## Event listeners

The code supports registering many types of event listeners that enable receiving notifications about important events as well as sometimes intervening in the way these events are handled. All listener interfaces extend SshdEventListener so they can be easily detected and distinguished from other EventListener(s).

In general, event listeners are **cumulative** - e.g., any channel event listeners registered on the SshClient/Server are automatically added to all sessions, *in addition* to any such listeners registered on the Session, as well as any specific listeners registered on a specific Channel - e.g.,

// Any channel event will be signalled to ALL the registered listeners  
 sshClient/Server.addChannelListener(new Listener1());  
 sshClient/Server.addSessionListener(new SessionListener() {  
 @Override  
 public void sessionCreated(Session session) {  
 session.addChannelListener(new Listener2());  
 session.addChannelListener(new ChannelListener() {  
 @Override  
 public void channelInitialized(Channel channel) {  
 channel.addChannelListener(new Listener3());  
 }  
 });  
 }  
 });

### IoServiceEventListener

This listener provides low-level events regarding connection establishment (by the client) or acceptance (by the server). The listener is registered on the IoServiceFactory via the FactoryManager-s (i.e., SshClient/Server#setIoServiceEventListener). Unlike other listeners defined in this section, it is **not cumulative** - i.e., one can setIoServiceEventListener but not addIoServiceEventListener - thus **replacing** any previously registered listener.

### SessionListener

Informs about session related events. One can modify the session - although the modification effect depends on the session's **state**. E.g., if one changes the ciphers *after* the key exchange (KEX) phase, then they will take effect only if the keys are re-negotiated. Furthermore, invoking some session API(s) - event getSomeValue at the wrong time might yield unexpected results. It is important to read the documentation very carefully and understand at which stage each listener method is invoked, what are the limitations and what are the repercussions of changes at that stage. In this context, it is worth mentioning that one can attach to sessions **arbitrary attributes** that can be retrieved by the user's code later on:

public static final AttributeKey<String> STR\_KEY = new AttributeKey<>();  
 public static final AttributeKey<Long> LONG\_KEY = new AttributeKey<>();  
  
 sshClient/Server.addSessionListener(new SessionListener() {  
 @Override  
 public void sessionEstablished(Session session) {  
 // examine the peer address or the connection context and set some attributes  
 }  
  
 @Override  
 public void sessionCreated(Session session) {  
 session.setAttribute(STR\_KEY, "Some string value");  
 session.setAttribute(LONG\_KEY, 3777347L);  
 // ...etc...  
 }  
  
 @Override  
 public void sessionClosed(Session session) {  
 String str = session.getAttribute(STR\_KEY);  
 Long l = session.getAttribute(LONG\_KEY);  
 // ... do something with the retrieved attributes ...  
 }  
 });

The attributes cache is automatically cleared once the session is closed.

### ChannelListener

Informs about channel related events - as with sessions, once can influence the channel to some extent, depending on the channel's **state**. The ability to influence channels is much more limited than sessions. In this context, it is worth mentioning that one can attach to channels **arbitrary attributes** that can be retrieved by the user's code later on and are cleared when channel is closed - same was as it is done for sessions.

### UnknownChannelReferenceHandler

Invoked whenever a message intended for an unknown channel is received. By default, the code **ignores** the vast majority of such messages and logs them at DEBUG level. For a select few types of messages the code generates an SSH\_CHANNEL\_MSG\_FAILURE packet that is sent to the peer session - see DefaultUnknownChannelReferenceHandler implementation. The user may register handlers at any level - client/server, session and/or connection service - the one registered "closest" to connection service will be used.

An **experimental** ChannelIdTrackingUnknownChannelReferenceHandler is available in *sshd-contrib* package that applies the "leniency" of the DefaultUnknownChannelReferenceHandler only if the unknown channel is one that has been assigned in the past - otherwise it throws an exception. In order to use it, the handler instance needs to be registered as **both** an UnknownChannelReferenceHandler and a ChannelListener.

### KexExtensionHandler

Provides hooks for implementing [KEX extension negotiation](https://tools.ietf.org/html/rfc8308).

**Note:** it can be used for monitoring the KEX mechanism and intervene in a more general case for other purposes as well. In any case, it is highly recommended though to read the interface documentation and also review the code that invokes it before attempting to use it. An **experimental** implementation example is available for the client side - see DefaultClientKexExtensionHandler.

### ReservedSessionMessagesHandler

Can be used to handle the following cases:

* [SSH\_MSG\_IGNORE](https://tools.ietf.org/html/rfc4253#section-11.2)
* [SSH\_MSG\_DEBUG](https://tools.ietf.org/html/rfc4253#section-11.3)
* [SSH\_MSG\_UNIMPLEMENTED](https://tools.ietf.org/html/rfc4253#section-11.4)
* Implementing a custom session heartbeat mechanism - for **both** [client](./client-setup.md#keeping-the-session-alive-while-no-traffic) or [server](./server-setup.md#providing-server-side-heartbeat).
* Any other unrecognized message received in the session.

**Note:** The handleUnimplementedMessage method serves both for handling SSH\_MSG\_UNIMPLEMENTED and any other unrecognized message received in the session as well.

class MyClientSideReservedSessionMessagesHandler implements ReservedSessionMessagesHandler {  
 @Override  
 public boolean handleUnimplementedMessage(Session session, int cmd, Buffer buffer) throws Exception {  
 switch(cmd) {  
 case MY\_SPECIAL\_CMD1:  
 ....  
 return true;  
 case MY\_SPECIAL\_CMD2:  
 ....  
 return true;  
 default:  
 return false; // send SSH\_MSG\_UNIMPLEMENTED reply if necessary  
 }  
 }  
 }  
  
 // client side  
 SshClient client = SshClient.setupDefaultClient();  
 // This is the default for ALL sessions unless specifically overridden  
 client.setReservedSessionMessagesHandler(new MyClientSideReservedSessionMessagesHandler());  
 // Adding it via a session listener  
 client.setSessionListener(new SessionListener() {  
 @Override  
 public void sessionCreated(Session session) {  
 // Overrides the one set at the client level.  
 if (isSomeSessionOfInterest(session)) {  
 session.setReservedSessionMessagesHandler(new MyClientSessionReservedSessionMessagesHandler(session));  
 }  
 }  
 });  
  
 try (ClientSession session = client.connect(user, host, port).verify(...timeout...).getSession()) {  
 // setting it explicitly  
 session.setReservedSessionMessagesHandler(new MyOtherClientSessionReservedSessionMessagesHandler(session));  
 session.addPasswordIdentity(password);  
 session.auth().verify(...timeout...);  
  
 ...use the session...  
 }  
  
  
 // server side  
 SshServer server = SshServer.setupDefaultServer();  
 // This is the default for ALL sessions unless specifically overridden  
 server.setReservedSessionMessagesHandler(new MyServerSideReservedSessionMessagesHandler());  
 // Adding it via a session listener  
 server.setSessionListener(new SessionListener() {  
 @Override  
 public void sessionCreated(Session session) {  
 // Overrides the one set at the server level.  
 if (isSomeSessionOfInterest(session)) {  
 session.setReservedSessionMessagesHandler(new MyServerSessionReservedSessionMessagesHandler(session));  
 }  
 }  
 });

**NOTE:** Unlike "regular" event listeners, the handler is not cumulative - i.e., setting it overrides the previous instance rather than being accumulated. However, one can use the EventListenerUtils and create a cumulative listener - see how SessionListener or ChannelListener proxies were implemented.

### SessionDisconnectHandler

This handler can be registered in order to monitor session disconnect initiated by the internal code due to various protocol requirements - e.g., unknown service, idle timeout, etc.. In many cases the implementor can intervene and cancel the disconnect by handling the problem somehow and then signaling to the code that there is no longer any need to disconnect. The handler can be registered globally at the SshClient/Server instance or per-session (via a SessionListener).

**NOTE(s):**

* This handler is non-cumulative - i.e., setting it replaces any existing previous handler instance.
* If any exception is thrown from one of the invoked callback methods then session disconnect proceeds as if the handler decided not to intervene.

### SignalListener

Informs about signal requests as described in [RFC 4254 - section 6.9](https://tools.ietf.org/html/rfc4254#section-6.9), "break" requests (sent as SIGINT) as described in [RFC 4335](https://tools.ietf.org/html/rfc4335) and "window-change" (sent as SIGWINCH) requests as described in [RFC 4254 - section 6.7](https://tools.ietf.org/html/rfc4254#section-6.7)