Cluster

Thierry EXCOFFIER

January 9, 2006

Abstract

Ce document n'est pas un polycopié, ce sont des notes personnelles. Elles ont pour but d'être comprise par moi et ne s'adressent pas à vous. Je ne me donne pas la peine de corriger le français et l'orthographe. Vous les utilisez donc à vos risques et périls

Contents

1 Introduction **2**

2 Fabriquer un *cluster* **2**
- 2.1 *Clusters* pour le calcul parallèle . . . 2
  - 2.1.1 Les machines . . . 3
  - 2.1.2 Le réseau . . . 3
  - 2.1.3 La surveillance . . . 3
  - 2.1.4 L'énergie . . . 4
  - 2.1.5 Le système d'exploitation . . . 4
- 2.2 *Clusters* pour la haute disponibilité . . . 4
  - 2.2.1 Évaluation du temps de panne . . . 4
  - 2.2.2 Un *cluster* n'est pas suffisant . . . 5
  - 2.2.3 Acheter une machine fiable . . . 5
  - 2.2.4 Fabriquer une machine fiable . . . 5
  - 2.2.5 Fabriquer une solution de stockage fiable . . . 5
  - 2.2.6 Accès à *Internet* . . . 6
  - 2.2.7 Énergie . . . 6
  - 2.2.8 *Clusters* répartis . . . 6

3 Logiciels pour les *clusters* **7**
- 3.1 Haute disponibilité des données . . . 7
  - 3.1.1 `DRBD` (*Distributed Remote Block Device*) . . . 7
  - 3.1.2 `GFS` (*Global File system*) de *RedHat* . . . 8
  - 3.1.3 `GPFS` (*General Parallel File system*) de `IBM` . . . 8
  - 3.1.4 `PVFS` (*Parallel Virtual File system*) . . . 8
  - 3.1.5 *Lustre* . . . 9
  - 3.1.6 *Coda* . . . 9
  - 3.1.7 *Peer to peer* . . . 9
- 3.2 Haute disponibilité des services . . . 9
  - 3.2.1 *Heart Beat* . . . 9
  - 3.2.2 Alternatives au basculement d'adresse `IP` . . . 11
- 3.3 Répartition de charge . . . 11
  - 3.3.1 `DNS` . . . 11
  - 3.3.2 `LVS` (*Linux Virtual Server*) . . . 11
  - 3.3.3 `MOSIX` . . . 12
- 3.4 Administration . . . 12
  - 3.4.1 Remplacement de machine . . . 12
  - 3.4.2 Surveillance . . . 12
  - 3.4.3 Lancement de commandes . . . 13
  - 3.4.4 Travaux en arrière plan (*batch system*) . . . 13
  - 3.4.5 Divers . . . 13
- 3.5 Calcul parallèle : `MPI` (*Message Passing Interface*) . . . 14
  - 3.5.1 Communicateur (groupe) . . . 14
  - 3.5.2 Exemple : programme minimal . . . 14
  - 3.5.3 Communication bipoint . . . 14
  - 3.5.4 Les types de donnée manipulables . . . 15
  - 3.5.5 Exemple : Calcul de l'ensemble de Mandelbrot . . . 15

3.5.6 Communications collectives . . . 18
3.5.7 Manipulation des communicateurs (groupes) . . . 19
3.5.8 Exemple : création de communicateur . . . 19
3.5.9 Topologies virtuelles . . . 20
3.6 Meta-systèmes du meta-ordinateur . . . 20
3.6.1 Bibliothèques et Web services . . . 20
3.6.2 *Globus* . . . 20
3.6.3 *Legion* . . . 21
3.6.4 Autres . . . 21

4 Réseaux pour les *clusters* **21**
4.1 Logiciel et protocoles . . . 21
4.1.1 Bidouiller les paramètres `TCPIP` . . . 21
4.1.2 Active messages (1992) . . . 22
4.1.3 `RDMA` (*Remote Direct Memory Access*) . . . 22
4.1.4 Mémoire partagée distribuée . . . 22
4.2 Matériels . . . 22
4.3 Conclusion . . . 22

5 Index **22**

1 Introduction

Un *cluster* est un groupe de machines travaillant ensemble. Ce cours présente les architectures physiques des *clusters* et leurs deux principales utilisations : La haute disponibilité et le parallélisme.

La haute disponibilité est un problème qui peut être résolu par des machines (*hardware*) spécialisés qui sont faites pour pouvoir tourner plusieurs années quelque soit les composants qui tombent en panne. Ces machines sont évidemment hors de prix (TANDEM maintenant fait partie d'HP). Ne pas confondre disponibilité continue et haute disponibilité.

Il est plus intéressant d'utiliser des machines "normales" et de remplacer "au vol" une machine par une autre quand il y a un problème. Pour certaines applications (Web) le remplacement n'est pas nécessaire car la charge de travail est répartie entre les machines.

Le parallélisme est un problème qui peut être résolu par des machines parallèles qui sont elles aussi hors de prix. L'achat de machines "normales" séparées est plus rentable économiquement.

Nous voyons donc que les *clusters* existent car il sont tous simplement moins cher et pour la plupart des problèmes aussi puissant qu'une machine dédiée `http://www.top500.org`. C'est la tendance depuis les années 1990.

Quelques critères :

- Disponibilité (*availability*)
- Évolutivité (*scalability*)
- Puissance (*performance*)
- Administrabilité

2 Fabriquer un *cluster*

Les deux utilisations des *clusters* sont bien différentes et leurs architectures n'ont rien à voir.

Par contre ce qui est commun :

- Des ordinateurs pas cher car grand public
- L'utilisation d'un réseau privé.

2.1 *Clusters* pour le calcul parallèle

Les informations qui sont ici concerne un *cluster* générique pour le calcul parallèle. Il est certain qu'il existe des applications parallèles qui de par leurs particularités nécessite une infrastructure différente :

- taille de disque dur (qui peut être nulle),
- débit et latence du réseau,
- vitesse des processeurs,
- nombre de processeurs par machine,
- capacité mémoire,
- machines avec des capacités identiques, ...

Si le *cluster* est formé de machines existantes sur un réseau existant et utilise les ressources libres de ces machines on appelle cela un `NOW` (*Network Of Workstation*)

2.1.1 Les machines

Ces *clusters* doivent contenir une grande quantité de machines. Les critères d'achat de ces machines :

- Le rapport prix/puissance qui doit être minimum (vieux processeur pas cher).
- Le rapport consommation électrique/puissance qui lui aussi doit être minimum (Processeur Crusoe de Trans-Meta).
- Le rapport volume/puissance qui doit être minimum.
- Pas besoin de carte graphique, de *floppy*, de lecteur de CD et dans certains cas de disque dur.

Les deux premiers peuvent être minimisés ensemble si la date de remplacement du matériel est prévue à l'avance. On peut aussi ajouter le troisième si l'on peut évaluer le prix de la location des locaux.

Ne pas oublier que les machines produisent beaucoup de chaleur et qu'il faudra obligatoirement climatiser si elles sont nombreuses dans un espace restreint.

Si les machines n'ont pas de disque dur, il faut les démarrer avec `DHCP` (*Dynamic Host Configuration Protocol*) et `PXE` (*Pre-boot eXecution Environment*). Ceci n'est possible que si les machines démarre très rarement et que les applications qu'elles supportent n'ont pas besoin d'espace disque et qu'elles fonctionnent longtemps. Comme ces machines ne font pas tourner de processus inutiles les seuls fichiers ouverts en écriture sont les fichiers de log qui pour la plupart peuvent être stockés en sur une machine éloignée sans passer par `NFS` (*Network File System*). Il ne faudra pas oublier de monter les systèmes de fichiers avec l'option `noatime` pour que les dates d'accès aux fichiers ne soient pas mise à jour car cela engendre des écritures sur les disques.

2.1.2 Le réseau

Pour connecter ces machines, il faut un *switch* monstrueux en nombre de ports et en capacité de transfert. Si peu de communication vous est nécessaire, on trouve des *switchs* 48 ports pour 450 Euros (non empilable évidemment et sans ports 1Gb/s). Les *switchs* n'ont pas besoin de beaucoup de fonctionnalités car normalement ils supportent un réseau privés de machines identiques se trouvant dans un espace restreint. On veut simplement de la performance pas des fonctionnalités.

Si le réseau comprend plusieurs *switchs* ils sont organisés en arbre et donc la latence des communications diffère suivant le couple de machine qui communiquent. Le *switch* racine doit être extrêmement performant.

Le plus gros système à l'heure actuelle connecte 640 octo-processeurs avec un *crossbar* de 640 ports.

Sinon, en utilisant du matériel standard on peut faire un réseau comportant des milliers de machines disposées sur une grille. Il y a un *switch* par colonne et par ligne de la grille. Chaque machine est raccordée à son *switch* de ligne et son *switch* de colonne. Avec un tel système des milliers de machines sont raccordé en ne faisant passer les paquets que par 2 *switchs* et une machine. (32*32 machines sur réseaux 1gb/s, bande passante des *switchs*~: 32gb/s, bande passante utile entre deux machines 1/32/32/2 gb/s. S'il y a un *switch* racine il doit supporter sur ses ports du 32*(32*32*1/32/32/2) = 16gb/s et en interne du 16*32 = 512gb/s. En 2004, on arrive à 720gb/s en interne avec 32 ports 10gb/s pour environ 10000 Euros. Ceci est plus rentable que d'acheter 32 *switchs* gigabit car ceux-ci sont encore peu courant.

Si l'on ne se limite pas à du matériel de grande distribution, on peut ne pas utiliser le réseau pour communiquer mais une architecture permettant de partager la mémoire des machines. Ceci permet de communiquer en s'affranchissant des couches réseau. Par exemple `NUMA` (*Non Uniform Memory acces*) ou `SCI` (*Scalable Coherent Interface* `IEEE 1596-1992`)

2.1.3 La surveillance

Quand le système fonctionne bien on peut le surveiller *via* le réseau. Mais au démarrage ?

Brancher un clavier et un écran sur chaque machine est évidemment impossible (prix, place, facilité d'utilisation). Si le nombre de machines ne dépasse pas quelques dizaines il est possible d'avoir un commutateur `KVM` (*Keyboard Video Mouse*) mais cela implique :

- D'en acheter un.
- D'acheter et de gérer beaucoup de câbles.
- D'avoir une carte graphique sur chaque PC.

Dans le cas de *Linux* on peut faire en sorte d'utiliser la liaison série (`RS232C`) pour faire tous les dialogues de la console (même au niveau du `BIOS`). On peut donc utiliser une machine avec beaucoup de connecteur `RS232C` pour gérer toutes les consoles, même si le réseau ne fonctionne pas encore. Pour plus d'info voir le *Remote Serial Console HowTo*.

On peut économiser cette infrastructure car dans un *cluster* on a normalement jamais besoin de dialoguer sur la console car :

- Si la machine à un problème matériel il faut de toute façon la remplacer et réparer.
- On s'arrange pour que la machine démarre dans tous les cas sans intervention manuelle. On peut imaginer qu'elle se réinstalle complètement si elle détecte un grave problème logiciel.
- Car les messages d'erreur affichés sur la console sont stockés sur le disque dur ou bien enregistré sur une machine distante *via* le protocole *syslog* configuré dans `/etc/syslog.conf`. Mais cela peut perturber les communications des applications parallèles.

2.1.4 L'énergie

Une machine allumée utilise environ 150Watts, donc pour 100 machines cela fait 15kW. Il faut donc une alimentation spéciale pour la salle.

Cela coûte 0.05Euro le kWh donc 6570Euro par an pour 100 machines.

Si les machines sont inutilisée il convient d'économiser de l'énergie. Une machine physiquement éteinte consomme 15W. Elle utilise cette énergie pour alimenter certaines fonctionnalités permettant le réveil à distance. Dans notre cas le `WOL` (*Wake On Lan*).

Il existe plein d'autres techniques d'économie d'énergie :

- Ne pas alimenter les parties de l'ordinateur non utilisées grâce à `ACPI` (*Advanced Configuration & Power Interface*). Par exemple l'`USB` (*Universal Serial Bus*), le port parallèle, ...
- Avoir des ventilateurs à vitesse variable.
- Ne pas mettre de cartes inutiles.

2.1.5 Le système d'exploitation

- Migration de processus (`MOSIX`) ?
- Entrée/Sortie parallèle sur le réseau ?

2.2 *Clusters* pour la haute disponibilité

La haute disponibilité implique que le travail est fait sans jamais avoir besoin d'attendre que les problèmes soient résolus. Une disponibilité de 5 neuf indique que le service est utilisable 99.999% du temps (5 minutes d'arrêt cumulé par an)

Avant de faire quoi que ce soit, il faut évaluer le coût pour l'entreprise si ces services s'arrêtent. Si un jour de panne coûte 1 million d'Euros, il est clair que l'entreprise investira ce million en matériel fiable afin de subir 10 minutes de panne par an plutôt qu'un jour. Par contre elle n'investira pas un million de plus pour ne subir qu'une minute de panne par an.

Pour rendre une architecture fiable il faut éliminer tous les problèmes qui peuvent causer l'arrêt du système. Pour cela il faut dédoubler toutes les parties du système.

Mais aussi certaines parties externes : alimentation électrique et air conditionné (obligatoire si les machines sont nombreuses).

Une fois ceci résolu, il reste la partie la plus complexe : les applications doivent évoluer et si les fichiers quelles manipulent changent de format elles doivent pouvoir lire l'ancien format car on ne peut pas traduire tous les fichiers instantanément.

2.2.1 Évaluation du temps de panne

Soit le `MTTF` (*Mean Time To Fêlure*), il est utilisé pour ce qui n'est pas réparable.

Soit le `MTBF` (*Mean Time Between Fêlures*), il est utilisé pour ce qui est réparable.

Soit le `MTTR` (*Mean Time To Repair*), on a $\texttt{MTBF} = \texttt{MTTF} + \texttt{MTTR}$

Disponibilité : $\frac{\texttt{MTTF}}{\texttt{MTBF}}$

La probabilité que la panne se produise dans la prochaine unité de temps est égale à : $\frac{1}{\texttt{MTTF}}$

Si l'unité est l'heure, la probabilité qu'il n'y ai pas de panne dans l'année est $(1 - \frac{1}{\texttt{MTTF}})^{365*24}$. Il y a 8760 heures dans une année.

Pour que les deux fonctionnent à un instant donné, il faut qu'elles ne soient pas tombées en panne durant le temps `MTTR` précédent. Probabilité panne simultanée : $(1 - (1 - \frac{1}{\texttt{MTTF}})^{\texttt{MTTR}})^2$

Si $\texttt{MTTF} = 1000$, $\texttt{MTTR} = 1$ (disponibilité machine seule : $\frac{1000}{1001} = 0.999$) le `MTTF` du système complet est de 1000000 On peut l'augmenter en diminuant le temps de réparation, mais celle-ci peut nécessiter la recopie de très gros volumes de données et donc un temps considérable, le temps de reconstruction de `RAID` se chiffre en de nombreuses heures. La disponibilité de la paire dépend du `MTTR`de la paire hors celui-ci est très grand car il nécessite de repartir des sauvegardes, si l'on suppose qu'il faut 100 heures pour réparer la disponibilité est de $\frac{1000000}{1000100} = 0.9999$.

Tous ceci n'est vrai que si la probabilité d'une panne est constante. Mais cela n'est pas vrai, elle évolue au cours du temps. De plus les calculs sur la paire sont fait en supposant que les pannes étaient des événements décorélés ce qui n'est pas vrai car les matériels on put être achetés en même temps et que l'environnement de fonctionnement est le même (température, vibration, courant électrique, ...)

De plus il ne faut oublier de compter les arrêts programmés à l'avance :

- pour faire des mises à jour,
- nettoyer les filtres,
- remplacer des composants défectueux,
- remplacer la machine.

Pour les petites entreprises il est moins important car il peut être fait durant des périodes ou l'entreprise ne fonctionne pas.

Ordres de grandeur des `MTTF` :

- Disque dur : 300000 à 1200000 heures (34 ans à 136 ans)

- Ventilateur d'excellente qualité : 65000 heures (7 ans)
- Processeur : 30 ans.
- Machine complète : 18000 heures (2 ans)
- *Cluster* de 9000 machines : 2 heures (durée mesurée expérimentalement sur un vrai déploiement)

2.2.2 Un *cluster* n'est pas suffisant

Utiliser l'architecture d'un *cluster* pour le parallélisme pour la haute disponibilité n'est pas possible sans un gros travail par rapport à une machine dédiée.

- La répartition de charge est faite en utilisant plusieurs adresses `IP` sur le même nom de machine : le serveur `DNS` (*Domain Name Service*) ne doit jamais tomber en panne, mais cela n'est pas un problème car `DNS` est un protocole fiable. De plus les clients qui on la malchance d'être sur celui qui tombe en panne sont bloqués un certain temps (cache `DNS`)
- La répartition de charge est faite par une passerelle, celle-ci ne dois jamais tomber en panne. Routage par `NAT` (*Network Address Translation*) ou *IP Tunneling* (réponse directe à celui qui fait la requête pour diminuer le trafic de la passerelle). Voir `LVS` (*Linux Virtual Server*)
- Dans tous les cas le stockage sur disque doit être fait de tel manière que la défaillance du disque ou du serveur n'arrête pas le service.
- L'accès à *Internet* n'est pas fiable.
- Le courant électrique n'est pas fiable.

2.2.3 Acheter une machine fiable

Dans tous les cas on retombe sur le problème de la machine unique qui contient des informations importantes et qui ne doit jamais tomber en panne. La réalité montre qu'à moins de payer des sommes astronomiques les matériels tombent toujours en panne que cela soit des onduleurs, des machines ou des disques.

La solution matériel du riche met en oeuvre des `SAN` (*Storage Area Network*) haute disponibilité utilisant la technique `RAID` (*Redundant Arrays of Inexpensive Disks*). À propos du `RAID`, chez nous, sur 4 disques, en 3 ans, 3 ont totalement perdu leur contenu. Et un autre à modifié le contenu des fichiers à cause d'un bug `SCSI` (*Small Computer System Interface*).

Il est possible de connecter simultanément un disque sur plusieurs machines. En interdisant l'accès simultané ou bien en l'autorisant avec un système de fichier le supportant : `GFS` (*Global File system*)

Oublions donc les solutions matériels onéreuses et voyons comment organiser des machines grand public pour résoudre le problème.

2.2.4 Fabriquer une machine fiable

Il est obligatoire d'avoir un service sur une adresse `IP`, ceci implique donc que lorsqu'une machine plante un autre prend sa place. Au moment du plantage toutes les requêtes en cours seront perdues mais la requête suivante aura une réponse. On appellera maître la machine serveuse et esclave sa remplaçante.

- Les deux machines doivent être configurées identiquement.
- L'esclave doit pouvoir s'apercevoir que le maître a planté.
- La communication entre les 2 ne peut passer par le réseau normal car les deux machines doivent pourvoir dialoguer même si elles ont la même adresse `IP`.
- Le vol d'adresse `IP` n'est pas instantané à cause du cache `ARP` (*Address Resolution Protocol*) des machines qui sont sur le même réseau local. Si le cache ne peut être vidé rapidement (cela dépend du matériel utilisé), il est possible de modifier son adresse `MAC` (*Media Access Control*)
- Quand le maître est réparé, il redémarre en prenant la place de l'esclave.

Tout ceci n'est possible que si l'on a un moyen fiable à 100% pour connecter les deux machines. C'est difficile à mettre en oeuvre si les deux machines sont éloignées l'une de l'autre car *Internet* ne peut être utilisé.

Cette solution est fiable à moins que les deux machines tombent en panne simultanément. Ceci n'est pas rare dans le cas de matériels identiques et se produit souvent avec les disques contenus dans les `RAID`.

2.2.5 Fabriquer une solution de stockage fiable

Il est souvent nécessaire d'avoir un stockage sur disque des données qui supporte les pannes. La méthode la plus simple et la plus économique est que le maître ne valide les écritures disque que quand l'esclave a lui aussi terminé l'écriture sur son propre disque. En cas de panne de l'un ou de l'autre il faut resynchroniser les disques. Tout ceci s'applique de la même façon pour les bases de données.

La synchronisation de données ce fait sur un réseau ne comprenant que ces deux machines avec un connexion directe des deux machines sans passer par du matériel réseau. Il y a moins de latence, moins de risque de panne, plus de bande passante et cela ne pénalise pas le réseau normal.

Un tandem de ce type nécessite deux machines identiques, au moins deux cartes réseaux par machine et une liaison `RS232C` pour être certain que le dialogue pourra toujours s'établir.

Cette solution permet d'avoir une solution avec des supports standard pour le stockage et donc sans utiliser de `RAID`.

Cette solution est évidemment moins performante qu'en utilisant un `SAN` mais elle ne coûte presque rien.

Ce n'est pas par ce que votre solution de stockage est fiable que vous n'avez pas besoin de faire des sauvegardes. Le but des sauvegardes est de :

- Pouvoir restaurer les données en cas de panne.
- Pouvoir restaurer les données en cas de catastrophe, à condition que les sauvegardes soient faîtes à distance.
- Pouvoir restaurer des données anciennes.

2.2.6 Accès à *Internet*

On ne peut pas supposer que l'accès à *Internet* soit fiable :

- Un simple coup de pelleteuse et vous pouvez vous retrouver isolé pour de nombreuse heures.
- Une simple erreur dans les tables de routage et plus personne ne peut vous atteindre.

Avec une fois 10 heures de panne dans l'année vous tombez à 99.9% de disponibilité.

Une seule chose est réellement fiable sur *Internet* : c'est le service `DNS`. Par ce que les algorithmes sont fiables et que les machines qui le supporte aussi.

Avoir deux fournisseurs d'accès à *Internet* n'est pas une solution parfaite :

- Il faut évidemment que les deux fournisseurs d'accès à *Internet* utilisent des technologies différentes, afin qu'un pelleteuse ne coupe pas les 2 câbles qui passent d'un seul coup (`ADSL` et câble).
- Il ne peuvent pas router chacun la même adresse `IP` chez vous.
- Il faut associer deux adresses `IP` au nom logique pour faire en plus de l'équilibrage de charge.
- Il faut mettre à jour le `DNS` en cas de panne pour supprimer l'une des adresses `IP`. **La mise à jour des caches est lente**.

Si vous utilisez un routeur n'oubliez pas qu'il peut tomber en panne (et aussi être attaqué par des virus). Il est plus fiable d'en utiliser 2.

```
I1 --- R1 --- Maitre
       |  \   /   |  .
       |   \ /    |  .
       |    X     |  .
       |   / \    |  .
       |  /   \   |  .
I2 --- R2 --- Esclave
```

Une autre solution plus rapide car sans basculement `IP`, mais moins fiable car elle n'évite pas les bugs dans les tables de routage est de trouver un fournisseur d'accès qui permette la communication simultanée *via* plusieurs types de liaison (`ISDN`, `ADSL`, câble, satellite, radio, ...). Il installe alors des routeurs fiable permettant d'utiliser tous ces flux. Ces fournisseurs d'accès garantissent 99.995% de disponibilité (moins de 26 minutes de panne par an). C'est évidemment la solution du riche.

2.2.7 Énergie

Mettre chacune des deux machines sur son propre onduleur. Des onduleurs grand public sont suffisants. Pas besoin d'onduleur ou l'on peut changer des alimentations sans interrompre le service.

Si vous voulez que cela continue à fonctionner sans courant longtemps. Cela coûte beaucoup plus cher, il est nécessaire :

- d'avoir un générateur de courant électrique
- ou être abonné à deux fournisseurs d'électricités utilisant des réseaux électriques différents (pas en France).

2.2.8 *Clusters* répartis

Cette solution doit être employée si la société est importante et est répartie sur plusieurs sites. Les machines sont alors réparties sur plusieurs sites. Lorsqu'un site a un problème les fichiers `DNS` sont mis à jours pour pointer vers les bons serveurs. Ceci résout les problèmes :

- Incendie, catastrophe, ...
- Coupures d'électricité.
- Mauvais routage au niveau d'*Internet*.

Il est préférable d'avoir en plus des liaisons spécialisées pour avoir une bonne bande passante entre les serveurs pour la duplication des données. Et aussi pour tester s'ils sont toujours vivant.

3 Logiciels pour les *clusters*

Il existe de nombreuses solution clef en main. Ces solutions sont payantes et de plus elles cachent les différents composants qui fonctionnent ensemble.

3.1 Haute disponibilité des données

3.1.1 `DRBD` (*Distributed Remote Block Device*)

C'est la solution la plus simple, elle permet de faire du `RAID1` (*mirroring*) entre deux machines. Si l'un des machines ou disque tombe en panne, l'autre contient toujours les données. Cette solution intégrée simple à visiblement détrôné `ENDB` (*Enhanced Network Block Device*) qui était plus généraliste et supportait de plus le `RAID5`.

Matériel nécessaire : deux machines avec chacune un disque de la même taille, pas obligatoirement du `RAID`, une connexion directe entre les deux machines afin de minimiser les problèmes de pannes, diminuer la latence et augmenter le débit.

L'un des deux disques est le primaire, l'autre est le secondaire. On ne peut écrire que sur le disque primaire. Avec les systèmes de fichier classiques, seul le primaire peut être utilisé. Le secondaire ne peut même pas être utilisé en lecture seulement (ou alors avec `GFS`).

Que ce passe-t-il quand le secondaire s'arrête ?

- Le primaire mémorise ce qu'il ne peut pas envoyer au secondaire. Il utilise un tableau de bits pour noter les secteurs à transférer.
- Quand le secondaire redémarre :
  - Si son disque a été changé alors une réplication totale est lancée.
  - Sinon, seul les données modifiées sont recopiée
- Si jamais le primaire plante à ce moment et que le secondaire démarre il trouvera un situation incohérente un administrateur devra dire ce qu'il faut faire : le passer en primaire ou attendre le redémarrage du primaire. Normalement ceci ne ce passe jamais.

Que ce passe-t-il quand le primaire plante ?

- On passe le secondaire en primaire tout simplement.
- Il mémorise tout ce qu'il ne peut envoyer à son nouveau secondaire.
- Quand l'ancien primaire redémarre, il devient secondaire et il se synchronise.
- Si le nouveau primaire plante avant le démarrage de l'ancien, il faut une intervention manuelle.

Arrêt planifié des deux machines : d'abord le secondaire puis le primaire. On redémarre le primaire puis le secondaire. Dans l'autre sens cela doit marcher aussi.

Arrêt non planifié des deux machines : tout se remet en route seulement au moment ou les deux machines sont d'accord sur leur statut. Cela reste bloqué quand une seul des deux démarre.

Le grand danger est de se retrouver avec deux primaires, ceci ne doit se produire en aucun cas. Il faut être **absolument** certain que l'autre machine n'est plus primaire.

Les écritures disques sont faites en utilisant un protocole parmi les 3 suivants :

- A : On considère que l'écriture distante est faite dès que la demande est partie sur le réseau.
- B : On considère que l'écriture distante est faite quand la machine distante confirme la réception du message.
- C : On considère que l'écriture distante est faite quand elle est stockée sur le disque distant.

Tant que l'une des deux machines est accessible, le contenu du disque est accessible et modifiable. Ceci même pendant une resynchronisation. Mais bien sur quand une machine plante, le système de fichier est mal démonté et il faut faire un `fsck` lorsque avant de monter le disque sur l'autre machine. Pour éviter le `fsck` qui peut durer des heures il faut utiliser un système de fichier journalisé comme *Ext3*, *Reiserfs*, ... Avec ces systèmes, quelque soit le moment ou la machine s'arrête les données sur le disque restent cohérentes.

Il est possible d'utiliser `DRBD` entre deux machines distantes sur *Internet*. On prend alors le protocole B et de gros zones temporaires afin que la latence ne soit pas trop importantes et le débit pas trop élevé. La fiabilité sera évidemment moins grande.

Si l'on désire pouvoir faire grossir la taille de ce disque fiable il suffit de le construire au dessus d'un disque `LVM`(*Logical Volume Manager*). Qui lui même peut être construit au dessus de disques `RAID` logiciel ou matériel afin de fiabiliser encore plus les données. N'oubliez pas que pour recopier 1TB avec une liaison 100Mb il faut 27 heures et que durant ce temps il est loin d'être improbable que l'autre disque plante aussi. D'autant plus qu'il est en utilisation constante puisqu'on le recopie complètement.

Autre point d'importance, il est obligatoire de faire des sauvegardes des données. Malheureusement cela n'a pas de sens de sauvegarder les données pendant qu'elles sont utilisées. On ne peut pas arrêter les services pendant la sauvegarde sinon ce n'est pas de la haute disponibilité. La solution est le *snapshot* (instantané) qui est disponible dans `LVM` et les autre outils de ce type, avec ceci ont arrête les applications le temps de faire un instantané et on les relance. On fait alors la sauvegarde de l'instantané.

Attention, les systèmes suivants fournissent un système de fichier réparti et sont donc beaucoup plus complexe.

Comment installer :

- Récupérez les sources de `DRBD`, ils tiennent sur un disquette.
- Dans le répertoire des sources : `make install install-tools`
- `depmod -a`
- Fichier de configuration identique sur les deux machines dans `/etc/drbd.conf` :

```
resource mon_disc {
protocol C;
incon-degr-cmd "echo '!DRBD! pri on incon-degr'|wall ; sleep 60 ; halt -f";
startup { wfc-timeout  0; degr-wfc-timeout 120; }
disk { on-io-error   detach; }
syncer { rate 10M; }
on machine_toto {
device     /dev/drbd0;
disk       /dev/hda6 ;
address    192.168.0.1:7788;
meta-disk  internal;
}
on machine_titi {
device    /dev/drbd0;
disk      /dev/hda6;
address   192.168.0.2:7788;
meta-disk internal;
}
}
```

- `mknod /dev/drbd0 b 147 0`
- Lancer le module : `/etc/init.d/drbd start`
- Regarder l'état : `cat /proc/drbd`
- Sur l'un dire qu'il est primaire et secondaire l'autre. L'option `--do-what-I-say` n'est utile que la première fois. `drbdadm -- --do-what-I-say primary mon_disc` et `drbdadm -- secondary mon_disc`
- Faire en sorte que cela démarre et s'arrête tout seul :
  `ln -s ../init.d/drbd /etc/rc2.d/S19drbd`
  `ln -s ../init.d/drbd /etc/rc0.d/K19drbd`
- Sur le primaire : `mkfs.ext3 /dev/drbd0 ; mount /dev/drbd0 /mnt`

3.1.2 GFS (*Global File system*) de *RedHat*

C'est la version la plus simple des systèmes de fichier répartis. Ce **système de fichier** autorise des montages simultané du même disque en lecture et en écriture.

Une mise en place simple est d'avoir plusieurs serveur de fichier branchés physiquement sur le même disque SAN *via* SCSI3, iSCSI, *Fibre Channel*.

Sinon on peut avoir des "clients" accédant à tous les disques en utilisant ENDB ou un autre outil du même type. Ajouter un disque nécessite de prévenir tous les "clients".

Ce système de fichiers est évidemment journalisé afin de toujours rester dans un état cohérent. On peut changer la taille disponible sans l'arrêter.

Les machines partageant les disques dialoguent entre elles afin de gérer les verrous.

Ce système de fichier est très lent. Il permet de résoudre le problème des pannes de machines, mais il faut des disques fiable. Toutes les machines et tous les disques doivent avoir deux ports de communication et il faut au moins deux *switchs* pour assurer la redondance des chemins réseaux.

3.1.3 GPFS (*General Parallel File system*) de IBM

Comme GFS mais en plus :

- Distribution des données sur le réseau.
- Réplication et *striping*.
- *Snapshot*.
- Payant.

3.1.4 PVFS (*Parallel Virtual File system*)

Logiciel libre.

Comme GPFS mais encore très expérimental.

Protocol sans état donc plus simple.

3.1.5 *Lustre*

Comme `GFS` mais les données et les meta données ne sont pas gérées par les mêmes serveurs. Les méta données doivent être distribuées par une paire de machine accédant au même disque totalement fiable.

Les données sont répartie sur les autres machines. Les autres machines peuvent tomber en panne sans problème à condition qu'il n'y en ai pas deux en même temps car *Lustre* fait du `RAID1`.

Les clients se connectent au serveur de meta données puis ensuite dialoguent avec les serveurs de données.

Cette solution est bonne mais elle nécessite un serveur de meta donnée totalement fiable.

La dernière version est payante, celle de l'année précédente est diffusée avec une licence `GPL` (*General Public Licence*).

Petite chose amusante, *Lustre*permet de stocker dans le système de fichier lui-même les montages qui doivent être fait sur les répertoires et fait ceux-ci au vol comme avec `HURD`. Cette fonctionnalité permet de se débarrasser des *automounter* comme `amd` et `autofs`.

3.1.6 *Coda*

C'est un système de fichier réparti, "ancêtre" de *Lustre* car ils ont le même papa. Mais avec les particularités suivantes :

- Les clients sont aussi des serveurs et leur espace disque est utilisé.
- Autorise les coupures réseaux et s'arrange pour tout synchroniser après.
- Le niveau de réplication que l'on veut.
- Déplace les fichiers vers l'endroit ou ils sont utilisés.

Ce système est puissant mais complexe.

Le gros avantage de *Coda* est que l'on peux utiliser de manière transparente la place disque situées sur toutes les machines.

3.1.7 *Peer to peer*

C'est le *cluster* formé par toutes les machines qui veulent bien y participer sans qu'il y ai d'administration centrale.

Pas d'application de niveau "production" orienté système de fichier pour le moment.

- OceanStore
- Ivy
- CFS Cooperative File System

Idées fondatrices de tous les nouveaux systèmes répartis :

- les `DHT` qui permettent de retrouver ou ce trouve l'information à l'aide d'une clef de type `MS5` ou `SHA` calculée à la fois sur les machines et les données, les machines stockant les données qui leur sont proche en terme de distance entre clefs.
- Le concept de clef publique et privée afin de protéger les données en lecture ou bien s'authentifier en tant qu'auteur.

3.2 Haute disponibilité des services

S'il est possible de faire fonctionner les services sur plusieurs machines simultanément, alors le problème est un problème de répartition de charge. Mais quand cela n'est pas possible, par exemple pour le stockage de donnée ou le routage, "il faut" que le service ne soit fait qu'une seule fois.

Pour qu'un service soit tout le temps disponible, il faut qu'une machine réponde toujours à l'adresse `IP` et fournisse le service. Il faut donc échanger deux machines instantanément.

3.2.1 *Heart Beat*

La fonctionnalité de ce système est simple : faire en sorte que le passage des services d'une machine sur une autre (le *takeover*) se passe bien en lançant les bonnes commandes au bon moment sur les deux machines quoi qu'il se passe au niveau logiciel ou matériel.

Heart Beat est une solution logicielle. La partie matérielle est en fait composée de matériels existants.

Pour faire basculer les services *Heart Beat* change les adresses `IP` publiques. Des adresses `IP` privées doivent exister afin que les machines puissent continuer à être sur le réseau même si elles n'ont plus l'adresse `IP` publique.

Solution 1 : Les deux machines sont seulement sur le réseau normal. Si elles sont séparées *Heart Beat* en déduis que l'autre machine est arrêtée, il lance donc les services sur l'ancien esclave. Si malheureusement le maître continuait à fonctionner, cela pourrait soulever de grave problèmes : deux machines écrivant simultanément sur le même disque ou répondant sur la même adresse `IP`. Pour éviter ce problème on peut utiliser *Stonith* pour éteindre l'ancien maître ou bien assurer plus de redondance sur la communication entre les deux machines.

Solution 2 : On ajoute une connexion série entre les deux machines, il suffit d'un câble *NULL Modem*. Ceci représente un faible investissement (5 Euro) car toutes les machines ont une liaison série `DB9`. On peut aussi utiliser une liaison réseau séparée dédiée pour les battements de cœur. La liaison série est préférée car elle n'est pas encombrée par toutes les couches `TCPIP`.

Heart Beat vérifie aussi si la machine est correctement connectée au réseau. Si pour une raison quelconque le maître n'est plus connecté au réseau alors l'esclave doit prendre la relève même si le maître fonctionne correctement. A moins bien sur que l'esclave ai le même problème. Mais ce cas est simple car généralement les deux peuvent encore communiquer *via* un autre moyen. Le basculement est rapide car il n'y a rien à attendre.

Heart Beat peut être utilisé par deux machines gérant un disque partagé ou bien utilisant `DRBD`. Dans le deuxième cas, elles ont déjà deux cartes réseau pour des raisons de performance.

Le déplacement des services est plus ou moins compliqué. Un serveur *Apache* ne pose aucun problème car les requêtes sont indépendantes (`HTTP` est un protocole sans état). Au pire des personnes devront recharger une page. De même pour `SMBFS`, `LDAP`. Par contre `NFS` nécessite un peu plus de travail car il stocke des informations sur les verrous de fichiers dans `/var/lib/nfs`, il faut donc stocker ces données dans le disque partagé. Il faut aussi lancer `rpc.statd` en spécifiant le nom de la machine publique sinon cela ne fonctionnera pas (option : `-n hostname`).

C'est *Heart Beat* qui configure les interfaces réseaux publiques et qui lance et arrête les services. **Jamais de lancement de services publique dans `/etc/rc...`.**

- Le maître plante.
- *Heart Beat* démarre les services sur l'esclave.
- Le disque `DRBD` local passe en primaire.
- Le disque local est monté.
- Le serveur `NFS` et ses adjoints sont lancés.
- L'adresse `IP` est volée : l'esclave devient maître.
- Un *broadcast* `IP` est envoyé pour faire mettre à jour les caches `ARP`.
- L'ancien esclave reçoit alors les requêtes `NFS`, les autres machines croient à un redémarrage instantané du maître.
- L'ancien maître redémarre et passe immédiatement esclave.
- `DRBD` se resynchronise.
- Si on le désire, on peut inverser automatiquement le maître et l'esclave si l'on a une machine maîtresse préférée.

Le problème qui fait vraiment peur à tous les administrateurs : le cas ou pour une raison ou une autre l'esclave croit le maître mort et le maître ne se rend pas compte qu'il est mort (bug de l'ordonnanceur, machine trop chargé, déni de service, ...). Dans ce cas on se retrouve avec deux machines avec la même adresse `IP` et modifiant le même disque en même temps. La solution est *Stonith*, elle permet à l'esclave de tuer le maître avant de passer lui aussi maître.

Heart Beat est en fait plus générale, il permet de vérifier le fonctionnement de plus de deux machines.

Configuration de *Heart Beat*.

- Installer le package `[heartbeat]`
- La doc est dans `/usr/share/doc/heartbeat`
- Les machines ont 2 interfaces réseau et un port série (`RS232C`)
- Dans `/etc/ha.d/ha.cf` mettre :

```
keepalive  1 # Battement de coeur toute les secondes
deadtime   5 # Mort au bout de 5 secondes sans battement de coeur
warntime        3 # Warning au bout de 3 secondes
initdead       10 # Temps maximum mis pour initialiser les services
realtime       on # Les processus des heartbeat sont prioritaires
auto_failback off # L'esclave reste maître après retour de l'ancien maître
ping IP1 IP2 ...  # On doit pouvoir joindre ces IP sinon on a un problème
node hostname1    # Nom court d'une machine du couple
node hostname2    # Nom court de l'autre machine
# Battement de coeur sur laison série
serial /dev/ttyS0 # Liaison série permettant de communiquer avec l'autre
baud        19200 # Vitesse de la liaison série
# Battement de coeur sur interface réseau
ucast eth1 IP     # On donne l'adresse IP de l'autre
```

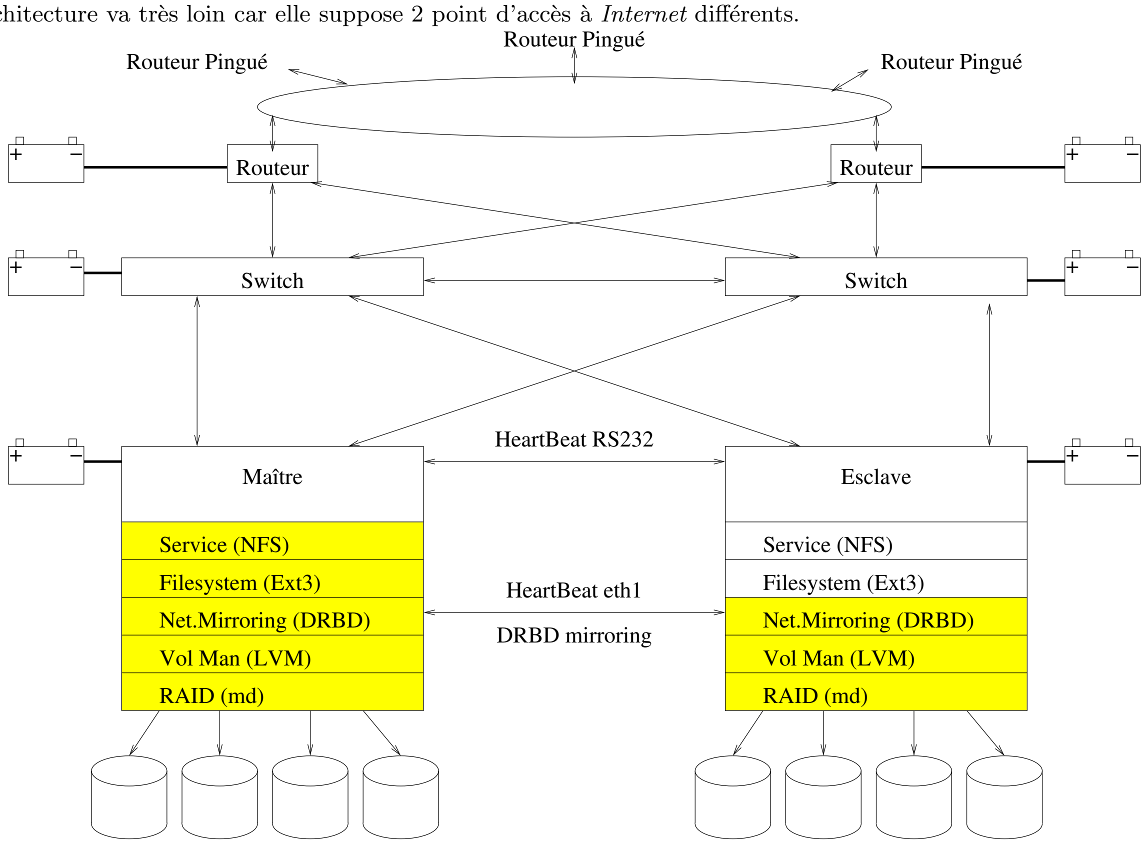
- L'interface `eth0` est configuré normalement.
- L'interface `eth1` est configuré pour un réseau local contenant seulement les deux machines.
- L'interface `eth0:0` (ou `eth2` est utilisé seulement par la machine maîtresse, c'est lui qui sert l'adresse `IP` qui bascule d'une machine à l'autre.
- Exemple de fichier `/etc/ha.d/haresources` dans lequel on défini les basculement `IP`, `DRBD` et le montage du système de fichier. Ce fichier est identique sur toutes les machines.

```
host1 192.168.0.2/24/eth0:0 drbddisk::my_disc Filesystem::/dev/drbd0::/mnt::ext3 nfs-common nfs-kernel-serve
host2 192.168.0.2/24/eth0:0 drbddisk::my_disc Filesystem::/dev/drbd0::/mnt::ext3 nfs-common nfs-kernel-serve
```

- Exemple de fichier `/etc/ha.d/authkeys` permettant une vérification minimale de l'identité de l'autre.

```
auth 2
2 sha1 unmotdepasepase
```

Exemple d'architecture très fiable que l'on peux réaliser avec uniquement des composants non fiable. Cette architecture va très loin car elle suppose 2 point d'accès à *Internet* différents.



3.2.2 Alternatives au basculement d'adresse `IP`

Le basculement d'adresse `IP` est simple à réaliser mais il y a quelques petits problèmes :

- Les machines qui sont sur le même réseau peuvent mettre du temps à s'en rendre compte car elles ont mémorisé l'adresse `MAC` de la machine pour éviter de la redemander à chaque fois. Mais ce n'est pas un problème pour les machines extérieurs car les *switchs* se mettent à jour généralement instantanément (il ne connaissent pas les `IP`)
- Le basculement d'`IP` interrompt les liaisons en cours. C'est normale en cas de panne mais pénible en cas de maintenance.

Le premier problème peut être résolu en modifiant l'adresse `MAC` de la carte réseau. Le problème est que cette opération n'est pas supportée par toutes les cartes.

Le deuxième problème est facilement soluble en changeant l'adresse `IP` du serveur dans les base de donnée `DNS`. Toute nouvelle connexion utilisera la nouvelle machine au lieu de l'ancienne. Cette solution est complexe à mettre en oeuvre s'il y a des écritures (`DRBD` par exemple). Une solution simple est de déporter le stockage sur un autre couple de serveur en utilisant des services d'accès au stockage utilisant des protocoles sans connexion.

3.3 Répartition de charge

La répartition de charge permet de répartir le travail entre plusieurs machines. Bien sur les accès en écriture doivent être centralisé (à moins d'utiliser `GFS` ou un équivalent).

3.3.1 `DNS`

C'est la solution la plus simple. Le serveur `DNS` donne une adresse `IP` "au hasard" dans un *cluster* quand on lui donne un nom de machine. L'inconvénient est que si l'une des machines du *cluster* s'arrête son adresse `IP` reste quand même dans les nombreux caches.

3.3.2 `LVS` (*Linux Virtual Server*)

Cette solution est simple :

- Toutes les requêtes vont vers la même machine qui doit être configurée en haute disponibilité.

- Elle fait suivre la requête à la machine du *cluster* qui est la moins chargée.
- Si le routage est fait en utilisant le *IP Tunneling* ou le *Direct Routing*. Dans les deux cas, le serveur répond directement au client sans repasser par la passerelle afin de ne pas la charger. L'*IP Tunneling* nécessite que le serveur destination le supporte et le *Direct Routing* nécessite que les machines soient sur le même réseau. C'est la solution la plus efficace.

Dans le cas du *Direct Routing* les machines doivent être sur le même réseau car `LVS` ne doit pas changer l'adresse `IP` en réécrivant le paquet. S'il le faisait alors le vrai serveur renverrais des réponses au client avec la mauvaise adresse `IP` et donc le client ne comprendrai pas que c'est la même connexion.

3.3.3 `MOSIX`

C'est un système parmi d'autre. Il est bâti au dessus de *Linux*. Il modifie le noyau du système pour que les applications n'ai pas à être modifiées.

Ce système permet la migration des tâches après leur lancement. Tous ces systèmes sont peu performant dans le cas ou il y a beaucoup d'entrées/sorties car celles-ci sont systématiquement redirigées *via* la machine qui a initialement lancé la tâche. En fait, cette redirection est faite au niveau des appels systèmes.

Ceci est due au fait que ces systèmes sont bâtis par dessus des systèmes d'exploitation mono-ordinateur qui ne prévoient pas la migration de tâche. Il faut aussi noter que `TCPIP` ne permet pas la migration des tâches car la destination est une adresse `IP` fixe. On pourrais détourner ceci en faisant du *multicast*.

3.4 Administration

3.4.1 Remplacement de machine

Dans tous les cas, si la machine n'est pas encore arrêté, on est obligé d'enlever la machine du *cluster*.

Si c'est une maintenance normale on a eu tout le temps de configurer la machine. Sinon, il faut rapidement la remplacer par une machine de réserve.

On suppose avoir le matériel suffisant pour construire la nouvelle machine. Sinon, on est un peu négligeant.

Configurer rapidement une machine.

- Le système de base est déjà installé avec tout les bons gestionnaires de périphériques.
- On suppose la configuration matériel quasiment identique, notamment le nommage des disques et leur partitionnement.
- Modifier l'adresse `MAC` dans le serveur `DHCP`("les" si on est prudent).
- Redémarrer la machine.
- La machine est prête (`DRBD` doit se resynchroniser automatiquement).

Outils pour que cela fonctionne.

- `DHCP` pour récupérer les informations lors du démarrage.
  - Nom de la machine.
  - Adresse `IP`.
  - Passerelle.
  - Serveurs `DNS`.
  - Nom de la racine du système de fichier.
- `cfengine` est un outil permettant de configurer et de vérifier le fonctionnement des machines. Cette solution est puissante mais nécessite de la programmation. Si on ne veut pas utiliser cet outil, il est tout de même nécessaire d'écrire un *script* permettant de configurer automatiquement une machine nue.
- *ClusterNFS* est une version d'`NFS` permettant de diffuser des fichiers différents suivant le nom de la machine (ou même de l'utilisateur). Ceci permet d'utiliser `NFS` pour charger la racine du système de fichier. Ce n'est pas du tout une bonne idée d'exporter à soi-même son propre système de fichier racine (verrou mortel). Mais théoriquement c'est réalisable.

3.4.2 Surveillance

Chaque système de gestion de *cluster* arrive avec ses outils pour vérifier son état.

- `DRBD` : `cat /proc/drbd`
- *Heart Beat* : `/etc/init.d/heartbeat status`
- `MOSIX` : `mon`

Sinon on peut utiliser l'outil classique de surveillance : `SNMP` (*Simple Network Management Protocol*)
Il permet de :

- demander des valeurs aux machines, routeurs, imprimantes, ...
- De modifier ces valeurs.
- De recevoir des événements (disque plein, trop d'erreur réseau, ...)

3.4.3 Lancement de commandes

Il est facile de lancer soit même une même commande sur plusieurs machines. Par contre quand leur nombre dépasse la dizaine cela commence à devenir compliqué car il faut paralléliser le lancement et résumer proprement les erreurs et le résultat des commandes. Il y a beaucoup d'outils permettant de faire ceci, en voici quelques un.

- `DSH` (*Dancer's Shell*) : Le problème est résolu en faisant un lancement récursif sur l'ensemble des clients (fenêtre de 4). Il utilise différentes commandes pour lancer l'exécution à distance : **`rsh`**, **`ssh`**. Il peut utiliser **`netgroup`** pour récupérer les listes de machines.

  `taktuk` est une version plus puissante dans laquelle on peut coder la topologie du réseau.
- `DVT` (*Distributed Virtual Terminals*), *Cluster SSH*, **`fanout`**, **`pconsole`** : Ils permettent de lancer un terminal permettant de taper des commandes sur plusieurs machines. L'un des terminal est particulier et tout ce qui est tapé dedans est envoyé à toutes les autres terminals. On peut éditer un fichier avec **`vi`** sur plein de machines à la fois !

clusterit : un ensemble d'outils.

3.4.4 Travaux en arrière plan (*batch system*)

Dans certains cas on ne veux pas faire (ou on ne peut pas faire) de parallélisme mais simplement lancer des processus qui consomment beaucoup de calculs.

- On veut pouvoir lancer les calculs sans se soucier de la machine qui les exécutera.
- On veut que chaque machine soit utilisée au maximum.
- On veut affecter des priorités aux calculs.
- Cela n'est pas une bonne idée de lancer tous les processus en même temps.
- Dans le cas général il est impossible pour le système de suspendre un processus en le stockant sur le disque pour pouvoir le reprendre après un redémarrage. Par contre c'est possible si le processus n'a pas de fichiers ouverts. Pour simplifier la tâche du système d'exploitation c'est au processus d'assurer la sauvegarde périodique de leur état interne afin de pouvoir redémarrer.
- Si le système peut suspendre un processus, il est alors facile de faire migrer le processus d'une machine à l'autre (pour la maintenance par exemple).

Il existe beaucoup de systèmes, les systèmes libre les plus connus permettant de lancer des calculs en arrière plan sur un *cluster* sont :

- `GNQS` (*Generalized Network Queuing System*): Un système de base simple, un peu mort.
- `PBS` (*Portable Batch System*) : Système payant mais avec une version bridée gratuite. Marche aussi sous *Fenêtres*. La version payante est bien vivante.
- *Condor* : Système bien vivant.
  - Optimisé pour utiliser le temps libre des ordinateurs de bureau.
  - Il supporte la migration de tâches grâce à l'utilisation de point de préemptions ou l'état de la tâche est sauvé.
  - Il déporte les appels système : les données traités n'ont pas besoin d'être accessibles sur les machines distantes.
  - Les programmes `MPI` ne sont pas suspendus et ne migrent pas.

3.4.5 Divers

`NTP` (*Network Time Protocol*) est un protocole permettant de synchroniser les horloges d'un groupe de machine. Si les horloges sont désynchronisées, de nombreux problèmes peuvent apparaître (`NFS` notamment).

Un *watchdog* est un chien de garde. L'idée est simple : un processus important doit toujours fonctionner, ce processus doit toujours dire au chien de garde "je suis vivant". Si pendant un certain temps le chien de garde n'a pas le message alors il redémarre la machine. Les chiens de garde peuvent être logiciel ou matériel. La version matérielle est plus fiable car la version logicielle peut planter à cause de ce qu'elle doit surveiller.

Stonith (*Shoot The Other Node In The Head*) est une `API` permettant d'éteindre une autre machine. Sa configuration définie comment éteindre l'autre. Cela peut être par une ré initialisation, l'arrêt à distance de son `UPS` (*Uninterruptible Power Supply*) ou un périphérique spécialisé. Si la demande passe par un serveur, celui-ci peut assurer que les deux machines ne s'éteignent pas mutuellement, mais est-ce un mal puisqu'elles ont toutes les deux un problème de communication (intéressant de tester avec un *ping* externe).

Pour paralléliser les compilations et les calculs sur un multi processeurs (ou avec `MOSIX`) il suffit de taper `make -j nb_processus`.

3.5 Calcul parallèle : MPI (*Message Passing Interface*)

MPI permet de faire communiquer simplement un ensemble de taches se trouvant sur des machines différentes afin de faire du parallélisme. Il gère seulement les communications, c'est les applications qui doivent se débrouiller pour répartir le travail.

Ces informations sont extraites de : http://www.llnl.gov/computing/tutorials/workshops/workshop/mpi/MAIN.html

MPI est une API pas un produit. On utilisera MPICH en TP. La première spécification date de 1994. Le but est de programmer sans ce soucier des couches basses pour la communication afin que les programmes soient portables.

3.5.1 Communicateur (groupe)

Les processus sont regroupés en communicateur, chaque processus du communicateur à un numéro (0, 1, 2, ...). MPI_COMM_WORLD est le groupe (communicateur) contenant tous les processus. MPI possède beaucoup de fonction permettant de gérer ces communicateurs.

3.5.2 Exemple : programme minimal

Le programme suivant affiche le nombre de processus qui tournent et le rang du processus qui s'exécute. Dans la version 1 de MPI on ne peut pas lancer d'autres processus, le programme va donc tourner seul sur la machine courante.

On peut aussi demander à MPI de lancer l'exécution sur plusieurs machines en même temps en listant les noms de machine dans un fichier. Ici, dix processus seront lancés. Les sorties standard et d'erreur des 10 processus sont redirigées vers celles de mpirun.

```
mpirun -np 10 -machinefile nom_des_machines nom_de_l'éxecutable_a_lancer
```

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc,char **argv)
{
    int  numtasks, rank, rc;

    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS)
        {
            printf ("Error starting MPI program. Terminating.\n");
            MPI_Abort(MPI_COMM_WORLD, rc);
        }

    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf ("Number of tasks= %d My rank= %d\n", numtasks, rank);

    /*******  do some work *******/

    MPI_Finalize();
}
```

3.5.3 Communication bipoint

MPI permet de faire communiquer par plein de méthodes différentes les processus. Les méthodes les plus simples sont les communications bipoints. Le tag est en entier permettant de filtrer les messages, un processus peut par exemple décider qu'une valeur du tag est urgente et que les message arrivant avec celle-ci doivent être traité prioritairement.

```
Blocking sends           MPI_Send (buffer,count,type,dest  ,tag,comm)
Blocking sends (Synchro) MPI_Ssend(buffer,count,type,dest  ,tag,comm,ierr)
Non-blocking sends       MPI_Isend(buffer,count,type,dest  ,tag,comm,request)
Blocking receive         MPI_Recv (buffer,count,type,source,tag,comm,status)
Non-blocking receive     MPI_Irecv(buffer,count,type,source,tag,comm,request)
```

Il ne faut pas modifier les données envoyées en mode non bloquant, elles ne sont pas forcément recopiées.

Pour savoir ou une communication non bloquante en est on utilise un test ou une attente.

```
MPI_Test    (request                        , flag, status)
MPI_Wait    (request                        ,       status)

MPI_Testany (count, array_of_requests, index, flag, status)
MPI_Waitany (count, array_of_requests, index      , status)
```

```
MPI_Testall (count, array_of_requests       , flag, array_of_statuses)
MPI_Waitall (count, array_of_requests             , array_of_statuses)

MPI_Testsome(incount, array_of_requests, outcount, array_of_offsets, array_of_statuses)
MPI_Waitsome(incount, array_of_requests, outcount, array_of_offsets, array_of_statuses)
```

3.5.4 Les types de donnée manipulables

Les types manipulables sont :

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

On peut créer les siens facilement dynamiquement. Par contre, la notion de pointeur n'est pas suporté comme dans `XDR` (*External Data Representation*est utilisé dans `NFS`). On ne peut donc pas transférer des arbres ou graphes directement, seulement un bloc de mémoire.

3.5.5 Exemple : Calcul de l'ensemble de Mandelbrot

Calcul de l'ensemble de Mandelbrot.

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <sys/resource.h>
#include <sys/time.h>

#define LAR 79
#define HAU 50
/* Défini le temps pris par les clients pour faire le calcul */
#define MAX_ITERATION 10000

void demande(int image[HAU][LAR], MPI_Request *requests, int ligne, int task)
{
  MPI_Send(&ligne                    // buffer
          , 1                        // 1 seul item dans le buffer
          , MPI_INT                  // C'est un tableau de 1 entier
          , task                     // La tache
          , 0                        // TAG
          , MPI_COMM_WORLD
          ) ;
  MPI_Irecv (image[ligne]            // endroit ou l'on stockera la réponse
            , LAR                    // On attend une ligne de pixels
            , MPI_INT                // Un entier par pixel (nb iterations)
            , task                   // Evidemment la même tache
            , 0                      // TAG
            , MPI_COMM_WORLD
            , &requests[task-1]      // -1 car la tache 0 est le maitre
            ) ;
}
void maitre()
{
  MPI_Request *requests ;
  MPI_Status status ;
  int image[HAU][LAR] ;          // Image que l'on calcule
  int numtasks, ligne, index, x ;

  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  requests = malloc( sizeof(*requests) * (numtasks-1) ) ;
  // Donne le travail initial
```

```c
  for(ligne=0; ligne < numtasks-1; ligne++)
    demande(image, requests, ligne, ligne+1) ;
  // Donne le travail restant au fur et à mesure.
  while( ligne < HAU )
    {
      MPI_Waitany(numtasks-1, requests, &index, &status) ;
      demande(image, requests, ligne, index + 1) ;
      ligne++ ;
      printf("Réponse de %d\n", index + 1) ;
    }
  // Termine les taches qui n'ont plus de travail en cours.
  while( ligne < HAU + numtasks - 1 )
    {
      MPI_Waitany(numtasks-1, requests, &index, &status) ;
      demande(image, requests, -1, index + 1) ;
      printf("Termine %d\n", index + 1) ;
      ligne++ ;
    }
  // Affiche le résultat
  for(ligne=0; ligne<HAU; ligne++)
    {
      for(x=0; x<LAR; x++)
        putchar( image[ligne][x] % 26 + 'a' ) ;
      putchar('\n') ;
    }
}
int mandelbrot(float x, float y)
{
  __complex long double z, z0 ;
  int n ;

  z0 = x + y * I ;
  z = 0. ;
  n = 0 ;
  while( (double)(z * conjf(z))  < 4 && n++ < MAX_ITERATION )
    z = z * z + z0 ;
  return n ;
}

enum statistiques { Statistique_Initialise, Statistique_Affiche } ;

void statistiques(enum statistiques s)
{
  static struct timeval tv_debut ;
  static struct rusage ru_debut ;
  struct timeval tv_fin ;
  struct rusage ru_fin ;
  int t, t_user, t_sys ;

  if ( s == Statistique_Initialise )
    {
      gettimeofday(&tv_debut, NULL) ;
      getrusage(RUSAGE_SELF, &ru_debut) ;
      return ;
    }
  gettimeofday(&tv_fin, NULL) ;
  t = (tv_fin.tv_sec - tv_debut.tv_sec) * 1000000
    + tv_fin.tv_usec - tv_debut.tv_usec ;
  getrusage(RUSAGE_SELF, &ru_fin) ;
  t_user = (ru_fin.ru_utime.tv_sec - ru_debut.ru_utime.tv_sec) *1000000
         + ru_fin.ru_utime.tv_usec - ru_debut.ru_utime.tv_usec ;
  t_sys = (ru_fin.ru_stime.tv_sec - ru_debut.ru_stime.tv_sec) *1000000
        + ru_fin.ru_stime.tv_usec - ru_debut.ru_stime.tv_usec ;
  printf("user=%f sys=%f real=%f %%utilisation=%g\n"
         , t_user / 1000000., t_sys / 1000000., t / 1000000.
         , ( (t_user+t_sys)/(float)t )*100
         ) ;
```

```
}

void esclave()
{
  int ligne, i, resultat[LAR] ;
  MPI_Status status ;

  statistiques(Statistique_Initialise) ;
  while(1)
    {
      MPI_Recv(&ligne     // buffer
              , 1         // 1 seul item dans le buffer
              , MPI_INT   // C'est un tableau d'entier
              , 0         // Message du Maître (ou MPI_ANY_SOURCE)
              , 0         // TAG
              , MPI_COMM_WORLD
              , &status) ;
      if ( ligne == -1 )
        {
          statistiques(Statistique_Affiche) ;
          return ;           // Fin tache
        }

      for(i=0; i<LAR; i++) // Calcul effectif
        resultat[i] = mandelbrot(((i/(float)LAR) - 0.5) * 4 - 0.5
                                , ((ligne/(float)HAU) - 0.5) * 3 ) ;
      MPI_Send(resultat           // buffer
              , LAR               // Nombre d'éléments
              , MPI_INT           //
              , 0                 // Le Maître
              , 0                 // TAG
              , MPI_COMM_WORLD) ;
    }
}

int main(int argc,char **argv)
{
    int rank, rc ;

    rc = MPI_Init(&argc,&argv) ;
    if ( rc != MPI_SUCCESS )
        {
            printf ("Error starting MPI program. Terminating.\n");
            MPI_Abort(MPI_COMM_WORLD, rc);
        }
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if ( rank == 0 )
      maitre() ;
    else
      esclave() ;

    MPI_Finalize();
    return 0 ;
}
```

Résultat d'éxécution :

```
mpirun -np 6 -machinefile nom_des_machines mandel
Réponse de 1
Réponse de 2
Réponse de 3
Réponse de 4
Réponse de 5
Réponse de 1
Réponse de 3
Réponse de 2
Réponse de 4
```

```
Réponse de 5
Réponse de 1
Réponse de 3
Réponse de 2
Réponse de 1
Réponse de 1
Réponse de 4
Réponse de 4
Réponse de 5
Réponse de 1
Réponse de 2
Termine 3
Termine 4
Termine 5
Termine 1
Termine 2
bbbbbbbbbbbbbbbbbbbbbbbccccccccccccccccccccccccccccccccccccccccccccccccccccbbb
bbbbbbbbbbbbbbbbbbbbbccccccccccccccccccccccccccccccccccccccccccccccccccccccccb
bbbbbbbbbbbbbbbbbbbcccccccccccddddddddddddddddddcccccccccccccccccccccccccccccc
bbbbbbbbbbbbbbbbbcccccccddddddddddddddddddeeefffeeedddcccccccccccccccccccccccc
bbbbbbbbbbbbbbbbcccccddddddddddddddddeeeeeeffgokhmfeeeedddcccccccccccccccccccc
bbbbbbbbbbbbbbbcccddddddddddddddddeeeeeeefffhjqcqihffeeeeddddccccccccccccccccc
bbbbbbbbbbbbbbccddddddddddddddddeeeeeeefggghilnrrrejhgfffeeeddddcccccccccccccc
bbbbbbbbbbbbbcdddddddddddddeeeeefffgjpljkjnquirhupmxihhjifeddddddccccccccccccc
bbbbbbbbbbbbcddddddddddeeffffffgghjdrrrrrrrrrrrrrrrrrerzjgfeddddddcccccccccccc
bbbbbbbbbbbcddddddddeeefqhgggggghhhjorrrrrrrrrrrrrrrrrrajifeeddddddcccccccccccc
bbbbbbbbbbbdddeeeeeeefffgilhplpnukjkorrrrrrrrrrrrrrrrrrrrrjgeeedddddcccccccccc
bbbbbbbbbbbeeeeeeffffgiimtrrrrrrrqrrrrrrrrrrrrrrrrrrrrrrhifeeeddddddcccccccccc
bbbbbbbbbbeffghhhghhijpqprrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrzhgfeeeddddddccccccccc
bbbbbbbbbbeffghhhghhijpqprrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrzhgfeeeddddddccccccccc
bbbbbbbbbbbeeeeeeffffgiimtrrrrrrrqrrrrrrrrrrrrrrrrrrrrrrhifeeeddddddcccccccccc
bbbbbbbbbbbdddeeeeeeefffgilhplpnukjkorrrrrrrrrrrrrrrrrrrrrjgeeedddddcccccccccc
bbbbbbbbbbbcddddddddeeefqhgggggghhhjorrrrrrrrrrrrrrrrrrajifeeddddddcccccccccccc
bbbbbbbbbbbbcddddddddddeeffffffgghjdrrrrrrrrrrrrrrrrrerzjgfeddddddcccccccccccc
bbbbbbbbbbbbbcdddddddddddddeeeeefffgjpljkjnquirhupmxihhjifeddddddccccccccccccc
bbbbbbbbbbbbbbccddddddddddddddddeeeeeeefggghilnrrrejhgfffeeeddddcccccccccccccc
bbbbbbbbbbbbbbbcccddddddddddddddddeeeeeeefffhjqcqihffeeeeddddccccccccccccccccc
bbbbbbbbbbbbbbbbcccccddddddddddddddddeeeeeeffgokhmfeeeedddcccccccccccccccccccc
bbbbbbbbbbbbbbbbbcccccccddddddddddddddddddeeefffeeedddcccccccccccccccccccccccc
bbbbbbbbbbbbbbbbbbbcccccccccccddddddddddddddddddcccccccccccccccccccccccccccccc
bbbbbbbbbbbbbbbbbbbbbccccccccccccccccccccccccccccccccccccccccccccccccccccccccb
user=0.029995 sys=0.000000 real=0.035391 %utilisation=84.7532
user=0.024996 sys=0.001000 real=0.037039 %utilisation=70.1855
user=0.027996 sys=0.000000 real=0.034871 %utilisation=80.2845
user=0.026996 sys=0.001000 real=0.034861 %utilisation=80.3075
user=0.029995 sys=0.001000 real=0.038782 %utilisation=79.9211
```

3.5.6 Communications collectives

Généralités :

- Ces communications sont faites entre **toutes** les machines du groupe.
- Ces primitives de communications sont bloquantes et doivent être utilisés “simultanément” sur toutes les machines du groupe.
- Dans certain cas le comportement de la fonction sera différent suivant que l'on est un émetteur ou un récepteur.
- Ces communications sont implémentées de manière plus performante que le simple appel des primitives bas niveau.
- Seul les plus bas niveau sont données, d'autres existent qui sont des combinaisons de celles de bas niveau.

MPI_Barrier(comm)	Blocage jusqu'à la synchronisation
MPI_Bcast(buffer,count,datatype,root,comm)	`root` envoit et les autres reçoivent
MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,root,comm)	Tronçonnage
MPI_Gather(sendbuf,sendcnt,sendtype,recvbuf,recvcount,recvtype,root,comm)	Collage
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)	Collage avec opération

3.5.7 Manipulation des communicateurs (groupes)

On a pas forcément besoin de faire communiquer tous les processus ensemble, il faut donc pouvoir créer des groupes de machines comme l'on veut.

L'utilisation de communicateurs permet de faire naviguer les messages dans des espaces différents afin qu'ils n'y ai pas de collisions. C'est nécessaire quand des fonctions dans une bibliothèques ont besoins de s'échanger des messages sans perturber les messages de l'application.

Le groupe est simplement un ensemble de machines qui peut être manipulé localement en mémoire avant de créer le communicateur qui nécessitera la synchronisation de toutes les machines.

Attention dans chaque communicateur les numéros des taches sont numérotés à partir de 0.

MPI_Comm_group(comm,group)	Récupère le groupe du communicateur
MPI_Group_incl(group,n,ranks,newgroup)	Création d'un groupe par extraction
MPI_Group_free(group)	Libère le groupe
MPI_Comm_create(comm,group,newcomm)	Création communicateur
MPI_Comm_free(comm)	Libération communicateur
MPI_Comm_dup(comm,newcomm)	Duplication

3.5.8 Exemple : création de communicateur

```
/* Les tâche paires dans MPI_COMM_WORLD calculent la somme de leur numéros
 * dans le communicateur MPI_COMM_WORLD.
 * Les impaires font de même.
 */
int main(int argc,char **argv)
{
  MPI_Group g_toutes, g_paires, g_impaires ;
  MPI_Comm            c_paires, c_impaires, c_le_mien ;
  int i, *indexes, rank, new_rank, numtasks ;
  int sendbuf, recvbuf ;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  indexes = malloc( sizeof(*indexes) * numtasks ) ;
  MPI_Comm_group(MPI_COMM_WORLD, &g_toutes);
  /* Création communicateur des paires */
  for(i=0; i<numtasks; i+=2)
    indexes[i/2] = i ;
  MPI_Group_incl(g_toutes, i/2, indexes, &g_paires);
  MPI_Comm_create(MPI_COMM_WORLD, g_paires, &c_paires);
  /* Création communicateur impaires */
  for(i=1; i<numtasks; i+=2)
    indexes[i/2] = i ;
  MPI_Group_incl(g_toutes, i/2, indexes, &g_impaires);
  MPI_Comm_create(MPI_COMM_WORLD, g_impaires, &c_impaires);
  /* Détermine dans quel groupe on communique */
  sendbuf = rank;
  if ( rank % 2 )
    c_le_mien = c_impaires ;
  else
    c_le_mien = c_paires ;
  /* Calcule la somme */
  MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, c_le_mien);
  MPI_Comm_rank(c_le_mien, &new_rank);
  printf("rank= %d newrank= %d recvbuf= %d  comm=%d\n"
         , rank, new_rank, recvbuf, c_le_mien);

  MPI_Finalize();
  return 0 ;
}
```

Résultat de l'exécution :

```
mpirun -np 8 -machinefile nom_des_machines comm
rank= 0 newrank= 0 recvbuf= 12  comm=134
rank= 7 newrank= 3 recvbuf= 16  comm=135
rank= 6 newrank= 3 recvbuf= 12  comm=134
rank= 2 newrank= 1 recvbuf= 12  comm=134
rank= 4 newrank= 2 recvbuf= 12  comm=134
```

```
rank= 3 newrank= 1 recvbuf= 16  comm=135
rank= 5 newrank= 2 recvbuf= 16  comm=135
rank= 1 newrank= 0 recvbuf= 16  comm=135
```

3.5.9 Topologies virtuelles

Ces topologies (grille et graphe) simplifient le travail du programmeur tout en permettant à une implémentation de `MPI` de faire des optimisations lors de la répartition des tâches sur les machines.

Voici quelques fonctions permettant de travailler avec des topologies de grille.

MPI_Cart_create(comm,ndims,dims[],periodique,reordonne,comm_new)	Création
MPI_Cart_coords(comm,rank,maxdims,coords[])	Récupère les coordonnées
MPI_Cart_shift(comm,dim,distance,voisin1,voisin2)	Dans les deux directions

Périodique indique que quand on envoie un message à travers un bord il arrive de l'autre coté.

Réordonne indique que l'ordre des taches à le droit d'être modifié.

3.6 Meta-systèmes du meta-ordinateur

Les "meta-systèmes" sont des environnements complets permettant de facilement lancer des tâches parallèles sans trop se soucier de l'endroit ou elles s'exécutent. Il permettent le partage des CPU, données, applications.

Et notamment dans le cas des "grilles" (fournisseurs temps CPU), certains programmes vont voir leur exécution utiliser des machines appartenant à plusieurs organisations différentes. Ces programmes doivent donc fonctionner sur des machines qui ne sont pas configurées identiquement, elles :

- pas la même gestion des droits d'accès ;
- n'ont pas les mêmes processeurs ;
- pas les mêmes accès aux fichiers ;
- peuvent être éteintes à tout moment;
- Sont connecté par Internet (insécure et lent)

Un *cluster* `MOSIX` n'est pas une "grille" car il est géré par une organisation unique. De plus il fonctionne de manière optimale sur réseau local seulement.

En plus les différentes implémentions permettent de définir des "services". Elles le font car elles ont elles-même besoins de créer des "services" pour fonctionner. Reste à savoir si vous avez besoin de cette optique service ou bien si seul le parallélisme vous intéresse. Quand un client demande un service le système se charge de le découvrir et de lancer si nécessaire.

Le standard en construction `OGSI` (*Open Grid Services Infrastructure*) défini les services de la grille, il utilise `WSDL` (*Web Services Description Language*) et `SOAP` (*Simple Object Access Protocol*).

Ces différents systèmes sont "compatibles" entre-eux. En effet ils offrent des passerelles permettant de passer de l'un à l'autre de manière transparente.

3.6.1 Bibliothèques et Web services

`DCE` (*Distributed Computing Environment*), `CORBA` (*Common Object Request Broker Architecture*), `DCOM` et `NET` : ces systèmes on pour but de simplifier la communication entre services et la découverte de services. `DCE` offre en plus des services tout fait, par exemple la gestion des utilisateurs et des fichiers.

Web service :

- Définition : Web Services Description Language (WSDL)
- Implémentation
- Déploiement. Nom, droits d'accès, publication méthode, persistance, ...
- Compilation et génération d'un fichier distribuable.
- Ajout du service.

Les services peuvent donner des informations sur eux-mêmes : coûts, vitesse, ...

3.6.2 *Globus*

Créé par un consortium.

Réponds au besoins d'une organisation virtuelle :

- Indépendance du système d'exploitation
- Auto destruction des tâches dépassant leur temps limite.
- Machines qui tombent en panne.
- Auditing.
- Web Services Description Language (WSDL) du W3. (SOAP) pour les appels.
  - Notification services

- the Meta Directory Service (MDS-2), which provides for information discovery through soft state registration : Grid Notification Protocol, for lifetime management of published information.
- Authentification (unique) et sécurité. Grid Security Infrastructure (GSI).
- Système de fichiers avec réplication intelligente.
- Mise à niveau des services (Numéro de version)
- Grid Resource Allocation and Management (GRAM) protocol.
- gatekeeper service, which provides for secure, reliable, service creation and management.

Globus est un ensemble de logiciel ouvert et gratuit. mais pas 100% libre (procès, mention).

Les services doivent être compilés sur toutes les machines.

Pas de nommage ou de vue globale du système, on peut perdre des objets si l'on oublie leur adresse.

3.6.3 *Legion*

C'est un système ancien (1993) avec des documentations compréhensibles. C'est un système simple que l'on peut prendre facilement en main. Il est actuellement commercialisé sous le nom de *Avaki*.

Les concepts généraux :

- Tout est objet : processeur, disque, classes, processus, fichier, ...
- Un objet est un processus s'il est actif, sinon il reste sur le disque.
- Les objets peuvent être organisés sous forme d'un système de fichier.
- Les droits d'accès aux objets sont ceux des fichiers
- On peut associer les attributs que l'on veut aux objets, des attributs standards permettent de décider comment faire exécuter les objets.
- Les objets peuvent rajouter leur couche de sécurité personnelle.
- Pour des utilisateurs de base, l'utilisation de *Legion* ce fait simplement avec des commandes interactives et l'utilisation d'une librairie C permettant de accéder au contenu des objets.

Contrairement à *Globus* les services doivent fournir certaines méthodes obligatoires afin d'autoriser entre autre la vue système de fichier. Pour l'utilisateur cette interface commune est souvent suffisante et donc plus simple à appréhender que de multiples interfaces.

D'autre part, *Globus* réutilise un ensemble de protocoles et d'outils existants, là aussi cela complique la tâche de l'utilisateur car il faut les apprendre. Avec *Legion* des objets comme les autres les réimplémente de manière naturelle.

Le but de *Legion* est de créer une machine virtuelle ce n'est pas celui de *Globus*.

Pas parallélisme fin car les objets communiquent lentement entre eux. Mais heureusement il y a un objet `MPI` qui permet de travailler sur plusieurs machines à la fois.

3.6.4 Autres

Berkeley NOW est un projet mort en 1997, son point fort était un système de fichier réparti de manière transparente entre toutes les machines. Il ne se préoccupait pas des problèmes de sécurité.

Nanos est un projet européen mort en 1999 c'est un système orienté programmation parallèle, avec son propre compilateur. Il ne se préoccupait pas des problèmes de sécurité.

Projet HPVM mort aussi en 1999.

Une autre façon encore expérimentale de créer une "grille" est d'utiliser des machines virtuelles (Voir *Xen*).

- On lance une machine virtuelle sur chaque machines faisant partie de la "grille".
- On alloue/comptabilise les ressources de la machine virtuelle.
- On peut faire migrer des machines virtuelles.
- La machine virtuelle offre un environnement uniforme sur toute la "grille".
- Le contenu de la machine virtuelle ne peut pas sortir de sa boite.
- Une même machine peut faire partie de plusieurs "grilles".

4 Réseaux pour les *clusters*

Plein de solutions, mais le futur semble assuré pour `IP` qui reste le protocole de choix même s'il sert à transmettre d'autre protocoles.

4.1 Logiciel et protocoles

4.1.1 Bidouiller les paramètres `TCPIP`

?

4.1.2 Active messages (1992)

C'est un méthode permettant de traiter plus rapidement les messages réseau et donc de diminuer la latence des communications. Les messages sont traité plus rapidement car ils ne franchissent pas toutes les couches `TCPIP` en étant recopié de multiple fois et en gérant des tampons mémoire.

Les messages contiennent la référence au gestionnaire qui va les recevoir.

Les messages contiennent des informations permettant au gestionnaire de savoir ou les stocker.

4.1.3 `RDMA` (*Remote Direct Memory Access*)

C'est une norme `IETF` (*Internet Engineering Task Force*). Elle au dessus de `TCPIP`.

Le but est d'éliminer les copies multiples des données et non de diminuer le coût des traitements `TCPIP`.

C'est le contenu du paquet qui est normalisé pas la façon dont on va l'analyser.

4.1.4 Mémoire partagée distribuée

`SCI` (*Scalable Coherent Interface*) (`ANSI`/IEEE? Standard 1596-1992) Il va plus loin que `RDMA`, Il gère vraiment une mémoire partagée. Il gère aussi les caches mémoire en les vidant si la mémoire à changée. (normé en 1992, peu utilisé)

4.2 Matériels

De plus en plus de carte réseau supportent `TOE` (*Transport Offload Engine*). Elles analyse complètement les paquets reçus et les stocke là ou l'application les demande sans aucune intervention du système d'exploitation.

La société Emulex (formerly Giganet) implémente le protocole `VIA` (*Virtual Interface Architecture*) , ou VI-IP (`VIA`sur `IP`) sur certaines de ses cartes réseaux. C'est en fait du `RDMA`.

Infiniband était au départ une architecture réseau devant remplacer le bus interne des ordinateurs afin d'augmenter la bande passante totale. Microsoft et IBM n'en ont pas voulus, mais les performances étant meilleur qu'ethernet ce support de comunication devient maintenant un moyen de communication entre machines. Il supporte la redondance des chemins. Il utilise des entêtes IPV6, les réseaux sont limités à 65000 “machines”.

Myrinet Protocol (`ANSI`/VITA 26-1998). Protocol de passage de message MG (le nouveau est MX Myrinet eXpress) Ce n'est pas du `IP`qui circule.

`iSCSI`(*Internet Small Computer System Interface*) est simplement une encapsulation de `SCSI` sur `IP`.

Fibre Channel(`ANSI`), longue distance, 4Gb/s, pas `IP` mais il existe un Internet *Fibre Channel* pour accéder à du *Fibre Channel* à partir d'Internet et l'inverse qui permet de faire de l'`IP` par dessus du *Fibre Channel*.

4.3 Conclusion

Cela évolue vite...

Index

ACPI, 4
ADSL, 6
amd, 9
ANSI, 22
Apache, 10
API, 13, 14
ARP, 5, 10
autofs, 9
Avaki, 21

BIOS, 3

cfengine, 12
cluster NFS, 12
clusterit, 13
Coda, 9
Condor, 13
CORBA, 20
crossbar, 3

DB9, 9
DCE, 20
DCOM, 20
DHCP, 3, 12
DHT, 9
direct routing, 12
DNS, 5, 6, 11, 12
DRBD, 7, 8, 10–12
DSH, 13
DVT, 13

ENDB, 7, 8
Ext3, 7

fanout, 13
Fenêtres, 13
floppy, 3
fsck, 7

GFS, 5, 7–9, 11
Globus, 20, 21
GNQS, 13
GPFS, 8
GPL, 9

heartbeat, 9, 10, 12
HTTP, 10
HURD, 9

IBM, 8
IEEE 1596-1992, 3
IETF, 22
Infiniband, 22
Internet, 5–7, 11
IP, 5, 6, 9–12, 21, 22
IP tunneling, 5, 12
iSCSI, 8, 22
ISDN, 6

KVM, 3

LDAP, 10
Legion, 21
Linux, 3, 12
Lustre, 9
LVM, 7
LVS, 5, 11, 12

MAC, 5, 11, 12
mirroring, 7
MOSIX, 4, 12, 13, 20
MPI, 13, 14, 20, 21
MPICH, 14
mpirun, 14
MS5, 9
MTBF, 4
MTTF, 4
MTTR, 4
multicast, 12

NAT, 5
NET, 20
NFS, 3, 10, 12, 13, 15
NOW, 2
NTP, 13
NUMA, 3

OGSI, 20

PBS, 13
pconsole, 13
Peer to peer, 9
PVFS, 8
PXE, 3

RAID, 4–7
RAID1, 7, 9
RAID5, 7
RDMA, 22
RedHat, 8
Reiserfs, 7
rpc.statd, 10
RS232C, 3, 6, 10
rsh, 13

SAN, 5, 6, 8
SCI, 3, 22
SCSI, 5, 22
SCSI3, 8
SHA, 9
SMBFS, 10
snapshot, 7, 8
SNMP, 12
SOAP, 20
ssh, 13
Stonith, 9, 10, 13
switch, 3, 8, 11
syslog, 3

taktuk, 13
TCPIP, 9, 12, 21, 22
TOE, 22

UPS, 13
USB, 4

VIA, 22

watchdog, 13
WOL, 4

WSDL, 20

XDR, 15
Xen, 21