Implementing a simple state-resume SSH client for Android phone *

Tuan Dao Moloud Shahbazi

Department of Computer Science and Engineering,
University of California, Riverside
{tdao006, mshah008}@cs.ucr.edu

SSH is a popular application to provide a secure channel for accessing the remote machines. SSH protocol is highly demanded by mobile users as well. But because of the intermittent nature of mobile communication, it needs to be adopted to the resource limitations of this type of users. In this paper, we modify SSH protocol to support data transfer resume and data compression. Our experiments shows the significant improve in amount of data traffic, power consumption and data transfer completion time. We have implemented three layers of ssh protocol, Transport, Authentication and Communication layer for both client and server.

1 Introduction

Secure shell (SSH) is a secured protocol that supports remote shell execution and data transfer between a client and a server host. SSH was first designed by Tatu Ylonen in 1995, at the Helsinki University of Technology, and since then it has been used widely throughout the world by both network administrators and normal users.

Mobile devices are more powerful these days, therefore there are more and more people use their smart phones for work while they are on the move. In this project, we want to implement a SSH client in Android, which is one of the most popular mobile platform. SSH could be a convenient tool for mobile users, since they can upload/download files from a remote machine, backup important files and even control a machine remotely.

Unfortunately, there is one characteristic of the SSH protocol that makes it less appealing to mobile platforms. The SSH protocol does not support resume the ongoing operations if the connection is broken. For instance, if somebody is downloading a large file through an SSH connection, for example via the SCP command, and the connection is down then he has to re-download the whole file from scratch. This drawback affects mobile users more than the traditional users, since the signals in mobile devices are less stable and easily to be interrupted, for instance because of the roaming process while users move from place to place.

Moreover, restarting the operations from scratch is expensive, both in network data price and the phone's power consumption. We make changes to the SSH protocol stack to support resume ongoing operations in case the connection

* Project proposal for CS253 by Prof. Harsha V. Madhyastha in Spring 2012

gets interrupted. Therefore, in our implementation, on-going data transfer will be resumed exactly at the point then interruption happens when the connection between the client and server is re-established.

2 Related Work

In [1], Mosh Mobile Shell, have been developed to manage SSH over an unstable channel. Mosh handles SSH connection either a clients IP is changed or the client gets disconnected and connects again. In other words, Mosh will restore the connection if it drops. However, this leads to poor performance. As an illustration, a Mosh client sends every keys which a user has typed to the server even though the command is not correct. In our system, not only we send a complete and valid commands to save resources, but also our system is able to resume transferring an incomplete file to the server. ConnectBot [6] is an android-based secure shell client. It creates multiple simultaneous SSH sessions to improve performance a user perceived. More than one million downloads of this application shows the need for mobile SSH client. ConnectBot, however, does not support restoring sessions.

3 Application design

In our implementation, we follow the SSH protocol architecture as describe in RFC 4251. SSH provides secure communication channels over an insecure network on the top of the TCP transport layer, thus we do not have to to pay attention to the data loss or unodered data problems. The SSH protocol consists of three main sub-layers, as shown in Fig. 1:

- The Transport Layer Protocol [2]: this layer provides server authentication, keys exchange method, data encryption and integrity, and compression service. This is the lowest layer in the SSH protocol and is established directly over a TCP connection. Client and server can support a different set of encryption and hashing algorithms, e.g. 3DES, EAS, HMAC-MD5, HMAC-SHA, etc. In order to successfully set up the transport connection, the client and server must agree on using the same set of algorithms. For simplicity, in our implementation, we use EAS-128 bits and HMAC-SHA on both the client and server.
- The User Authentication Protocol [3]: After the transport connection has been set up, authentication data will be transfer securely between the client and server. Server can support various ways to authenticate SSH users, e.g. using password, user's public key and host-based authentication. In our implementation, we only support the plain password authentication. Password can be sent as plaintext between client and server over the secure connection already set up.
- The Connection Protocol [4]: This protocol runs over the User Authentication Protocol. This protocol provides logical channels to be used in many

different purposes, e.g. X11 forwarding, TCP forwarding, interactive sessions, data transfer sessions, etc. In our implementation, we only support remote shell execution and data transfer between clients and server.

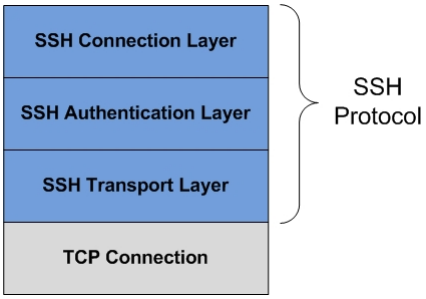

Fig. 1: SSH protocol stack

Users can execute a script/program or start a data transfer session by sending a command to server. If the command starts with either "download" or "upload" words, it is considered a data transfer command. Otherwise, it is considered a remote execution command, then server will execute the command with the same privilege of the user who client used to log in and response with the execution result to client.
If it is a data transfer command, a new data transfer operation is executed.

4 Implementation

4.1 SSH client/server

In this project, we implement all the three sub-layers of the SSH protocol as defined in section 3. Our implementation consists of a server component which runs on a Linux server and a client component which runs on an Android device. Both components are written in Java. We used the JCE (Java Cryptography Extension) to support encrypt/decrypt messages, calculate hash code and exchange keys. For data compression, we used the zlib library, which is commonly used to compress data in Linux kernel.

We also make change to how clients send command to server compared to the original SSH protocol. In the original protocol, whenever a user hits a key, the key code is immediately sent to the server and the character is displayed on user's screen when server sends echo back. The command is executed when the user hits the "enter" button. The reason for that is they want to emulate network delay when users typing their commands. When network delay between client and server is low, the characters appear immediately on user's screen, otherwise it could take a short time before the characters are displayed.

When working with the mobile devices, we noticed that the number of typing mistakes was quite high. Therefore, we only send the whole command from client to server when the user hit then "send"' button to reduce bandwidth from sending incorrect key stroke.

In addition, in our implementation we detect interruptions between client and server by catching the SocketException in data transfer. When an interruption happens, the client will reconnect to server automatically. When the connection is re-established, the client reacts depends on the on-going command:

- If the on-going command is a data transfer command, the transfer is resumed as defined in section 3.
- If the on-going command is a executive command, just notifies the user that the connection has been established again and he or she will decide whether to re-execute the command again.

The main reason for not resuming an executive command is that we do not save the state of the server when the connection get interupted. Without the saved state, it is impossible to resume the command at the server. For instance, if the user was executing a program that appends data into a file and the connection went down. When the connection came up, we could not simply append all the data again to that file.

Support saving state at server would requires a lot more work and consideration. One of the main issue is the security problem on the server, saving state would consume resources on the server until the interrupted client comes back. It could be come an security hole to be exploited to launch attack on the server if there are a large number of clients executing some scripts, then disconnect at the same time, similar to the SYN attack on TCP protocol. Therefore, when a client is disconnected, our server just simply closes the socket and deletes all the information about that client. When that client comes back, it has to create a whole new session but can transfer data starting from a specific point depends on when the connection is interrupted.

For clients, if after a predefined of number of unsuccessful retries, the reconnection process is aborted.

4.2 Data transfer operations

The message passing between client and server for downloading a file from server to the client is shown in figure 2. After sending a command for downloading a file on the server, the client sends the digest message of that file to the server. The file is divided into chunks and the message digest is calculated for each chunk. All the digests of the file chunks then will be sent to the server in a digest message. Server after receiving the message digest message, calculate the digest for each chunk in the source file and compare it with the received digest for that chunk. After figuring out the first difference between digest of the source file chunk and received digest, it make a pointer to the beginning of that chunk and send this pointer to the client.

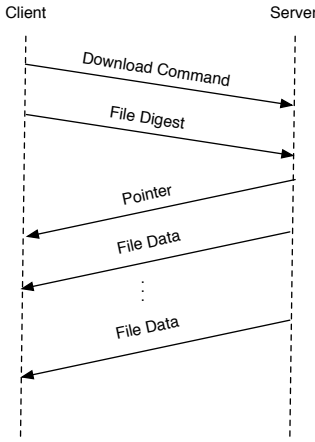

Fig. 2: Mobile ssh message passing for the download command.

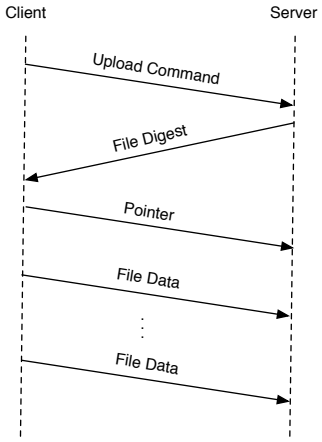

Fig. 3: Mobile ssh message passing for upload command.

After sending the pointer, server starts sending the data from the byte that pointer shows. Client attach these received data from the server to the place in file that pointer shows.

Figure 3 shows the message passing between client and server during an upload command. After receiving a upload command from user, client sends this command to the server. As the destination is on the server side, server will calculate the message digest same as in download and sends it to the client. Client uses this digest message to compare the digests of source and destination files and find the first occurrence of chunk difference and sends this pointer to the client. Client after receiving the message containing the pointer, start sending the data from the place in the file that pointer shows. Server starts attaching the received data to the file from the place pointer shows.

4.3 Auto Reconnection

In all the communications between clients and server, we check for network disconnections. When there is an interruption in the TCP connection between client and server, a SocketException is thrown in our program. So, in our implementation we catch this kind of exception and do the reconnection as follows:

```
try{
    //sendingorreceivingbetweenclientsandserver
```

```
}
catch(SocketExceptionex){
    reconnect();
}
```

In the reconnect operation, client will retry for a predefined number of times (in our implementation, we set this value as 5 retries). Each retry is 10 second after the previous one. If reconnection is successful, for data transfer commands, the transfer operation is automatically resume, for remote command execution we just notice users there is a disconnection between client and server and that user should re-execute the command based on user decision. We do not support resume command execution because it could lead to unexpected error such as append the same data to some file in the server twice.
The impact of data compression on our SSH implementation:
In our implementation, we support data compression using zlib library, which is a commonly used library to compress data in many Linux distribution.
We apply compression when transferring text file with size from 500kb to 50MB between client and server. The compress ratio for these text files are from 40% to 57%. We consider the impact of data compression on:

- The difference in the amount of data sent over the network
- The difference in the time used to send data (time when without compression time when using compression)
- The difference in power consumption (power consumption without compression power consumption when using compression)

5 Evaluation

For evaluation, we consider the difference between our implementation and the original SSH protocol on three main attributes:

- Total power consumption
- Total amount of data transfered over the network
- Total data transfer time

We used the Power Tutor [5] tool for power meter since this tool support measure power consumption of a specific application and on specific components, such as LCD, Wifi interface. The result from Power Tutor is not highly accurate, however we only consider the difference between the value from the original SSH protocol and our implementation, the result is still reasonable. For each experiments, we conducted three different tests and calculate the average numbers.

5.1 Effects of Transfer Resume

To evaluate the performance of our mobile ssh client, we are looking at the total amount of data sent over the network to transfer a file, the power consumption

of a mobile device and the completion time of a data transfer process. We did our measurements for different file sizes in presence of link interruption in the middle of a data transfer. We create one interruption during the transferring process of a data file after certain percentage of data has been transferred. We change this percentage value in our experiments to 15%, 50% and 85%. To do this, we run the server on a typical desktop machine while the client is running on an android phone. Both client and server connects to the same access point.

To emulate the network connection interruption, we modified the client to close the connection after downloading certain amount of data. The client auto reconnects and continue downloading after new connection has been stablished. We compare two cases where the ssh handles the resume after disconnection and the other case where the server restart the operation.
Figure 4 shows the difference of total data transferred for different file sizes with restarting minus resuming the transfer operation after disconnection happens. In this figure, we plotted this difference for different cases where interruption happens. As the figure shows, the difference in total amount of data transferred is positive which shows that without state resume more data is transferred over the network to transfer a file with the same size. This difference increases with increasing in the size of data, as expected.

Figures 5 and 6 shows the same case studies as Figure 4 to show the difference in power consumption and time to deliver a data file. These two figures show that with state resume in data transfer in presence of connection interruption, the more power is saved and the delivery time is reduced. Similarly, the difference is more in transferring larger data files. the reason is that we emulate the interruption after certain percentage of data has been transferred, so without data resuming the larger amount of data need to be resend for larger data size.

To measure the power consumption, we use the PowerTutor tool. The estimated value is not very accurate, however we only focus on the difference between the two power consumption values. Therefore, we believe the value shown in Figure 5 is quite reasonable. From Figure 5 one interesting observation can be made about the power consumption in our device. The power used in network operations is much higher than the power used in CPU operation. So the power saved because of fewer data transfer reduces the total power consumption, despite using more CPU power to compress/decompress data.

5.2 Effects of Data Compression

We also consider the affect of using data compression for SSH protocol. We did the experiments by downloading text files which size is from 500kb to 50MB in both compressed and uncompressed mode and compared the differences. The text files contain tweet entries from different users, with compress ratios chang-

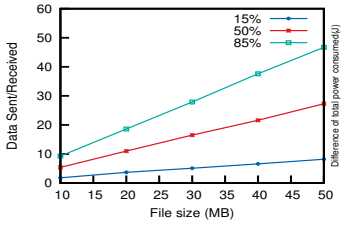

Fig. 4: Difference between total data transferred for two cases with state resume and without state resume for a download command for different file sizes.

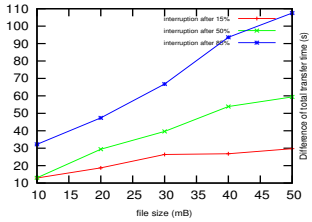

Fig. 5: Difference between total power consumption of Mobile device for two cases with State resume and without state resume for a download command for different file sizes.

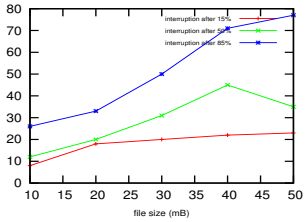

Fig. 6: Difference between data transfer time for two cases with State resume and without state resume for a download command for different file sizes.

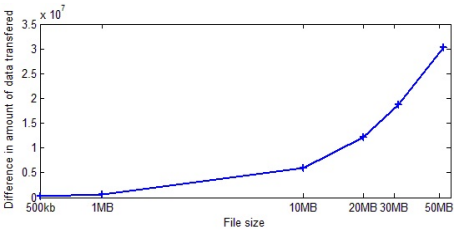

Fig. 7: Difference in the amount of data transfered

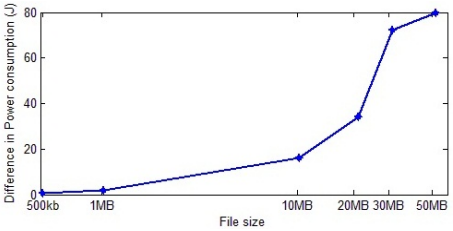

Fig. 8: Difference in transfer time

ing from 40% to 57%. In these experiments, we did not consider connection interruption between clients and server.

Fig. 7 and Fig. 9 shows the saved amount of data transfer over the network and the difference in transfering time. The ratios of saved amount of data transfered are also around 50%, similar to the compress ratios. We also realize that the time saved by using compression is noticeable, which means the time to send data the network is much higher than CPU time needed to compress data. In those experiments, clients and server are in the same local wifi network, in reality the difference will be even higher.

Fig. 8 shows the power we could save by using compression. We considered the total power is sum of the power used by the CPU and the network interface to transmit data. Again, the experiments show that the power we saved by transmitting fewer data over the network is much higher than the power we had to spend more for compressing data. For file size of 50MB, the saved power is about 80 J, approximately one third of the total power used to send the file.

We also tested the effects of data compression on binary data files. However, the results are bad as applying compression on those files could even expand the file size. Therefore, we only recommend data compression when a large amount of text data is transferred between clients and server.

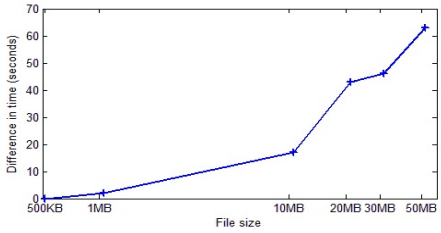

Fig. 9: Difference in Power Consumption

6 Conclusion

SSH is a powerful tool which is widely used by computer users to access remote machines. In this project we have considered the issues related to the mobile users of SSH protocol. our proposed mobile SSH design with resume capability for data transfer results in significant reduction in the network traffic and power consumption while the time to transfer the whole data file decreases, too.We also apply data compression to reduce the total size of the data transferred. The results show that the total power consumption reduces because the computation power of compression is significantly less than the power needed to transfer the data. The experimental results show that the gain obtained by applying resume and compression increases with increase in the size of data file.

References

1. K. Winstein and H. Balakrishnan, "Mosh: An Interactive Remote Shell for Mobile Clients," *Draft Paper* , April 2012.
2. Request for Comments 4253, The Secure Shell (SSH) Transport Protocol. , Jan 2006.
3. Request for Comments 4252, The Secure Shell (SSH) Authentication Protocol. , Jan 2006.
4. Request for Comments 4254, The Secure Shell (SSH) Connection Protocol. , Jan 2006.
5. Power Tutor, *http://powertutor.org/*
6. Connectbot, *http://code.google.com/p/connectbot/*