**Setting up SSH**

The goal of this side-note is to explain how to set up SSH to for the use of MPI. For this purpose, it is necessary to set up SSH so that the server authenticates the client via public key, and the client also accepts the server without the need of user intervention.

**Public key authentication concepts**

**Public key cryptography**

When talking of encryption, the first thing that comes to mind is what is called *symmetric encryption* or *secret key encryption*, where the same (secret) key is used both for encryption and decryption.

However, there are encryption schemes where the encryption and decription keys are distinct (but, of course, they come in pairs), and obtaining the decryption key from the encryption one, while clearly possible, is way too CPU demanding to be practical.

Such an encryption scheme is called *asymmetric encription* or *public key encryption*.

The latter name is justified by the way an asymmetric encryption scheme is used:

1. The recepient of encrypted messages generates a key pair (the paired encryption and decryption keys).
2. The decryption key is published, to make it available to all potential senders; for this reason, the encryption key is also named *public key*.
3. The sender encrypts the message and sends it to the receiver.
4. Only the receiver — the holder of the decryption key, also known as *private key* — is able to decrypt the message.

There are several asymmetric encryption algorithms in use, one of the best known is RSA (Rivest, Shamir, Adelman).

**Digital signatures**

The goal of a digital signature is to allow the receiver of a message to verify if the message was indeed sent by the declared sender and that it had not been modified by anyone else.

The digital signature schemes make use of a pair of transforms using a pair of keys, similarly to the asymmetric cryptography.

A digital signature scheme works as follows:

1. The sender of signed messages creates a key pair. The key pair consists in a *signing key* (private) that is used for creating signatures, and a *verifying key* (public) that is used for verifying signatures.
2. The sender makes public the verifying key.
3. For each message to be sent, the sender creates a *signature*, which is a sequence of bits (somewhat like a hash), created from the message and the signing key.
4. The sender sends the message, together with its signature.
5. The receiver verifies that the signature matches the message, using the verifying key.
6. The security of the scheme relies in the fact that creating a pair of a message and a matching signature is impractically hard, as long as one doesn't know the signing key but only the verifying key.

**Public key authentication of sessions**

When a secured client opens a session to a server, the first thing they do is agree on a unique, randomly generated, *session key*, which is used for encrypting (via a symmetrical algorithm, which is faster) the communication. Agreeing upon a session key can be done via an asymetric cryptography algorithm (the client generates the session key and sends it to the server using the server's public key, and then the server decrypts it using its private key). However, there are better schemes, such as Diffie-Hellman key agreement.

After the session key was established, each party must authenticate the other. More precisely, each party must verify the identity of the other holder of the session key. Note that someone can send and understand messages sent over the encrypted session if and only if that someone knows the session key.

In order to authenticate, the server sends the client the signature of the session key, and the client verifies the signature using the server's public key. In order for this to work, the client must have the server's public key from a trusted source.

In many applications, the server authenticates the client via passwords. However, public key authentication, as above, by having the client sign the session key, can also be performed. To set this up, the server must have the user's public key, and the client program must have access to the user private key.

**Server authentication in SSH**

In SSH, when starting the server for the first time, it generates several key pairs, one for each authentication algorithm it can use. They are stored in files under the /etc/ssh directory. The files come in pairs: one with the .pub extension, which is world-readable (but only root writable), containing the public key, and one with the same name but without the .pub extension, readable only by root, containing the corresponding private key.

The client expects the server public key to be either under the user's directory in ~/.ssh/known\_hosts or in the system-wide configuration in /etc/ssh/ssh\_known\_hosts . Both files are text files and have the same format: on each line, there is the name of the host (or several names and IPs that denote the same host), followed by the signing algorithm name, the public key of the host and an optional comment.

So, to set up the server authentication, copy the public key file from the server and add its content as a line in the known hosts file on the client, while adding the host name in front of it.

**Client authentication in SSH**

Setting up client authentication is a bit harder, as it consists in 3 steps: creating the key pair, putting the public key on the server in the appropriate location, and making the private key available to the client.

**Generating key pairs**

The key pair is generated by using the ssh-keygen command. You can specify the signing algorithm for which the key is to be generated, the length of the key, the file to write the key to, an optional comment, and the password to use for encrypting the private key (more on this later).

Example: generate a 2048-bit RSA key, and write the private key in file keyFile and the public key in keyFile.pub:

ssh-keygen -t rsa -b 2048 -f keyFile

**Setting up the server**

Create a file ~/.ssh/authorized\_keys under the user's home directory and put there the public key(s) (one per line) that are allowed to be used for authenticating as that user.

**Simple client authentication**

If the private key file is not encrypted, it can be used for authentication in either of the following two ways:

1. specify it with the -i option to SSH: ssh -i keyFile <host>...
2. put the file under the name ~/.ssh/id\_rsa (or similar for other algorithms)

Note that an unencrypted private key file is a security risk.

**Setting up the authentication agent**

The best way to handle the private key file is to store it encrypted on disk and to use a process called *authentication agend* to hold the key in decrypted form.

The authentication agent is usually launched upon login and is set up in such a way as all processes created by the current user under that session can communicate to it. If it is not properly started, it can be started by ssh-agend bash . This way, the agent starts and launches a shell; all processes launched from that shell can access the agent.

Adding a key to the agent is done by ssh-add keyFile. The user is prompted for the key password at that moment.

The ssh command automatically tries to contact the agent and to use any key it holds.