Radboud University Nijmegen

Computer Science

BACHELOR THESIS

Auomatically Learning a Model of the SSH2 Transport Layer

Author:
Mirjam van Nahmen
3004120
m.vannahmen@student.ru.nl

Supervisor:
Erik Poll, Fides Aarts, Joeri de Ruiter
Institute for Computing and Information Sciences
Radboud University Nijmegen

February 19, 2014

Abstract

This thesis is about experiments to automatically learn a finite state model of the SSH Transport Layer of an implementation of SSH. To be more specifically, the learned model is about the server side of the communication during the Transport Layer.
To apply the learning approach is a specific implementation needed, for this paper is for the OpenSSH implementation chosen. But it is just an example. With the resulting program is it possible to check every SSH server.
The description of the SSH protocol is not always precisely how some steps of the protocol should be implemented. The resulting model gives an overview which steps are possible and which answer the server really returns during the conversation. This can show the differences between the OpenSSH implementation and the formal description.
To learn the model the tool "LearnLib" and a modified SSH implementation JSch is used. LearnLib is a tool for automata learning and can automatically learn a model. With a modified SSH client LearnLib is able to send requests to the server and use the answers of the server to create a model. The main work of this project was to modify the SSH client "JSch" in such a way that it can be connected with LearnLib. Some problems during this modification will also be shown. Because LearnLib acts as a client parallelism of SSH is left out of this project.

Contents

1 Introduction

The SSH protocol is one of the oldest and most used secure communication protocols. There are also many different implementations of the SSH server and clients but it is not always possible to see if they are exactly implemented as in the original definitions (the RFC 4250-54[5], [6], [7], [8] and [9]) and sometimes these definitions are not very exactly. For instance sometimes it is necessary to view debug messages to troubleshoot any SSH connection issue, but in the documentation it is allowed to choose if the implementation ignores a message which is labelled with a SSH_MSG_DEBUG, or not.
An exact picture of one implementation of a SSH server can be given by a learned model. In order to build such a model LearnLib is a useful tool, which can use a given alphabet to give a 'System Under Test'(SUT) words as input and receives the response of this SUT. An alphabet contains all possible inputs of the SUT, e.g. connect, "SSH_MSG_KEXINIT", "SSH_MSG_KEXINIT_DH", "SSH_MSG_NEWKYS",... . A word is then a sequence of the alphabet, e.g. ("SSH_MSG_KEXINIT", connect, "SSH_MSG_KEXINIT"). With this information is LearnLib able to create a model which is almost exactly the SUT. To create the words LearnLib uses the L*-algorithm.
In case of the project the SUT the SSH server OpenSSH [1] and with the input alphabet LearnLib is able to control the SSH client which sends requests to the SSH server.

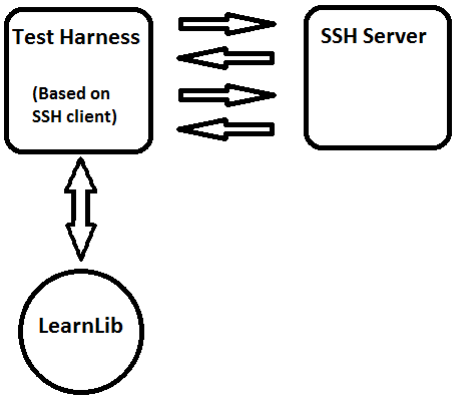


Figure 1: Project structure

Figure 4 shows the structure of this project. On one side is LearnLib which communicates with a Test Harness and the Test Harness communicates with the SSH server. The Test Harness is the modified SSH client. LearnLib uses the input alphabet to say which request should be sent to the server in which order.
Usually the client asks the server to begin a conversation with a connection request followed by a request to get the version number of the server, so that the client can compare the version of the server with the supported versions of the client. The next step of the client should be to ask to begin with a key exchange. But what happens if the client asks directly for a key exchange without a request for the version number? LearnLib forms almost every possible order of the various requests in form of so called words of the input alphabet. A word can be seen as a name of a request. This word will be sent to the Test Harness. The function of the Test Harness is now to translate the names in the word to the function which sends the associated requests to the server. After sending a request the Test Harness should wait for the responses of the server translates the responses of the server into an understandable word for LearnLib and sends it to LearnLib.
Because this is a Bachelor thesis and to delimit this project, only the Transport Layer will be considered.

[1] http://www.openssh.org/

The Transport Layer is the bottom layer of the SSH protocol. More over SSH and the layer see section 2.1 or the very good German article[3].
Nothing further will be said about a model of the exactly architecture of other layers of SSH. This paper also deals only with the security or correctness of the tested implementation on the Transport Layer.
Chapter 2 gives some background information of SSH and the utilized programs. In the first section SSH is described and some detailed information about the Transport Layer is provide. The second section describes the used programs/tools. Chapter 3 describes which preparation needed for the testing. Chapter 4 will describe how the testing was done for this project. There were some problems to run the tests. And in the last two sections the results and the project will be evaluated and the conclusion of this thesis will be drawn.

2 Background

To understand the next steps in this project it is useful to describe first what SSH is, especially the architecture which is described in the [SSHARCH]. Here it will also be described what the result should look like and which programs/tools were used for this project.

2.1 SSH

SSH is a communication protocol that provides secure login, file transfer and TCP/IP connection through an untrusted network. For the authentication it uses a cryptographic authentication, automatic session encryption and integrity protection for transferred data [10].
SSH is specified in five RFCs: RFCs 4250-4524 [5][6][8][7][9]. In these RFCs the common notation is the overall architecture and three sub-protocols described. The sub-protocols are the transport[SSH-TRANS], authentication[SSH-AUTH] and connection[SSH-CONN] layer. The advantage of this is the modularity of the layers. With this the protocol is very flexible and can independently choose if the direction of the communication will negotiate the methods of the key exchange and the encryption. In the following section the layers will be described shortly. However the cryptographic algorithms won't be described here, because these will be negotiated.[3]
The following picture shows the protocols and their relation to each other which are used in a login session. The dashed line shows which component starts which other component. The thin solid lines show the communication for the authentication. The thick solid line shows the data and the lines with gray background are the protocols.[3]

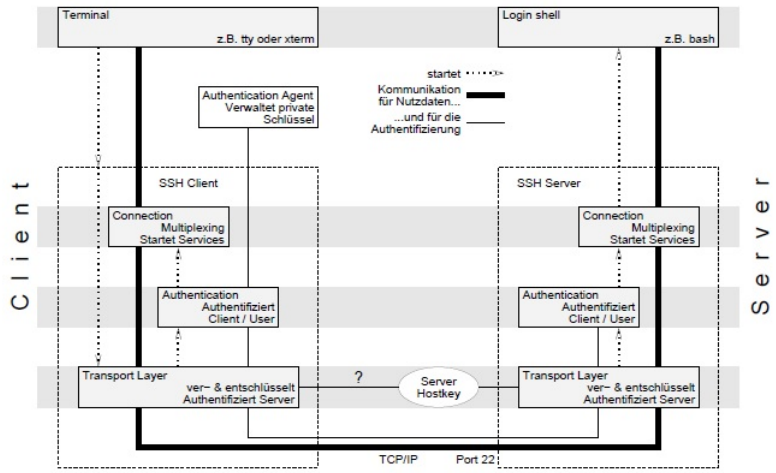


Figure 2: SSH Sub-Protcol Relations[3]

2.1.1 SSH Transport Layer Protocol

The RFC 4253 is the important RFC for this paper. It describes the first sub-protocol **SSH Transport Layer Protocol**. This sub-protocol begins a session, thus in this RFC is described how to set up a connection including creating the session keys, the authentication of the server and it finishes with initialization of the data exchange.[4]
The format of the packets which are used during a session is, with some exceptions the beginning of the Transport Layer Protocol, the so called Binary Packet Protocol. This packet contains one byte for the message number, which determines the type of the message. For the Transport Layer the numbers 1-49

are reserved. [5][4][8]
The Transport Layer Protocol can also be divided into four levels:

1. the protocol identification phase - which version of SSH is run - SSH1 or SSH2
2. the algorithm negotiation phase - which key exchange algorithm is used
3. the key exchange - do the key exchange
4. the service request phase - initialization of the further sub-protocols.

The sequence of the sending and receiving of the packets can be shown as the following:

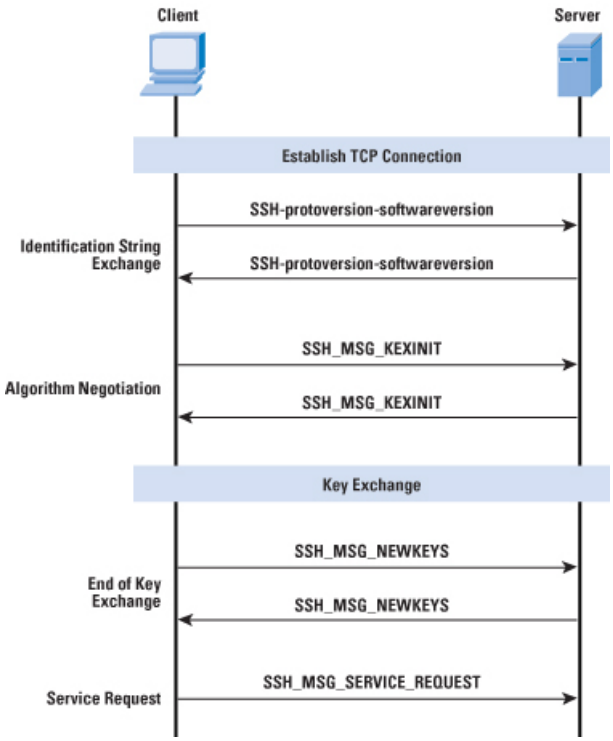


Figure 3: SSH Transport Layer Protocol Packet Exchanges[3]

To establish the connection the client first connects to the server via the port 22. After this, both sends the version string "SSH-${protocol version}-${softwareversion}$< CR >< LF >$", so that there are no misunderstandings by using different versions.
After the version string the data will be sent in the format of Binary Packets Protocol (BDP) which is specified in the [SSH-TRANS][8]. The format of the packets as described in [SSH-TRANS] is this[8]:

$unit32$	$packet\_length$
$byte$	$padding\_length$
$byte[n1]$	$payload; n1 = packet\_length - padding\_length - 1$
$byte[n2]$	$randompadding; n2 = padding\_length$
$byte[m]$	$mac(MessageAuthenticationCode - MAC); m = mac\_length$

The type of the packets are defined of the message numbers. This is defined in the [SSH-NUMBERS][5]. The important numbers for this project are[5]:

MessageID	*Value*
SS_MSG_DISCONNECT	1
SSH_MSG_IGNORE	2
SSH_MSG_UNIMPLEMENTED	3
SSH_MSG_DEBUG	4
SSH_MSG_SERVICE_REQUEST	5
SSH_MSG_SERVICE_ACCEPT	6
SSH_MSG_KEXINIT	20
SSH_MSG_NEWKEYS	21
SSH_MSG_KEXDH_INIT	30
SSH_MSG_KEXDH_REPLY	31

For the key exchange two algorithms can be chosen. The "diffie-hellman-group1-sha1" and "diffie-hellman-group14-sha1". With this algorithms the Transport Layer guarantees the central security objectivs of SSH, the confidentiality and integrity. [4] If the encryption algorithm is found every packet will encrypt with it. [3]

To encrypt and sign, the protocol can use almost any public key format, encoding and algorithm. Currently defined are "ssh-dss - Raw DSS Key", "ssh-rsa - Raw RSA Key", "pgp-sign-rsa - OpenPGP certificates (RSA key)" and "pgp-sign-dss - OpenPGP certificates (DSS key)" [5]

2.2 Programs/Tools

2.2.1 Test harness

The main part of the project is to build the Test Harness which has as main function to send requests to and receive the responds of the server. A Test Harness is a thus a SSH client with some special functions.

Figure 4: Test Harness structure

The conditions of such a client must be:

- The program code must be freely available (i.e. Open Source), otherwise it is not possible to adjust the program in the way, that I can determine with the test harness the contents of the packets.
- Because the SSH1 is not the actual used protocol, and had some disadvantages [2] the program must use at least SSH2 or both SSH1 and SSH2.
- Because LearnLib is written in Java it makes it easier to learn the model if the client and therefore the harness is also written in Java.
- And important for every client implementation is also that it adheres to the RFCs, which define the SSH. So it is representative of the SSH definition.

Anyone who searched for "SSH Java implementation" in google knows, that there are a lot of programs. Most of them are some small projects without good documentation and/or explicitly designed according

to the definitions in the RFCs. After checking in the descriptions and documentations of a lot of them, there were two favorites that fit the conditions. **MidpSSH** and **JSch**.

MidpSSH

MidpSSH is a simple SSH and Telnet client implementation in Java for mobile devices.[2]
It is based on Floyd SSH and Telnet Floyd by Radek Polak [3] The main features are:

- SSH1 and Telnet support, and now SSH2 support,
- Macros for entering frequently typed commands,
- Traffic usage reporting,
- Free and Open Source (GPL).

JSch

JSch is a pure Java implementation of the SSH client protocol.[4] It is a known and often used implementation, which is also used in many IDE(Eclipse, Netbeans...). The features of this implementation:

- standardized on IETF Secure Shell working group,
- SSH1 and SSH2 support,
- Free and Open Source (GPL).

In this project JSch is chosen as client implementation because it is an often used and very common library for the SSH communication in programs written in Java.

The following task was to modify the client to get the possibility to control the order of the requests of the client. This is done by splitting methods which control the communication and catch out the responding packets of the server. For more details see 3.1.

2.2.2 LearnLib

Finally a program is needed which makes a model out of the information which makes the test harness available. A common tool is LearnLib. This framework is made in a project group of the TU Dortmund for Automata learning.
With automata learning it is possible to construct a finite automate, which matches a goal-automate using observation. LearnLib tries to construct a model by trying out operations (in our case sending different types of SSH packets).
LearnLib gives a framework to use different learn algorithms and analyse their characteristics.[5] [1]

[2] http://www.midpssh.org
[3] http://phoenix.inf.upol.cz/~polakr/
[4] http://jcraft.com/jsch/
[5] http://ls5-www.cs.tu-dortmund.de/projects/learnlib/index.php

3 Implementing the test harness

3.1 Harness

As described above the test harness should be used to make it possible to send and receive the packets separately which are not in the normal order. It should also give an interface for the model based testing. This interface should contain the functions to send and receive the packets and also to change some settings of the packets.
So remember the figure 3, for every arrow is at least one function is needed. (For instance sendVersion(); and receiveVersion();)(See Harness.java**??**) But because JSch is a library which uses only the default packet sequence there are no functions to call for only one packet sending or receiving. After a longer and deeper look JSch seems to have two important classes, JSch.java and Session.java. The class Session.java controls the communication. This means that there will be decided which step is next and the needed functions will be called and the needed states will be set. So I made a new class which is similar to the original NewSession.java but with the difference that this class is based on the described interface,(see NewSession.javaA.3). To use this class and interface some changes were needed in the Session.java. First the different steps must be split, so that the sending of the packets can be separately called. This includes also the changing and setting of different states. Because the packets can be sent in a sequence which is not by default, some states must be set by hand to ensure the validity of the packets. Because if the client sends a packet and waits for the answer, the send and receive functions are always in one function call of the Session.java. The output of the functions of the harness should be the message numbers as integer. This is done by hand in the Session.java. Because the packet will be read by a so called socket and put in a buffer, it is very easy to extract the message number from this buffer.
Of course this is not the detailed description of the changes but for a detailed look what had be changed to make this work see (See the java class Session.javaA.2, Harness.java**??** and NewSession.javaA.3)

3.1.1 Problems

Problems during the implementation:
In the beginning there were some problems, how to use the client implementation. This is a library and the small documentation is more intended for software developers who use only the already established methods. It can be seen more as a black box for using a SSH connection. But to implement the harness a deeper look is needed. To understand the inside of the library I had to read the code mostly without any documentation or JavaDoc. So it takes really a lot of time to read and understand the code and to get a detailed impression of the data flow.
For the project it was also interesting to look at how they built a packet. In the beginning of the Session.java stand all needed message numbers, so the first idea was to look at these numbers to find the data flow of the packets. But this was not the easiest way. They work not only with these numbers, often boolean or bytes are used to set a state and write a packet on an abstract level. Afterwards the best strategy to find the data flow and the building way was to look first at the boolean and bytes. Also to set the states on the right way was a little bit tricky. Sometimes informations from the old packet are needed, but if this packet is all corrupt, therefore does not contains the right values, is it impossible to get the needed information. It is not possible to compute the key after a corrupt SSH_MSG_KEXINIT, even if the states are set hard code with the values of a default session.

Results

Manual testing

To understand how to make a state machine or a model and to get a first look at the results, here a model is made by hand. Thus this makes it possible to call several steps in different order manually to simulate different situations and note the result of the states. Because it is almost impossible to check every possible combination of requests and this test is only geared to get a first impression only some different request sequences are tested. First to get a reference the common order of requests is sent(1.), after that it is tried to connect request after request(2.-5.), at last it is tried to send every request two times(6.-9.).
These tests are done by hard coding a sequence of requests in the main.java.
The outputs in the console look like this:
Result with the default sequence:
begin
Start...
Connected...
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1
Sever Version: 2.0
Good Server version(0) otherwise(-1): 0
KexInit1: SSH_MSG_KEXINIT = 20
KexInitDH: SSH_MSG_KEXDH_REPLY = 31
NewKeys: SSH_MSG_NEWKEYS = 21
SSH_MSG_SERVICE_REQUEST_accepted: SSH_MSG_SERVICE_ACCEPT = 6

This program calls every step in the right sequence so that it simulates a normal beginning of a session. But it is not really easy to read. A commonly used notation to describe the communication between a server and a client is to use arrows. Because we are looking from the client side and speak to the server we use → if we send something to the server and ← if we get a message from the server. The important information in every message is the message number, this is the information which is notated here.

The translated results of my handmade testing:
1. Result with the default sequence:
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT

2. Result with two times connect after recieving msg number:
→ connect
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT

← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT

3. Results with connect after Kexinit:

→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT

4. Results with connect after KexInit_DH:

→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT

5. Results connect after NEW_KEYS:

→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ connect
→ SEND_VERSION
← Packet corrupt

6. Result after 2 times kexInit:
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXINIT
← Packet corrupt

7. Result after 2 times kexinit_DH:
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_KEXDH_INIT
← Packet corrupt

8. Result after 2 times Newkeys:
→ connect
→ SEND_VERSION
← VERSION_RECEIVED
→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_NEWKEYS
← Packet corrupt

9. Result after 2 times request:
→ connect
→ SEND_VERSION
← VERSION_RECEIVED

→ SSH_MSG_KEXINIT
← SSH_MSG_KEXINIT
→ SSH_MSG_KEXDH_INIT
← Kex_DH_Reply
→ SSH_MSG_NEWKEYS
← SSH_MSG_NEWKEYS
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT
→ SSH_MSG_SERVICE_REQUEST
← SSH_MSG_SERVICE_ACCEPT

Now it is possible to set the output manually to a model. It is possible to do this manually via every sequence and set the message numbers or strings as state and the function name as step.

Set together manually it gets this result:

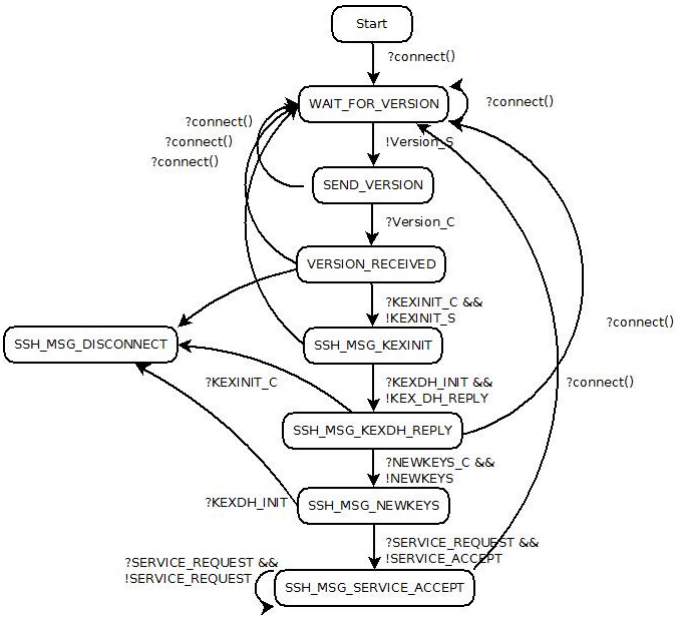


Figure 5: The partial model obtained by manual testing

In some steps the send and received values are together because, there is only the one sequence order possible, otherwise the states are no longer valid. As notation of sent requests of the client the request a has '?', which is comparable with an → in the notation of the results of the manual testing. The '!' is for the other way around which is like the ← above. As one can see here the OpenSSH server goes through the protocol and if a packet from the client is not as expected sends the server a disconnect and closes the session. Only the reconnect, in order to start a new session, is possible at every state, but this is a secure step. And of course the client is allowed to send a SERVICE-REQUEST as often as he wants. But then the keys are defined and negotiated.

This is the hand made model, which allows for the fact that it is not complete. This is a short overview over the standard sequence and what happens if some steps are sent twice. To make this model it took at least three to four hours, thus you need time for this and it contains not every possible step.

Another method could be to test the program with model based testing. During model based testing a system will test if it fullfills the conditions which have been given by a model. The model of the manual testing is not complete enough. It shows only the conditions of the tests cases and there are still a multitude of possible test cases.
A better solution is to make the model automatically use a tool which can test all possible test cases. One tool to learn state machines is LearnLib. LearnLib uses a very straight strategy of two steps. First learn a hypothetical model and then test if this model is correct. If it finds a counter example it uses the counter example to learn a new hypothetical model and test it again. This continues until it has the maximum of test done without finding a counter model.

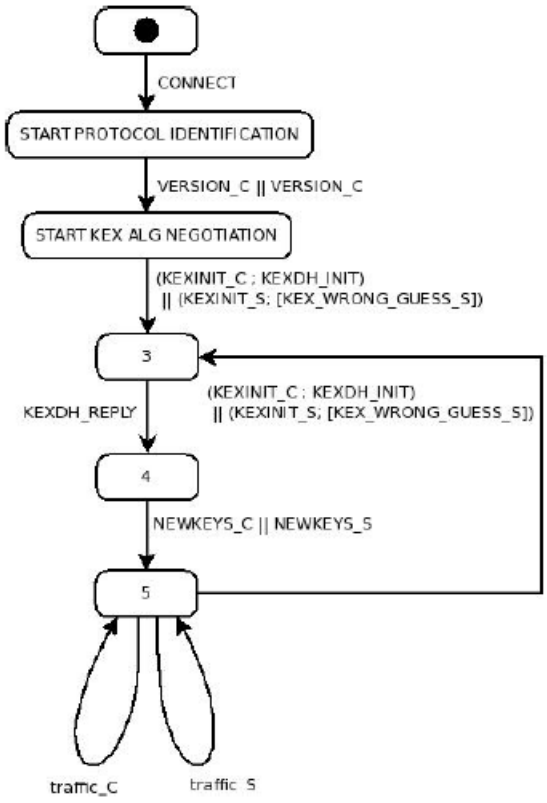


Figure 6: Model made of the RFCs "SSH2 doing Diffie-Hellman key exchange, allowing for guessing of the key exchange algorithm (i.e. KEXDH INIT may be sent before KEXINIT S) and an erroneous initial key exchange packet by the other party (the optional KEX WRONG GUESS S that is ignored)."[4]

Figure 6 shows a model which is made by Erik Poll and Aleksy Schubert of the definition of the SSH, the RFC 4250-4254. Comparing the two models shows, that the default path is correct. This means that a communications in which the usual order of requests work is used, it is the same in both figure 5 and figure 6. Further shows that some steps are not tested at this time. E.g. the step KEXINIT from the state after NEWKEYS. To check every possible state an automatic way should be better.

Automatically Testing

As said in the introduction, LearnLib is instead to sends all possible combinations of steps in the form of an alphabet to the Harness and uses the results to make a model.

LearnLib

LearnLib is a tool to automatically make a model in form of afinite state machine. As mentioned in the end of the section about the hand made model, it uses a two step strategy. First learn a hypothetical model and then test if it is correct.
LearnLib communicates with a so called alphabet, these are words which can be sent to the system under test. But the Test Harness understands only function calls. Thus the alphabet of LearnLib must be translated to the method calls of the Test Harness. This is the task of the Wrapper. The Wrapper receives the words of LearnLib and calls the right methods the Wrapper also receives the responses of the Test Harness and translates them to understandable words for LearnLib.

Wrapper

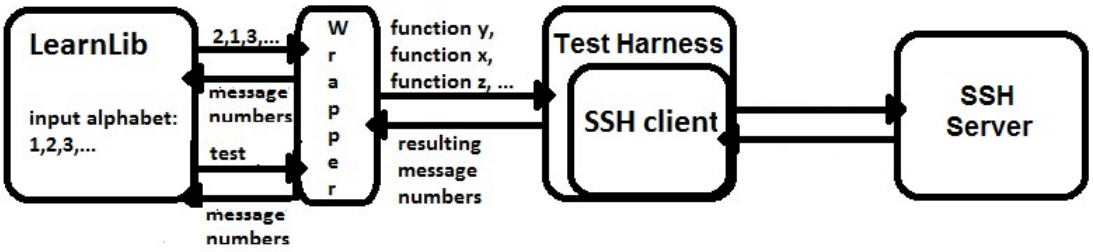
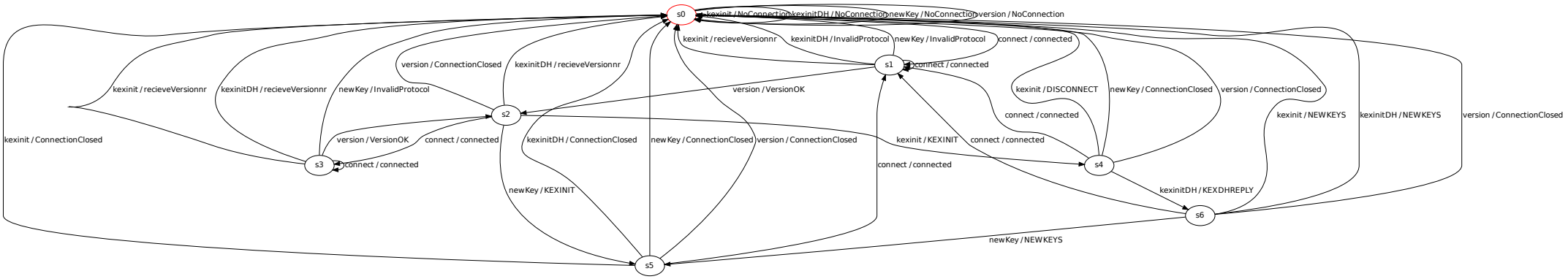


Figure 7: Connection between LearnLib and the SSH client

The wrapper forms the connection between the system under test, the SSH communication, and LearnLib. Therefore it connects to the communication socket and translates the input from LearnLib. LearnLib always gets an alphabet as input. The alphabet contains all possible names for the requests, e.g. "SSH_MSG_KEXINIT", "SSH_MSG_KEXINIT_DH", "SSH_MSG_NEWKYS",... . With this alphabet LearnLib forms "words" for the system under test understands. "Words" mean a sequence of inputs. In our case the system gets only an order of function calls, e.g. ("SSH_MSG_KEXINIT", connect, "SSH_MSG_KEXINIT"). LearnLib understands only Strings, so for every request is the name of the request chosen. After some test it seems that LearnLib has some problems with underscores '_', therefore it is chosen to write the names without them. So one of the tasks of the wrapper is to connect the names to the function calls.

Results

The resulting model of LearnLib looks like:


s0
s1
s2
s3
s4
s5
s6
kexinit / NoConnection
kexinitDH / NoConnection
newKey / NoConnection
version / NoConnection
kexinit / recieveVersionnr
kexinitDH / InvalidProtocol
newKey / InvalidProtocol
connect / connected
version / ConnectionClosed
kexinitDH / recieveVersionnr
connect / connected
kexinit / recieveVersionnr
kexinitDH / recieveVersionnr
newKey / InvalidProtocol
version / VersionOK
kexinit / DISCONNECT
newKey / ConnectionClosed
version / ConnectionClosed
connect / connected
kexinit / NEWKEYS
kexinitDH / NEWKEYS
version / ConnectionClosed
kexinit / ConnectionClosed
version / VersionOK
connect / connected
kexinitDH / ConnectionClosed
newKey / ConnectionClosed
version / ConnectionClosed
kexinit / KEXINIT
connect / connected
connect / connected
kexinitDH / KEXDHREPLY
newKey / KEXINIT
newKey / NEWKEYS

The learned model shows that the default path works fine.

$$
\begin{gathered}
s0 \\
\rightarrow connect/connected \\
s1 \\
\rightarrow version/versionOK \\
s2 \\
\rightarrow kexinit/KEXINIT \\
s4 \\
\rightarrow kexinitDH/KEXDHREPLY \\
s6 \\
\rightarrow newkeys/NEWKEYS \\
s5
\end{gathered}
$$

At every state every possible request is tested. This is an indication that the model is fully tested. Here fully means that only the requests of the transport layer are in the alphabet used.
But it is not only possible to reach the end state $s5$ via the default path; it can also be reached as follows:

$$
\begin{gathered}
s0 \\
\rightarrow connect/connected \\
s1 \\
\rightarrow version/versionOK \\
s2 \\
\rightarrow newkeys/KEXINIT \\
s5
\end{gathered}
$$

This happens because only the SSH Transport Layer is tested and not the entire SSH. A manual test showed, that it is not possible to go further than to state $s5$.
Other than that there is nothing very surprising in the model. If a connect is sent after every state it is possible to continue with a version number otherwise, if a wrong request is sent it goes directly to the begin state or first to an error state and then to the begin state.
During the automatically testing also some problems arose. The main problem was that OpenSSH is seen as BlackBox, which means that there were some unknown behaviour. E.g. after more than 20 connection requests the server blocks the requests. This happens because OpenSSH disconnects after N time requests sending or don't send a request in M seconds. LearnLib has a big problem with this behaviour. Because the responds of the server becomes non-deterministic and for LearnLib it is not possible to make a non-deterministic automata. A solution was to restarts the server during every reset of LearnLib.

Conclusion

The results shows that the Transport layer of the OpenSSH server works good. This is not a really surprising result, but good to know it for sure. Now it is clear that the server works at that part precisely on the RFCs which are the definitions of the SSH.

Evaluation OpenSSH

But still were the tests only on the transport layer protocol, thus they say nothing about the other two sub protocols or the hole communication. Also is parallelism not considered. In the result model can be seen that the OpenSSH server goes through the protocol from state s0 to state s1 to the next state s2 and to state s4 and then to s5 and at the end to s6. But if the client sends during this communicatioin a packet which is not as expected, sends the server an error request and close the session. This means, that the client must again begin in state s1. Only the reconnect, thus to start a new session, is at every state possible, but this is a secure step, because it leaves the encrypted state and go through an plain state. And of course the client is allowed to send a SERVICE-REQUEST as many as he wants. But then are the keys defined and negotiated and the data are already encrypted. But we can based on results in the section before conclude, that the OpenSSH Transport Layer Protocol is pretty correct through the definitions in the [SSH-TANS].

Evaluation Project

For this project the client implementation JSch was a bad choice. The implementation of the session and how the packets were made is not really useful to make the Harness that I wanted. Also the documentation was really not meant for further programming/reprogramming. Because it contains almost no JavaDoc and the other documentation is only for users of the library. A better choice had been to write an own program from scratch, which contains the needed functions of a client. This approach would perhaps need only less than the half of the code which is now used for the client. Also cost it not so much time to understand the packet building, sending and receiving. Which would reduce the needed time also a lot.

Future work

In this project it is only the transport layer protocol tested. A further work would be to test what happens if also requests of the other layers would be tried during the transport layer session. The RFCs over the other layers of the SSH protocol are also not always very clear. It could also be interesting to test these layers with the same sort of model based tests, to show how the server react at every state if he receives not the expected packet.
Maybe it could also be interesting to look what happens if packets will be to the server send, which have not the expected structure. E.g. a packet with a request number for a data packet, but after this it sends only a version string and not the expected data length and data.
The resulting program (Harness and wrapper) can also used to test other SSH server. With this it can be possible to find fingerprints of different servers and thus it is a possibility to identify the server implementation. Maybe some implementations uses some solutions which has no unimplemented parts and thus answers never with an unimplemented code. To find features like this can be used to identify implementations of SSH. This can be helpful to find server with an implementation, which has a known bug or error, to find targets for a hacker attack.

References

[1] Fakultät für Informatik TU Dortmund. Learnlib. `http://ls5-www.cs.tu-dortmund.de/projects/learnlib/index.php`.

[2] Malte Gronau. Sicherheit in vernetzten Systemen - SSH-Secure Shell. *Hauptseminar WS 2004/05, University Siegen*, February 2005. `http://www.bs.fb12.uni-siegen.de/lehre/ws0405/sec/13_Ausarbeitung.pdf`.

[3] Stefan Klinger. Das secure shell protokoll. *The Protocols that Rund the Internet, Fachbereich Informatik, Universität Konstanz*, 2003.

[4] Erik Poll and Aleksy Schubert. Rigorous specifications of the SSH Transport Layer. *ICIS–R11004, Radboud University Nijmegen*, 2011.

[5] T. Ylonen. RFC 4250 - The Secure Shell (SSH) protocol assigned numbers. *Internet Official Protocol Standards*, January 2006.

[6] T. Ylonen. RFC 4251 - The Secure Shell (SSH) Protocol Architecture. *Internet Official Protocol Standards*, January 2006.

[7] T. Ylonen. RFC 4252 - The Secure Shell (SSH) Authentication Protocol. *Internet Official Protocol Standards*, January 2006.

[8] T. Ylonen. RFC 4253 - The Secure Shell (SSH) Transport Layer Protocol. *Internet Official Protocol Standards*, January 2006.

[9] T. Ylonen. RFC 4254 - The Secure Shell (SSH) Connection Protocol. *Internet Official Protocol Standards*, January 2006.

[10] Tatu Ylonen. SSH secure login connections over the internet. June 7, 1996.

A Appendix

A.1 SSHTestService.java

Wrapper for automatic testing

```
 package com.jcraft.jsch;
/**
 * Wrapper for automatic testing
 * @author      Mirjam van Nahmen  <m.vannahmen @ student.ru.nl>
 * @version     1.0
 * @since       2014-01-31
 */
import static com.jcraft.jsch.NewSession.jsch;
import static com.jcraft.jsch.NewSession.kexinitDH;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.net.Socket;
import java.util.*;
import sun.security.ssl.*;

@SuppressWarnings("deprecation")
public class SSHTestService {

  JSch jsch;
  Session session;
  //SSH-Login information
  String hostname = "localhost";
  String UserName = "Miri";
    String Password = "Hallo";
    String HostKeyAlias = "localhost";
    String Host = "127.0.0.1";

  //Booleans to check the previous state
  boolean connected = false;
  boolean versioned = false;
  boolean kexinited = false;
  boolean kexinitedDH = false;
  boolean newKeyed = false;

  //Cache for first-state
  private Session initState;
  private Session connectedState;
  private JSch initJSch;
  private Buffer initBuf;
  //Cache for second-state
  private Session versionedState;
  private Buffer versionBuf;
  private JSch versionJSch;
  //Cache for third-state
  private Session kexInitState;
  private Buffer kexInitBuf;
  private JSch kexInitJSch;
  //Cache for fourth-state
  private Session kexInitDHState;
  private Buffer kexInitDHBuf;
  private JSch kexInitDHJSch;
  private KeyExchange kexInitDHkex; //Key information
  //Cache for fifth-state
  private Session newKeyState;
  private Buffer newKeyBuf;
  private JSch newKeyJSch;


```

```
64  public static SSHTestService createSSHTestService() throws Exception {
65    SSHTestService service = new SSHTestService();
66                return service;
67  }
68
69    // Set up connection and set all caches
70  public SSHTestService() throws Exception {
71            initConnection(); //set up connection
72
73            initJSch = jsch;
74            initState=session;
75            session.connect();
76            byte[] V_C = Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
77            session.version(session.i, session.j,V_C);
78
79            versionJSch = jsch;
80            versionedState = session;
81            versionBuf = session.buf;
82            reset();
83
84  }
85   //set up connection
86  public void initConnection() throws JSchException
87  {
88    jsch = new JSch();
89    session = new Session(jsch);
90    session.setUserName(UserName);
91    session.setPassword(Password);
92    session.setHostKeyAlias(Password);
93    session.setHost(host);
94    java.util.Properties config = new java.util.Properties();
95    config.put("StrictHostKeyChecking", "no");
96    session.setConfig(config);
97    System.out.println("Start...");
98  }
99
100 //Reset server and set new caches and init connection
101 public void reset() throws Exception
102 {
103 //______Restart server_____
104   Process p = Runtime.getRuntime().exec("sudo /home/miri/test.sh");
105   p.waitFor();
106   Thread.sleep(100);
107   BufferedReader reader = new BufferedReader(new InputStreamReader(p.getErrorStream()));
108   String line = reader.readLine();
109
110   System.out.println("restart Server:"+line);
111 //_____________________________
112
113   initConnection();
114 // set caches
115   initJSch = jsch;
116   initState = session;
117   initBuf = session.buf;
118   session.connect();
119   byte[] V_C=Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
120   session.version(session.i, session.j,V_C);
121
122   versionJSch = jsch;
123   versionedState = session;
124   versionBuf = session.buf;
125
126   session.kexinit(0, 0, 0, 0);
127
128   kexInitJSch = jsch;
129   kexInitState = session;
130   kexInitBuf = session.buf;
131
132   session.kexDH_init();
```

```
133
134     kexInitDHJSch = jsch;
135     kexInitDHState = session;
136     kexInitDHBuf = session.buf;
137     kexInitDHkex = session.tmp_kex;
138   //set actual state back to initial state
139     jsch = initJSch;
140     session = initState;
141     session.buf = initBuf;
142
143   //reset all booleans
144     connected = false;
145     versioned = false;
146     kexinited = false;
147     kexinitedDH = false;
148   }
149
150   //send connect request
151   public void connect() throws Exception {
152     session.connect();
153     connected = true;
154   }
155
156   /*
157   * Check the input of LearnLib which request should send
158   * @param  input: input of LearnLib
159   * @return: the name of the request number which where response of the server
160   */
161   public String processSymbol(String input) throws Exception {
162     String inAction = input;
163     System.out.println(inAction);
164
165     //If LearnLib sends the request to send the versionnumber
166     if (inAction.equals("version")) {
167       String outAction;
168       if(connected == false){ //If the server is not connected it is not possible to send a
              request
169         kexinited=false;
170         return "NoConnection";
171             }
172
173       byte[] V_C = Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
174         //send request and store response
175       outAction = session.version(session.i, session.j,V_C);
176       //check response
177             if (outAction == "ConnectionClosed") {
178                  connected=false;
179             }
180       if(outAction == "VersionOK")
181       {
182         versioned=true;
183         versionJSch = jsch;
184         versionedState = session;
185         versionBuf = session.buf;
186         kexinited=false;
187       }
188
189             return outAction.toString();
190
191       //If LearnLib sends the request to send the KEXINIT
192             } else if (inAction.equals("kexinit")) {
193         if(connected == false)
194         {
195           return "NoConnection"; //If the server is not connected it is not possible to
                  send a request
196         }
197         //If the previous state was not the send version, put the actual state of the
              client in the right state
198         if(versioned == false)
```

```
199         {
200           jsch = versionJSch;
201           session = versionedState;
202           session.buf = versionBuf;
203         }
204                 //send request and store response
205                 String outAction;
206             outAction = MSGToString(session.kexinit(0,0,0,0));
207         //prepare the state according to the response
208         if (outAction == "ConnectionClosed") {
209           connected=false;
210         }
211         if(outAction == "KEXINIT")
212         {
213           kexInitJSch = jsch;
214           kexInitState = session;
215           kexInitBuf = session.buf;
216           kexinited = true;
217         }
218         if (outAction == "DISCONNECT")
219         {
220           connected = false;
221         }
222         if(outAction == "recieveVersionnr")
223         {
224           reset();
225         }
226         if(outAction == "NEWKEYS")
227         {
228            reset();
229            versioned = false;
230            kexinited = false;
231
232         }
233         System.out.println(outAction);
234         return outAction;
235       }
236       //If LearnLib sends the request to send the KEXINIT_DH
237       else if (inAction.equals("kexinitDH")) {
238         if(connected == false){
239           return "NoConnection"; //If the server is not connected it is not possible to
                  send a request
240         }
241         //If the previous state was not the KEXINIT, put the actual state of the client in
              the right state
242         if(kexinited == false)
243         {
244           jsch = kexInitJSch;
245           session = kexInitState;
246           session.buf = kexInitBuf;
247         }
248         //send request and store response
249         String outAction;
250             outAction = MSGToString(session.kexDH_init());
251
252         //prepare the state according to the response
253         if (outAction == "ConnectionClosed") {
254           session = new Session(jsch);
255           session.setUserName(UserName);
256           session.setPassword(Password);
257           session.setHostKeyAlias(Password);
258           session.setHost(host);
259           java.util.Properties config = new java.util.Properties();
260           config.put("StrictHostKeyChecking", "no");
261           session.setConfig(config);
262           connect();
263           byte[] V_C=Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
264           session.version(session.i, session.j,V_C);
265           session.kexinit(0,0,0,0);
```

```
266           connected=false;
267           versioned=false;
268           kexinited=false;
269           kexinitedDH=false;
270         }
271         if (outAction == "DISCONNECT")
272         {
273           connected = false;
274         }
275         if(outAction == "recieveVersionnr" || outAction == "InvalidProtocol")
276         {
277           reset();
278         }
279         if(outAction == "KEXDHREPLY")
280         {
281           kexinitedDH = true;
282           kexinited = false;
283           kexInitDHJSch = jsch;
284           kexInitDHState = session;
285           kexInitDHBuf = session.buf;
286           kexInitDHkex = session.tmp_kex;
287
288         }
289         if(outAction == "NEWKEYS")
290         {
291            reset();
292            versioned = false;
293            kexinited = false;
294
295         }
296
297         System.out.println(outAction);
298         return outAction;
299                 }
300
301       //If LearnLib sends the request to send the NEWKEY
302             else if (inAction.equals("newKey")) {
303         if(connected == false){
304                     return "NoConnection"; //If the server is not connected it is not
                            possible to send a request
305                 }
306                  //If the previous state was not the KEXINIT_DH, put the actual state of
                        the client in the right state
307         if(kexinitedDH == false){
308           jsch = kexInitDHJSch;
309           session = kexInitDHState;
310           session.buf = kexInitDHBuf;
311           session.tmp_kex = kexInitDHkex;
312         }
313         //send the request and store the response of the server
314         String outAction;
315         outAction = MSGToString(session.newKey());
316
317         //prepare the state according to the response
318         if (outAction == "ConnectionClosed") {
319           session = new Session(jsch);
320           session.setUserName(UserName);
321           session.setPassword(Password);
322           session.setHostKeyAlias(Password);
323           session.setHost(host);
324           java.util.Properties config = new java.util.Properties();
325           config.put("StrictHostKeyChecking", "no");
326           session.setConfig(config);
327           connect();
328           byte[] V_C=Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
329           session.version(session.i, session.j,V_C);
330           session.kexinit(0,0,0,0);
331           connected=false;
332           versioned=false;
```

```
333         }
334         if (outAction == "DISCONNECT")
335         {
336           connected = false;
337         }
338         if(outAction == "recieveVersionnr" || outAction == "InvalidProtocol")
339         {
340           reset();
341         }
342 
343         if(outAction == "NEWKEYS")
344         {
345           newKeyed = true;
346           kexinitedDH = false;
347           kexinited = false;
348           newKeyJSch = jsch;
349           newKeyState = session;
350           newKeyBuf = session.buf;
351 
352         }
353         System.out.println(outAction);
354         return outAction;
355     }
356     else {
357         System.out.println("Unknown input symbol ...");
358         System.exit(0);
359     }
360 
361     return null;
362   }
363 
364   /**
365   * Transelate the output of the server (the int's which the Session class sends) into the
        names for LearnLib
366   * @param msg: the output of the server send of the session.class
367   * @return name of request(String) without underscores, because LearnLib has problems with
         it.
368   public static String MSGToString(int msg)
369   {
370     switch(msg){
371       case 1: return "DISCONNECT";
372       case 2: return "IGNORE";
373       case 3: return "UNIMPLEMENTED";
374       case 4: return "DEBUG";
375       case 5: return "SERVICEREQUEST";
376       case 6: return "SERVICEACCEPT";
377       case 20: return "KEXINIT";
378       case 21: return "NEWKEYS";
379       case 30: return "KEXDHINIT";
380       case 31: return "KEXDHREPLY";
381       case -1: return "ConnectionClosed";
382       case -5: return "recieveVersionnr";
383       case -67: return "KEYEXCHANGEFAILED";
384       case 46: return "InvalidProtocol";
385       default: return ("somthing else"+ msg);
386     }
387   }
388 
389   public static void main(String[] args) throws Exception {
390   SSHTestService ssh = new SSHTestService();
391   ssh.reset();
392   }
393 }
```

A.2 Session.java

This version of the session.class of the JSch contains splitted function for every request.

```
1 package com.jcraft.jsch;
2 /**
3  * @modified from     Mirjam van Nahmen  <m.vannahmen @ student.ru.nl>
4  * @version         1.0
5  * @since            2014-01-31
6  * This version of the session.class of the JSch contains splitted function for every
       request.
7  */
8
9 import java.io.*;
10 import java.net.*;
11
12 public class Session implements Runnable{
13 // Message numbers of the SSH
14   static final int SSH_MSG_DISCONNECT=                     1;
15   static final int SSH_MSG_IGNORE=                         2;
16   static final int SSH_MSG_UNIMPLEMENTED=                  3;
17   static final int SSH_MSG_DEBUG=                          4;
18   static final int SSH_MSG_SERVICE_REQUEST=                5;
19   static final int SSH_MSG_SERVICE_ACCEPT=                 6;
20   static final int SSH_MSG_KEXINIT=                       20;
21   static final int SSH_MSG_NEWKEYS=                       21;
22   static final int SSH_MSG_KEXDH_INIT=                    30;
23   static final int SSH_MSG_KEXDH_REPLY=                   31;
24   static final int SSH_MSG_KEX_DH_GEX_GROUP=              31;
25   static final int SSH_MSG_KEX_DH_GEX_INIT=               32;
26   static final int SSH_MSG_KEX_DH_GEX_REPLY=              33;
27   static final int SSH_MSG_KEX_DH_GEX_REQUEST=            34;//End TPL
28   static final int SSH_MSG_GLOBAL_REQUEST=                80;
29   static final int SSH_MSG_REQUEST_SUCCESS=               81;
30   static final int SSH_MSG_REQUEST_FAILURE=               82;
31   static final int SSH_MSG_CHANNEL_OPEN=                  90;
32   static final int SSH_MSG_CHANNEL_OPEN_CONFIRMATION=     91;
33   static final int SSH_MSG_CHANNEL_OPEN_FAILURE=          92;
34   static final int SSH_MSG_CHANNEL_WINDOW_ADJUST=         93;
35   static final int SSH_MSG_CHANNEL_DATA=                  94;
36   static final int SSH_MSG_CHANNEL_EXTENDED_DATA=         95;
37   static final int SSH_MSG_CHANNEL_EOF=                   96;
38   static final int SSH_MSG_CHANNEL_CLOSE=                 97;
39   static final int SSH_MSG_CHANNEL_REQUEST=               98;
40   static final int SSH_MSG_CHANNEL_SUCCESS=               99;
41   static final int SSH_MSG_CHANNEL_FAILURE=              100;
42
43   private static final int PACKET_MAX_SIZE = 256 * 1024;
44   //MI: if their was an error MSG
45   private boolean isDisconnect=false;   //if their was a disconnect msg during the buffer
         reading
46   private boolean isDebuged=false; //if their was a debug msg during the buffer reading
47   private boolean isUnimplemented=false; //if their was an unimplemented msg during the
         buffer reading
48   private boolean isIgnored=false; //if their was an ingnor msg during the buffer reading
49
50   private byte[] V_S;                                  //MI: server version
51   private byte[] V_C=Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION); //MI: client version
52
53   private byte[] I_C; // the payload of the client's SSH_MSG_KEXINIT
54   private byte[] I_S; // the payload of the server's SSH_MSG_KEXINIT
55   private byte[] K_S; // the host key
56
57   private byte[] session_id;
58
59   private byte[] IVc2s;
60   private byte[] IVs2c;
61   private byte[] Ec2s;
62   private byte[] Es2c;
63   private byte[] MACc2s;
```

```
64   private byte[] MACs2c;
65
66   private int seqi=0;
67   private int seqo=0;
68
69   String[] guess=null;
70   private Cipher s2ccipher;
71   private Cipher c2scipher;
72   private MAC s2cmac;
73   private MAC c2smac;
74   //MI: private byte[] mac_buf;
75   private byte[] s2cmac_result1;
76   private byte[] s2cmac_result2;
77
78   private Compression deflater;
79   private Compression inflater;
80
81   private IO io;
82   Socket socket;
83   private int timeout=0;
84   int i, j;
85
86   volatile boolean isConnected=false;
87
88   private boolean isAuthed=false;
89
90   private Thread connectThread=null;
91   private Object lock=new Object();
92
93   boolean x11_forwarding=false;
94   boolean agent_forwarding=false;
95
96   InputStream in=null;
97   OutputStream out=null;
98
99   static Random random;
100
101  Buffer buf;
102  Packet packet;
103
104  SocketFactory socket_factory=null;
105
106  static final int buffer_margin = 32 + // maximum padding length
107                                   20 + // maximum mac length
108                                   32;  // margin for deflater; deflater may inflate data
109
110  private java.util.Hashtable config=null;
111
112  Proxy proxy=null;
113  private UserInfo userinfo;
114
115  private String hostKeyAlias=null;
116  private int serverAliveInterval=0;
117  private int serverAliveCountMax=1;
118
119  private IdentityRepository identityRepository = null;
120
121  protected boolean daemon_thread=false;
122
123  private long kex_start_time=0L;
124
125  int max_auth_tries = 6;
126  int auth_failures = 0;
127  //MI: connectionsettings
128  String host="127.0.0.1";
129  int port=22;
130
131  String username="Miri";
132  byte[] password= Util.str2byte("Hallo");
```

```
133
134   JSch jsch;
135
136   Session(JSch jsch) throws JSchException{
137     super();
138     this.jsch=jsch;
139     buf=new Buffer();
140     packet=new Packet(buf);
141   }
142
143   public String connect() throws JSchException, IOException, Exception{
144      return connect(timeout);
145   }
146 int connectTimeout;
147   public String connect(int connectTimeout) throws JSchException, IOException, Exception
148   {
149
150    this.connectTimeout=connectTimeout;
151   /*MI: if(isConnected){
152       throw new JSchException("session is already connected");
153     }*/
154
155     io=new IO();
156     if(random==null){
157       try{
158   Class c=Class.forName(getConfig("random"));
159         random=(Random)(c.newInstance());
160       }
161       catch(Exception e){
162         throw new JSchException(e.toString(), e);
163       }
164     }
165     Packet.setRandom(random);
166
167     if(JSch.getLogger().isEnabled(Logger.INFO)){
168       JSch.getLogger().log(Logger.INFO,
169                            "Connecting to "+host+" port "+port);
170     }
171
172     try {
173
174
175       if(proxy==null){
176         InputStream in;
177         OutputStream out;
178   if(socket_factory==null){
179           socket=Util.createSocket(host, port, connectTimeout);
180     in=socket.getInputStream();
181     out=socket.getOutputStream();
182   }
183   else{
184           socket=socket_factory.createSocket(host, port);
185     in=socket_factory.getInputStream(socket);
186     out=socket_factory.getOutputStream(socket);
187   }
188   //if(timeout>0){ socket.setSoTimeout(timeout); }
189         socket.setTcpNoDelay(true);
190         io.setInputStream(in);
191         io.setOutputStream(out);
192       }
193       else{
194   synchronized(proxy){
195           proxy.connect(socket_factory, host, port, connectTimeout);
196     io.setInputStream(proxy.getInputStream());
197     io.setOutputStream(proxy.getOutputStream());
198           socket=proxy.getSocket();
199   }
200       }
201
```

```
202       if(connectTimeout>0 && socket!=null){
203         socket.setSoTimeout(connectTimeout);
204       }
205
206       isConnected=true;
207
208
209       if(JSch.getLogger().isEnabled(Logger.INFO)){
210         JSch.getLogger().log(Logger.INFO,
211                              "Connection established");
212       }
213
214
215
216     }catch(Exception e){}
217
218     if(isConnected)
219          return "-1";
220       return "-2";
221     }
222
223
224
225   //MI: V_C set the version outside of the class
226     public String version(int i, int j,byte[] V_C) throws IOException, JSchException
227     {
228       jsch.addSession(this);
229       {
230   // Some Cisco devices will miss to read '\n' if it is sent separately.
231   byte[] foo=new byte[V_C.length+1];
232   System.arraycopy(V_C, 0, foo, 0, V_C.length);
233   foo[foo.length-1]=(byte)'\n';
234   io.put(foo, 0, foo.length);
235       }
236
237       while(true){
238         i=0;
239         j=0;
240         while(i<buf.buffer.length){
241           try{
242               j=io.getByte();
243           }catch(SocketException e)
244           {
245               return "ConnectionClosed";
246           }
247         //  System.out.print((char)j);
248           if(j<0)break;
249           buf.buffer[i]=(byte)j; i++;
250           if(j==10)break;
251         }
252
253         if(j<0){
254             return "ConnectionClosed";//new String(new byte[]{buf.getCommand()});
255           // MI: throw new JSchException("connection is closed by foreign host");
256         }
257
258         if(buf.buffer[i-1]==10){    // 0x0a
259           i--;
260           if(i>0 && buf.buffer[i-1]==13){  // 0x0d
261             i--;
262           }
263         }
264
265        if(i<=3 ||
266            ((i!=buf.buffer.length) &&
267             (buf.buffer[0]!='S'||buf.buffer[1]!='S'||
268              buf.buffer[2]!='H'||buf.buffer[3]!='-'))){
269           // It must not start with 'SSH-'
270          // System.err.println(new String(buf.buffer, 0, i));
```

```
271           continue;
272         }
273        //System.out.println("Sever Version: "+ (char)buf.buffer[4]+"."+(char)buf.buffer[6])
               ;
274        if(i==buf.buffer.length ||
275            i<7 ||                                        // SSH-1.99 or SSH-2.0
276            (buf.buffer[4]=='1' && buf.buffer[6]!='9')  // SSH-1.5
277            ){
278             System.out.println("old Server version ");//+(char)buf.buffer[4]+"."+(char)buf.
                   buffer[6]);
279             return "ConnectionClosed";
280           //MI: throw new JSchException("invalid server's version string");
281         }
282       break;
283       }
284
285       V_S=new byte[i]; System.arraycopy(buf.buffer, 0, V_S, 0, i);
286       //System.err.println("V_S: ("+i+") ["+new String(V_S)+"]");
287
288      if(JSch.getLogger().isEnabled(Logger.INFO)){
289         JSch.getLogger().log(Logger.INFO,
290                              "Remote version string: "+Util.byte2str(V_S));
291         JSch.getLogger().log(Logger.INFO,
292                              "Local version string: "+Util.byte2str(V_C));
293       }
294       return "VersionOK";
295     }
296
297   /**
298    * send kexinit with manual settings
299    * @param kex_contant: default=0,only the first=1,empty=2
300    * @param server_host_key_algorithms_contant: default=0,empty=1
301    * @param cipher_c2s_contant: default=0,only the first=1,empty=2
302    * @param cipher_s2c_contant: default=0,only the first=1,empty=2
303    * @return the message number of the server
304    * @throws Exception
305    */
306   int kexinit(int kex_contant,int server_host_key_algorithms_contant, int
           cipher_c2s_contant, int cipher_s2c_contant) throws Exception
307   {
308       send_kexinit( kex_contant,server_host_key_algorithms_contant, cipher_c2s_contant,
               cipher_s2c_contant);
309
310       try{
311           buf=read(buf);
312       }catch(SocketException e)
313       {
314           return -1;//MI: Connection Closed
315       }catch(IOException e)
316             {
317                 return -1; //End of IOStream -> Connection closed
318             }
319
320       //MI: if not versioned server sends verionnr...abfangen und
321       // verwerten
322
323       //System.out.println(strTmp);
324       if(buf.tmpinput!=null)// == "SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1")
325       {   return -5;}
326       //MI: returns the MSG if there is one of the ERROr MSG
327       if(isDisconnect)
328         return 1;//new String(new byte[]{SSH_MSG_DISCONNECT});
329       if(isDebuged)
330         return 4;//new String(new byte[]{SSH_MSG_DEBUG});
331       if(isUnimplemented)
332         return 3;//new String(new byte[]{SSH_MSG_UNIMPLEMENTED});
333       if(isIgnored)
334         return 2;//new String(new byte[]{SSH_MSG_IGNORE});
335
```

```
336       byte tmp = buf.getCommand();
337       if(tmp!=SSH_MSG_KEXINIT){
338         //MI: in_kex=false;
339           System.out.println("no SSH_MSG_KEXINIT recieved");
340   //MI: throw new JSchException("invalid protocol: "+buf.getCommand());
341       }
342
343       if(JSch.getLogger().isEnabled(Logger.INFO)){
344         JSch.getLogger().log(Logger.INFO,
345                              "SSH_MSG_KEXINIT received");
346       }
347
348
349       return buf.getCommand(); //"SSH_MSG_KEXINIT";
350   }
351      KeyExchange tmp_kex;
352
353     public int kexDH_init() throws Exception
354     {
355       //MI: get the algorithm information out of the packet
356       KeyExchangeANDString ks=receive_kexinit(buf);
357       KeyExchange kex=ks.kex;
358       if(ks.tmp!=null)
359       {   System.err.println("kexDH_init "+ks.tmp);
360           return -5;
361
362       }
363       else if(kex!=null){
364         byte tmp;
365         while(true){
366             try{
367                 buf=read(buf);
368             }catch(SocketException e)
369             {
370                 return -1; //Connection closed
371             }catch(IOException e)
372             {
373                 return -1; //End of IOStream -> Connection closed
374             }
375
376           tmp = buf.getCommand();
377           if(kex.getState()==tmp){
378             kex_start_time=System.currentTimeMillis();
379             boolean result=kex.next(buf);
380             if(!result){
381               System.out.println("verify: "+result);
382               in_kex=false;
383               return tmp;
384             }
385           }
386           else{
387             in_kex=false;
388             System.out.println("tmpinput: "+kex.tmpForVersionnr);
389             System.out.println("invalid protocol(kex): "+buf.getCommand());
390             return tmp;
391           }
392           if(kex.getState()==KeyExchange.STATE_END){
393             break;
394           }
395         }
396         tmp_kex=kex;
397         return tmp;
398       }
399       return -10;
400     }
401
402     public int newKey() throws JSchException, Exception
403     {
404       KeyExchange kex = tmp_kex;
```

```
405       try{ checkHost(host, port, kex); }
406       catch(JSchException ee){
407         in_kex=false;
408         throw ee;
409       }
410
411       send_newkeys();
412
413
414       //MI: receive SSH_MSG_NEWKEYS(21)
415        try{
416                  buf=read(buf);
417             }catch(SocketException e)
418             {
419                 return -1; //Connection closed
420             }catch(IOException e)
421             {
422                 return -1; //End of IOStream -> Connection closed
423             }
424
425
426       //MI: temp for buffer
427       byte tmp =buf.getCommand();
428       if(tmp==SSH_MSG_NEWKEYS){
429
430         if(JSch.getLogger().isEnabled(Logger.INFO)){
431           JSch.getLogger().log(Logger.INFO,
432                                "SSH_MSG_NEWKEYS received");
433         }
434
435   receive_newkeys(buf, kex);
436
437       }
438       else{
439         in_kex=false;
440         System.out.println("invalid protocol(newkyes): "+tmp);
441       }
442       return (int)tmp;
443     }
444
445     public int service_accepted() throws JSchException, Exception{
446
447      try{
448
449       try{
450         String s = getConfig("MaxAuthTries");
451         if(s!=null){
452           max_auth_tries = Integer.parseInt(s);
453         }
454       }
455       catch(NumberFormatException e){
456         throw new JSchException("MaxAuthTries: "+getConfig("MaxAuthTries"), e);
457       }
458
459
460       boolean auth=false;
461       boolean auth_cancel=false;
462
463       UserAuth ua=null;
464       try{
465   Class c=Class.forName(getConfig("userauth.none"));
466         ua=(UserAuth)(c.newInstance());
467       }
468       catch(Exception e){
469         throw new JSchException(e.toString(), e);
470       }
471
472       //MI: send SSH_MSG_SERVICE_REQUEST and recieve SSH_MSG_SERVICE_ACCEPT(6)
473       auth=ua.start(this);
```

```
474
475       return ua.service_recieved;
476      /* MI:
477       String cmethods=getConfig("PreferredAuthentications");
478
479       String[] cmethoda=Util.split(cmethods, ",");
480
481       String smethods=null;
482       if(!auth){
483         smethods=((UserAuthNone)ua).getMethods();
484         if(smethods!=null){
485           smethods=smethods.toLowerCase();
486         }
487         else{
488           // methods: publickey,password,keyboard-interactive
489           //smethods="publickey,password,keyboard-interactive";
490           smethods=cmethods;
491         }
492       }
493
494       String[] smethoda=Util.split(smethods, ",");
495
496       int methodi=0;
497
498       loop:
499       while(true){
500
501   while(!auth &&
502         cmethoda!=null && methodi<cmethoda.length){
503
504           String method=cmethoda[methodi++];
505           boolean acceptable=false;
506           for(int k=0; k<smethoda.length; k++){
507             if(smethoda[k].equals(method)){
508               acceptable=true;
509               break;
510             }
511           }
512           if(!acceptable){
513             continue;
514           }
515
516           //System.err.println("  method: "+method);
517
518           if(JSch.getLogger().isEnabled(Logger.INFO)){
519             String str="Authentications that can continue: ";
520             for(int k=methodi-1; k<cmethoda.length; k++){
521               str+=cmethoda[k];
522               if(k+1<cmethoda.length)
523                 str+=",";
524             }
525             JSch.getLogger().log(Logger.INFO,
526                                  str);
527             JSch.getLogger().log(Logger.INFO,
528                                  "Next authentication method: "+method);
529           }
530
531     ua=null;
532           try{
533             Class c=null;
534             if(getConfig("userauth."+method)!=null){
535               c=Class.forName(getConfig("userauth."+method));
536               ua=(UserAuth)(c.newInstance());
537             }
538           }
539           catch(Exception e){
540             if(JSch.getLogger().isEnabled(Logger.WARN)){
541               JSch.getLogger().log(Logger.WARN,
542                                    "failed to load "+method+" method");
```

```
543             }
544           }
545
546     if(ua!=null){
547             auth_cancel=false;
548       try{
549         auth=ua.start(this);
550               if(auth &&
551                  JSch.getLogger().isEnabled(Logger.INFO)){
552                 JSch.getLogger().log(Logger.INFO,
553                                      "Authentication succeeded ("+method+").");
554               }
555       }
556       catch(JSchAuthCancelException ee){
557         auth_cancel=true;
558       }
559       catch(JSchPartialAuthException ee){
560               String tmp = smethods;
561               smethods=ee.getMethods();
562               smethoda=Util.split(smethods, ",");
563               if(!tmp.equals(smethods)){
564                 methodi=0;
565               }
566         //System.err.println("PartialAuth: "+methods);
567         auth_cancel=false;
568         continue loop;
569       }
570       catch(RuntimeException ee){
571         throw ee;
572       }
573       catch(JSchException ee){
574               throw ee;
575       }
576       catch(Exception ee){
577         //System.err.println("ee: "+ee); // SSH_MSG_DISCONNECT: 2 Too many authentication
              failures
578               if(JSch.getLogger().isEnabled(Logger.WARN)){
579                 JSch.getLogger().log(Logger.WARN,
580                                      "an exception during authentication\n"+ee.toString());
581               }
582               break loop;
583       }
584     }
585   }
586         break;
587       }
588
589       if(!auth){
590         if(auth_failures >= max_auth_tries){
591           if(JSch.getLogger().isEnabled(Logger.INFO)){
592             JSch.getLogger().log(Logger.INFO,
593                                  "Login trials exceeds "+max_auth_tries);
594           }
595         }
596         if(auth_cancel)
597           throw new JSchException("Auth cancel");
598         throw new JSchException("Auth fail");
599       }
600
601       if(connectTimeout>0 || timeout>0){
602         socket.setSoTimeout(timeout);
603       }
604
605       isAuthed=true;
606
607       synchronized(lock){
608         if(isConnected){
609           connectThread=new Thread(this);
610           connectThread.setName("Connect thread "+host+" session");
```

```
611           if(daemon_thread){
612             connectThread.setDaemon(daemon_thread);
613           }
614           connectThread.start();
615         }
616         else{
617           //MI: The session has been already down and
618           //MI: we don't have to start new thread.
619         }
620       }//MI: return (int) service_accepted;*/
621     }
622     catch(Exception e) {
623       in_kex=false;
624       if(isConnected){
625   try{
626     packet.reset();
627     buf.putByte((byte)SSH_MSG_DISCONNECT);
628     buf.putInt(3);
629     buf.putString(Util.str2byte(e.toString()));
630     buf.putString(Util.str2byte("en"));
631     write(packet);
632     disconnect();
633           return SSH_MSG_DISCONNECT;
634   }
635   catch(Exception ee){
636   }
637       }
638       isConnected=false;
639       //e.printStackTrace();
640       if(e instanceof RuntimeException) throw (RuntimeException)e;
641       if(e instanceof JSchException) throw (JSchException)e;
642       throw new JSchException("Session.connect: "+e);
643     }
644     finally{
645       Util.bzero(this.password);
646       this.password=null;
647     }
648   }
649
650
651   @SuppressWarnings("empty-statement")
652   private KeyExchangeANDString receive_kexinit(Buffer buf) throws Exception {
653     String tmpinput=new String(buf.buffer,"UTF-8");
654     String tmpForVersionnr = null;
655     if(tmpinput.contains("SSH-"))//"SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1")
656     {
657         tmpForVersionnr = tmpinput;
658         System.out.println("recieve_kexinit"+tmpinput);
659     }
660     else{
661         int j=buf.getInt();
662         if(j!=buf.getLength()){    // packet was compressed and
663           buf.getByte();           // j is the size of deflated packet.
664           I_S=new byte[buf.index-5];
665         }
666         else{
667           I_S=new byte[j-1-buf.getByte()];
668         }
669        System.arraycopy(buf.buffer, buf.s, I_S, 0, I_S.length);
670
671       // if(!in_kex){     // We are in rekeying activated by the remote!
672        //  send_kexinit(0,0,0,0);
673       // }
674
675
676         //MI: guess=KeyExchange.guess(I_S, I_C);
677        // MI: set guess allways on normal session state
678         guess=new String[]{"diffie-hellman-group1-sha1","ssh-rsa","aes128-ctr","aes128-ctr
             ","hmac-md5","hmac-md5","none","none","","",""};
```

```
679         /* System.out.println("guess ");
680          for(int k = 0; k< guess.length;k++)
681              System.out.print("\""+guess[k]+"\""+",");
682          if(guess==null){
683            throw new JSchException("Algorithm negotiation fail");
684          }*/
685
686          /*if(!isAuthed &&
687             (guess[KeyExchange.PROPOSAL_ENC_ALGS_CTOS].equals("none") ||
688              (guess[KeyExchange.PROPOSAL_ENC_ALGS_STOC].equals("none")))){
689            throw new JSchException("NONE Cipher should not be chosen before authentification
                   is successed.");
690          }*/
691       }
692       KeyExchange kex=null;
693       if(tmpForVersionnr==null)
694       {
695           try{
696                 Class c=Class.forName(getConfig(guess[KeyExchange.PROPOSAL_KEX_ALGS]));
697                 kex=(KeyExchange)(c.newInstance());
698               }
699          catch(Exception e){
700                 throw new JSchException(e.toString(), e);
701               }
702
703          kex.init(this, V_S, V_C, I_S, I_C);
704       }
705 //          kex.init(this, V_S, V_C, I_S, I_C);
706 //          kex.tmpForVersionnr=tmpForVersionnr;
707
708       KeyExchangeANDString ks = new KeyExchangeANDString(kex,tmpForVersionnr);
709       return ks;
710     }
711
712     boolean in_kex=false;
713     public void rekey() throws Exception {
714       send_kexinit(0,0,0,0);
715     }
716
717     /**
718      * make the kex_init_packet manually
719      * @param kex_contant: default=0,only the first=1,empty=2
720      * @param server_host_key_algorithms_contant: default=0,empty=1
721      * @param cipher_c2s_contant: default=0,only the first=1,empty=2
722      * @param cipher_s2c_contant
723      * @throws Exception
724      */
725     void send_kexinit(int kex_contant,int server_host_key_algorithms_contant, int
           cipher_c2s_contant, int cipher_s2c_contant) throws Exception {
726
727       //MI: cipher_c2s_contant: default: 0, only the first: 1, empty:2
728       String ciphers2c, cipherc2s;
729       if(cipher_c2s_contant==0){
730           cipherc2s=getConfig("cipher.c2s");//"aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,
                  blowfish-cbc,aes192-cbc,aes256-cbc"
731       }else if(cipher_c2s_contant==1){
732           cipherc2s = "aes128-ctr";
733       }else{
734           cipherc2s="asdf";
735       }
736
737       //MI: cipher_s2c_contant: default: 0, only the first: 1, empty:2
738       if(cipher_s2c_contant==0){
739           ciphers2c=getConfig("cipher.s2c");//"aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,
                  blowfish-cbc,aes192-cbc,aes256-cbc"
740       }else if(cipher_s2c_contant==1){
741           ciphers2c = "aesabc-cbc";
742       }else{
743           ciphers2c="asdf";
```

```
744     }
745
746     String[] not_available_ciphers=checkCiphers(getConfig("CheckCiphers")); //aes256-ctr,
            aes192-ctr,aes128-ctr,aes256-cbc,aes192-cbc,aes128-cbc,3des-ctr,arcfour,arcfour128,
            arcfour256
747
748     if(cipher_c2s_contant==0 && cipher_s2c_contant==0 && not_available_ciphers!=null &&
            not_available_ciphers.length>0){
749       cipherc2s=Util.diffString(cipherc2s, not_available_ciphers);
750       ciphers2c=Util.diffString(ciphers2c, not_available_ciphers);
751       if(cipherc2s==null || ciphers2c==null){
752         throw new JSchException("There are not any available ciphers.");
753       }
754     }
755
756     //MI: kex_algorithms contant: default: 0, only the first: 1, empty:2
757     String kex;
758     if(kex_contant==0){
759         kex=getConfig("kex");//"diffie-hellman-group1-sha1,diffie-hellman-group14-sha1,
                diffie-hellman-group-exchange-sha1"
760         String[] not_available_kexes=checkKexes(getConfig("CheckKexes"));//"diffie-hellman-
                group14-sha1
761         if(not_available_kexes!=null && not_available_kexes.length>0){
762           kex=Util.diffString(kex, not_available_kexes);
763           if(kex==null){
764             throw new JSchException("There are not any available kexes.");
765           }
766         }
767     }else if(kex_contant==1){
768         kex=getConfig("diffie-hellman-group1-sha1");
769     }else{
770         kex=" ";
771     }
772
773     //MI: server_host_key_algorithms_contant: default: 0, empty: 1
774     String server_host_key_algorithms;
775     if(server_host_key_algorithms_contant==0)
776         server_host_key_algorithms=getConfig("server_host_key");
777     else
778         server_host_key_algorithms=" ";
779
780     in_kex=true;
781     kex_start_time=System.currentTimeMillis();
782
783     // byte      SSH_MSG_KEXINIT(20)
784     // byte[16]  cookie (random bytes)
785     // string    kex_algorithms
786     // string    server_host_key_algorithms
787     // string    encryption_algorithms_client_to_server
788     // string    encryption_algorithms_server_to_client
789     // string    mac_algorithms_client_to_server
790     // string    mac_algorithms_server_to_client
791     // string    compression_algorithms_client_to_server
792     // string    compression_algorithms_server_to_client
793     // string    languages_client_to_server
794     // string    languages_server_to_client
795     Buffer buf = new Buffer();                // send_kexinit may be invoked
796     Packet packet = new Packet(buf);          // by user thread.
797     packet.reset();
798     buf.putByte((byte) SSH_MSG_KEXINIT);
799     synchronized(random){
800       random.fill(buf.buffer, buf.index, 16); buf.skip(16);
801     }
802     buf.putString(Util.str2byte(kex));
803     buf.putString(Util.str2byte(server_host_key_algorithms));
804     buf.putString(Util.str2byte(cipherc2s));
805   //set normal state
806             cipherc2s=getConfig("cipher.c2s");//"aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,
                  blowfish-cbc,aes192-cbc,aes256-cbc"
```

```
807     buf.putString(Util.str2byte(ciphers2c));
808   //set normal state
809             ciphers2c=getConfig("cipher.s2c");//"aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,
                    blowfish-cbc,aes192-cbc,aes256-cbc"
810             if(cipher_c2s_contant==0 && cipher_s2c_contant==0 && not_available_ciphers!=
                    null && not_available_ciphers.length>0){
811                  cipherc2s=Util.diffString(cipherc2s, not_available_ciphers);
812                  ciphers2c=Util.diffString(ciphers2c, not_available_ciphers);
813                   if(cipherc2s==null || ciphers2c==null){
814                     throw new JSchException("There are not any available ciphers.");
815                   }
816             }
817     buf.putString(Util.str2byte(getConfig("mac.c2s")));
818     buf.putString(Util.str2byte(getConfig("mac.s2c")));
819     buf.putString(Util.str2byte(getConfig("compression.c2s")));
820     buf.putString(Util.str2byte(getConfig("compression.s2c")));
821     buf.putString(Util.str2byte(getConfig("lang.c2s")));
822     buf.putString(Util.str2byte(getConfig("lang.s2c")));
823     buf.putByte((byte)0);
824     buf.putInt(0);
825
826     buf.setOffSet(5);
827     I_C=new byte[buf.getLength()];
828     buf.getByte(I_C);
829
830     write(packet);
831
832     if(JSch.getLogger().isEnabled(Logger.INFO)){
833       JSch.getLogger().log(Logger.INFO,
834                            "SSH_MSG_KEXINIT sent");
835     }
836
837   }
838
839   private void send_newkeys() throws Exception {
840     // send SSH_MSG_NEWKEYS(21)
841     packet.reset();
842     buf.putByte((byte)SSH_MSG_NEWKEYS);
843     write(packet);
844
845     if(JSch.getLogger().isEnabled(Logger.INFO)){
846       JSch.getLogger().log(Logger.INFO,
847                            "SSH_MSG_NEWKEYS sent");
848     }
849   }
850
851   private void checkHost(String chost, int port, KeyExchange kex) throws JSchException {
852     String shkc=getConfig("StrictHostKeyChecking");
853
854     if(hostKeyAlias!=null){
855       chost=hostKeyAlias;
856     }
857
858     //System.err.println("shkc: "+shkc);
859
860     byte[] K_S=kex.getHostKey();
861     String key_type=kex.getKeyType();
862     String key_fprint=kex.getFingerPrint();
863
864     if(hostKeyAlias==null && port!=22){
865       chost=("["+chost+"]:"+port);
866     }
867
868     HostKeyRepository hkr=jsch.getHostKeyRepository();
869
870     String hkh=getConfig("HashKnownHosts");
871     if(hkh.equals("yes") && (hkr instanceof KnownHosts)){
872       hostkey=((KnownHosts)hkr).createHashedHostKey(chost, K_S);
873     }
```

```
874     else{
875       hostkey=new HostKey(chost, K_S);
876     }
877 
878     int i=0;
879     synchronized(hkr){
880       i=hkr.check(chost, K_S);
881     }
882 
883     boolean insert=false;
884 
885     if((shkc.equals("ask") || shkc.equals("yes")) &&
886        i==HostKeyRepository.CHANGED){
887       String file=null;
888       synchronized(hkr){
889   file=hkr.getKnownHostsRepositoryID();
890       }
891       if(file==null){file="known_hosts";}
892 
893       boolean b=false;
894 
895       if(userinfo!=null){
896         String message=
897 "WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!\n"+
898 "IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!\n"+
899 "Someone could be eavesdropping on you right now (man-in-the-middle attack)!\n"+
900 "It is also possible that the "+key_type+" host key has just been changed.\n"+
901 "The fingerprint for the "+key_type+" key sent by the remote host is\n"+
902 key_fprint+".\n"+
903 "Please contact your system administrator.\n"+
904 "Add correct host key in "+file+" to get rid of this message.";
905 
906         if(shkc.equals("ask")){
907           b=userinfo.promptYesNo(message+
908                                  "\nDo you want to delete the old key and insert the new
                                       key?");
909         }
910         else{  // shkc.equals("yes")
911           userinfo.showMessage(message);
912         }
913       }
914 
915       if(!b){
916         throw new JSchException("HostKey has been changed: "+chost);
917       }
918 
919       synchronized(hkr){
920         hkr.remove(chost,
921                    (key_type.equals("DSA") ? "ssh-dss" : "ssh-rsa"),
922                    null);
923         insert=true;
924       }
925     }
926 
927     if((shkc.equals("ask") || shkc.equals("yes")) &&
928        (i!=HostKeyRepository.OK) && !insert){
929       if(shkc.equals("yes")){
930   throw new JSchException("reject HostKey: "+host);
931       }
932       //System.err.println("finger-print: "+key_fprint);
933       if(userinfo!=null){
934   boolean foo=userinfo.promptYesNo(
935 "The authenticity of host '"+host+"' can't be established.\n"+
936 key_type+" key fingerprint is "+key_fprint+".\n"+
937 "Are you sure you want to continue connecting?"
938             );
939   if(!foo){
940     throw new JSchException("reject HostKey: "+host);
941   }
```

```
942   insert=true;
943       }
944       else{
945   if(i==HostKeyRepository.NOT_INCLUDED)
946     throw new JSchException("UnknownHostKey: "+host+". "+key_type+" key fingerprint is "+
          key_fprint);
947   else
948           throw new JSchException("HostKey has been changed: "+host);
949       }
950     }
951
952     if(shkc.equals("no") &&
953        HostKeyRepository.NOT_INCLUDED==i){
954       insert=true;
955     }
956
957     if(i==HostKeyRepository.OK &&
958        JSch.getLogger().isEnabled(Logger.INFO)){
959       JSch.getLogger().log(Logger.INFO,
960                            "Host '"+host+"' is known and mathces the "+key_type+" host key
                                 ");
961     }
962
963     if(insert &&
964        JSch.getLogger().isEnabled(Logger.WARN)){
965       JSch.getLogger().log(Logger.WARN,
966                            "Permanently added '"+host+"' ("+key_type+") to the list of
                                 known hosts.");
967     }
968
969     if(insert){
970       synchronized(hkr){
971   hkr.add(hostkey, userinfo);
972       }
973     }
974   }
975
976 //public void start(){ (new Thread(this)).start();  }
977
978   public Channel openChannel(String type) throws JSchException{
979     if(!isConnected){
980       throw new JSchException("session is down");
981     }
982     try{
983       Channel channel=Channel.getChannel(type);
984       addChannel(channel);
985       channel.init();
986       return channel;
987     }
988     catch(Exception e){
989       //e.printStackTrace();
990     }
991     return null;
992   }
993
994   // encode will bin invoked in write with synchronization.
995   public void encode(Packet packet) throws Exception{
996 //System.err.println("encode: "+packet.buffer.getCommand());
997 //System.err.println("        "+packet.buffer.index);
998 //if(packet.buffer.getCommand()==96){
999 //Thread.dumpStack();
1000 //}
1001     if(deflater!=null){
1002       compress_len[0]=packet.buffer.index;
1003       packet.buffer.buffer=deflater.compress(packet.buffer.buffer,
1004                                              5, compress_len);
1005       packet.buffer.index=compress_len[0];
1006     }
1007     if(c2scipher!=null){
```

```
1008        //packet.padding(c2scipher.getIVSize());
1009        packet.padding(c2scipher_size);
1010        int pad=packet.buffer.buffer[4];
1011        synchronized(random){
1012    random.fill(packet.buffer.buffer, packet.buffer.index-pad, pad);
1013        }
1014      }
1015      else{
1016        packet.padding(8);
1017      }
1018
1019      if(c2smac!=null){
1020        c2smac.update(seqo);
1021        c2smac.update(packet.buffer.buffer, 0, packet.buffer.index);
1022        c2smac.doFinal(packet.buffer.buffer, packet.buffer.index);
1023      }
1024      if(c2scipher!=null){
1025        byte[] buf=packet.buffer.buffer;
1026        c2scipher.update(buf, 0, packet.buffer.index, buf, 0);
1027      }
1028      if(c2smac!=null){
1029        packet.buffer.skip(c2smac.getBlockSize());
1030      }
1031    }
1032
1033    int[] uncompress_len=new int[1];
1034    int[] compress_len=new int[1];
1035
1036    private int s2ccipher_size=8;
1037    private int c2scipher_size=8;
1038    public Buffer read(Buffer buf) throws Exception{
1039      int j=0;
1040
1041    isDisconnect=false;
1042    isDebuged=false;
1043    isUnimplemented=false;
1044    isIgnored=false;
1045    String tmpinput;
1046      while(true){
1047        buf.reset();
1048    //    System.out.println(buf.buffer.toString()+" index "+ buf.index + " size "+
          s2ccipher_size+ " read "+new String(buf.buffer,"UTF-8"));
1049
1050        io.getByte(buf.buffer, buf.index, s2ccipher_size);
1051        buf.index+=s2ccipher_size;
1052        if(s2ccipher!=null){
1053          s2ccipher.update(buf.buffer, 0, s2ccipher_size, buf.buffer, 0);
1054        }
1055        j=((buf.buffer[0]<<24)&0xff000000)|
1056          ((buf.buffer[1]<<16)&0x00ff0000)|
1057          ((buf.buffer[2]<< 8)&0x0000ff00)|
1058          ((buf.buffer[3]    )&0x000000ff);
1059
1060        buf.tmpinput=null;
1061        //Mi: check if versionnr is send(connect-> kexinit(without version)
1062        tmpinput=new String(buf.buffer,"UTF-8");
1063        if(tmpinput.contains("SSH-"))//"SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1.1")
1064        {
1065              buf.tmpinput=tmpinput;
1066              break;
1067        }
1068
1069       // System.out.println(new String(buf.buffer,"UTF-8"));
1070
1071        // RFC 4253 6.1. Maximum Packet Length
1072        if(j<5 || j>PACKET_MAX_SIZE){
1073          start_discard(buf, s2ccipher, s2cmac, j, PACKET_MAX_SIZE);
1074        }
1075        int need = j+4-s2ccipher_size;
```

```
1076        System.out.println("index: "+buf.index+" j: "+j+" s2ccipher_size: "+s2ccipher_size);
1077       //if(need<0){
1078       //  throw new IOException("invalid data");
1079       //}
1080      System.out.println("bufferbyte: "+ buf.buffer[0]);// buffer: "+new String(buf.buffer,"
              UTF-8"));
1081
1082       if((buf.index+need)>buf.buffer.length){
1083         byte[] foo=new byte[buf.index+need];
1084         System.arraycopy(buf.buffer, 0, foo, 0, buf.index);
1085         buf.buffer=foo;
1086       }
1087
1088       if((need%s2ccipher_size)!=0){
1089         String message="Bad packet length "+need;
1090         if(JSch.getLogger().isEnabled(Logger.FATAL)){
1091           JSch.getLogger().log(Logger.FATAL, message);
1092         }
1093         start_discard(buf, s2ccipher, s2cmac, j, PACKET_MAX_SIZE-s2ccipher_size);
1094       }
1095
1096       if(need>0){
1097   io.getByte(buf.buffer, buf.index, need); buf.index+=(need);
1098   if(s2ccipher!=null){
1099     s2ccipher.update(buf.buffer, s2ccipher_size, need, buf.buffer, s2ccipher_size);
1100   }
1101       }
1102
1103       if(s2cmac!=null){
1104   s2cmac.update(seqi);
1105   s2cmac.update(buf.buffer, 0, buf.index);
1106
1107         s2cmac.doFinal(s2cmac_result1, 0);
1108   io.getByte(s2cmac_result2, 0, s2cmac_result2.length);
1109         if(!java.util.Arrays.equals(s2cmac_result1, s2cmac_result2)){
1110           if(need > PACKET_MAX_SIZE){
1111             throw new IOException("MAC Error");
1112           }
1113           start_discard(buf, s2ccipher, s2cmac, j, PACKET_MAX_SIZE-need);
1114           continue;
1115   }
1116       }
1117
1118       seqi++;
1119
1120       if(inflater!=null){
1121         //inflater.uncompress(buf);
1122   int pad=buf.buffer[4];
1123   uncompress_len[0]=buf.index-5-pad;
1124   byte[] foo=inflater.uncompress(buf.buffer, 5, uncompress_len);
1125   if(foo!=null){
1126     buf.buffer=foo;
1127     buf.index=5+uncompress_len[0];
1128   }
1129   else{
1130     System.err.println("fail in inflater");
1131     break;
1132   }
1133       }
1134
1135       int type=buf.getCommand()&0xff;
1136       //System.err.println("read: "+type);
1137       if(type==SSH_MSG_DISCONNECT){
1138         buf.rewind();
1139         buf.getInt();buf.getShort();
1140   int reason_code=buf.getInt();
1141   byte[] description=buf.getString();
1142   byte[] language_tag=buf.getString();
1143   //throw new JSchException("SSH_MSG_DISCONNECT: "+
```

```
1144   //          reason_code+
1145   //      " "+Util.byte2str(description)+
1146   //      " "+Util.byte2str(language_tag));
1147   isDisconnect=true;
1148         break;
1149       }
1150       else if(type==SSH_MSG_IGNORE){
1151           isIgnored=true;
1152       }
1153       else if(type==SSH_MSG_UNIMPLEMENTED){
1154         buf.rewind();
1155         buf.getInt();buf.getShort();
1156   int reason_id=buf.getInt();
1157         if(JSch.getLogger().isEnabled(Logger.INFO)){
1158           JSch.getLogger().log(Logger.INFO,
1159                                "Received SSH_MSG_UNIMPLEMENTED for "+reason_id);
1160         }
1161         isUnimplemented=true;
1162       }
1163       else if(type==SSH_MSG_DEBUG){
1164         buf.rewind();
1165         buf.getInt();buf.getShort();
1166         isDebuged=true;
1167 /*
1168   byte always_display=(byte)buf.getByte();
1169   byte[] message=buf.getString();
1170   byte[] language_tag=buf.getString();
1171   System.err.println("SSH_MSG_DEBUG:"+
1172          " "+Util.byte2str(message)+
1173          " "+Util.byte2str(language_tag));
1174 */
1175       }
1176       else if(type==SSH_MSG_CHANNEL_WINDOW_ADJUST){
1177           buf.rewind();
1178           buf.getInt();buf.getShort();
1179     Channel c=Channel.getChannel(buf.getInt(), this);
1180     if(c==null){
1181     }
1182     else{
1183       c.addRemoteWindowSize(buf.getInt());
1184     }
1185       }
1186       else if(type==UserAuth.SSH_MSG_USERAUTH_SUCCESS){
1187         isAuthed=true;
1188         if(inflater==null && deflater==null){
1189           String method;
1190           method=guess[KeyExchange.PROPOSAL_COMP_ALGS_CTOS];
1191           initDeflater(method);
1192           method=guess[KeyExchange.PROPOSAL_COMP_ALGS_STOC];
1193           initInflater(method);
1194         }
1195         break;
1196       }
1197       else{
1198         break;
1199       }
1200     }
1201     buf.rewind();
1202
1203     //Mi: check tmpinput
1204     /*
1205     if(tmpinput!=null)
1206     {
1207         buf = new Buffer();
1208         byte[] tmp=tmpinput.getBytes();
1209         buf.buffer=tmp;
1210     }*/
1211     return buf;
1212   }
```

```
1213
1214  private byte start_discard(Buffer buf, Cipher cipher, MAC mac,
1215                             int packet_length, int discard) throws JSchException,
                                       IOException{
1216    MAC discard_mac = null;
1217
1218    if(cipher==null||!cipher.isCBC()){
1219        return buf.getCommand();// throw new JSchException("Packet corrupt");
1220    }
1221
1222    if(packet_length!=PACKET_MAX_SIZE && mac != null){
1223      discard_mac = mac;
1224    }
1225
1226    discard -= buf.index;
1227
1228    while(discard>0){
1229      buf.reset();
1230      int len = discard>buf.buffer.length ? buf.buffer.length : discard;
1231      io.getByte(buf.buffer, 0, len);
1232      if(discard_mac!=null){
1233        discard_mac.update(buf.buffer, 0, len);
1234      }
1235      discard -= len;
1236    }
1237
1238    if(discard_mac!=null){
1239      discard_mac.doFinal(buf.buffer, 0);
1240    }
1241
1242    throw new JSchException("Packet corrupt");
1243  }
1244
1245  byte[] getSessionId(){
1246    return session_id;
1247  }
1248
1249  private void receive_newkeys(Buffer buf, KeyExchange kex) throws Exception {
1250    updateKeys(kex);
1251    in_kex=false;
1252  }
1253  private void updateKeys(KeyExchange kex) throws Exception{
1254    byte[] K=kex.getK();
1255    byte[] H=kex.getH();
1256    HASH hash=kex.getHash();
1257
1258    if(session_id==null){
1259      session_id=new byte[H.length];
1260      System.arraycopy(H, 0, session_id, 0, H.length);
1261    }
1262
1263    /*
1264      Initial IV client to server:     HASH (K || H || "A" || session_id)
1265      Initial IV server to client:     HASH (K || H || "B" || session_id)
1266      Encryption key client to server: HASH (K || H || "C" || session_id)
1267      Encryption key server to client: HASH (K || H || "D" || session_id)
1268      Integrity key client to server:  HASH (K || H || "E" || session_id)
1269      Integrity key server to client:  HASH (K || H || "F" || session_id)
1270    */
1271
1272    buf.reset();
1273    buf.putMPInt(K);
1274    buf.putByte(H);
1275    buf.putByte((byte)0x41);
1276    buf.putByte(session_id);
1277    hash.update(buf.buffer, 0, buf.index);
1278    IVc2s=hash.digest();
1279
1280    int j=buf.index-session_id.length-1;
```

```
1281
1282      buf.buffer[j]++;
1283      hash.update(buf.buffer, 0, buf.index);
1284      IVs2c=hash.digest();
1285
1286      buf.buffer[j]++;
1287      hash.update(buf.buffer, 0, buf.index);
1288      Ec2s=hash.digest();
1289
1290      buf.buffer[j]++;
1291      hash.update(buf.buffer, 0, buf.index);
1292      Es2c=hash.digest();
1293
1294      buf.buffer[j]++;
1295      hash.update(buf.buffer, 0, buf.index);
1296      MACc2s=hash.digest();
1297
1298      buf.buffer[j]++;
1299      hash.update(buf.buffer, 0, buf.index);
1300      MACs2c=hash.digest();
1301
1302      try{
1303        Class c;
1304        String method;
1305
1306        method=guess[KeyExchange.PROPOSAL_ENC_ALGS_STOC];
1307        c=Class.forName(getConfig(method));
1308        s2ccipher=(Cipher)(c.newInstance());
1309        while(s2ccipher.getBlockSize()>Es2c.length){
1310          buf.reset();
1311          buf.putMPInt(K);
1312          buf.putByte(H);
1313          buf.putByte(Es2c);
1314          hash.update(buf.buffer, 0, buf.index);
1315          byte[] foo=hash.digest();
1316          byte[] bar=new byte[Es2c.length+foo.length];
1317    System.arraycopy(Es2c, 0, bar, 0, Es2c.length);
1318    System.arraycopy(foo, 0, bar, Es2c.length, foo.length);
1319    Es2c=bar;
1320        }
1321        s2ccipher.init(Cipher.DECRYPT_MODE, Es2c, IVs2c);
1322        s2ccipher_size=s2ccipher.getIVSize();
1323
1324        method=guess[KeyExchange.PROPOSAL_MAC_ALGS_STOC];
1325        c=Class.forName(getConfig(method));
1326        s2cmac=(MAC)(c.newInstance());
1327        MACs2c = expandKey(buf, K, H, MACs2c, hash, s2cmac.getBlockSize());
1328        s2cmac.init(MACs2c);
1329        //mac_buf=new byte[s2cmac.getBlockSize()];
1330        s2cmac_result1=new byte[s2cmac.getBlockSize()];
1331        s2cmac_result2=new byte[s2cmac.getBlockSize()];
1332
1333        method=guess[KeyExchange.PROPOSAL_ENC_ALGS_CTOS];
1334        c=Class.forName(getConfig(method));
1335        c2scipher=(Cipher)(c.newInstance());
1336        while(c2scipher.getBlockSize()>Ec2s.length){
1337          buf.reset();
1338          buf.putMPInt(K);
1339          buf.putByte(H);
1340          buf.putByte(Ec2s);
1341          hash.update(buf.buffer, 0, buf.index);
1342          byte[] foo=hash.digest();
1343          byte[] bar=new byte[Ec2s.length+foo.length];
1344    System.arraycopy(Ec2s, 0, bar, 0, Ec2s.length);
1345    System.arraycopy(foo, 0, bar, Ec2s.length, foo.length);
1346    Ec2s=bar;
1347        }
1348        c2scipher.init(Cipher.ENCRYPT_MODE, Ec2s, IVc2s);
1349        c2scipher_size=c2scipher.getIVSize();
```

```
1350
1351        method=guess[KeyExchange.PROPOSAL_MAC_ALGS_CTOS];
1352        c=Class.forName(getConfig(method));
1353        c2smac=(MAC)(c.newInstance());
1354        MACc2s = expandKey(buf, K, H, MACc2s, hash, c2smac.getBlockSize());
1355        c2smac.init(MACc2s);
1356
1357        method=guess[KeyExchange.PROPOSAL_COMP_ALGS_CTOS];
1358        initDeflater(method);
1359
1360        method=guess[KeyExchange.PROPOSAL_COMP_ALGS_STOC];
1361        initInflater(method);
1362      }
1363      catch(Exception e){
1364        if(e instanceof JSchException)
1365          throw e;
1366        throw new JSchException(e.toString(), e);
1367        //System.err.println("updatekeys: "+e);
1368      }
1369    }
1370
1371
1372    /*
1373     * RFC 4253  7.2. Output from Key Exchange
1374     * If the key length needed is longer than the output of the HASH, the
1375     * key is extended by computing HASH of the concatenation of K and H and
1376     * the entire key so far, and appending the resulting bytes (as many as
1377     * HASH generates) to the key.  This process is repeated until enough
1378     * key material is available; the key is taken from the beginning of
1379     * this value.  In other words:
1380     *   K1 = HASH(K || H || X || session_id)   (X is e.g., "A")
1381     *   K2 = HASH(K || H || K1)
1382     *   K3 = HASH(K || H || K1 || K2)
1383     *   ...
1384     *   key = K1 || K2 || K3 || ...
1385     */
1386    private byte[] expandKey(Buffer buf, byte[] K, byte[] H, byte[] key,
1387                             HASH hash, int required_length) throws Exception {
1388      byte[] result = key;
1389      int size = hash.getBlockSize();
1390      while(result.length < required_length){
1391        buf.reset();
1392        buf.putMPInt(K);
1393        buf.putByte(H);
1394        buf.putByte(result);
1395        hash.update(buf.buffer, 0, buf.index);
1396        byte[] tmp = new byte[result.length+size];
1397        System.arraycopy(result, 0, tmp, 0, result.length);
1398        System.arraycopy(hash.digest(), 0, tmp, result.length, size);
1399        Util.bzero(result);
1400        result = tmp;
1401      }
1402      return result;
1403    }
1404
1405    /*public*/ /*synchronized*/ void write(Packet packet, Channel c, int length) throws
         Exception{
1406      long t = getTimeout();
1407      while(true){
1408        if(in_kex){
1409          if(t>0L && (System.currentTimeMillis()-kex_start_time)>t){
1410            throw new JSchException("timeout in wating for rekeying process.");
1411          }
1412          try{Thread.sleep(10);}
1413          catch(java.lang.InterruptedException e){};
1414          continue;
1415        }
1416        synchronized(c){
1417
```

```
1418          if(c.rwsize<length){
1419            try{
1420              c.notifyme++;
1421              c.wait(100);
1422            }
1423            catch(java.lang.InterruptedException e){
1424            }
1425            finally{
1426              c.notifyme--;
1427            }
1428          }
1429
1430          if(c.rwsize>=length){
1431            c.rwsize-=length;
1432            break;
1433          }
1434
1435        }
1436        if(c.close || !c.isConnected()){
1437    throw new IOException("channel is broken");
1438        }
1439
1440        boolean sendit=false;
1441        int s=0;
1442        byte command=0;
1443        int recipient=-1;
1444        synchronized(c){
1445    if(c.rwsize>0){
1446      long len=c.rwsize;
1447            if(len>length){
1448              len=length;
1449            }
1450            if(len!=length){
1451              s=packet.shift((int)len,
1452                             (c2scipher!=null ? c2scipher_size : 8),
1453                             (c2smac!=null ? c2smac.getBlockSize() : 0));
1454            }
1455      command=packet.buffer.getCommand();
1456      recipient=c.getRecipient();
1457      length-=len;
1458      c.rwsize-=len;
1459      sendit=true;
1460    }
1461        }
1462        if(sendit){
1463    _write(packet);
1464          if(length==0){
1465            return;
1466          }
1467    packet.unshift(command, recipient, s, length);
1468        }
1469
1470        synchronized(c){
1471          if(in_kex){
1472            continue;
1473          }
1474          if(c.rwsize>=length){
1475            c.rwsize-=length;
1476            break;
1477          }
1478        }
1479      }
1480      _write(packet);
1481    }
1482
1483    public void write(Packet packet) throws Exception{
1484      // System.err.println("in_kex="+in_kex+" "+(packet.buffer.getCommand()));
1485      long t = getTimeout();
1486      while(in_kex){
```

```
1487        if(t>0L && (System.currentTimeMillis()-kex_start_time)>t){
1488          throw new JSchException("timeout in wating for rekeying process.");
1489        }
1490        //MI: without TPL
1491        byte command=packet.buffer.getCommand();
1492        //System.err.println("command: "+command);
1493        if(command==SSH_MSG_KEXINIT ||
1494           command==SSH_MSG_NEWKEYS ||
1495           command==SSH_MSG_KEXDH_INIT ||
1496           command==SSH_MSG_KEXDH_REPLY ||
1497           command==SSH_MSG_KEX_DH_GEX_GROUP ||
1498           command==SSH_MSG_KEX_DH_GEX_INIT ||
1499           command==SSH_MSG_KEX_DH_GEX_REPLY ||
1500           command==SSH_MSG_KEX_DH_GEX_REQUEST ||
1501           command==SSH_MSG_DISCONNECT){
1502          break;
1503        }
1504        try{Thread.sleep(10);}
1505        catch(java.lang.InterruptedException e){};
1506      }
1507      _write(packet);
1508    }
1509
1510    private void _write(Packet packet) throws Exception{
1511      synchronized(lock){
1512        encode(packet);
1513        if(io!=null){
1514          io.put(packet);
1515          seqo++;
1516        }
1517      }
1518    }
1519
1520    Runnable thread;
1521    public void run(){
1522      thread=this;
1523
1524      byte[] foo;
1525      Buffer buf=new Buffer();
1526      Packet packet=new Packet(buf);
1527      int i=0;
1528      Channel channel;
1529      int[] start=new int[1];
1530      int[] length=new int[1];
1531      KeyExchange kex=null;
1532
1533      int stimeout=0;
1534      try{
1535        while(isConnected &&
1536        thread!=null){
1537          try{
1538            buf=read(buf);
1539            stimeout=0;
1540          }
1541          catch(InterruptedIOException/*SocketTimeoutException*/ ee){
1542            if(!in_kex && stimeout<serverAliveCountMax){
1543              sendKeepAliveMsg();
1544              stimeout++;
1545              continue;
1546            }
1547            else if(in_kex && stimeout<serverAliveCountMax){
1548              stimeout++;
1549              continue;
1550            }
1551            throw ee;
1552          }
1553
1554    int msgType=buf.getCommand()&0xff;
1555
```

```
1556   if(kex!=null && kex.getState()==msgType){
1557           kex_start_time=System.currentTimeMillis();
1558     boolean result=kex.next(buf);
1559     if(!result){
1560       throw new JSchException("verify: "+result);
1561     }
1562     continue;
1563   }
1564 
1565         switch(msgType){
1566   case SSH_MSG_KEXINIT:
1567 //System.err.println("KEXINIT");
1568     KeyExchangeANDString ks=receive_kexinit(buf);
1569           kex=ks.kex;
1570     break;
1571 
1572   case SSH_MSG_NEWKEYS:
1573 //System.err.println("NEWKEYS");
1574           send_newkeys();
1575     receive_newkeys(buf, kex);
1576     kex=null;
1577     break;
1578 
1579   case SSH_MSG_CHANNEL_DATA:
1580           buf.getInt();
1581           buf.getByte();
1582           buf.getByte();
1583           i=buf.getInt();
1584     channel=Channel.getChannel(i, this);
1585     foo=buf.getString(start, length);
1586     if(channel==null){
1587       break;
1588     }
1589 
1590           if(length[0]==0){
1591       break;
1592           }
1593 
1594 try{
1595     channel.write(foo, start[0], length[0]);
1596 }
1597 catch(Exception e){
1598 //System.err.println(e);
1599   try{channel.disconnect();}catch(Exception ee){}
1600 break;
1601 }
1602     int len=length[0];
1603     channel.setLocalWindowSize(channel.lwsize-len);
1604     if(channel.lwsize<channel.lwsize_max/2){
1605             packet.reset();
1606       buf.putByte((byte)SSH_MSG_CHANNEL_WINDOW_ADJUST);
1607       buf.putInt(channel.getRecipient());
1608       buf.putInt(channel.lwsize_max-channel.lwsize);
1609             synchronized(channel){
1610               if(!channel.close)
1611                 write(packet);
1612             }
1613       channel.setLocalWindowSize(channel.lwsize_max);
1614     }
1615     break;
1616 
1617         case SSH_MSG_CHANNEL_EXTENDED_DATA:
1618           buf.getInt();
1619     buf.getShort();
1620     i=buf.getInt();
1621     channel=Channel.getChannel(i, this);
1622     buf.getInt();                    // data_type_code == 1
1623     foo=buf.getString(start, length);
1624     //System.err.println("stderr: "+new String(foo,start[0],length[0]));
```

```
1625      if(channel==null){
1626        break;
1627      }
1628
1629           if(length[0]==0){
1630        break;
1631           }
1632
1633      channel.write_ext(foo, start[0], length[0]);
1634
1635      len=length[0];
1636      channel.setLocalWindowSize(channel.lwsize-len);
1637      if(channel.lwsize<channel.lwsize_max/2){
1638              packet.reset();
1639        buf.putByte((byte)SSH_MSG_CHANNEL_WINDOW_ADJUST);
1640        buf.putInt(channel.getRecipient());
1641        buf.putInt(channel.lwsize_max-channel.lwsize);
1642              synchronized(channel){
1643                if(!channel.close)
1644                  write(packet);
1645              }
1646        channel.setLocalWindowSize(channel.lwsize_max);
1647      }
1648      break;
1649
1650    case SSH_MSG_CHANNEL_WINDOW_ADJUST:
1651           buf.getInt();
1652      buf.getShort();
1653      i=buf.getInt();
1654      channel=Channel.getChannel(i, this);
1655      if(channel==null){
1656        break;
1657      }
1658      channel.addRemoteWindowSize(buf.getInt());
1659      break;
1660
1661    case SSH_MSG_CHANNEL_EOF:
1662           buf.getInt();
1663           buf.getShort();
1664           i=buf.getInt();
1665      channel=Channel.getChannel(i, this);
1666      if(channel!=null){
1667        //channel.eof_remote=true;
1668        //channel.eof();
1669        channel.eof_remote();
1670      }
1671      /*
1672      packet.reset();
1673      buf.putByte((byte)SSH_MSG_CHANNEL_EOF);
1674      buf.putInt(channel.getRecipient());
1675      write(packet);
1676      */
1677      break;
1678    case SSH_MSG_CHANNEL_CLOSE:
1679           buf.getInt();
1680      buf.getShort();
1681      i=buf.getInt();
1682      channel=Channel.getChannel(i, this);
1683      if(channel!=null){
1684  //        channel.close();
1685        channel.disconnect();
1686      }
1687      /*
1688           if(Channel.pool.size()==0){
1689        thread=null;
1690      }
1691      */
1692      break;
1693    case SSH_MSG_CHANNEL_OPEN_CONFIRMATION:
```

```
1694          buf.getInt();
1695    buf.getShort();
1696    i=buf.getInt();
1697    channel=Channel.getChannel(i, this);
1698    if(channel==null){
1699      //break;
1700    }
1701          int r=buf.getInt();
1702          long rws=buf.getUInt();
1703          int rps=buf.getInt();
1704
1705          channel.setRemoteWindowSize(rws);
1706          channel.setRemotePacketSize(rps);
1707          channel.open_confirmation=true;
1708          channel.setRecipient(r);
1709          break;
1710  case SSH_MSG_CHANNEL_OPEN_FAILURE:
1711          buf.getInt();
1712    buf.getShort();
1713    i=buf.getInt();
1714    channel=Channel.getChannel(i, this);
1715    if(channel==null){
1716      //break;
1717    }
1718    int reason_code=buf.getInt();
1719    //foo=buf.getString();  // additional textual information
1720    //foo=buf.getString();  // language tag
1721          channel.setExitStatus(reason_code);
1722          channel.close=true;
1723    channel.eof_remote=true;
1724    channel.setRecipient(0);
1725    break;
1726  case SSH_MSG_CHANNEL_REQUEST:
1727          buf.getInt();
1728    buf.getShort();
1729    i=buf.getInt();
1730    foo=buf.getString();
1731          boolean reply=(buf.getByte()!=0);
1732    channel=Channel.getChannel(i, this);
1733    if(channel!=null){
1734      byte reply_type=(byte)SSH_MSG_CHANNEL_FAILURE;
1735      if((Util.byte2str(foo)).equals("exit-status")){
1736        i=buf.getInt();             // exit-status
1737        channel.setExitStatus(i);
1738        reply_type=(byte)SSH_MSG_CHANNEL_SUCCESS;
1739      }
1740      if(reply){
1741        packet.reset();
1742        buf.putByte(reply_type);
1743        buf.putInt(channel.getRecipient());
1744        write(packet);
1745      }
1746    }
1747    else{
1748    }
1749    break;
1750  case SSH_MSG_CHANNEL_OPEN:
1751          buf.getInt();
1752    buf.getShort();
1753    foo=buf.getString();
1754    String ctyp=Util.byte2str(foo);
1755          if(!"forwarded-tcpip".equals(ctyp) &&
1756       !("x11".equals(ctyp) && x11_forwarding) &&
1757       !("auth-agent@openssh.com".equals(ctyp) && agent_forwarding)){
1758            //System.err.println("Session.run: CHANNEL OPEN "+ctyp);
1759      //throw new IOException("Session.run: CHANNEL OPEN "+ctyp);
1760      packet.reset();
1761      buf.putByte((byte)SSH_MSG_CHANNEL_OPEN_FAILURE);
1762      buf.putInt(buf.getInt());
```

```
1763        buf.putInt(Channel.SSH_OPEN_ADMINISTRATIVELY_PROHIBITED);
1764        buf.putString(Util.empty);
1765        buf.putString(Util.empty);
1766        write(packet);
1767      }
1768      else{
1769        channel=Channel.getChannel(ctyp);
1770        addChannel(channel);
1771        channel.getData(buf);
1772        channel.init();
1773
1774        Thread tmp=new Thread(channel);
1775        tmp.setName("Channel "+ctyp+" "+host);
1776              if(daemon_thread){
1777                tmp.setDaemon(daemon_thread);
1778              }
1779        tmp.start();
1780        break;
1781      }
1782    case SSH_MSG_CHANNEL_SUCCESS:
1783            buf.getInt();
1784      buf.getShort();
1785      i=buf.getInt();
1786      channel=Channel.getChannel(i, this);
1787      if(channel==null){
1788        break;
1789      }
1790      channel.reply=1;
1791      break;
1792    case SSH_MSG_CHANNEL_FAILURE:
1793      buf.getInt();
1794      buf.getShort();
1795      i=buf.getInt();
1796      channel=Channel.getChannel(i, this);
1797      if(channel==null){
1798        break;
1799      }
1800      channel.reply=0;
1801      break;
1802    case SSH_MSG_GLOBAL_REQUEST:
1803      buf.getInt();
1804      buf.getShort();
1805      foo=buf.getString();       // request name
1806      reply=(buf.getByte()!=0);
1807      if(reply){
1808        packet.reset();
1809        buf.putByte((byte)SSH_MSG_REQUEST_FAILURE);
1810        write(packet);
1811      }
1812      break;
1813    case SSH_MSG_REQUEST_FAILURE:
1814    case SSH_MSG_REQUEST_SUCCESS:
1815            Thread t=grr.getThread();
1816            if(t!=null){
1817              grr.setReply(msgType==SSH_MSG_REQUEST_SUCCESS? 1 : 0);
1818              t.interrupt();
1819            }
1820      break;
1821    default:
1822            //System.err.println("Session.run: unsupported type "+msgType);
1823      throw new IOException("Unknown SSH message type "+msgType);
1824    }
1825        }
1826      }
1827      catch(Exception e){
1828        in_kex=false;
1829        if(JSch.getLogger().isEnabled(Logger.INFO)){
1830          JSch.getLogger().log(Logger.INFO,
```

```
1831                                "Caught an exception, leaving main loop due to " + e.
                                        getMessage());
1832        }
1833        //System.err.println("# Session.run");
1834        //e.printStackTrace();
1835      }
1836      try{
1837        disconnect();
1838      }
1839      catch(NullPointerException e){
1840        //System.err.println("@1");
1841        //e.printStackTrace();
1842      }
1843      catch(Exception e){
1844        //System.err.println("@2");
1845        //e.printStackTrace();
1846      }
1847      isConnected=false;
1848    }
1849
1850    public void disconnect(){
1851      if(!isConnected) return;
1852      //System.err.println(this+": disconnect");
1853      //Thread.dumpStack();
1854      if(JSch.getLogger().isEnabled(Logger.INFO)){
1855        JSch.getLogger().log(Logger.INFO,
1856                             "Disconnecting from "+host+" port "+port);
1857      }
1858      /*
1859      for(int i=0; i<Channel.pool.size(); i++){
1860        try{
1861          Channel c=((Channel)(Channel.pool.elementAt(i)));
1862    if(c.session==this) c.eof();
1863        }
1864        catch(Exception e){
1865        }
1866      }
1867      */
1868
1869      Channel.disconnect(this);
1870
1871      isConnected=false;
1872
1873      PortWatcher.delPort(this);
1874      ChannelForwardedTCPIP.delPort(this);
1875      ChannelX11.removeFakedCookie(this);
1876
1877      synchronized(lock){
1878        if(connectThread!=null){
1879          Thread.yield();
1880          connectThread.interrupt();
1881          connectThread=null;
1882        }
1883      }
1884      thread=null;
1885      try{
1886        if(io!=null){
1887    if(io.in!=null) io.in.close();
1888    if(io.out!=null) io.out.close();
1889    if(io.out_ext!=null) io.out_ext.close();
1890        }
1891        if(proxy==null){
1892          if(socket!=null)
1893      socket.close();
1894        }
1895        else{
1896    synchronized(proxy){
1897      proxy.close();
1898    }
```

```
1899    proxy=null;
1900        }
1901      }
1902      catch(Exception e){
1903  //      e.printStackTrace();
1904      }
1905      io=null;
1906      socket=null;
1907  //    synchronized(jsch.pool){
1908  //      jsch.pool.removeElement(this);
1909  //    }
1910
1911      jsch.removeSession(this);
1912
1913      //System.gc();
1914    }
1915
1916    public int setPortForwardingL(int lport, String host, int rport) throws JSchException{
1917      return setPortForwardingL("127.0.0.1", lport, host, rport);
1918    }
1919    public int setPortForwardingL(String boundaddress, int lport, String host, int rport)
          throws JSchException{
1920      return setPortForwardingL(boundaddress, lport, host, rport, null);
1921    }
1922    public int setPortForwardingL(String boundaddress, int lport, String host, int rport,
          ServerSocketFactory ssf) throws JSchException{
1923      PortWatcher pw=PortWatcher.addPort(this, boundaddress, lport, host, rport, ssf);
1924      Thread tmp=new Thread(pw);
1925      tmp.setName("PortWatcher Thread for "+host);
1926      if(daemon_thread){
1927        tmp.setDaemon(daemon_thread);
1928      }
1929      tmp.start();
1930      return pw.lport;
1931    }
1932    public void delPortForwardingL(int lport) throws JSchException{
1933      delPortForwardingL("127.0.0.1", lport);
1934    }
1935    public void delPortForwardingL(String boundaddress, int lport) throws JSchException{
1936      PortWatcher.delPort(this, boundaddress, lport);
1937    }
1938    public String[] getPortForwardingL() throws JSchException{
1939      return PortWatcher.getPortForwarding(this);
1940    }
1941
1942    public void setPortForwardingR(int rport, String host, int lport) throws JSchException{
1943      setPortForwardingR(null, rport, host, lport, (SocketFactory)null);
1944    }
1945    public void setPortForwardingR(String bind_address, int rport, String host, int lport)
          throws JSchException{
1946      setPortForwardingR(bind_address, rport, host, lport, (SocketFactory)null);
1947    }
1948    public void setPortForwardingR(int rport, String host, int lport, SocketFactory sf)
          throws JSchException{
1949      setPortForwardingR(null, rport, host, lport, sf);
1950    }
1951    public void setPortForwardingR(String bind_address, int rport, String host, int lport,
          SocketFactory sf) throws JSchException{
1952      ChannelForwardedTCPIP.addPort(this, bind_address, rport, host, lport, sf);
1953      setPortForwarding(bind_address, rport);
1954    }
1955
1956    public void setPortForwardingR(int rport, String daemon) throws JSchException{
1957      setPortForwardingR(null, rport, daemon, null);
1958    }
1959    public void setPortForwardingR(int rport, String daemon, Object[] arg) throws
          JSchException{
1960      setPortForwardingR(null, rport, daemon, arg);
1961    }
```

```
1962   public void setPortForwardingR(String bind_address, int rport, String daemon, Object[]
           arg) throws JSchException{
1963     ChannelForwardedTCPIP.addPort(this, bind_address, rport, daemon, arg);
1964     setPortForwarding(bind_address, rport);
1965   }
1966
1967   private class GlobalRequestReply{
1968     private Thread thread=null;
1969     private int reply=-1;
1970     void setThread(Thread thread){
1971       this.thread=thread;
1972       this.reply=-1;
1973     }
1974     Thread getThread(){ return thread; }
1975     void setReply(int reply){ this.reply=reply; }
1976     int getReply(){ return this.reply; }
1977   }
1978   private GlobalRequestReply grr=new GlobalRequestReply();
1979   private void setPortForwarding(String bind_address, int rport) throws JSchException{
1980     synchronized(grr){
1981     Buffer buf=new Buffer(100); // ??
1982     Packet packet=new Packet(buf);
1983
1984     String address_to_bind=ChannelForwardedTCPIP.normalize(bind_address);
1985
1986     grr.setThread(Thread.currentThread());
1987
1988     try{
1989       // byte SSH_MSG_GLOBAL_REQUEST 80
1990       // string "tcpip-forward"
1991       // boolean want_reply
1992       // string  address_to_bind
1993       // uint32  port number to bind
1994       packet.reset();
1995       buf.putByte((byte) SSH_MSG_GLOBAL_REQUEST);
1996       buf.putString(Util.str2byte("tcpip-forward"));
1997       buf.putByte((byte)1);
1998       buf.putString(Util.str2byte(address_to_bind));
1999       buf.putInt(rport);
2000       write(packet);
2001     }
2002     catch(Exception e){
2003       grr.setThread(null);
2004       if(e instanceof Throwable)
2005         throw new JSchException(e.toString(), (Throwable)e);
2006       throw new JSchException(e.toString());
2007     }
2008
2009     int count = 0;
2010     int reply = grr.getReply();
2011     while(count < 10 && reply == -1){
2012       try{ Thread.sleep(1000); }
2013       catch(Exception e){
2014       }
2015       count++;
2016       reply = grr.getReply();
2017     }
2018     grr.setThread(null);
2019     if(reply != 1){
2020       throw new JSchException("remote port forwarding failed for listen port "+rport);
2021     }
2022     }
2023   }
2024   public void delPortForwardingR(int rport) throws JSchException{
2025     ChannelForwardedTCPIP.delPort(this, rport);
2026   }
2027
2028   private void initDeflater(String method) throws JSchException{
2029     if(method.equals("none")){
```

```
2030        deflater=null;
2031        return;
2032      }
2033      String foo=getConfig(method);
2034      if(foo!=null){
2035        if(method.equals("zlib") ||
2036           (isAuthed && method.equals("zlib@openssh.com"))){
2037          try{
2038            Class c=Class.forName(foo);
2039            deflater=(Compression)(c.newInstance());
2040            int level=6;
2041            try{ level=Integer.parseInt(getConfig("compression_level"));}
2042            catch(Exception ee){ }
2043            deflater.init(Compression.DEFLATER, level);
2044          }
2045          catch(NoClassDefFoundError ee){
2046            throw new JSchException(ee.toString(), ee);
2047          }
2048          catch(Exception ee){
2049            throw new JSchException(ee.toString(), ee);
2050            //System.err.println(foo+" isn't accessible.");
2051          }
2052        }
2053      }
2054    }
2055    private void initInflater(String method) throws JSchException{
2056      if(method.equals("none")){
2057        inflater=null;
2058        return;
2059      }
2060      String foo=getConfig(method);
2061      if(foo!=null){
2062        if(method.equals("zlib") ||
2063           (isAuthed && method.equals("zlib@openssh.com"))){
2064          try{
2065            Class c=Class.forName(foo);
2066            inflater=(Compression)(c.newInstance());
2067            inflater.init(Compression.INFLATER, 0);
2068          }
2069          catch(Exception ee){
2070            throw new JSchException(ee.toString(), ee);
2071        //System.err.println(foo+" isn't accessible.");
2072          }
2073        }
2074      }
2075    }
2076
2077    void addChannel(Channel channel){
2078      channel.setSession(this);
2079    }
2080
2081    public void setProxy(Proxy proxy){ this.proxy=proxy; }
2082    public void setHost(String host){ this.host=host; }
2083    public void setPort(int port){ this.port=port; }
2084    void setUserName(String username){ this.username=username; }
2085    public void setUserInfo(UserInfo userinfo){ this.userinfo=userinfo; }
2086    public UserInfo getUserInfo(){ return userinfo; }
2087    public void setInputStream(InputStream in){ this.in=in; }
2088    public void setOutputStream(OutputStream out){ this.out=out; }
2089    public void setX11Host(String host){ ChannelX11.setHost(host); }
2090    public void setX11Port(int port){ ChannelX11.setPort(port); }
2091    public void setX11Cookie(String cookie){ ChannelX11.setCookie(cookie); }
2092    public void setPassword(String password){
2093      if(password!=null)
2094        this.password=Util.str2byte(password);
2095    }
2096    public void setPassword(byte[] password){
2097      if(password!=null){
2098        this.password=new byte[password.length];
```

```
2099        System.arraycopy(password, 0, this.password, 0, password.length);
2100      }
2101    }
2102
2103    public void setConfig(java.util.Properties newconf){
2104      setConfig((java.util.Hashtable)newconf);
2105    }
2106
2107    public void setConfig(java.util.Hashtable newconf){
2108      synchronized(lock){
2109        if(config==null)
2110          config=new java.util.Hashtable();
2111        for(java.util.Enumeration e=newconf.keys() ; e.hasMoreElements() ;) {
2112          String key=(String)(e.nextElement());
2113          config.put(key, (String)(newconf.get(key)));
2114        }
2115      }
2116    }
2117
2118    public void setConfig(String key, String value){
2119      synchronized(lock){
2120        if(config==null){
2121          config=new java.util.Hashtable();
2122        }
2123        config.put(key, value);
2124      }
2125    }
2126
2127    public String getConfig(String key){
2128      Object foo=null;
2129      if(config!=null){
2130        foo=config.get(key);
2131        if(foo instanceof String) return (String)foo;
2132      }
2133      foo=jsch.getConfig(key);
2134      if(foo instanceof String) return (String)foo;
2135      return null;
2136    }
2137
2138    public void setSocketFactory(SocketFactory sfactory){
2139      socket_factory=sfactory;
2140    }
2141    public boolean isConnected(){ return isConnected; }
2142    public int getTimeout(){ return timeout; }
2143    public void setTimeout(int timeout) throws JSchException {
2144      if(socket==null){
2145        if(timeout<0){
2146          throw new JSchException("invalid timeout value");
2147        }
2148        this.timeout=timeout;
2149        return;
2150      }
2151      try{
2152        socket.setSoTimeout(timeout);
2153        this.timeout=timeout;
2154      }
2155      catch(Exception e){
2156        if(e instanceof Throwable)
2157          throw new JSchException(e.toString(), (Throwable)e);
2158        throw new JSchException(e.toString());
2159      }
2160    }
2161    public String getServerVersion(){
2162      return Util.byte2str(V_S);
2163    }
2164    public String getClientVersion(){
2165      return Util.byte2str(V_C);
2166    }
2167    public void setClientVersion(String cv){
```

```
2168      V_C=Util.str2byte(cv);
2169    }
2170
2171    public void sendIgnore() throws Exception{
2172      Buffer buf=new Buffer();
2173      Packet packet=new Packet(buf);
2174      packet.reset();
2175      buf.putByte((byte)SSH_MSG_IGNORE);
2176      write(packet);
2177    }
2178
2179    private static final byte[] keepalivemsg=Util.str2byte("keepalive@jcraft.com");
2180    public void sendKeepAliveMsg() throws Exception{
2181      Buffer buf=new Buffer();
2182      Packet packet=new Packet(buf);
2183      packet.reset();
2184      buf.putByte((byte)SSH_MSG_GLOBAL_REQUEST);
2185      buf.putString(keepalivemsg);
2186      buf.putByte((byte)1);
2187      write(packet);
2188    }
2189
2190    private HostKey hostkey=null;
2191    public HostKey getHostKey(){ return hostkey; }
2192    public String getHost(){return host;}
2193    public String getUserName(){return username;}
2194    public int getPort(){return port;}
2195    public void setHostKeyAlias(String hostKeyAlias){
2196      this.hostKeyAlias=hostKeyAlias;
2197    }
2198    public String getHostKeyAlias(){
2199      return hostKeyAlias;
2200    }
2201
2202    /**
2203     * Sets the interval to send a keep-alive message.  If zero is
2204     * specified, any keep-alive message must not be sent.  The default interval
2205     * is zero.
2206     * @param interval the specified interval, in milliseconds.
2207     * @see #getServerAliveInterval()
2208     */
2209    public void setServerAliveInterval(int interval) throws JSchException {
2210      setTimeout(interval);
2211      this.serverAliveInterval=interval;
2212    }
2213
2214    /**
2215     * Returns setting for the interval to send a keep-alive message.
2216     * @see #setServerAliveInterval(int)
2217     */
2218    public int getServerAliveInterval(){
2219      return this.serverAliveInterval;
2220    }
2221
2222    /**
2223     * Sets the number of keep-alive messages which may be sent without
2224     * receiving any messages back from the server.  If this threshold is
2225     * reached while keep-alive messages are being sent, the connection will
2226     * be disconnected.  The default value is one.
2227     * @param count the specified count
2228     * @see #getServerAliveCountMax()
2229     */
2230    public void setServerAliveCountMax(int count){
2231      this.serverAliveCountMax=count;
2232    }
2233
2234    /**
2235     * Returns setting for the threshold to send keep-alive messages.
2236     * @see #getServerAliveCountMax(int)
```

```
2237     */
2238   public int getServerAliveCountMax(){
2239     return this.serverAliveCountMax;
2240   }
2241
2242   public void setDaemonThread(boolean enable){
2243     this.daemon_thread=enable;
2244   }
2245
2246   private String[] checkCiphers(String ciphers){
2247     if(ciphers==null || ciphers.length()==0)
2248       return null;
2249
2250     if(JSch.getLogger().isEnabled(Logger.INFO)){
2251       JSch.getLogger().log(Logger.INFO,
2252                            "CheckCiphers: "+ciphers);
2253     }
2254
2255     java.util.Vector result=new java.util.Vector();
2256     String[] _ciphers=Util.split(ciphers, ",");
2257     for(int i=0; i<_ciphers.length; i++){
2258       if(!checkCipher(getConfig(_ciphers[i]))){
2259         result.addElement(_ciphers[i]);
2260       }
2261     }
2262     if(result.size()==0)
2263       return null;
2264     String[] foo=new String[result.size()];
2265     System.arraycopy(result.toArray(), 0, foo, 0, result.size());
2266
2267     if(JSch.getLogger().isEnabled(Logger.INFO)){
2268       for(int i=0; i<foo.length; i++){
2269         JSch.getLogger().log(Logger.INFO,
2270                              foo[i]+" is not available.");
2271       }
2272     }
2273
2274     return foo;
2275   }
2276
2277   static boolean checkCipher(String cipher){
2278     try{
2279       Class c=Class.forName(cipher);
2280       Cipher _c=(Cipher)(c.newInstance());
2281       _c.init(Cipher.ENCRYPT_MODE,
2282               new byte[_c.getBlockSize()],
2283               new byte[_c.getIVSize()]);
2284       return true;
2285     }
2286     catch(Exception e){
2287       return false;
2288     }
2289   }
2290
2291   private String[] checkKexes(String kexes){
2292     if(kexes==null || kexes.length()==0)
2293       return null;
2294
2295     if(JSch.getLogger().isEnabled(Logger.INFO)){
2296       JSch.getLogger().log(Logger.INFO,
2297                            "CheckKexes: "+kexes);
2298     }
2299
2300     java.util.Vector result=new java.util.Vector();
2301     String[] _kexes=Util.split(kexes, ",");
2302     for(int i=0; i<_kexes.length; i++){
2303       if(!checkKex(this, getConfig(_kexes[i]))){
2304         result.addElement(_kexes[i]);
2305       }
```

```
2306      }
2307      if(result.size()==0)
2308        return null;
2309      String[] foo=new String[result.size()];
2310      System.arraycopy(result.toArray(), 0, foo, 0, result.size());
2311
2312      if(JSch.getLogger().isEnabled(Logger.INFO)){
2313        for(int i=0; i<foo.length; i++){
2314          JSch.getLogger().log(Logger.INFO,
2315                               foo[i]+" is not available.");
2316        }
2317      }
2318
2319      return foo;
2320    }
2321
2322    static boolean checkKex(Session s, String kex){
2323      try{
2324        Class c=Class.forName(kex);
2325        KeyExchange _c=(KeyExchange)(c.newInstance());
2326        _c.init(s ,null, null, null, null);
2327        return true;
2328      }
2329      catch(Exception e){ return false; }
2330    }
2331
2332    /**
2333     * Sets the identityRepository, which will be referred
2334     * in the public key authentication.  The default value is null.
2335     *
2336     * @param identityRepository
2337     *
2338     * @see #getIdentityRepository()
2339     */
2340    public void setIdentityRepository(IdentityRepository identityRepository){
2341      this.identityRepository = identityRepository;
2342    }
2343
2344    /**
2345     * Gets the identityRepository.  If this.identityRepository is null,
2346     * JSch#getIdentityRepository() will be invoked.
2347     *
2348     * @see JSch#getIdentityRepository()
2349     */
2350    IdentityRepository getIdentityRepository(){
2351      if(identityRepository == null)
2352        return jsch.getIdentityRepository();
2353      return identityRepository;
2354    }
2355  }
```

A.3 NewSession.java

Test Wrapper for the manual testing

```
1 package com.jcraft.jsch;
2 /**
3  * Test Wrapper for the manual testing
4  * @author      Mirjam van Nahmen  <m.vannahmen @ student.ru.nl>
5  * @version     1.0
6  * @since       2014-01-31
7  */
8 import java.io.IOException;
9 import java.util.logging.Level;
10 import java.util.logging.Logger;
11
12 public class NewSession
13 {
14     static JSch jsch;
15     static Session session;
16
17    //Main to create a sequence of requests
18     public static void main(String [] args)
19     {
20         try {
21       //Init connection
22             System.out.println("begin");
23             jsch = new JSch();
24             session = new Session(jsch);
25             session.setUserName("Miri");
26             session.setPassword("Hallo");
27             session.setHostKeyAlias("localhost");
28             session.setHost("127.0.0.1");
29             java.util.Properties config = new java.util.Properties();
30             config.put("StrictHostKeyChecking", "no");
31             session.setConfig(config);
32             System.out.println("Start...");
33
34       //Begin sequence, in this case
35       // "connect, sendVersion, KEXINIT, connect, KEXINITDH,
36             connect();
37             System.out.println("Connected... ");
38             System.out.println("Good Server version(0) otherwise(-1): "+version());
39       System.out.println("KexInit1: "+MSGToString(kexinit()));
40       //put actual state in cache
41       JSch tmpsch = jsch;
42             Session tmpsession = session;
43             Buffer tempbuf = session.buf;
44             //send again a connect
45       System.out.println("Connected... ");
46             //set actual to the state of the cache
47       jsch = tmpsch;
48             session = tmpsession;
49             session.buf = tempbuf;
50
51             System.out.println("KexInitDH: "+MSGToString(kexinitDH()));
52             System.out.println("NewKeys: "+MSGToString(newKeys()));
53             System.out.println("Service_Request_accepted: "+MSGToString(service_accepted())
                   );
54             session.disconnect();
55         }
56         catch (JSchException ex) {
57            session.disconnect(); Logger.getLogger(NewSession.class.getName()).log(Level.
                  SEVERE, null, ex);
58         } catch (Exception ex) {
59            session.disconnect(); Logger.getLogger(NewSession.class.getName()).log(Level.
                  SEVERE, null, ex);
60         }
61     }
62
63     /**
```

```
64      * Connect to the server.
65      * Only with default values, otherwise no connection
66      * @return 0 if the connection is established
67      **/
68     public static String connect() throws JSchException, IOException, Exception{
69         session.connect();
70         return "Connected";
71     }
72 
73   /**
74    * send the version number
75    * @return if the right version is send
76    **/
77     public static String version() throws IOException, JSchException{
78         byte[] V_C=Util.str2byte("SSH-2.0-JSCH-"+JSch.VERSION);
79         return session.version(session.i, session.j,V_C);
80     }
81 
82   /**
83   * send kexinit with manual settings
84   * @param kex_contant: default=0
85   * @param server_host_key_algorithms_contant: default=0
86   * @param cipher_c2s_contant: default=0
87   * @param cipher_s2c_contant: default=0
88   **/
89     public static int kexinit() throws Exception{
90         return session.kexinit(0,0,0,0);
91     }
92 
93   /**
94    * send the KEXINIT_DH
95    * @return the message number
96    **/
97     public static int kexinitDH() throws Exception{
98         return session.kexDH_init();
99     }
100 
101  /**
102   * send the the SERVICE_ACCEPTED
103   * @return the message number
104   **/
105    private static int service_accepted() throws JSchException, Exception {
106        return session.service_accepted();
107    }
108 
109  /**
110   * send the NEWKEY
111   * @return the message number
112   **/
113    private static int newKeys() throws JSchException, Exception {
114        return session.newKey();
115    }
116 
117    /**
118     * convert the integer of the message number to the associated String
119     * @param int msg
120     * @return String SSH-MSG-XXX
121     **/
122    public static String MSGToString(int msg){
123        switch(msg){
124            case 1: return "SSH_MSG_DISCONNECT = 1";
125            case 2: return "SSH_MSG_IGNORE = 2";
126            case 3: return "SSH_MSG_UNIMPLEMENTED = 3";
127            case 4: return "SSH_MSG_DEBUG = 4";
128            case 5: return "SSH_MSG_SERVICE_REQUEST = 5";
129            case 6: return "SSH_MSG_SERVICE_ACCEPT = 6";
130            case 20: return "SSH_MSG_KEXINIT = 20";
131            case 21: return "SSH_MSG_NEWKEYS = 21";
132            case 30: return "SSH_MSG_KEXDH_INIT = 30";
```

```
133             case 31: return "SSH_MSG_KEXDH_REPLY = 31";
134             case -5: return "sendVersion";
135             default: return "somthing else"+msg;
136         }
137     }
138 }
```