on the server side. For this purpose SSHD provides an ScpShell class that provides a good enough implementation of the **limited** command types that an SCP client is likely to require. For this purpose, the ScpCommandFactory also implements ShellFactory which spawns the limited ScpShell support.

ScpCommandFactory factory = new ScpCommandFactory.Builder()  
 .with(...)  
 .with(...)  
 .build()  
 ;  
 sshd.setCommandFactory(factory);  
 sshd.setShellFactory(factory);

**Note:** a similar result can be achieved if activating SSHD from the command line by specifying -o ShellFactory=scp

## Remote-to-remote transfer

The code provides an ScpTransferHelper class that enables copying files between 2 remote accounts without going through the local file system.

ClientSession src = ...;  
 ClientSession dst = ...;  
 // Can also provide a listener in the constructor in order to be informed of the actual transfer progress  
 ScpRemote2RemoteTransferHelper helper = new ScpRemote2RemoteTransferHelper(src, dst);  
 // can be repeated for as many files as necessary using the same helper  
 helper.transferFile("remote/src/path/file1", "remote/dst/path/file2");

## References

* [How the SCP protocol works](https://chuacw.ath.cx/development/b/chuacw/archive/2019/02/04/how-the-scp-protocol-works.aspx)
* [scp.c](https://github.com/cloudsigma/illumos-omnios/blob/master/usr/src/cmd/ssh/scp/scp.c)