SSH

J. Ryan

14 janvier 2003

1 SSH (Secure SHell)

Les réseaux informatiques actuels ne sont pas sûrs. Tout un chacun peut y écouter les communications des autres, et même les modifier s'il y met un peu d'énergie. Cette caractéristique rend ces réseaux impropres à toute utilisation, où l'intégrité et la confidentialité des données ainsi que l'authentification des correspondants sont fondamentaux (voir Annexe A). Les techniques de chiffrement associées à des protocoles sécurisés vont permettre de résoudre ces problèmes.

SSH est à la fois la définition d'un protocole et un ensemble de programmes utilisant ce protocole, destinés à permettre aux utilisateurs d'ouvrir, depuis une machine cliente (Voir Annexe B), des sessions interactives à distance sur des serveurs et de transférer des fichiers entre les deux. La figure résume cette situation (en noir, les machines impliquées ; en bleu, les applications ; en rouge, la connexion encryptée) :

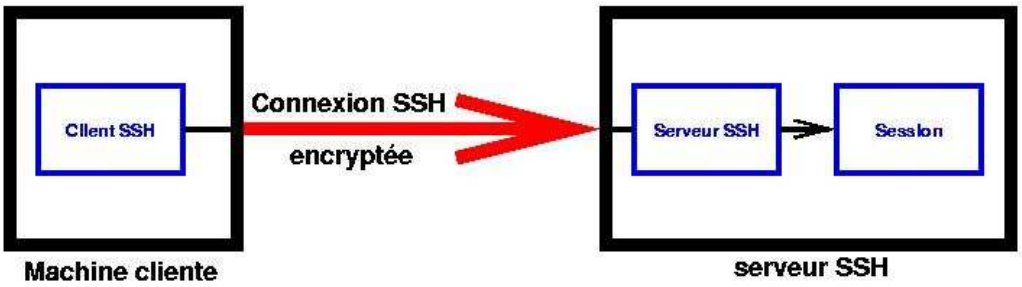

FIG. 1 – Connection SSH encryptée

SSH est basée sur une identification des utilisateurs en utilisant les protocoles d'identification RSA et DSA.

1.1 Le système RSA (DSA est semblable)

Le premier algorithme de chiffrement à clef publique a été développé par R.Merckle et M.Hellman en 1977. Il fut vite obsolète grâce aux travaux de Shamir, Zippel et Herlestman, de célèbres cryptanaliseurs.

En 1978, l'algorithme à clé publique de Rivest Shamir et Adelman (RSA) apparaît. Il sert encore à l'aube de l'an 2000 à protéger les codes nucléaires de l'armée américaine et soviétique...

1.2 fonctionnement de RSA

Le fonctionnement du cryptosystème RSA est basé sur la difficulté de factoriser deux entiers.

Soit deux nombres premiers p et q, et d un entier premier avec p-1 et q-1. Le triplet (p,q,d) constitue ainsi la clé privée.

La clé publique est alors le doublet (n,e) créé à l'aide de la clé privée par les transformations suivantes :

$n = p * q$

$e = 1/dmod((p-1)(q-1))$

Si un utilisateur désire envoyer un message M à un destinataire, il lui suffit de se procurer la clé publique (n,e) de ce dernier, puis de calculer le message chiffré c :

$c = m^e mod(n)$

L'utilisateur envoie ensuite le message chiffré c au destinataire, qui est capable de le déchiffrer à l'aide de sa clé privé (p,q,d) :

$m = m^{ed} mod(n) = c^d mod(n)$

1.3 Remarques

- Il suffit de générer une seule paire de clefs : une clef privée bien protégée et une clef publique (qui n'a pas besoin d'être protégée) que l'on peut mettre sur toute machine où l'on désire se connecter.
- Les développeurs de SSH ont mis quelques protections supplémentaires. *ssh* ne pourra utiliser la clef privée si les droits de lecture ou d'écriture sont attachés à ce fichier.
- Une autre protection est de crypter la clef privée à l'aide d'une passphrase (au moins 7 caractères). Lors de la génération des clefs privées et publiques à l'aide de *ssh-keygen* , une passphrase est demandée. Si aucune passphrase n'est entrée , la clef privée n'est pas cryptée et ouvre la porte de l'insécurité ....

1.4 Génération de clefs privées et publiques *ssh-keygen*

ssh-keygen par défaut génère des clefs avec le système RSA , avec l'option -t DSA , le système DSA est utilisé.

```
bash-2.05$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home1/ryan/.ssh/id_dsa): (tapper Entree)
Enter passphrase (empty for no passphrase): (Entrer une phrase magique)
Enter same passphrase again:(De nouveau entrer une phrase magique)
Your identification has been saved in /home1/ryan/.ssh/id_dsa.
Your public key has been saved in /home1/ryan/.ssh/id_dsa.pub.
The key fingerprint is:
a5:ab:ab:96:10:be:f5:d4:c2:6d:86:ef:dd:39:cf:60 ryan@jupiter
```

1.5 Installation de la clef privée sur remotebox

On veut se connecter sur la machine remotebox par ssh.
Il faut y mettre la clef publique ;

Pour cela

- Etape 1 : on copie la clef sur remotebox
  `scp ~/.ssh/id_dsa.pub ryan@remotebox`
- Etape 2 : On la met au bon endroit :
  On se connecte à la remotebox
  `ssh remotebox`
  Pour l'instant, il faut encore donner son mot de passe.
  Puis
  ```
  cat id_dsa.pub >> ~/.ssh/authorized_keys
  exit
  ```

1.6 Connection sur remotebox

```
ssh remotebox
```

Cette fois ci, remotebox nous demande la passphrase qui a permis d'encrypter la clef :

```
Enter passphrase for key '/home/ryan/.ssh/id_dsa':
```

Si aucune faute de frappe n'a été faite, on est sur remotebox.

A priori, on n'a pas gagné grand chose , à part remplacer un mot de passe par une passphrase.

Mais *ssh-agent* va nous aider.

1.7 *ssh-agent*

ssh-agent est un programme que permet d'utiliser *ssh* de façon plus agréable tout en restant sécurisé. Ce programme met en mémoire cache votre clef privée decryptée. *ssh* communique avec *ssh-agent* et récupère cette clef privée decryptée sans avoir à vous demander la passphrase.

```
ssh-agent  bash
```

donne à votre shell bash des variables d'environnement qui permettront à *ssh* de communiquer avec *ssh-agent*.

Maintenant il reste à mettre en mémoire cache la clef privée decryptée grâce à *ssh-add*

1.8 *ssh-add*

Par défaut, quand vous entrez

```
ssh-add
```

On vous répond

```
Need passphrase for /home/ryan/.ssh/id_dsa
Enter passphrase for /home/ryan/.ssh/id_dsa
(Entrer la  passphrase)
```

Cette action a mis la clef privée decryptée dans la mémoire cache associée a *ssh-agent* et maintenant

1.9 Connection sur remotebox

```
ssh remotebox
```

Cette fois ci, remotebox ne nous demande plus rien et on est sur remotebox.

ssh-agent et *ssh-add* n'ont besoin d'etre fait qu'une seule fois en début de session. Vous pourrez vous connecter autant de fois que vous le désirez sur remotebox sans passphrase depuis cette session. Par contre si vous fermer votre session sur la machine locale et vous reloguez ensuite, il faudra de nouveau activer *ssh-agent* et *ssh-add*.

1.10 Résumé

- Création des clefs sur la machine locale : `ssh-keygen -t dsa`
- Copie de la clef publique sur remotebox
  ```
  scp ~/.ssh/id_dsa.pub ryan@remotebox
  ssh remotebox
  cat id_dsa.pub >> ~/.ssh/authorized_keys
  exit
  ```
- Pour mettre en mémoire la clef privée décryptée par *ssh-agent*
  ```
  ssh-agent bash
  ssh-add
  ```
- Et maintenant on se connecte en douceur sur remotebox.

1.11 Utilisation de la couche X locale

Il est possible de créer des fenêtres X sur votre machine locale, fenêtres lancées depuis un service situé sur la machine distante.

Exemple : Vous voulez utiliser matlab qui n'existe pas sur votre machine locale mais qui est installé sur remotebox.

Il suffit de se connecter sur remotebox avec l'option -X

```
ssh -X remotebox  (vous etes sur remotebox)
matlab
```

La fenêtre matlab apparaîtra sur votre machine locale.

1.12 Connection à une troisième machine

Vous êtes sur **lappy** et voulez vous connecter sur **notrust1** , machine connue de **trustbox** par une connection sécurisée.

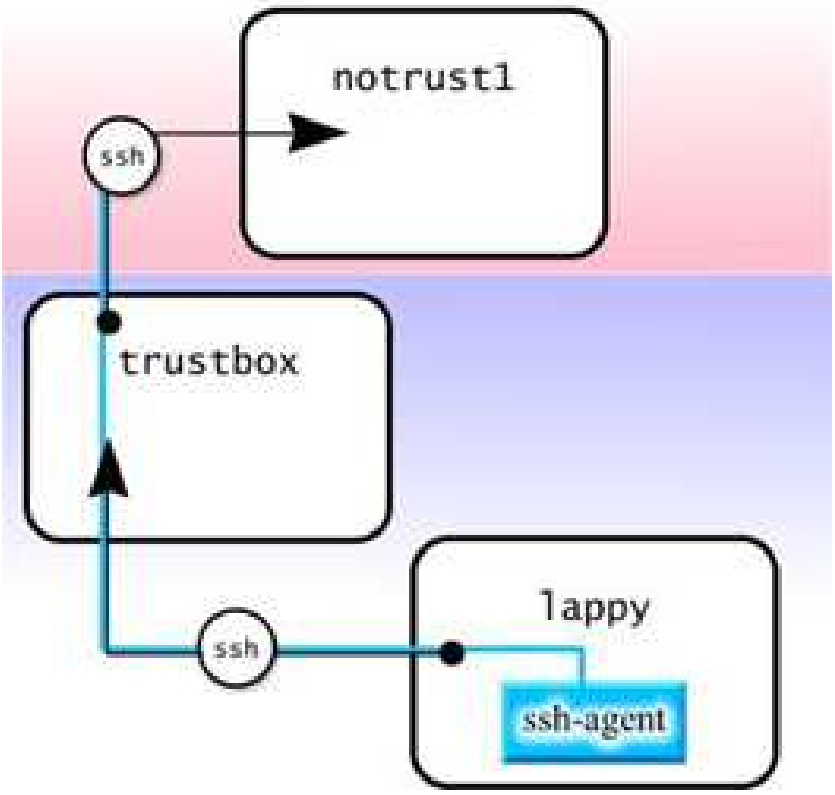

FIG. 2 – Autre connection SSH encryptée

Il suffit de copier votre clef publique sur **trustbox** et **notrust1** par la commande scp :

```
scp ~/.ssh/id_dsa.pub ryan@trustbox
ssh trustbox
cat id_dsa.pub >> ~/.ssh/authorized_keys
scp ~/id_dsa.pub ryan@notrust1
ssh notrust1
cat id_dsa.pub >> ~/.ssh/authorized_keys
exit
exit     (on est revenu sur lappy)
```

Et pour se connecter après avoir lancé *ssh-agent* et *ssh-add* sur **lappy** :

```
1)  ssh -A trustbox
```

Aucun mot de passe ou passphrase ne sera demandé.
L'option -A permet de garder une connection avec *ssh-agent* de **lappy**.
Et maintenant depuis trustbox

```
2)  ssh  notrust1
```

permet l'accès à **notrust1** sans mot de passe ou passphrase, car le *ssh* de **trustbox** a connaissance de la clef privée décryptée située dans la mémoire cache de *ssh-agent* de **lappy**.

Une autre manière de procéder est de passer par un tunnel.

2 Tunnels SSH

Avant de parler des *tunnels SSH*, rappelons le point suivant qui est nécessaire à leur compréhension (pour plus d'explication , voir Annexe C). Une application utilisant le réseau se connecte à un *serveur* donné sur un *port* donné, correspondant à un service spécifique. Par exemple, votre outil de courrier électronique se connecte au serveur `mail.fr` sur le port 25 (port **SMTP**) pour lui confier le message que vous désirez envoyer à votre correspondant.

Les clients *SSH* offrent la possibilité de créer des *tunnels SSH*. A travers un menu ou par des options sur la ligne de commande (selon le client utilisé), on peut spécifier le serveur et le port (service) de ce serveur qui seront la cible du tunnel. Notons que ce serveur est en général différent du serveur *SSH* sur lequel le client a ouvert une session interactive. Le logiciel client *SSH* va alors ouvrir ce même port sur la machine cliente *SSH* (il est possible de faire ouvrir un port différent, mais ceci est rarement utilisé dans la pratique). Quand une application se connecte à ce port, ouvert sur le client *SSH*, le logiciel *SSH* sur le client *SSH* et sur le serveur *SSH* collaborent pour que les données transmises sur ce port parviennent au serveur cible du tunnel, et inversément, les données émises par ce serveur parviennent à l'application connectée sur le port source du tunnel.

En résumé, on a la situation illustrée sur la figure suivante :

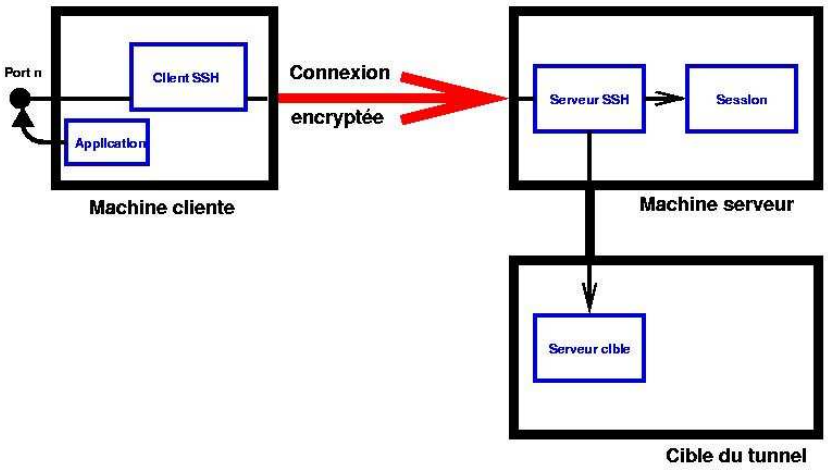

FIG. 3 – Tunnel

2.1 Transmission de ports

– Pour acceder à un service sur un serveur depuis la machine locale en passant par une machine portail "remote"

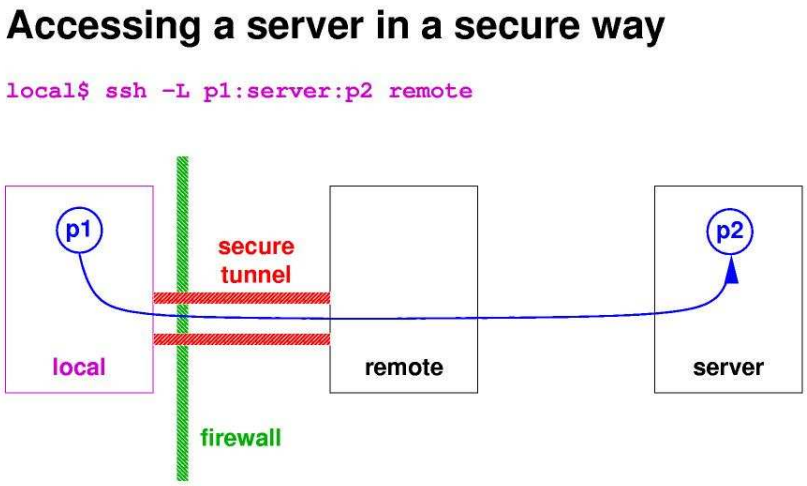

FIG. 4 – Accès à un service sur un serveur

– Création de la connection du port p1 du serveur local au port p2 du serveur cible en passant la machine remote
`ssh -L p1:serveur:p2 remote`

– Permettre à un service du serveur associé au port p2 d'acceder à un port local p1.

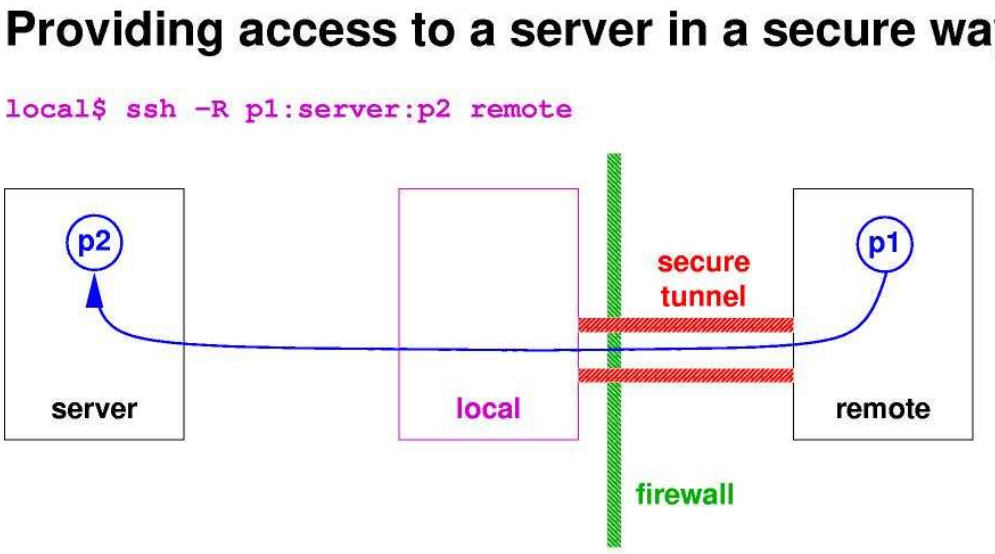

FIG. 5 – Permettre à un service du serveur d'accéder à un port local

– Création de la connection sur le serveur
  `ssh -R p1:serveur:p2 remote`

2.2 Exemples d'utilisation de tunnels SSH

Voici quelques exemples simples pour illustrer l'utilisation des tunnels *SSH* :

– l'utilisateur ouvre un tunnel pour le port 22 (*ssh*) du serveur cible de son choix.
  `ssh -L 8022:serveur:22 remote`
  Alors, en entrant
  `ssh -p 8022 localhost`
  sur la ligne de commande , il se trouvera connecté au serveur cible et verra le *prompt* l'invitant à entrer son *username* et son mot de passe pour la procédure de *ssh* sur le serveur cible.

– l'utilisateur ouvre un tunnel pour le port 23 (*telnet*) du serveur cible de son choix.
  `ssh -L 8023:serveur:23 remote`
  Alors, en entrant
  `telnet localhost`

sur la ligne de commande (ou bien, de manière équivalente, en demandant à son *browser* d'ouvrir l'*URL* `telnet://localhost/`) il se trouvera connecté au serveur cible et verra le *prompt* l'invitant à entrer son *username* et son mot de passe pour la procédure de *login* sur le serveur cible.

– l'utilisateur ouvre un tunnel pour le port 80 (*HTTP* ou *Web*) par exemple sur le serveur cible `www.jupiter.math.univ-paris13.fr`. En demandant à son *browser* de visiter l'*URL* `http://localhost/`, il accédera en fait à la page d'accueil du serveur *Web* de l'université.

Comme on le voit, l'utilisation des tunnels *SSH* passe par une connexion locale (sur le client *SSH*) pour atteindre le serveur cible. A ce sujet, rappelons une convention utilisée par toutes les implémentations du protocole de communication *TCP/IP* : la machine locale peut se contacter sous le nom `localhost` et l'adresse numérique **127.0.0.1**

3 Annexe A : Sécurité des réseaux

3.1 L'intégrité des données

On doit être en mesure de détecter si une donnée à été frauduleusement modifée, et/ou être en mesure d'empêcher totalement ce genre de modification.

Une solution est de chiffrer le message avec une clé.

3.2 Algorithmes de chiffrement

3.2.1 Algorithmes à clé privée

Dans les algorithmes à clé privée, la clé de chiffrement, est identique à la clé de déchiffrement. Cette clé, unique, doit donc être gardée secrète par son propriétaire. L'inconvénient évident de ce système, est que l'envoyeur et le destinataire du message doivent avoir convenu de la clé avant l'envoi du message. Ils doivent donc disposer d'un canal sûr, pour s'échanger des clés. Ceci pose problème sur Internet où il n'y a pas de canaux sûrs. Un exemple d'un tel algorithme est le très fameux DES (Data Encryption System).

3.2.2 Algorithmes à clé publique

Dans ce cas, les clés de chiffrement et de déchiffrement sont distinctes, et généralement symétriques : la clé de chiffrement permet de déchiffrer ce qui a été chiffré avec la clé de déchiffrement, et vice versa. L'heureux possesseur d'une telle paire de clés, en rend une (au choix) publique, c'est-à-dire qu'il la donne à tout le monde, dans une sorte d'annuaire. Tout correspondant qui veut envoyer un message, chiffre son message à l'aide de la clé publique du destinataire. Seul le possesseur de la clé secrète correspondant à cette clé publique pourra déchiffrer le message.

Les algorithmes de chiffrement à clé publique permettent aussi à l'envoyeur de signer son message. En effet, il lui suffit de chiffrer, le message (ou une fonction de ce message) avec sa propre clé secrète. Le destinataire déchiffrera cette fonction avec la clé publique de l'envoyeur et sera ainsi certain de l'identité de l'envoyeur, puisqu'il est le seul à posséder la clé secrète qui permette de faire un tel chiffrement. Subtil n'est-ce pas. RSA (Rivest, Shamir, Adleman, les 3 inventeurs) est un tel algorithme. DSA en est une variation.

3.3 Algorithmes mixtes

Les algorithmes à clé publique sont assez lents. La méthode généralement utilisée pour envoyer un message, est de tirer au hasard une clé secrète, chiffrer le message avec un algorithme à clé privée en utilisant cette clé, puis chiffrer cette clé aléatoire elle-même avec la clé publique du destinataire. Ceci permet d'avoir la sécurité des systèmes à clé publique, avec la performance des systèmes à clé privée.

3.4 Différentes attaques

Que peut encore faire un éventuel pirate face à cet arsenal. Les attaques usuelles sont :

3.4.1 Lecture frauduleuse d'un message

Impossible à cause du chiffrement.

3.4.2 Modification frauduleuse d'un message

Impossible pour la même raison.

3.4.3 Envoi d'un faux message

Impossible à cause de la signature électronique.

3.4.4 Duplication d'un message

C'est le cas où une personne mal intentionnée, copie un message en le voyant passer (sans le comprendre), mais se doute de sa teneur, et le renvoie identique un peu plus tard. On peut imaginer qu'un tel message était un ordre de versement sur un compte banquaire. Pour empêcher cela, au début de la communication, le destinataire tire aléatoirement une clé, l'envoi à l'envoyeur, qui va la joindre au message (avant chiffrement). Quand le scélérat tentera de rejouer la communication, une clé aléatoire différente (espérons) sera choisie, et le récepteur se rendra compte de la supercherie.

4 Annexe B : Système client/serveur

De nombreuses applications fonctionnent selon un environnement client/serveur, cela signifie que des **machines clientes** (des machines faisant partie du réseau) contactent un **serveur**, une machine généralement très puissante en terme de capacités d'entrée-sortie, qui leur fournit des **services**. Ces services sont des programmes fournissant des données telles que l'heure, des fichiers, une connexion, ...

Les services sont exploités par des programmes, appelés **programmes clients**, s'exécutant sur les machines clientes. On parle ainsi de client FTP, client de messagerie, ..., lorsque l'on désigne un programme, tournant sur une machine cliente, capable de traiter des informations qu'il récupère auprès du serveur (dans le cas du client FTP il s'agit de fichiers, tandis que pour le client messagerie il s'agit de courrier électronique).

Dans un environnement purement Client/serveur, les ordinateurs du réseau (les clients) ne peuvent voir que le serveur, c'est un des principaux atouts de ce modèle. Avantages de l'architecture client/serveur

Le modèle client/serveur est particulièrement recommandé pour des réseaux nécessitant un grand niveau de fiabilité, ses principaux atouts sont :

- **des ressources centralisées** : étant donné que le serveur est au centre du réseau, il peut gérer des ressources communes à tous les utilisateurs, comme par exemple une base de données centralisée, afin d'éviter les problèmes de redondance et de contradiction
- **une meilleure sécurité** : car le nombre de points d'entrée permettant l'accès aux données est moins important
- **une administration au niveau serveur** : les clients ayant peu d'importance dans ce modèle, ils ont moins besoin d'être administrés
- **un réseau évolutif** : grâce à cette architecture il est possible de supprimer ou rajouter des clients sans perturber le fonctionnement du réseau et sans modifications majeures

Inconvénients du modèle client/serveur

L'architecture client/serveur a tout de même quelques lacunes parmi lesquelles :

- **un coût élevé** dû à la technicité du serveur
- **un maillon faible** : le serveur est le seul maillon faible du réseau client/serveur, étant donné que tout le réseau est architecturé autour de lui ! Heureusement, le serveur a une grande tolérance aux pannes (notamment grâce au système RAID)

Fonctionnement d'un système client/serveur

Un système client/serveur fonctionne selon le schéma suivant :

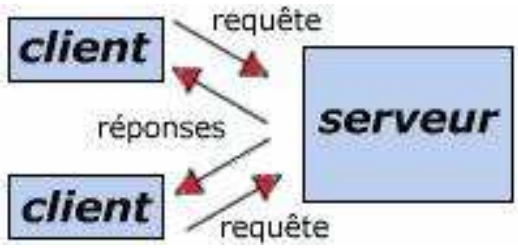

FIG. 6 – Modèle client/serveur

5 Annexe C : Quelques notions d'applications réseau

L'utilisation du réseau par les applications est basée sur le modèle client/serveur :

- l'application cliente (avec laquelle interagit l'utilisateur) aura besoin d'un service fourni par un serveur distant. Elle contacte ce dernier à travers le réseau pour lui soumettre une requête.
- Le serveur transmet les données demandées au client, par la même connexion réseau (celle-ci est bi-directionnelle).
- Une même machine peut être serveur pour plusieurs services simultanément. Ils sont distingués par le *port* que doit utiliser le client pour contacter tel ou tel service.
- Ces ports sont implémentés par l'utilisation d'entiers (entre 1 et 65536) auxquels on a souvent donnés un nom : par exemple, on parle sans distinction du "port 80", du "port HTTP" ou du "port Web" pour dénoter le port d'un serveur *Web* que doit contacter votre *browser* (application cliente) pour obtenir la page *HTML* que vous lui demander. Voir ci-dessous pour une liste de tels ports.

Un résumé en image :

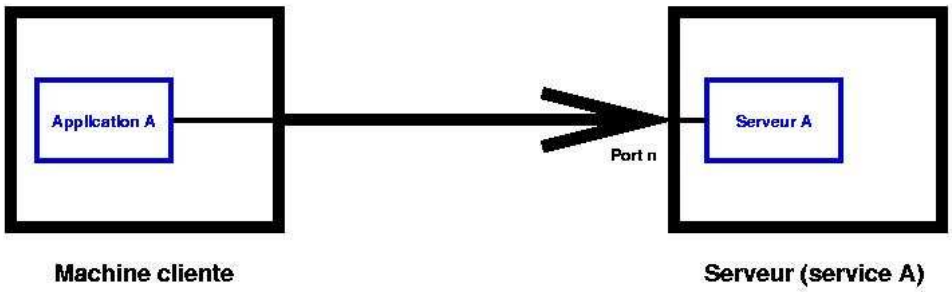

FIG. 7 – Réseau

5.1 Tableau des ports usuels

Il existe des milliers de ports (ceux-ci sont codés sur 16 bits,il y a donc 65536 possibilités), c'est pourquoi une assignation standard a été mise au point, afin d'aider à la configuration des réseaux.

Voici certaines de ces assignations par défaut :

Port (numérique)	Nom	Client typique
21	FTP	outils FTP (WS-ftp, ncftp, browser pour les URL `ftp://serveur/fichier`)
22	SSH	clients SSH (ssh, ttssh, niftytelnet, mindterm)
23	telnet	telnet, émulateurs de terminal (TeraTerm)
25	SMTP, Mail	outils de mail (Eudora, Netscape, Outlook) pour envoyer des messages
80	HTTP	Web browser (Internet Explorer, Netscape)
110	POP	outils de mail (Eudora, Netscape, Outlook) pour chercher les messages
119	NNTP	News lecteurs de News (xrn, Netscape, Outlook)
513	LOGIN	rlogin
514	SHELL	rsh, rcp
1494	ICA	Independant Computer Architecture de Citrix
3389	MSRDP	Terminal Server/Remote Display Protocol de Microsoft
5800	VNC	Applications Virtual Network Computer (port de chargement du client Java)
5900	VNC	Applications Virtual Network Computer

Les ports 0 à 1023 sont les ports reconnus ou réservés (Well Known Ports). Ils sont assignés par le IANA (Internet Assigned Numbers Authority) et sont, sur beaucoup de systèmes, uniquement utilisables par les processus système ou les programmes exécutés par des utilisateurs privilégiés. Un administrateur réseau peut toutefois lier des services aux ports de son choix.
Les ports 1024 à 49151 sont appelés *ports enregistrés* (Registered Ports)
Les ports 49152 à 65535 sont les *ports dynamiques ou privés*

Ainsi, un serveur (un ordinateur que l'on contacte et qui propose des services tels que FTP, Telnet, ...) possède des numéros de port fixes auxquels l'administrateur réseau a associé des services. Ainsi, les ports d'un serveur sont généralement compris entre 0 et 1023 (fourchette de valeurs associées à des services connus).

Du côté du client, le port est choisi aléatoirement parmi ceux disponibles par le système d'exploitation. Ainsi, les ports du clients ne seront jamais compris entre 0 et 1023 car cet intervalle de valeurs représente les *ports connus*.

5.2 Exemples

- Vous ouvrez votre outil de *mail* (*Eudora*, *Netscape*, *Outlook*, etc.) et cliquez sur le bouton "Recevoir" pour aller chercher les messages que vous avez reçus. Selon la description ci-dessus, votre outil se connecte au serveur où arrivent ces messages, soit sur le port 110 (**POP**), soit sur le port 143 (**IMAP**), et demande de recevoir les nouvelles missives. (**POP** et **IMAP** sont deux manières fonctionnellement équivalentes d'aller chercher son courrier électronique sur un serveur, la différence principale étant qu'avec **IMAP** une copie des messages reste sur le serveur.)
  Si c'est votre responsable informatique qui a configuré votre poste de travail, tout ceci peut rester caché et vous ne savez pas quel serveur est contacté, ni quel port (110 ou 143) est utilisé, mais ces renseigments figurent toujours quelque part dans les menus de configuration de votre outil de mail.
- Vous rédigez alors une réponse à votre correspondant, que vous envoyez par le bouton "Envoyer" : votre outil de *mail* se connecte au serveur **SMTP** (par exemple `mail.epfl.ch`) sur le port 25 et lui transmet votre message pour qu'il s'occupe de son acheminement.
  Comme ci-dessus, vous ne savez peut-être pas quel serveur **SMTP** est utilisé, mais ce renseigment figure quelque part dans les menus de configuration de votre outil de mail.
- Un exemple simple où le nom du serveur apparaît toujours clairement vous est fourni chaque fois que vous entrez un *URL* (**U**niversal **R**esource **L**ocator) dans votre *browser*. Ces *URL* ont le format général :
  `http://serveur.domaine/d1/d2/.../page.html`
  Pour chercher la page ainsi référencée, votre *browser* contacte le serveur `serveur.domaine` (par exemple `www.epfl.ch`) sur le port 80 (port **HTTP**) et lui demande le fichier `d1/d2/.../page.html`.
- Les *URL* peuvent aussi contenir un port explicite autre que le port 80 (port **HTTP**), comme suit :
  `http://serveur.domaine:1234/d1/d2/.../page.html`
  Pour chercher la page ainsi référencée, votre *browser* contacte le serveur `serveur.domaine` (par exemple `www.jupiter.math.univ-paris13.fr`) sur le port 1234 et lui demande le fichier `d1/d2/.../page.html`.
  Il s'agit-là d'un des rares cas où un port numérique est visible à l'utilisateur.

6 Annexe D : man des principales fonctions

6.1 man ssh

```
SSH(1)

NAME
     ssh - OpenSSH SSH client (remote login program)

SYNOPSIS
     ssh [-l login_name] hostname | user@hostname [command]

     ssh [-afgknqstvxACNPTX1246] [-b bind_address] [-c cipher_spec]
         [-e escape_char] [-i identity_file] [-l login_name] [-m mac_spec]
         [-o option] [-p port] [-F configfile] [-L port:host:hostport] [-R
         port:host:hostport] [-D port] hostname | user@hostname [command]

DESCRIPTION
     ssh (SSH client) is a program for logging into a remote machine and for
     executing commands on a remote machine.  It is intended to replace rlogin
     and rsh, and provide secure encrypted communications between two
     untrusted hosts over an insecure network.  X11 connections and arbitrary
     TCP/IP ports can also be forwarded over the secure channel.

     ssh connects and logs into the specified hostname.  The user must prove
     his/her identity to the remote machine using one of several methods
     depending on the protocol version used:

   SSH protocol version 1

     First, if the machine the user logs in from is listed in /etc/hosts.equiv
     or /etc/ssh/shosts.equiv on the remote machine, and the user names are
     the same on both sides, the user is immediately permitted to log in.
     Second, if .rhosts or .shosts exists in the user's home directory on the
     remote machine and contains a line containing the name of the client
     machine and the name of the user on that machine, the user is permitted
     to log in.  This form of authentication alone is normally not allowed by
     the server because it is not secure.

     The second authentication method is the rhosts or hosts.equiv method com-
```

bined with RSA-based host authentication. It means that if the login would be permitted by $HOME/.rhosts, $HOME/.shosts, /etc/hosts.equiv, or /etc/ssh/shosts.equiv, and if additionally the server can verify the client's host key (see /etc/ssh/ssh_known_hosts and $HOME/.ssh/known_hosts in the FILES section), only then login is permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing. [Note to the administrator: /etc/hosts.equiv, $HOME/.rhosts, and the rlogin/rsh protocol in general, are inherently insecure and should be disabled if security is desired.]

As a third authentication method, ssh supports RSA based authentication. The scheme is based on public-key cryptography: there are cryptosystems where encryption and decryption are done using separate keys, and it is not possible to derive the decryption key from the encryption key. RSA is one such system. The idea is that each user creates a public/private key pair for authentication purposes. The server knows the public key, and only the user knows the private key. The file $HOME/.ssh/authorized_keys lists the public keys that are permitted for logging in. When the user logs in, the ssh program tells the server which key pair it would like to use for authentication. The server checks if this key is permitted, and if so, sends the user (actually the ssh program running on behalf of the user) a challenge, a random number, encrypted by the user's public key. The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the private key, proving that he/she knows the private key but without disclosing it to the server.

ssh implements the RSA authentication protocol automatically. The user creates his/her RSA key pair by running ssh-keygen(1). This stores the private key in $HOME/.ssh/identity and the public key in $HOME/.ssh/identity.pub in the user's home directory. The user should then copy the identity.pub to $HOME/.ssh/authorized_keys in his/her home directory on the remote machine (the authorized_keys file corresponds to the conventional $HOME/.rhosts file, and has one key per line, though the lines can be very long). After this, the user can log in without giving the password. RSA authentication is much more secure than rhosts authentication.

The most convenient way to use RSA authentication may be with an authentication agent. See ssh-agent(1) for more information.

If other authentication methods fail, ssh prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

SSH protocol version 2

When a user connects using protocol version 2 similar authentication methods are available. Using the default values for PreferredAuthentications, the client will try to authenticate first using the hostbased method; if this method fails public key authentication is attempted, and finally if this method fails keyboard-interactive and password authentication are tried.

The public key method is similar to RSA authentication described in the previous section and allows the RSA or DSA algorithm to be used: The client uses his private key, $HOME/.ssh/id_dsa or $HOME/.ssh/id_rsa, to sign the session identifier and sends the result to the server. The server checks whether the matching public key is listed in $HOME/.ssh/authorized_keys and grants access if both the key is found and the signature is correct. The session identifier is derived from a shared Diffie-Hellman value and is only known to the client and the server.

If public key authentication fails or is not available a password can be sent encrypted to the remote host for proving the user's identity.

Additionally, ssh supports hostbased or challenge response authentication.

Protocol 2 provides additional mechanisms for confidentiality (the traffic is encrypted using 3DES, Blowfish, CAST128 or Arcfour) and integrity (hmac-md5, hmac-sha1). Note that protocol 1 lacks a strong mechanism for ensuring the integrity of the connection.

Login session and remote execution

When the user's identity has been accepted by the server, the server either executes the given command, or logs into the machine and gives the user a normal shell on the remote machine. All communication with the remote command or shell will be automatically encrypted.

If a pseudo-terminal has been allocated (normal login session), the user may use the escape characters noted below.

If no pseudo tty has been allocated, the session is transparent and can be used to reliably transfer binary data. On most systems, setting the escape character to ‘‘none’’ will also make the session transparent even if a tty is used.

The session terminates when the command or shell on the remote machine exits and all X11 and TCP/IP connections have been closed. The exit status of the remote program is returned as the exit status of ssh.

Escape Characters

When a pseudo terminal has been requested, ssh supports a number of functions through the use of an escape character.

A single tilde character can be sent as ~~ or by following the tilde by a character other than those described below. The escape character must always follow a newline to be interpreted as special. The escape character can be changed in configuration files using the EscapeChar configuration directive or on the command line by the -e option.

The supported escapes (assuming the default ‘~’) are:

~. Disconnect

~^Z Background ssh

~# List forwarded connections

~&%
Background ssh at logout when waiting for forwarded connection / X11 sessions to terminate

~? Display a list of escape characters

~C Open command line (only useful for adding port forwardings using the -L and -R options)

~R Request rekeying of the connection (only useful for SSH protocol version 2 and if the peer supports it)

X11 and TCP forwarding

If the ForwardX11 variable is set to ‘‘yes’’ (or, see the description of the -X and -x options described later) and the user is using X11 (the DISPLAY environment variable is set), the connection to the X11 display is automatically forwarded to the remote side in such a way that any X11 programs started from the shell (or command) will go through the encrypted channel, and the connection to the real X server will be made from the local machine. The user should not manually set DISPLAY. Forwarding of X11 connections can be configured on the command line or in configuration files.

The DISPLAY value set by ssh will point to the server machine, but with a display number greater than zero. This is normal, and happens because ssh creates a ‘‘proxy’’ X server on the server machine for forwarding the connections over the encrypted channel.

ssh will also automatically set up Xauthority data on the server machine. For this purpose, it will generate a random authorization cookie, store it in Xauthority on the server, and verify that any forwarded connections carry this cookie and replace it by the real cookie when the connection is opened. The real authentication cookie is never sent to the server machine (and no cookies are sent in the plain).

If the user is using an authentication agent, the connection to the agent is automatically forwarded to the remote side unless disabled on the command line or in a configuration file.

Forwarding of arbitrary TCP/IP connections over the secure channel can be specified either on the command line or in a configuration file. One possible application of TCP/IP forwarding is a secure connection to an electronic purse; another is going through firewalls.

Server authentication

ssh automatically maintains and checks a database containing identifications for all hosts it has ever been used with. Host keys are stored in $HOME/.ssh/known_hosts in the user’s home directory. Additionally, the

file /etc/ssh/ssh_known_hosts is automatically checked for known hosts. Any new hosts are automatically added to the user's file. If a host's identification ever changes, ssh warns about this and disables password authentication to prevent a trojan horse from getting the user's password. Another purpose of this mechanism is to prevent man-in-the-middle attacks which could otherwise be used to circumvent the encryption. The StrictHostKeyChecking option can be used to prevent logins to machines whose host key is not known or has changed.

The options are as follows:

-a Disables forwarding of the authentication agent connection.

-A Enables forwarding of the authentication agent connection. This can also be specified on a per-host basis in a configuration file.

-b bind_address
Specify the interface to transmit from on machines with multiple interfaces or aliased addresses.

-c blowfish|3des|des
Selects the cipher to use for encrypting the session. 3des is used by default. It is believed to be secure. 3des (triple-des) is an encrypt-decrypt-encrypt triple with three different keys. blowfish is a fast block cipher, it appears very secure and is much faster than 3des. des is only supported in the ssh client for interoperability with legacy protocol 1 implementations that do not support the 3des cipher. Its use is strongly discouraged due to cryptographic weaknesses.

-c cipher_spec
Additionally, for protocol version 2 a comma-separated list of ciphers can be specified in order of preference. See Ciphers for more information.

-e ch|^ch|none
Sets the escape character for sessions with a pty (default: '~'). The escape character is only recognized at the beginning of a line. The escape character followed by a dot ('.') closes the connection, followed by control-Z suspends the connection, and

followed by itself sends the escape character once. Setting the character to ‘‘none’’ disables any escapes and makes the session fully transparent.

-f Requests ssh to go to background just before command execution. This is useful if ssh is going to ask for passwords or passphrases, but the user wants it in the background. This implies -n. The recommended way to start X11 programs at a remote site is with something like ssh -f host xterm.

-g Allows remote hosts to connect to local forwarded ports.

-i identity_file
Selects a file from which the identity (private key) for RSA or DSA authentication is read. The default is $HOME/.ssh/identity for protocol version 1, and $HOME/.ssh/id_rsa and $HOME/.ssh/id_dsa for protocol version 2. Identity files may also be specified on a per-host basis in the configuration file. It is possible to have multiple -i options (and multiple identities specified in configuration files).

-I smartcard_device
Specifies which smartcard device to use. The argument is the device ssh should use to communicate with a smartcard used for storing the user's private RSA key.

-k Disables forwarding of Kerberos tickets and AFS tokens. This may also be specified on a per-host basis in the configuration file.

-l login_name
Specifies the user to log in as on the remote machine. This also may be specified on a per-host basis in the configuration file.

-m mac_spec
Additionally, for protocol version 2 a comma-separated list of MAC (message authentication code) algorithms can be specified in order of preference. See the MACs keyword for more information.

-n Redirects stdin from /dev/null (actually, prevents reading from stdin). This must be used when ssh is run in the background. A common trick is to use this to run X11 programs on a remote

machine. For example, ssh -n shadows.cs.hut.fi emacs &%
will
start an emacs on shadows.cs.hut.fi, and the X11 connection will be automatically forwarded over an encrypted channel. The ssh program will be put in the background. (This does not work if ssh needs to ask for a password or passphrase; see also the -f option.)

-N Do not execute a remote command. This is useful for just forwarding ports (protocol version 2 only).

-o option
Can be used to give options in the format used in the configuration file. This is useful for specifying options for which there is no separate command-line flag.

-p port
Port to connect to on the remote host. This can be specified on a per-host basis in the configuration file.

-P Use a non-privileged port for outgoing connections. This can be used if a firewall does not permit connections from privileged ports. Note that this option turns off RhostsAuthentication and RhostsRSAAuthentication for older servers.

-q Quiet mode. Causes all warning and diagnostic messages to be suppressed.

-s May be used to request invocation of a subsystem on the remote system. Subsystems are a feature of the SSH2 protocol which facilitate the use of SSH as a secure transport for other applications (eg. sftp). The subsystem is specified as the remote command.

-t Force pseudo-tty allocation. This can be used to execute arbitrary screen-based programs on a remote machine, which can be very useful, e.g., when implementing menu services. Multiple -t options force tty allocation, even if ssh has no local tty.

-T Disable pseudo-tty allocation.

-v Verbose mode. Causes ssh to print debugging messages about its progress. This is helpful in debugging connection, authentication, and configuration problems. Multiple -v options increases the verbosity. Maximum is 3.

-x Disables X11 forwarding.

-X Enables X11 forwarding. This can also be specified on a per-host basis in a configuration file.

-C Requests compression of all data (including stdin, stdout, stderr, and data for forwarded X11 and TCP/IP connections). The compression algorithm is the same used by gzip(1), and the ‘‘level’’ can be controlled by the CompressionLevel option. Compression is desirable on modem lines and other slow connections, but will only slow down things on fast networks. The default value can be set on a host-by-host basis in the configuration files; see the Compression option.

-F configfile
Specifies an alternative per-user configuration file. If a configuration file is given on the command line, the system-wide configuration file (/etc/ssh/ssh_config) will be ignored. The default for the per-user configuration file is $HOME/.ssh/config.

-L port:host:hostport
Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to port on the local side, and whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to host port hostport from the remote machine. Port forwardings can also be specified in the configuration file. Only root can forward privileged ports. IPv6 addresses can be specified with an alternative syntax: port/host/hostport

-R port:host:hostport
Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to port on the remote side, and whenever a connection is made to this port, the connec-

tion is forwarded over the secure channel, and a connection is made to host port hostport from the local machine. Port forwardings can also be specified in the configuration file. Privileged ports can be forwarded only when logging in as root on the remote machine. IPv6 addresses can be specified with an alternative syntax: port/host/hostport

-D port
Specifies a local ‘‘dynamic’’ application-level port forwarding. This works by allocating a socket to listen to port on the local side, and whenever a connection is made to this port, the connection is forwarded over the secure channel, and the application protocol is then used to determine where to connect to from the remote machine. Currently the SOCKS4 protocol is supported, and ssh will act as a SOCKS4 server. Only root can forward privileged ports. Dynamic port forwardings can also be specified in the configuration file.

-1 Forces ssh to try protocol version 1 only.

-2 Forces ssh to try protocol version 2 only.

-4 Forces ssh to use IPv4 addresses only.

-6 Forces ssh to use IPv6 addresses only.

CONFIGURATION FILES
ssh may additionally obtain configuration data from a per-user configuration file and a system-wide configuration file. The file format and configuration options are described in ssh_config(5).

ENVIRONMENT
ssh will normally set the following environment variables:

DISPLAY
The DISPLAY variable indicates the location of the X11 server. It is automatically set by ssh to point to a value of the form ‘‘hostname:n’’ where hostname indicates the host where the shell runs, and n is an integer
= 1. ssh uses this special value to
forward X11 connections over the secure channel. The user should

normally not set DISPLAY explicitly, as that will render the X11 connection insecure (and will require the user to manually copy any required authorization cookies).

HOME Set to the path of the user's home directory.

LOGNAME
Synonym for USER; set for compatibility with systems that use this variable.

MAIL Set to the path of the user's mailbox.

PATH Set to the default PATH, as specified when compiling ssh.

SSH_ASKPASS
If ssh needs a passphrase, it will read the passphrase from the current terminal if it was run from a terminal. If ssh does not have a terminal associated with it but DISPLAY and SSH_ASKPASS are set, it will execute the program specified by SSH_ASKPASS and open an X11 window to read the passphrase. This is particularly useful when calling ssh from a .Xsession or related script. (Note that on some machines it may be necessary to redirect the input from /dev/null to make this work.)

SSH_AUTH_SOCK
Identifies the path of a unix-domain socket used to communicate with the agent.

SSH_CLIENT
Identifies the client end of the connection. The variable contains three space-separated values: client ip-address, client port number, and server port number.

SSH_ORIGINAL_COMMAND
The variable contains the original command line if a forced command is executed. It can be used to extract the original arguments.

SSH_TTY
This is set to the name of the tty (path to the device) associated with the current shell or command. If the current session

has no tty, this variable is not set.

TZ The timezone variable is set to indicate the present timezone if it was set when the daemon was started (i.e., the daemon passes the value on to new connections).

USER Set to the name of the user logging in.

Additionally, ssh reads $HOME/.ssh/environment, and adds lines of the format ‘‘VARNAME=value’’ to the environment.

FILES

$HOME/.ssh/known_hosts
Records host keys for all hosts the user has logged into that are not in /etc/ssh/ssh_known_hosts. See sshd(8).

$HOME/.ssh/identity, $HOME/.ssh/id_dsa, $HOME/.ssh/id_rsa
Contains the authentication identity of the user. They are for protocol 1 RSA, protocol 2 DSA, and protocol 2 RSA, respectively. These files contain sensitive data and should be readable by the user but not accessible by others (read/write/execute). Note that ssh ignores a private key file if it is accessible by others. It is possible to specify a passphrase when generating the key; the passphrase will be used to encrypt the sensitive part of this file using 3DES.

$HOME/.ssh/identity.pub, $HOME/.ssh/id_dsa.pub, $HOME/.ssh/id_rsa.pub
Contains the public key for authentication (public part of the identity file in human-readable form). The contents of the $HOME/.ssh/identity.pub file should be added to $HOME/.ssh/authorized_keys on all machines where the user wishes to log in using protocol version 1 RSA authentication. The contents of the $HOME/.ssh/id_dsa.pub and $HOME/.ssh/id_rsa.pub file should be added to $HOME/.ssh/authorized_keys on all machines where the user wishes to log in using protocol version 2 DSA/RSA authentication. These files are not sensitive and can (but need not) be readable by anyone. These files are never used automatically and are not necessary; they are only provided for the convenience of the user.

$HOME/.ssh/config

This is the per-user configuration file. The file format and configuration options are described in ssh_config(5).

$HOME/.ssh/authorized_keys
: Lists the public keys (RSA/DSA) that can be used for logging in as this user. The format of this file is described in the sshd(8) manual page. In the simplest form the format is the same as the .pub identity files. This file is not highly sensitive, but the recommended permissions are read/write for the user, and not accessible by others.

/etc/ssh/ssh_known_hosts
: Systemwide list of known host keys. This file should be prepared by the system administrator to contain the public host keys of all machines in the organization. This file should be world-readable. This file contains public keys, one per line, in the following format (fields separated by spaces): system name, public key and optional comment field. When different names are used for the same machine, all such names should be listed, separated by commas. The format is described on the sshd(8) manual page.

  The canonical system name (as returned by name servers) is used by sshd(8) to verify the client host when logging in; other names are needed because ssh does not convert the user-supplied name to a canonical name before checking the key, because someone with access to the name servers would then be able to fool host authentication.

/etc/ssh/ssh_config
: Systemwide configuration file. The file format and configuration options are described in ssh_config(5)\.

/etc/ssh/ssh_host_key, /etc/ssh/ssh_host_dsa_key,
/etc/ssh/ssh_host_rsa_key
: These three files contain the private parts of the host keys and are used for RhostsRSAAuthentication and HostbasedAuthentication. If the protocol version 1 RhostsRSAAuthentication method is used, ssh must be setuid root, since the host key is readable only by root. For protocol version 2, ssh uses ssh-keysign(8)to access the host keys for HostbasedAuthentication. This eliminates the

requirement that ssh be setuid root when that authentication method is used. By default ssh is not setuid root.

$HOME/.rhosts
: This file is used in .rhosts authentication to list the host/user pairs that are permitted to log in. (Note that this file is also used by rlogin and rsh, which makes using this file insecure.) Each line of the file contains a host name (in the canonical form returned by name servers), and then a user name on that host, separated by a space. On some machines this file may need to be world-readable if the user's home directory is on a NFS partition, because sshd(8) reads it as root. Additionally, this file must be owned by the user, and must not have write permissions for anyone else. The recommended permission for most machines is read/write for the user, and not accessible by others.

  Note that by default sshd(8) will be installed so that it requires successful RSA host authentication before permitting .rhosts authentication. If the server machine does not have the client's host key in /etc/ssh/ssh_known_hosts, it can be stored in $HOME/.ssh/known_hosts. The easiest way to do this is to connect back to the client from the server machine using ssh; this will automatically add the host key to $HOME/.ssh/known_hosts.

$HOME/.shosts
: This file is used exactly the same way as .rhosts. The purpose for having this file is to be able to use rhosts authentication with ssh without permitting login with rlogin or rsh(1)\footnote{See U

/etc/hosts.equiv
: This file is used during .rhosts authentication. It contains canonical hosts names, one per line (the full format is described on the sshd(8)\footnote{. If the client host is found in this file, login is automatically permitted provided client and server user names are the same. Additionally, successful RSA host authentication is normally required. This file should only be writable by root.

/etc/ssh/shosts.equiv
: This file is processed exactly as /etc/hosts.equiv. This file may be useful to permit logins using ssh but not using

rsh/rlogin.

/etc/ssh/sshrc
: Commands in this file are executed by ssh when the user logs in just before the user's shell (or command) is started. See the sshd(8 manual page for more information.

$HOME/.ssh/rc
: Commands in this file are executed by ssh when the user logs in just before the user's shell (or command) is started. See the sshd(8) manual page for more information.

$HOME/.ssh/environment
: Contains additional definitions for environment variables, see section ENVIRONMENT above.

DIAGNOSTICS

ssh exits with the exit status of the remote command or with 255 if an error occurred.

AUTHORS

OpenSSH is a derivative of the original and free ssh 1.2.12 release by Tatu Ylonen. Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt and Dug Song removed many bugs, re-added newer features and created OpenSSH. Markus Friedl contributed the support for SSH protocol versions 1.5 and 2.0.

6.2 man 1 ssh-agent

```
 SSH-AGENT(1)

NAME
     ssh-agent - authentication agent

SYNOPSIS
     ssh-agent [-a bind_address] [-c | -s] [-d] [command [args ...]]
     ssh-agent [-c | -s] -k

DESCRIPTION
     ssh-agent is a program to hold private keys used for public key authenti-
     cation (RSA, DSA).  The idea is that ssh-agent is started in the begin-
     ning of an X-session or a login session, and all other windows or pro-
     grams are started as clients to the ssh-agent program.  Through use of
     environment variables the agent can be located and automatically used for
     authentication when logging in to other machines using ssh(1).

     The options are as follows:

     -a bind_address
             Bind the agent to the unix-domain socket bind_address.  The
             default is /tmp/ssh-XXXXXXXX/agent.<%
ppid>%
.

     -c      Generate C-shell commands on stdout.  This is the default if
             SHELL looks like it's a csh style of shell.

     -s      Generate Bourne shell commands on stdout.  This is the default if
             SHELL does not look like it's a csh style of shell.

     -k      Kill the current agent (given by the SSH_AGENT_PID environment
             variable).

     -d      Debug mode.  When this option is specified ssh-agent will not
             fork.

     If a commandline is given, this is executed as a subprocess of the agent.
```

When the command dies, so does the agent.

The agent initially does not have any private keys. Keys are added using ssh-add(1). When executed without arguments, ssh-add(1) adds the files $HOME/.ssh/id_rsa, $HOME/.ssh/id_dsa and $HOME/.ssh/identity. If the identity has a passphrase, ssh-add(1) asks for the passphrase (using a small X11 application if running under X11, or from the terminal if running without X). It then sends the identity to the agent. Several identities can be stored in the agent; the agent can automatically use any of these identities. ssh-add -l displays the identities currently held by the agent.

The idea is that the agent is run in the user's local PC, laptop, or terminal. Authentication data need not be stored on any other machine, and authentication passphrases never go over the network. However, the connection to the agent is forwarded over SSH remote logins, and the user can thus use the privileges given by the identities anywhere in the network in a secure way.

There are two main ways to get an agent setup: Either the agent starts a new subcommand into which some environment variables are exported, or the agent prints the needed shell commands (either sh(1) or csh(1) syntax can be generated) which can be evalled in the calling shell. Later ssh(1) looks at these variables and uses them to establish a connection to the agent.

The agent will never send a private key over its request channel. Instead, operations that require a private key will be performed by the agent, and the result will be returned to the requester. This way, private keys are not exposed to clients using the agent.

A unix-domain socket is created and the name of this socket is stored in the SSH_AUTH_SOCK environment variable. The socket is made accessible only to the current user. This method is easily abused by root or another instance of the same user.

The SSH_AGENT_PID environment variable holds the agent's process ID.

The agent exits automatically when the command given on the command line terminates.

FILES

$HOME/.ssh/identity

Contains the protocol version 1 RSA authentication identity of the user.

$HOME/.ssh/id_dsa

Contains the protocol version 2 DSA authentication identity of the user.

$HOME/.ssh/id_rsa

Contains the protocol version 2 RSA authentication identity of the user.

/tmp/ssh-XXXXXXXX/agent.<%
ppid>%

Unix-domain sockets used to contain the connection to the authentication agent. These sockets should only be readable by the owner. The sockets should get automatically removed when the agent exits.

AUTHORS

OpenSSH is a derivative of the original and free ssh 1.2.12 release by Tatu Ylonen. Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt and Dug Song removed many bugs, re-added newer features and created OpenSSH. Markus Friedl contributed the support for SSH protocol versions 1.5 and 2.0.

6.3 man 1 ssh-add

```
 SSH-ADD(1)

NAME
     ssh-add - adds RSA or DSA identities to the authentication agent

SYNOPSIS
     ssh-add [-lLdDxX] [-t life] [file ...]
     ssh-add -s reader
     ssh-add -e reader

DESCRIPTION
     ssh-add adds RSA or DSA identities to the authentication agent,
     ssh-agent(1)}.  When run without arguments, it adds the files
     $HOME/.ssh/id_rsa, $HOME/.ssh/id_dsa and $HOME/.ssh/identity.  Alterna-
     tive file names can be given on the command line.  If any file requires a
     passphrase, ssh-add asks for the passphrase from the user.  The
     passphrase is read from the user's tty.  ssh-add retries the last
     passphrase if multiple identity files are given.

     The authentication agent must be running and must be an ancestor of the
     current process for ssh-add to work.

     The options are as follows:

     -l      Lists fingerprints of all identities currently represented by the
             agent.

     -L      Lists public key parameters of all identities currently repre-
             sented by the agent.

     -d      Instead of adding the identity, removes the identity from the
             agent.

     -D      Deletes all identities from the agent.

     -x      Lock the agent with a password.

     -X      Unlock the agent.
```

-t life
Set a maximum lifetime when adding identities to an agent. The lifetime may be specified in seconds or in a time format specified in sshd(8).

-s reader
Add key in smartcard reader.

-e reader
Remove key in smartcard reader.

FILES

$HOME/.ssh/identity
Contains the protocol version 1 RSA authentication identity of the user.

$HOME/.ssh/id_dsa
Contains the protocol version 2 DSA authentication identity of the user.

$HOME/.ssh/id_rsa
Contains the protocol version 2 RSA authentication identity of the user.

Identity files should not be readable by anyone but the user. Note that ssh-add ignores identity files if they are accessible by others.

ENVIRONMENT

DISPLAY and SSH_ASKPASS
If ssh-add needs a passphrase, it will read the passphrase from the current terminal if it was run from a terminal. If ssh-add does not have a terminal associated with it but DISPLAY and SSH_ASKPASS are set, it will execute the program specified by SSH_ASKPASS and open an X11 window to read the passphrase. This is particularly useful when calling ssh-add from a .Xsession or related script. (Note that on some machines it may be necessary to redirect the input from /dev/null to make this work.)

SSH_AUTH_SOCK
Identifies the path of a unix-domain socket used to communicate

with the agent.

DIAGNOSTICS

Exit status is 0 on success, 1 if the specified command fails, and 2 if ssh-add is unable to contact the authentication agent.

AUTHORS

OpenSSH is a derivative of the original and free ssh 1.2.12 release by Tatu Ylonen. Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt and Dug Song removed many bugs, re-added newer features and created OpenSSH. Markus Friedl contributed the support for SSH protocol versions 1.5 and 2.0.

6.4 man 1 ssh-keygen

```
SSH-KEYGEN(1)

NAME
     ssh-keygen - authentication key generation, management and conversion

SYNOPSIS
     ssh-keygen [-q] [-b bits] -t type [-N new_passphrase] [-C comment]
                [-f output_keyfile]
     ssh-keygen -p [-P old_passphrase] [-N new_passphrase] [-f keyfile]
     ssh-keygen -i [-f input_keyfile]
     ssh-keygen -e [-f input_keyfile]
     ssh-keygen -y [-f input_keyfile]
     ssh-keygen -c [-P passphrase] [-C comment] [-f keyfile]
     ssh-keygen -l [-f input_keyfile]
     ssh-keygen -B [-f input_keyfile]
     ssh-keygen -D reader
     ssh-keygen -U reader [-f input_keyfile]

DESCRIPTION
     ssh-keygen generates, manages and converts authentication keys for
     ssh(1).  ssh-keygen can create RSA keys for use by SSH protocol version 1
     and RSA or DSA keys for use by SSH protocol version 2. The type of key to
     be generated is specified with the -t option.

     Normally each user wishing to use SSH with RSA or DSA authentication runs
     this once to create the authentication key in $HOME/.ssh/identity,
     $HOME/.ssh/id_dsa or $HOME/.ssh/id_rsa.  Additionally, the system admin-
     istrator may use this to generate host keys, as seen in /etc/rc.

     Normally this program generates the key and asks for a file in which to
     store the private key.  The public key is stored in a file with the same
     name but ``.pub'' appended.  The program also asks for a passphrase.  The
     passphrase may be empty to indicate no passphrase (host keys must have an
     empty passphrase), or it may be a string of arbitrary length.  A
     passphrase is similar to a password, except it can be a phrase with a
     series of words, punctuation, numbers, whitespace, or any string of char-
     acters you want.  Good passphrases are 10-30 characters long, are not
     simple sentences or otherwise easily guessable (English prose has only
```

1-2 bits of entropy per character, and provides very bad passphrases), and contain a mix of upper and lowercase letters, numbers, and non-alphanumeric characters. The passphrase can be changed later by using the -p option.

There is no way to recover a lost passphrase. If the passphrase is lost or forgotten, a new key must be generated and copied to the corresponding public key to other machines.

For RSA1 keys, there is also a comment field in the key file that is only for convenience to the user to help identify the key. The comment can tell what the key is for, or whatever is useful. The comment is initialized to ‘‘user@host’’ when the key is created, but can be changed using the -c option.

After a key is generated, instructions below detail where the keys should be placed to be activated.

The options are as follows:

-b bits
: Specifies the number of bits in the key to create. Minimum is 512 bits. Generally 1024 bits is considered sufficient, and key sizes above that no longer improve security but make things slower. The default is 1024 bits.

-c
: Requests changing the comment in the private and public key files. This operation is only supported for RSA1 keys. The program will prompt for the file containing the private keys, for the passphrase if the key has one, and for the new comment.

-e
: This option will read a private or public OpenSSH key file and print the key in a ‘SECSH Public Key File Format’ to stdout. This option allows exporting keys for use by several commercial SSH implementations.

-f filename
: Specifies the filename of the key file.

-i
: This option will read an unencrypted private (or public) key file in SSH2-compatible format and print an OpenSSH compatible private

(or public) key to stdout. ssh-keygen also reads the 'SECSH Public Key File Format'. This option allows importing keys from several commercial SSH implementations.

-l Show fingerprint of specified public key file. Private RSA1 keys are also supported. For RSA and DSA keys ssh-keygen tries to find the matching public key file and prints its fingerprint.

-p Requests changing the passphrase of a private key file instead of creating a new private key. The program will prompt for the file containing the private key, for the old passphrase, and twice for the new passphrase.

-q Silence ssh-keygen. Used by /etc/rc when creating a new key.

-y This option will read a private OpenSSH format file and print an OpenSSH public key to stdout.

-t type
Specifies the type of the key to create. The possible values are ''rsa1'' for protocol version 1 and ''rsa'' or ''dsa'' for protocol version 2.

-B Show the bubblebabble digest of specified private or public key file.

-C comment
Provides the new comment.

-D reader
Download the RSA public key stored in the smartcard in reader.

-N new_passphrase
Provides the new passphrase.

-P passphrase
Provides the (old) passphrase.

-U reader
Upload an existing RSA private key into the smartcard in reader.

FILES

$HOME/.ssh/identity
: Contains the protocol version 1 RSA authentication identity of the user. This file should not be readable by anyone but the user. It is possible to specify a passphrase when generating the key; that passphrase will be used to encrypt the private part of this file using 3DES. This file is not automatically accessed by ssh-keygen but it is offered as the default file for the private key. ssh(1) will read this file when a login attempt is made.

$HOME/.ssh/identity.pub
: Contains the protocol version 1 RSA public key for authentication. The contents of this file should be added to $HOME/.ssh/authorized_keys on all machines where the user wishes to log in using RSA authentication. There is no need to keep the contents of this file secret.

$HOME/.ssh/id_dsa
: Contains the protocol version 2 DSA authentication identity of the user. This file should not be readable by anyone but the user. It is possible to specify a passphrase when generating the key; that passphrase will be used to encrypt the private part of this file using 3DES. This file is not automatically accessed by ssh-keygen but it is offered as the default file for the private key. ssh(1) will read this file when a login attempt is made.

$HOME/.ssh/id_dsa.pub
: Contains the protocol version 2 DSA public key for authentication. The contents of this file should be added to $HOME/.ssh/authorized_keys on all machines where the user wishes to log in using public key authentication. There is no need to keep the contents of this file secret.

$HOME/.ssh/id_rsa
: Contains the protocol version 2 RSA authentication identity of the user. This file should not be readable by anyone but the user. It is possible to specify a passphrase when generating the key; that passphrase will be used to encrypt the private part of this file using 3DES. This file is not automatically accessed by ssh-keygen but it is offered as the default file for the private key. ssh(1) will read this file when a login attempt is made.

$HOME/.ssh/id_rsa.pub
: Contains the protocol version 2 RSA public key for authentication. The contents of this file should be added to $HOME/.ssh/authorized_keys on all machines where the user wishes to log in using public key authentication. There is no need to keep the contents of this file secret.

AUTHORS

OpenSSH is a derivative of the original and free ssh 1.2.12 release by Tatu Ylonen. Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt and Dug Song removed many bugs, re-added newer features and created OpenSSH. Markus Friedl contributed the support for SSH protocol versions 1.5 and 2.0.

Table des matières