Flags d'un Segment TCP

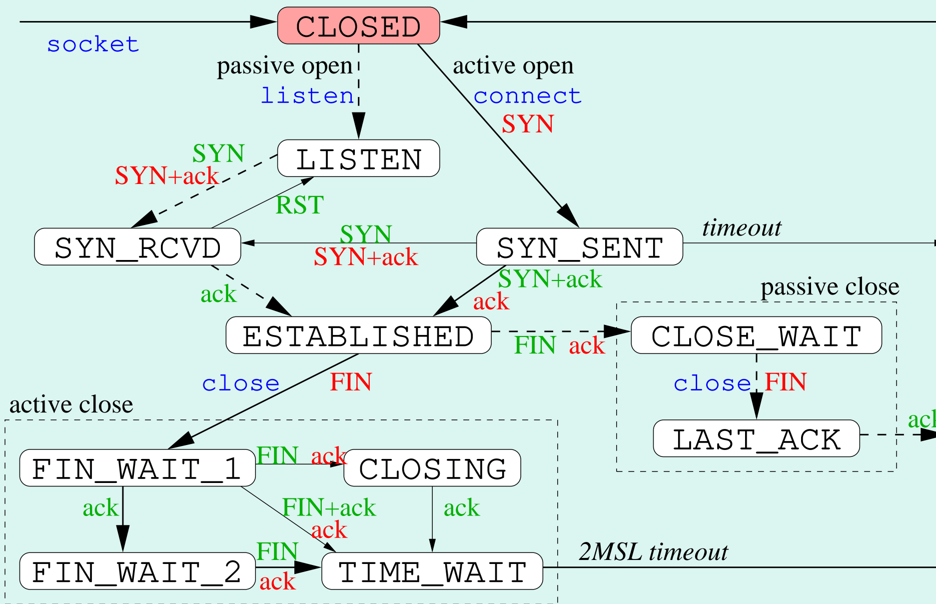
ⓒ 2006 Marc Feeley

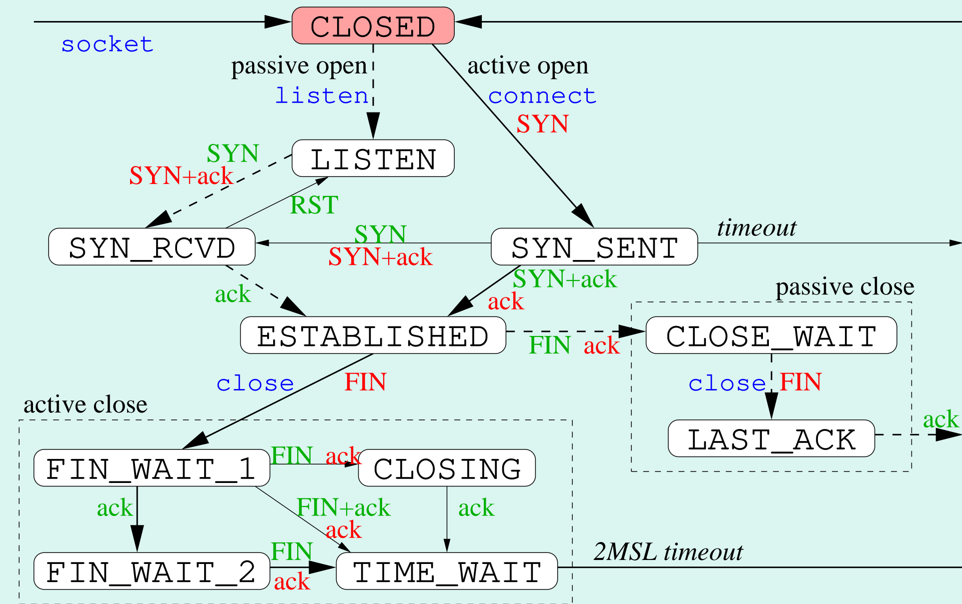
- Chaque segment TCP peut contenir un des flags suivants:
  - **SYN**: synchroniser les accusés de reception
  - **FIN**: signaler la fin d'une connexion
  - **RST**: mise-à-zéro d'une connexion
- Chaque segment TCP contient aussi un accusé de reception qui indique combien de données ont été reçues correctement
- Les données, flags et accusés de reception peuvent être combinées dans un même segment pour minimiser le nombre de paquets envoyés

États d'un Socket

Établissement d'une Connexion (1)

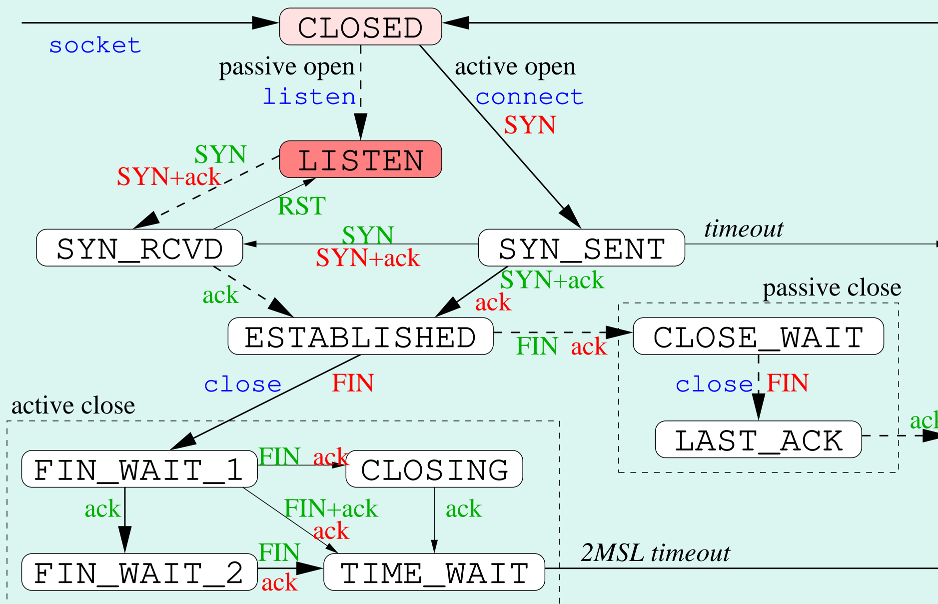
1) Serveur Client

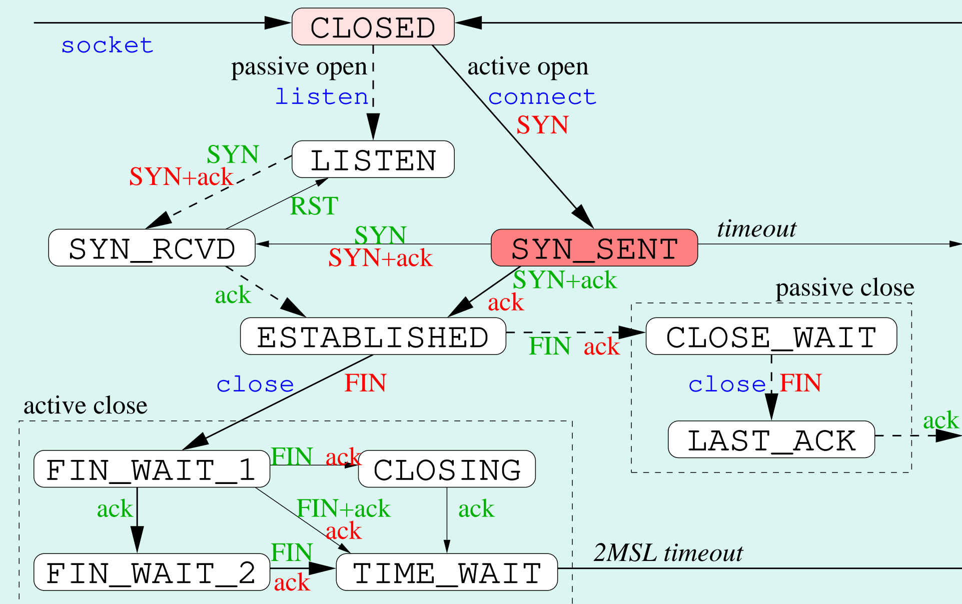




Établissement d’une Connexion (2)

2) Serveur | Client

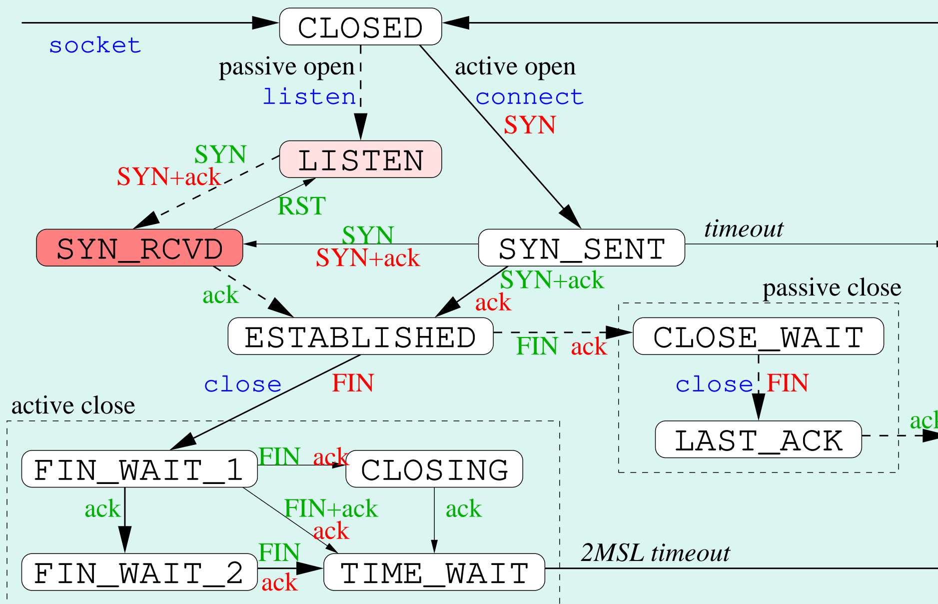


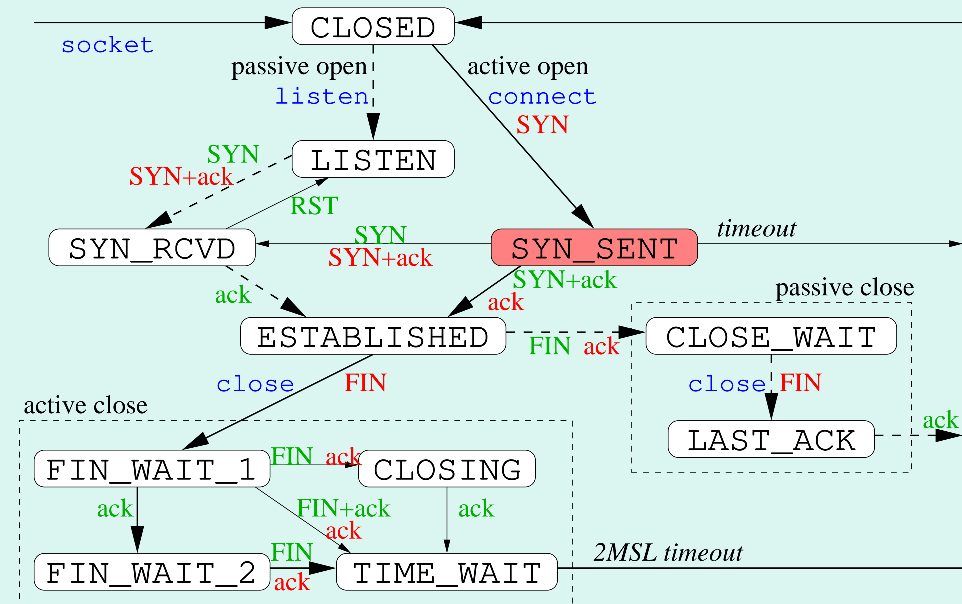


Établissement d'une Connexion (3)

ⓒ 2006 Marc Feeley

3) Serveur Client

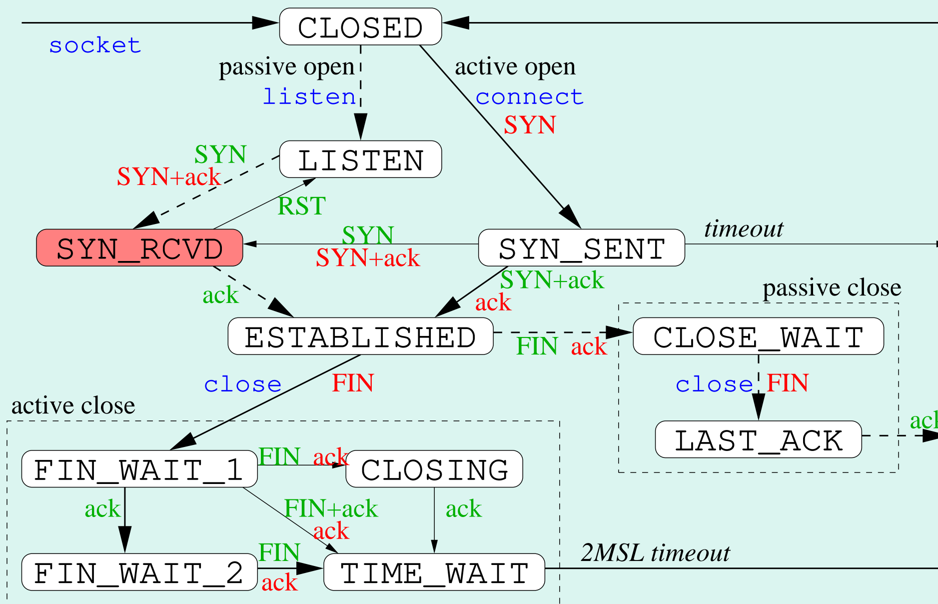


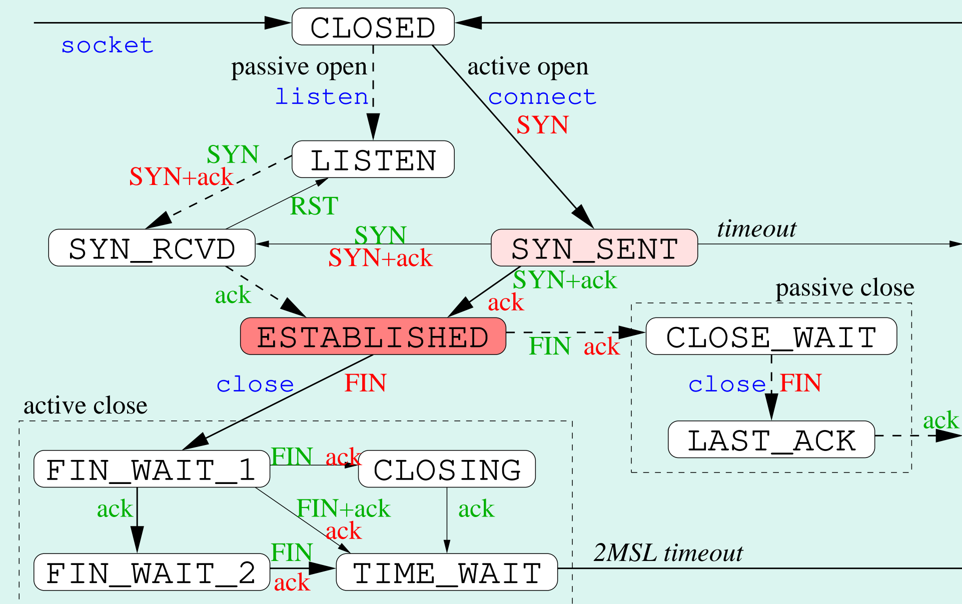


Établissement d'une Connexion (4)

© 2006 Marc Feeley

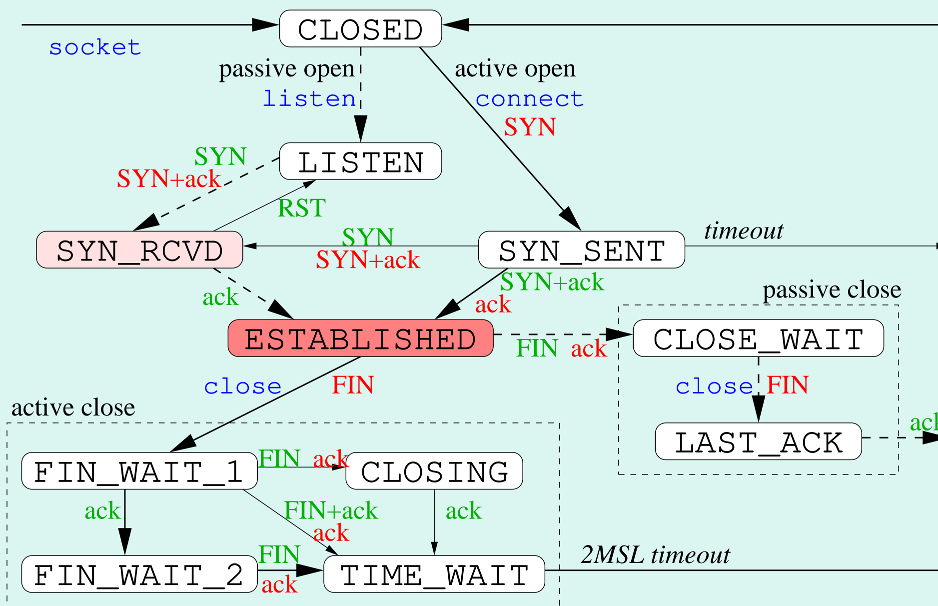
4) Serveur Client

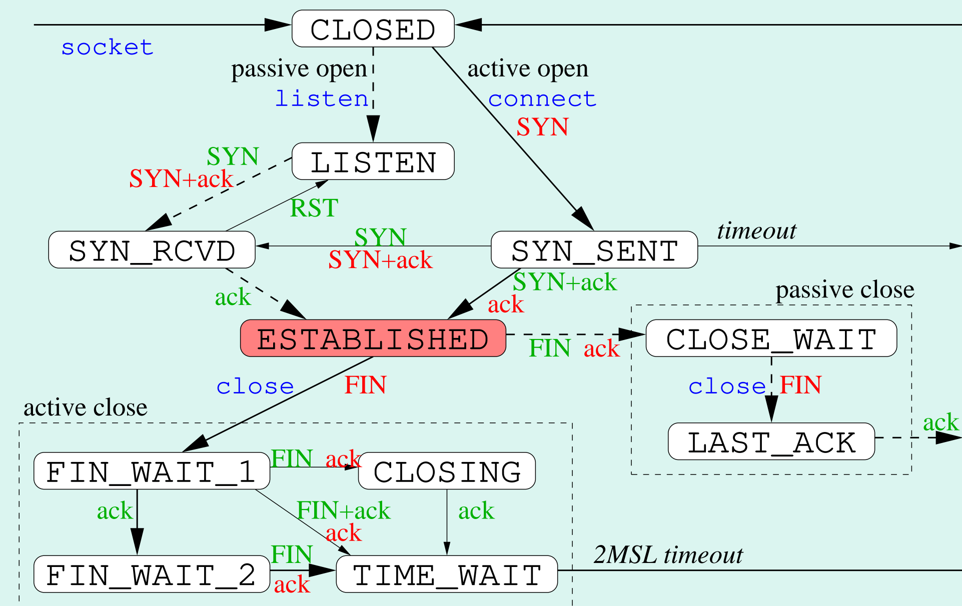




© 2006 Marc Feeley

5) Serveur — Client

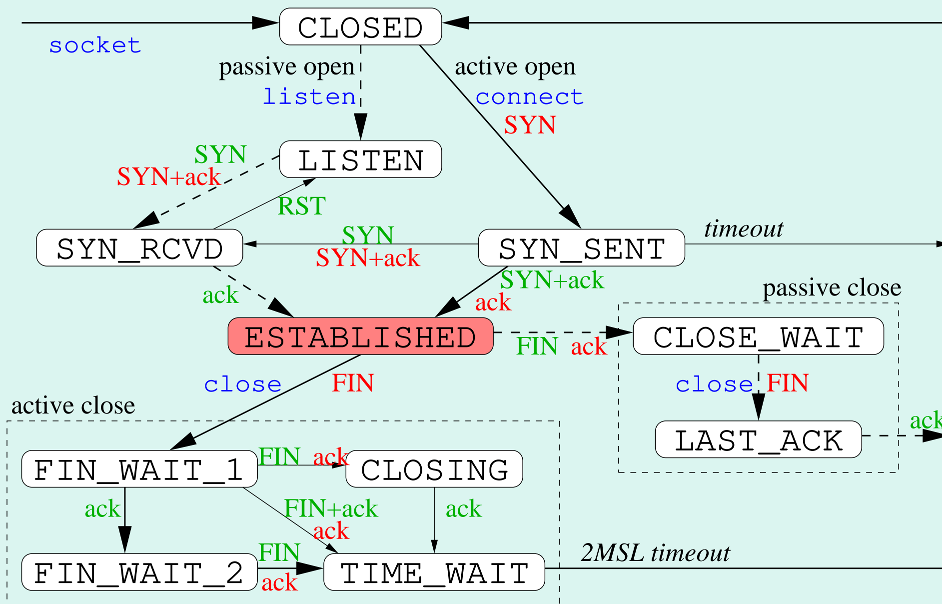


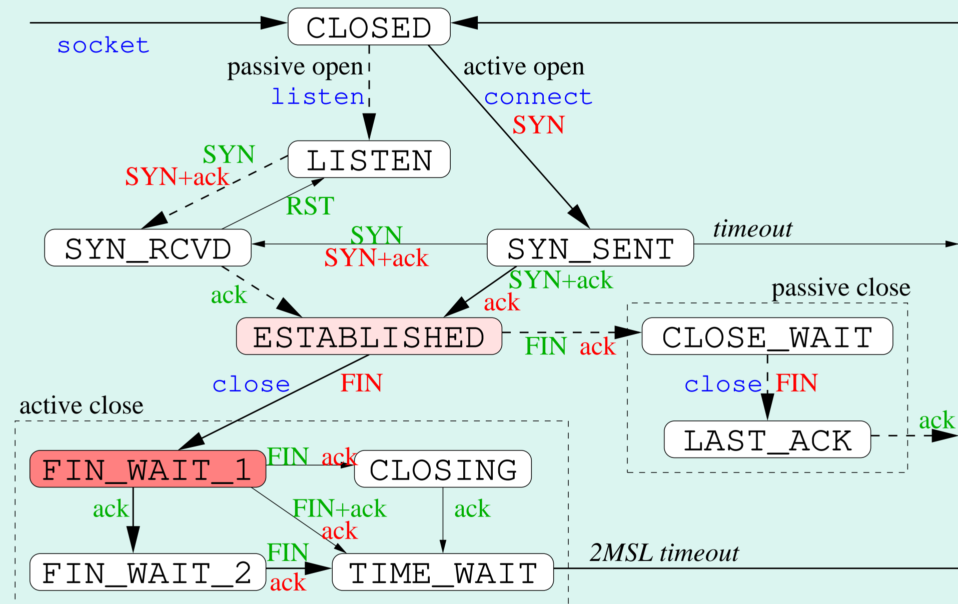


Fermeture d'une Connexion (1)

© 2006 Marc Feeley

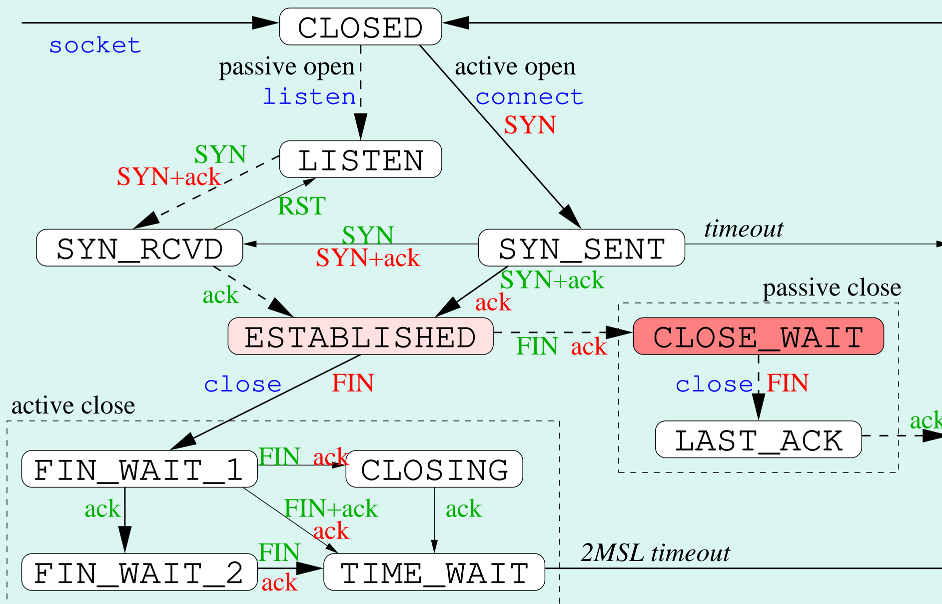
1) Serveur Client

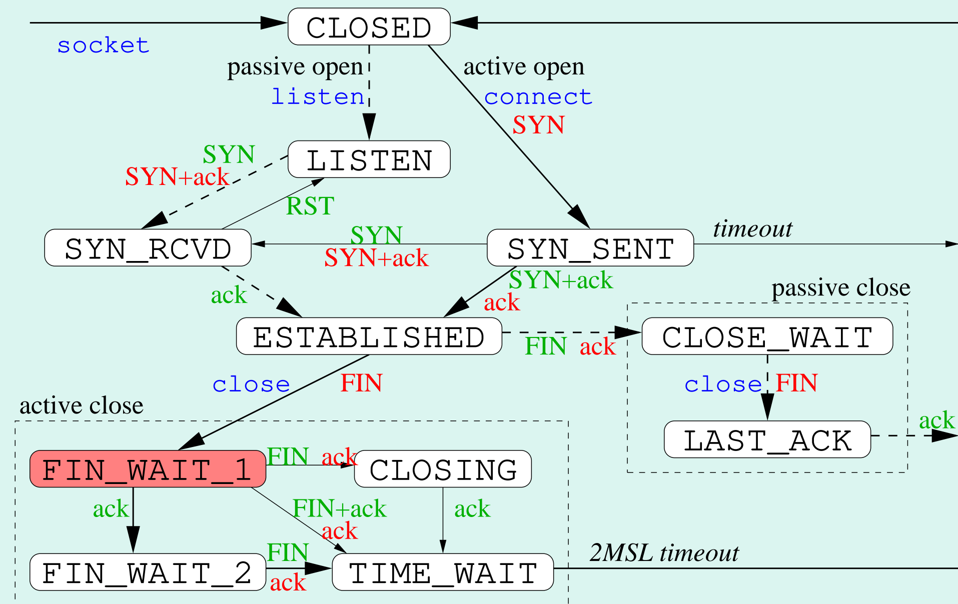




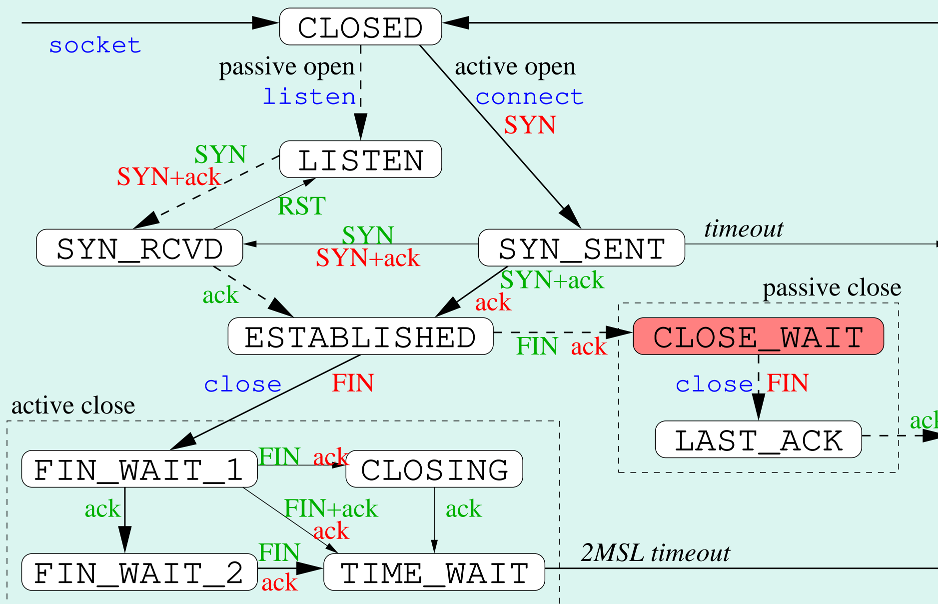
Fermeture d'une Connexion (2)

2) Serveur Client

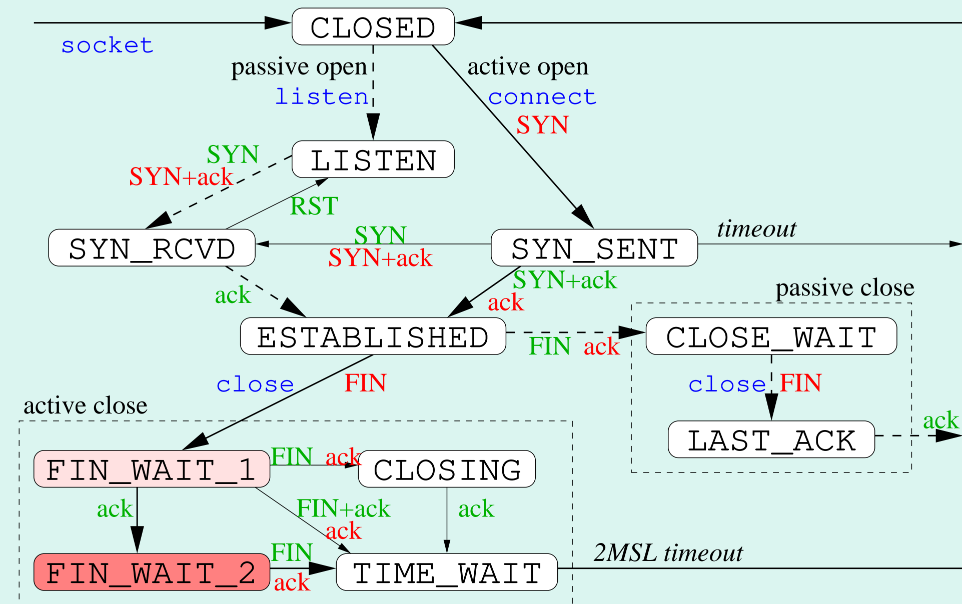




3) Serveur

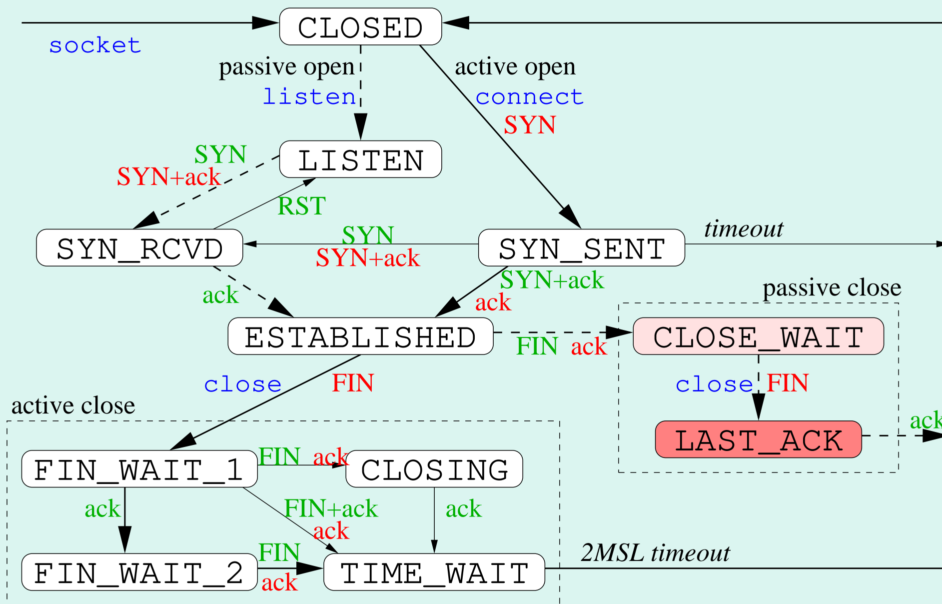


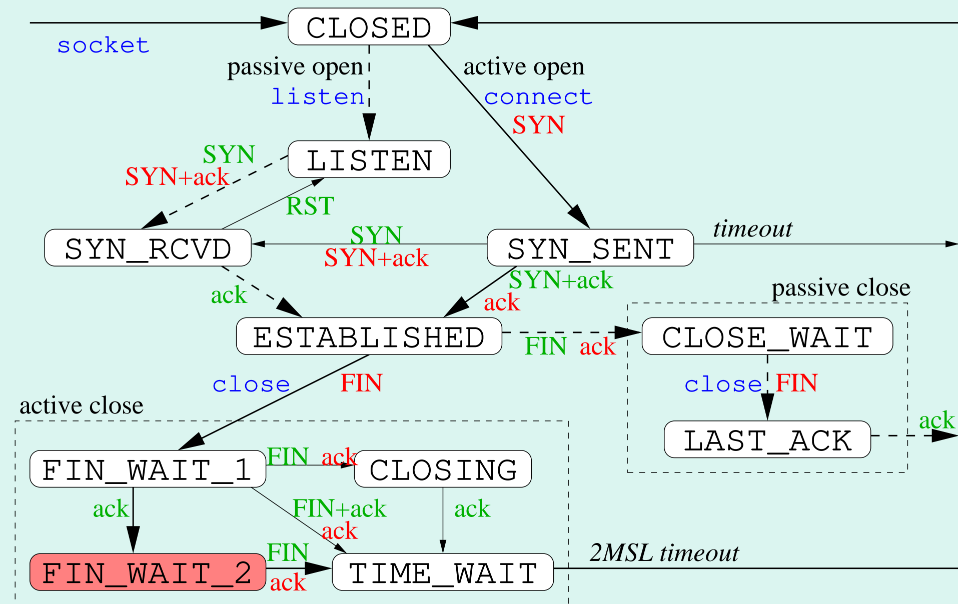
Client



ⓒ 2006 Marc Feeley

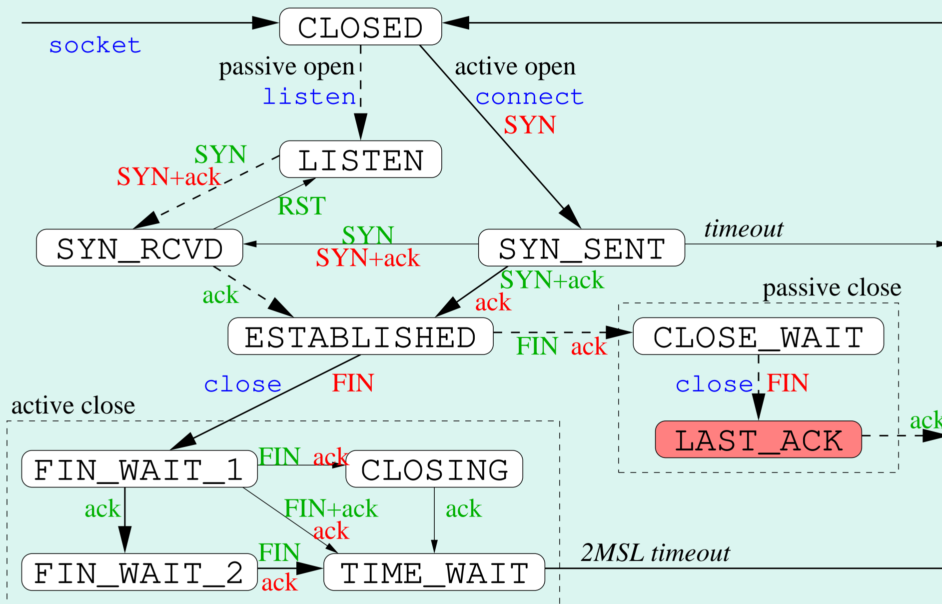
4) Serveur Client

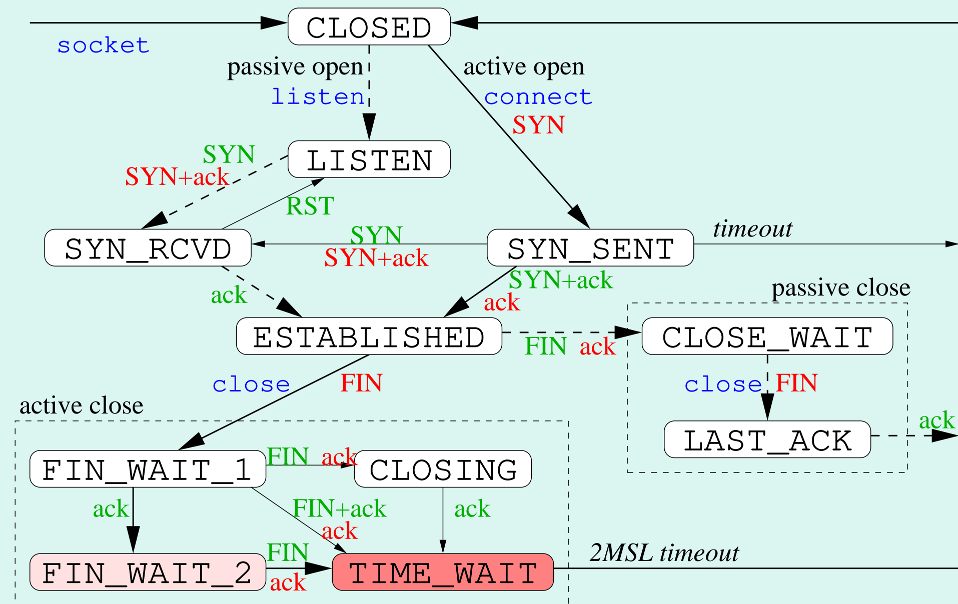




Fermeture d’une Connexion (5)

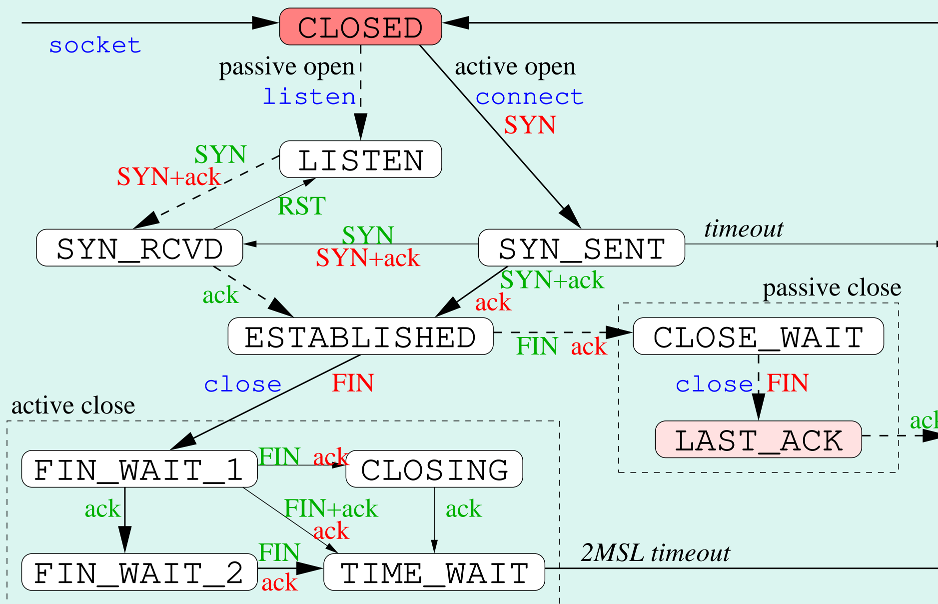
5) Serveur Client

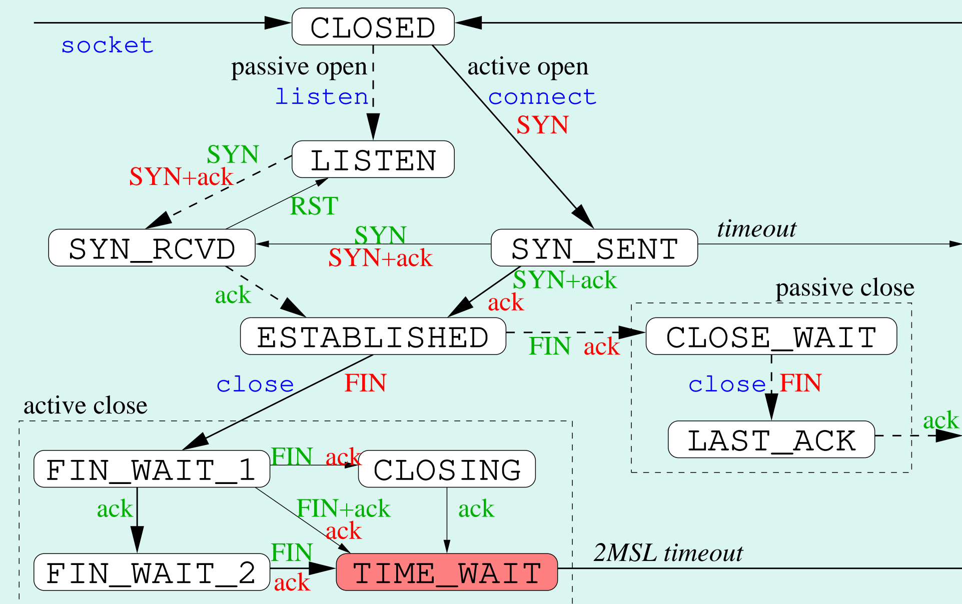




© 2006 Marc Feeley

6) Serveur Client

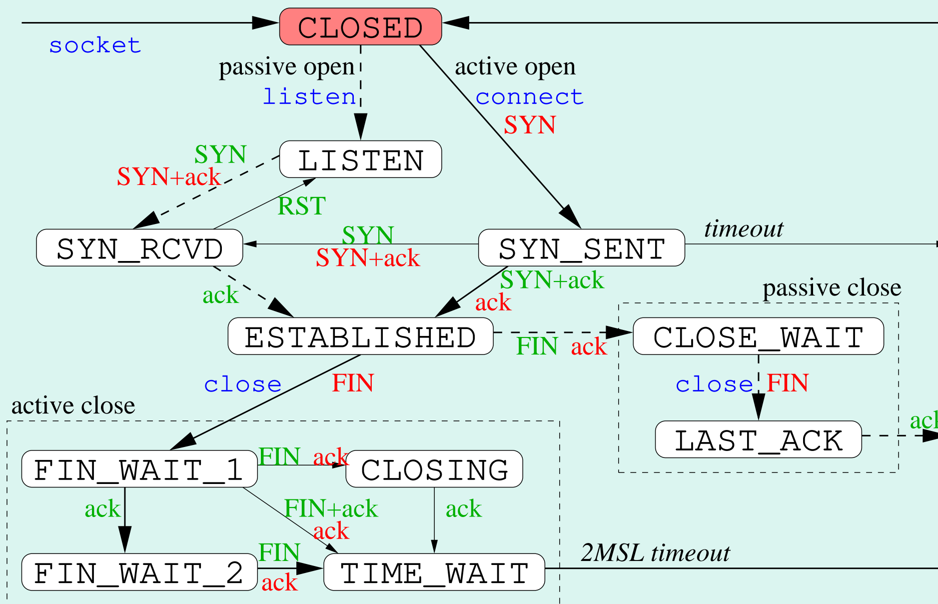


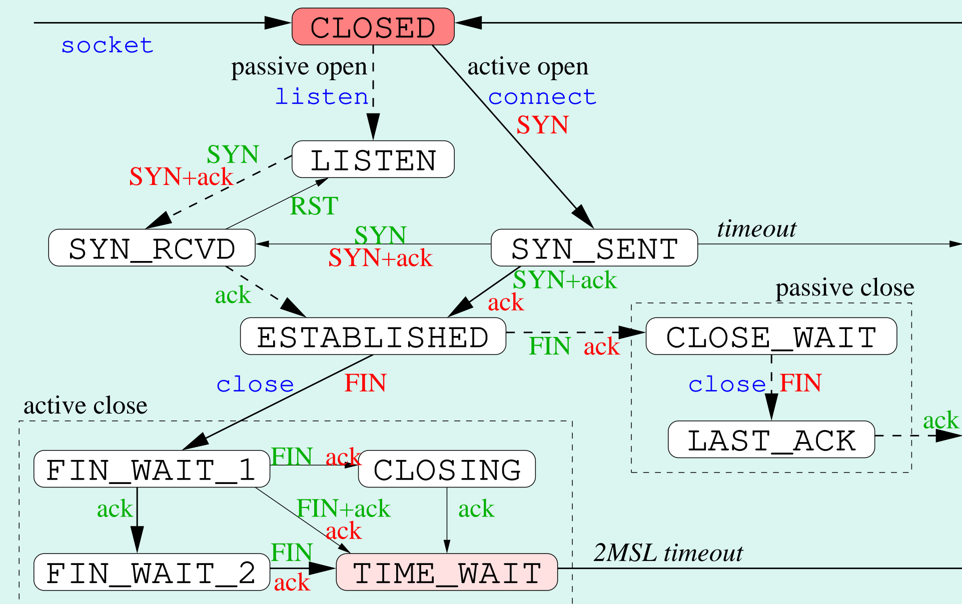


Fermeture d’une Connexion (7)

© 2006 Marc Feeley

7) Serveur Client





Fermeture Partielle (1)

© 2006 Marc Feeley

- Une lecture sur un socket bloque s'il n'y a **pas de données reçues** et la **connexion n'est pas fermée**
- Une lecture sur un socket retourne "fin de fichier" (`read` retourne 0) si la **connexion est fermée**
- Parfois un des processus (serveur ou client) désire indiquer à l'autre qu'il a **fini d'envoyer des données** sur la connexion mais qu'il peut encore en recevoir
- Par exemple, envois d'une requête de **longueur indéterminée** (une "fin de fichier" représente la fin de la requête pour le serveur)
- Si le client utilise `close` à la fin de la requête, le serveur va lire une fin de fichier **mais le client ne pourra pas lire la réponse du serveur**

© 2006 Marc Feeley

- L'appels système `shutdown` permet de fermer sélectivement chacun des sens d'une connexion (écriture ou lecture)

  `int shutdown (int` fd`, int` how`);`

  how doit être `SHUT_RD`, `SHUT_WR` ou `SHUT_RDWR`

- Exemple :

```
1. int fd = socket (PF_INET, SOCK_STREAM, 0);
2.
3. connect (fd, ...);
4.
5. write (fd, ...);
6.
7. shutdown (fd, SHUT_WR);
8.
9. read (fd, ...);
10.
11. close (fd);
```

Mémoire Secondaire

© 2006 Marc Feeley

- La mémoire principale (RAM) est rapide, mais **chère** et **volatile**
- La **mémoire secondaire** permet de stocker une **grande quantité** d'information de façon **permanente** et à **faible prix**
- Le temps d'accès est **beaucoup plus grand** que pour la RAM (dans le meilleur cas, quelques millisecondes par accès)
- Exemples typiques:
  - Disque dur (capacité élevée)
  - Floppy et ZIP (faible capacité)
  - CD-RW, CD-ROM, CD-R et WORM (capacité moy.)
  - Ruban magnétique (capacité très élevée)

Systèmes de Fichier (1)

ⓒ 2006 Marc Feeley

- Un système de fichier est une **organisation** des données permanentes dans la mémoire secondaire et des **méthodes pour gérer** ces données
- Facteurs à considérer:
  - **Identification**: syntaxe des noms de fichier
  - **Types permis**: réguliers (texte ou binaire), sous-répertoires, périphériques (“devices”), etc
  - **Attributs**: de partage? date de création/accès?
  - **Performance**: latence/débit d’accès aux fichiers séquentiel/aléatoire, utilisation économe de l’espace
  - **Fiabilité**: peut-il y avoir perte d’information? outils pour réparer les inconsistances? “backups”?
  - **Partage d’information**: fichiers accessibles à d’autres

Systèmes de Fichier (2)

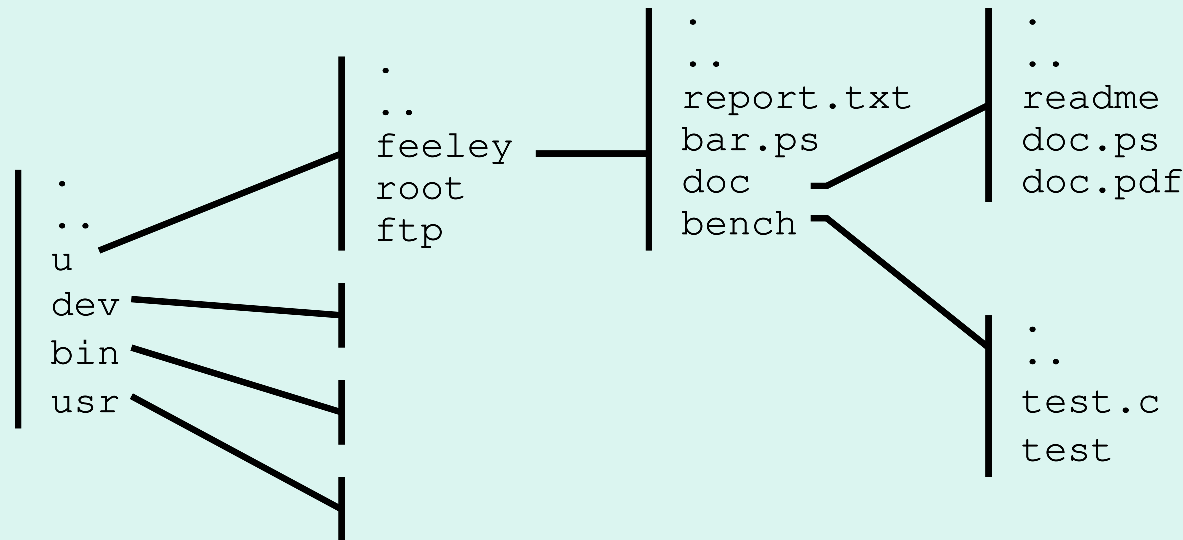
ⓒ 2006 Marc Feeley

- Un fichier normal est une **séquence d'octets** d'une certaine longueur
- Opérations sur les fichiers (et fonction UNIX)
  - **Création**: allouer une entrée de répertoire (`creat`)
  - **Ouverture/Fermeture**: début/fin des accès au contenu du fichier, le descripteur obtenu à l'ouverture contient un **index** de lecture/écriture (`open`/`close`)
  - **Écriture**: écrit des données à l'index (ce qui demande possiblement d'allouer de l'espace pour étendre le fichier) et l'avance (`write`)
  - **Lecture**: lit des données à l'index et l'avance (`read`)
  - **Repositionnement**: change l'index (`lseek`)
  - **Élimination**: élimine le fichier et libère l'espace associé (`unlink`)

Identification de Fichiers (1)

© 2006 Marc Feeley

- La majorité des systèmes de fichier utilisent une **organisation hiérarchique** sous forme d'arbre
- Le **répertoire racine** ("root") est à la racine de l'arbre; les répertoires peuvent contenir des **sous-répertoires**

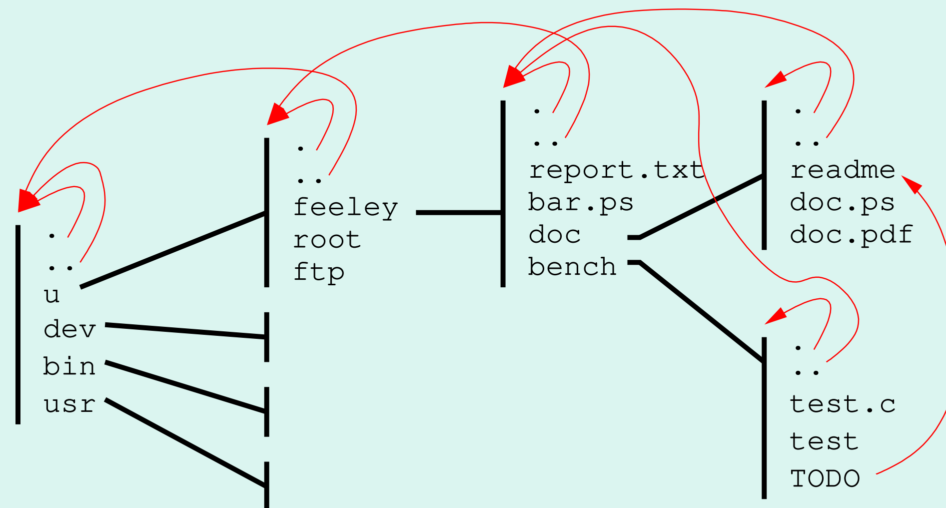


- Une position particulière dans l'arbre peut-être identifiée de façon unique par le **chemin** qui mène à cette position depuis la racine

Identification de Fichiers (2)

© 2006 Marc Feeley

- Certains systèmes de fichier permettent des **alias** (un fichier peut avoir plus d'un nom)
- UNIX permet les liens **durs** ("hard link") et **symboliques**, qui peuvent introduire des cycles
  Pour créer un lien: `ln [-s] existant nouveau`
- Les répertoires UNIX contiennent toujours les sous-répertoires "." et ".." qui sont des liens durs vers le répertoire lui-même et son parent



Identification de Fichiers (3)

ⓒ 2006 Marc Feeley

- Un **compteur de référence** est requis pour compter les **liens durs** et effacer le fichier lorsque = 0

```
% mkdir test
% cd test
% echo aaa > a
% echo bbbbb > b
% ln b b2
% echo ccccccc > c
% ln c c2
% ln -s c c3
% ls -al
total 36
drwxrwxr-x    2 feeley   feeley       4096 Dec  3 08:28 .
drwxrwxr-x    7 feeley   feeley      12288 Dec  3 08:00 ..
-rw-rw-r--    1 feeley   feeley          4 Dec  3 08:30 a
-rw-rw-r--    2 feeley   feeley          6 Dec  3 08:30 b
-rw-rw-r--    2 feeley   feeley          6 Dec  3 08:30 b2
-rw-rw-r--    2 feeley   feeley          8 Dec  3 08:30 c
-rw-rw-r--    2 feeley   feeley          8 Dec  3 08:30 c2
lrwxrwxrwx    1 feeley   feeley          1 Dec  3 08:30 c3 -> c
% cat c3
ccccccc
% rm c
% cat c2
ccccccc
% cat c3
cat: c3: No such file or directory
% rm c3
% cat c2
ccccccc
% rm c2
```

Identification de Fichiers (4)

ⓒ 2006 Marc Feeley

- Chaque processus UNIX a un **répertoire de travail présent**, qui peut être obtenu avec `getwd` et se changer avec `chdir`
- Les noms de fichiers non-absolus sont relatifs au répertoire de travail présent
- La **syntaxe des chemins de fichier** varie entre systèmes
  - **UNIX**: `/a/b/c` (absolu), `xyz` et `x/y/z` (relatifs)
  - **Windows**: `C:\a\b\c` (absolu), `\a\b\c` (relatif au disque présent), `xyz` et `x\y\z` (relatifs)
  - **MACOS**: `a:b:c` (absolu sur disque `a`), `xyz` et `:x:y:z` et `::x:y:z` (relatifs)

Routines `chdir` et `getwd` (1)

© 2006 Marc Feeley

- L'appel système `chdir` change le "répertoire de travail" du processus (`open` se servira de ce répertoire si le nom de fichier est relatif)

```
int chdir (char* path);
```

- L'appel système `getwd` copie le chemin du "répertoire de travail" dans le tableau *buf* (au plus `PATH_MAX` caractères dans un chemin)

```
char* getwd (char* buf);
```

Routines `chdir` et `getwd` (2)

ⓒ 2006 Marc Feeley

- Exemple :

```
int main (int argc, char* argv[])
{ char dir[PATH_MAX+1];
  getwd (dir);
  printf ("working dir = %s\n", dir);
  chdir ("/etc");
  getwd (dir);
  printf ("working dir = %s\n", dir);
  return 0;
}
```

- Exécution:

```
% pwd
/home/feeley/ift2240
% ./a.out
working dir = /home/feeley/ift2240
working dir = /etc
```

Routine `creat`

© 2006 Marc Feeley

- L'appel système `creat` permet de créer une nouvelle entrée dans un répertoire

  `int creat (char*` *filename*`, mode_t` *mode*`);`

- Cela est équivalent à un appel à `open` :

  `open (`*filename*`, O_CREAT | O_TRUNC | O_WRONLY,` *mode*`);`

Routine `umask`

© 2006 Marc Feeley

- L'appel système `umask` change le mode de création de fichier du processus

```
mode_t umask (mode_t mask);
```

- Chaque bit à 1 dans le $mask$ est un bit de permission qui sera **retiré** du $mode$ passé à `open` lors de la création d'un fichier
- Le masque de défaut est 022 (ne pas permettre les écritures du groupe ou des autres usagers)

Routines `link`, `symlink` et `unlink` (1)

- Les appels système `link` et `symlink` permettent de créer des liens durs et symboliques vers des fichiers existants, et l'appel système `unlink` permet d'éliminer un lien dur

```
int link (char* existant, char* nouveau);
int symlink (char* existant, char* nouveau);
int unlink (char* existant);
```

- Pour `link` *existant* et *nouveau* doivent être sur le même système de fichier

- `link` incrémente le nombre de liens vers *existant*

- `unlink` décrémente le nombre de liens vers *existant*; lorsque ça tombe à 0 **et** plus aucun processus n'a ce fichier d'ouvert, l'espace de ce fichier est récupéré

Routines `link`, `symlink` et `unlink` (2)

© 2006 Marc Feeley

- Exemple :

```
int main (int argc, char *argv[])
{ int fd = creat ("test1", 0644);
  write (fd, "allo\n", 5);
  close (fd);
  link ("test1", "test2");
  link ("test2", "test3");
  unlink ("test1");
  return 0;
}
```

- Exécution:

```
% ./a.out
% ls -l test*
-rw-r--r--   2 feeley  feeley      5 Apr  2 21:27 test2
-rw-r--r--   2 feeley  feeley      5 Apr  2 21:27 test3
% cat test2
allo
% cat test3
allo
```

Routines `stat` et `lstat` (1)

- L'appel système `stat` obtient des informations sur un fichier donné (et `lstat` ne traverse pas les liens symboliques)

```
int stat (char* filename, struct stat* buf);
```

- Structure `struct stat`:

```
struct stat {
  dev_t         st_dev;      /* device */
  ino_t         st_ino;      /* inode */
  mode_t        st_mode;     /* type and protection */
  nlink_t       st_nlink;    /* number of hard links */
  uid_t         st_uid;      /* user ID of owner */
  gid_t         st_gid;      /* group ID of owner */
  dev_t         st_rdev;     /* device type (if inode devic
  off_t         st_size;     /* total size, in bytes */
  unsigned long st_blksize;  /* blocksize for filesystem I/
  unsigned long st_blocks;   /* number of blocks allocated
  time_t        st_atime;    /* time of last access */
  time_t        st_mtime;    /* time of last modification *
  time_t        st_ctime;    /* time of last change */
};
```

ⓒ 2006 Marc Feeley

- Exemple :

```
void info (char* filename)
{ struct stat s;

  if (lstat (filename, &s) < 0) return;

  printf ("file %s is a ", filename);
  if (S_ISREG(s.st_mode))  printf ("regular file");
  if (S_ISDIR(s.st_mode))  printf ("directory");
  if (S_ISCHR(s.st_mode))  printf ("character device");
  if (S_ISBLK(s.st_mode))  printf ("block device");
  if (S_ISFIFO(s.st_mode)) printf ("fifo");
  if (S_ISLNK(s.st_mode))  printf ("symbolic link");// lsta
  printf (" with size=%ld\n", s.st_size);
}

int main (int argc, char* argv[])
{ info ("/etc/passwd");
  info ("/dev");
  info ("/dev/ttyS0");
  info ("/dev/modem");
  info ("/dev/hda1");
  info ("/u/feeley/x");
  return 0;
}
```

Routines `stat` et `lstat` (3)

© 2006 Marc Feeley

- Exécution

```
% mkfifo ~/x
% ls -ld /etc/passwd /dev /dev/ttyS0 /dev/modem /dev/hda1 ~/x
drwxr-xr-x 8 root   root  36864 Dec 31 20:48 /dev
brw-rw---- 1 root   disk  3,   1 May  5  1998 /dev/hda1
lrwxrwxrwx 1 root   root       5 Jun  4  2001 /dev/modem -> ttyS0
crw------- 1 root   tty   4,  64 May  5  1998 /dev/ttyS0
-rw-r--r-- 1 root   root     720 Nov 11  2001 /etc/passwd
prw-r----- 1 feeley users      0 Jan 13  2005 /u/feeley/x
% ./a.out
file /etc/passwd is a regular file with size=720
file /dev is a directory with size=36864
file /dev/ttyS0 is a character device with size=0
file /dev/modem is a symbolic link with size=5
file /dev/hda1 is a block device with size=0
file /u/feeley/x is a fifo with size=0
% cat < ~/x &
% cat > ~/x
hello
hello
world
world
<CTRL-D>
%
```

© 2006 Marc Feeley

- Les appels système `chmod` et `chown` permettent de modifier les permissions et le propriétaire et groupe d'appartenance d'un fichier

  ```
  int chmod (char* filename, mode_t mode);
  int chown (char* filename, uid_t u, gid_t g);
  ```

  -1 pour u ou g ne modifie pas ce champ

Routine getpwnam

ⓒ 2006 Marc Feeley

- L'appel système `getpwnam` permet d'obtenir des informations sur un usager à partir de son "username"

```
struct passwd *getpwnam (char* name);
```

- Structure `struct passwd`:

```
struct passwd {
  char  *pw_name;   /* user name */
  char  *pw_passwd; /* user password */
  uid_t pw_uid;     /* user id */
  gid_t pw_gid;     /* group id */
  char  *pw_gecos;  /* real name */
  char  *pw_dir;    /* home directory */
  char  *pw_shell;  /* shell program */
};
```

ⓒ 2006 Marc Feeley

- L'appel système `utime` permet de modifier les dates de dernier accès et de dernière modification

  `int utime (char*` *filename*`, struct utimbuf*` *am*`);`

  am pointe vers une structure contenant les dates de dernier accès et de dernière modification

- Structure `struct utimbuf`:

```
struct utimbuf {
  time_t actime;   /* access time */
  time_t modtime;  /* modification time */
};
```

- `time_t` est un type entier; le nombre de **secondes depuis l'epoch** (minuit, 1 janvier 1970)

Routine `utime` (2)

© 2006 Marc Feeley


```
void info (char *filename, time_t *access, time_t *modif)
{ struct stat s;

  if (lstat (filename, &s) < 0) exit (1);

  *access = s.st_atime;   *modif = s.st_mtime;

  printf ("access = %s", ctime (access));
  printf ("modif  = %s", ctime (modif));
}

int main (int argc, char *argv[])
{ char *filename = argv[1];
  time_t access, modif;
  struct utimbuf t;

  info (filename, &access, &modif);

  t.actime = access-10;   t.modtime = 0;

  if (utime (filename, &t) < 0) exit (1);

  info (filename, &access, &modif);

  return 0;
}
```

ⓒ 2006 Marc Feeley

- Exécution

```
% date
Sun Apr  2 20:41:42 EDT 2006
% echo allo > test
% date
Sun Apr  2 20:42:15 EDT 2006
% cat test
allo
% date
Sun Apr  2 20:42:22 EDT 2006
% ls -l test
-rw-r--r--   1 feeley  feeley  5 Apr  2 20:41 test
% ./a.out test
access = Sun Apr  2 20:42:20 2006
modif  = Sun Apr  2 20:41:55 2006
access = Sun Apr  2 20:42:10 2006
modif  = Wed Dec 31 19:00:00 1969
% ls -l test
-rw-r--r--   1 feeley  feeley  5 Dec 31  1969 test
```

Routines `mkdir` et `rmdir` (1)

© 2006 Marc Feeley

- La création d'un répertoire UNIX se fait avec `mkdir`

  `int mkdir (char*` *path*`, mode_t` *mode*`);`

- Le *mode* indique les permissions d'accès:

  0700 = lecture, écriture, accès par nom (propriétaire)

  0070 = lecture, écriture, accès par nom (groupe)
  0007 = lecture, écriture, accès par nom (autres usagers)

  En *mode* "lecture", on peut obtenir l'ensemble des noms de fichiers dans le répertoire

  En *mode* "accès par nom", on peut seulement accéder à un fichier si on connait son nom (et on a en plus les permissions nécessaires sur ce fichier)

- `rmdir` élimine un répertoire **vide**

  `int rmdir (char*` *path*`);`

© 2006 Marc Feeley

- Exemple

```
/* create directory "test" listable by group and
 * accessible by specific name by all, and file
 * "test/xxx" readable by everyone. */

int fd;
if (mkdir ("test", 0751) < 0)
  perror ("mkdir");
else if ((fd = creat ("test/xxx", 0444)) < 0)
  perror ("creat");
else if (close (fd) < 0)
  perror ("close");
else if (rmdir ("test") < 0)
  perror ("rmdir");

% ./a.out
rmdir: Directory not empty
% ls -la test
total 16
drwxr-x--x  2 feeley feeley   4096 Jan  2 16:35 .
drwx--x--x  7 feeley feeley  12288 Jan  2 16:35 ..
-r--r--r--  1 feeley feeley      0 Jan  2 16:35 xxx
% ./a.out
mkdir: File exists
```

Routines `opendir`, `readdir`, etc (1)

© 2006 Marc Feeley

- `opendir` permet d'énumérer les entrées d'un répertoire

```
DIR* opendir (char* path);
struct dirent* readdir (DIR* dir);
int closedir (DIR* dir);
```

- Structure `struct dirent`:

```
struct dirent {
  ...
  unsigned char d_type;   /* file type */
  char d_name[256];       /* file name */
};
```

© 2006 Marc Feeley

- Exemple : parcours récursif d'un répertoire

```
% ./a.out /home/feeley/test
|-> .
|-> ..
|-> readme            [size=1000]
|-> doc
|    |-> .
|    |-> ..
|    |-> report.doc        [size=10000]
|    |-> report.ps         [size=50000]
|-> test.c            [size=2000]
|-> bin
|    |-> .
|    |-> ..
|    |-> i386
|    |    |-> .
|    |    |-> ..
|    |    |-> a.out        [size=4000]
|    |    |-> find         [size=8000]
|    |-> ppc
|    |    |-> .
|    |    |-> ..
|    |    |-> a.out        [size=9000]
total = 84000 bytes
```

```
int list (char* path, int indent)
{ int i, n = 0;
  char old[PATH_MAX+1];
  getwd (old);
  if (chdir (path) == 0)
    { DIR* dir = opendir (".");
      if (dir != NULL)
        { struct dirent* e;
          while ((e = readdir (dir)) != NULL)
            { struct stat s;
              for (i=0; i<indent; i++) printf ("|     ");
              printf ("|-> %s", e->d_name);
              if (lstat (e->d_name, &s) == 0)
                { if (S_ISDIR(s.st_mode))
                    { printf ("\n");
                      if (strcmp (e->d_name, ".") != 0 &&
                          strcmp (e->d_name, "..") != 0)
                        n += list (e->d_name, indent+1);
                    }
                  else
                    { printf ("\t[size=%ld]\n", s.st_size);
                      n += s.st_size;
                    }
                }
            }
          closedir (dir);
        }
      chdir (old);
    }
  return n;
```

```
int main (int argc, char* argv[])
{ printf ("total = %d bytes\n", list (argv[1], 0));
  return 0;
}
```

Routines `lseek` et `lseek64` (1)

© 2006 Marc Feeley

- Le cannal d'E/S créé lors de l'ouverture d'un fichier mémorise la **position dans le fichier du prochain octet qui sera lu ou écrit**

- Les appels système `lseek` et `lseek64` permettent de connaître la position et la changer

  ```
  off_t lseek (int fd, off_t pos, int rel);
  off64_t lseek64 (int fd, off64_t pos, int rel);
  ```

  `off_t` et `off64_t` sont des types entiers représentant une position

  rel est un code qui indique comment interpréter pos :
  `SEEK_SET` = à partir du début,
  `SEEK_CUR` = à partir de la position présente,
  `SEEK_END` = à partir de la fin

```
#define N 100000

struct dossier { char nom[43], tel[11], nas[10]; };

int tab; /* descripteur de fichier de la table */

void init (void)
{ int i = N;
  struct dossier d;
  d.nom[0] = '\0';
  while (i-- > 0) write (tab, &d, sizeof (d));
}

void lire (struct dossier* d, int i)
{ lseek (tab, i * sizeof(struct dossier), SEEK_SET);
  read (tab, (char*)d, sizeof(struct dossier));
}

void ecrire (struct dossier* d, int i)
{ lseek (tab, i * sizeof(struct dossier), SEEK_SET);
  write (tab, (char*)d, sizeof(struct dossier));
}
```

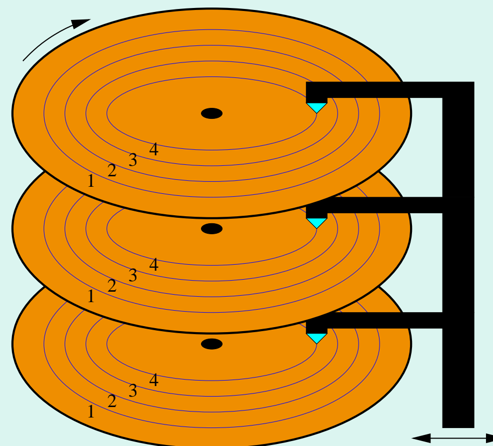
```
int hash (char* str)
{ char* p = str;
  int h = 0;
  while (*p != '\0')
    h = (((h>>8) + *p++) * 331804471) & 0x7fffffff;
  return h;
}

void ajouter (struct dossier* d)
{ struct dossier t;
  int i = hash (d->nom) - 1;
  do { i = (i+1) % N;
       lire (&t, i);
     } while (t.nom[0] != '\0' &&
              strcmp (d->nom, t.nom) != 0);
  ecrire (d, i);
}

void chercher (char* nom, struct dossier* d)
{ int i = hash (nom) - 1;
  do { i = (i+1) % N;
       lire (d, i);
     } while (d->nom[0] != '\0' &&
              strcmp (nom, d->nom) != 0);
}
```

```
int main (int argc, char *argv[])
{
  struct dossier d;

  tab = open ("bd", O_RDWR|O_CREAT, 0644);

  init ();

  strcpy (d.nom, "etienne");
  strcpy (d.tel, "3435766");
  strcpy (d.nas, "111111111");
  ajouter (&d);

  strcpy (d.nom, "marc");
  strcpy (d.tel, "5551212");
  strcpy (d.nas, "222222222");
  ajouter (&d);

  chercher ("etienne", &d);

  printf ("tel = %s\n", d.tel);

  return 0;
}
```

Disque Dur (1)

ⓒ 2006 Marc Feeley

- Mémoire secondaire la plus répandue = **disque dur**
- Dans un enclos hermétique (sans poussière) un certain nombre de **plateaux** recouverts d'une substance magnétique tournent rapidement (p.e. 5400 RPM)
- Un ensemble de têtes de lectures qui peuvent **lire** et **écrire** des informations sur les plateaux se déplacent en unisson ("**seek**") en fonction des commandes envoyées par le contrôleur de disque dur



Disque Dur (2)

ⓒ 2006 Marc Feeley

- Chaque plateau contient un certain nombre de **pistes** ("track"); chaque piste contient un certain nombre de **secteurs**
- L'ensemble des $i^{\text{ème}}$ pistes de tous les plateaux forment un **cylindre**
- La capacité totale du disque est

$$\text{capacité} = N_P * N_T * N_S * OS$$

  où N_P=nombre de pistes par plateau, N_T=nombre de têtes (normalement 2 par plateau), N_S=nombre de secteurs par piste, OS=nombre d'octets par secteur
- Exemple typique: $N_P = 1011$, $N_T = 15$, $N_S = 44$, $OS = 512$, capacité=325MB

Disque Dur (3)

© 2006 Marc Feeley

- L'unité de transfert de donnée est le **secteur** (typiquement de 512 à 4096 octets)
- Chaque secteur a une adresse **"CHS"** qui l'identifie (sur PC: Cylinder=0..$N_P - 1$, Head=0..$N_T - 1$, Sector=1..N_S)
- L'approche **"LBA"** (Logical Block Addressing) est une autre façon de numéroter les secteurs (de 0 à $N - 1$, où N = nombre total de secteurs)

$$\text{adresse LBA} = (C * N_T + H) * N_S + S - 1$$

Disque Dur (4)

ⓒ 2006 Marc Feeley

- Le temps d'accès dépend du temps de positionnement de la tête sur la bonne piste ST (**"seek time"**), la latence additionnelle de rotation du disque RL (**"rotational latency"**), le temps de transfert des secteurs accédés TT

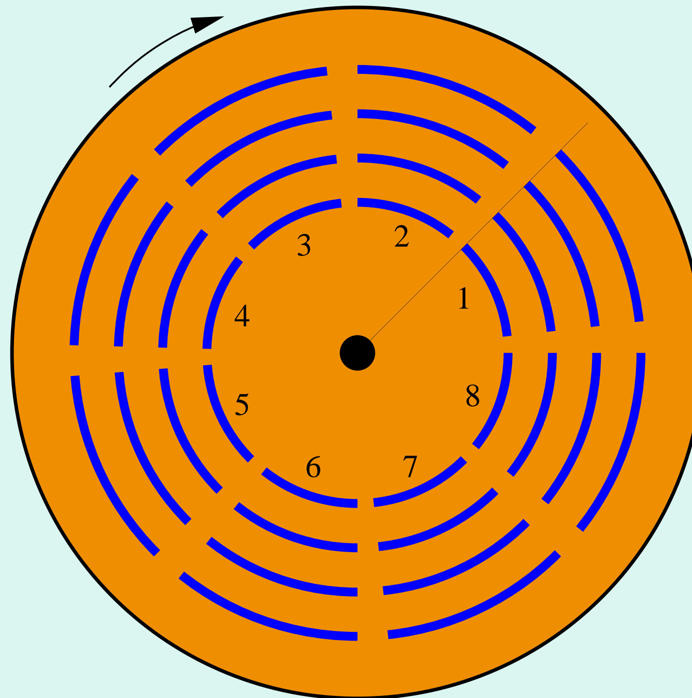
  temps d'accès = $ST + RL + TT$

- Valeurs typiques: ST = 0 .. 40 msecs, RL = 0 .. 11 msecs (pour un disque 5400 RPM)
- ST dépend de la distance de déplacement d (nombre de pistes), l'accélération maximale des têtes, la vitesse maximale des têtes, et le temps de stabilisation
- Un modèle simple: $ST = d * k_1 + k_2$
- Une mesure moyenne de ST = 1/3 du ST maximal

Disque Dur (5)

© 2006 Marc Feeley

- Une organisation simple est de placer le secteur $i+1$ immédiatement après le secteur i

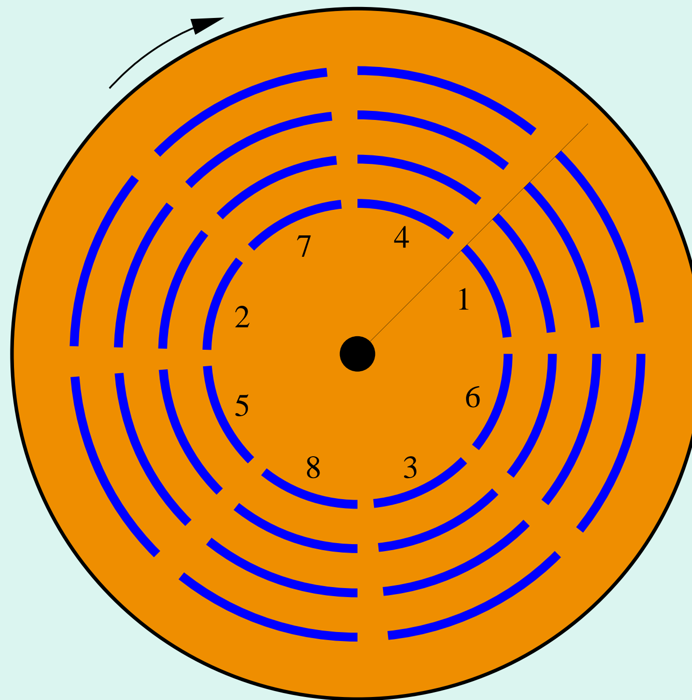


- Dans ce cas, RL est très petit si après le secteur i on accède au secteur $i+1$

Disque Dur (6)

© 2006 Marc Feeley

- Lorsque le temps entre les accès aux secteurs i et $i+1$ n'est pas petit, il peut être avantageux d'**entrelacer les secteurs** pour que l'accès au secteur $i+1$ n'ait pas à attendre une rotation complète du disque



- L'entrelacement optimal dépend de la vitesse du processeur, du SE, de l'application, etc.

ⓒ 2006 Marc Feeley

- Les disques récents n’ont pas une géométrie CHS fixe
- Le nombre de secteurs par piste n’est pas constant, il **augmente à la périphérie du disque** pour que la densité des bits par unité de surface soit approximativement constant
- La relation entre CHS et LBA est plus complexe
- Pour demeurer compatible avec les vieux SE qui supposent une géométrie CHS fixe dans leurs échanges avec le contrôleur de disque, le disque déclare des valeurs N_P, N_T et N_S qui ne correspondent pas à la réalité et **font eux même une traduction de ce CHS logique en un CHS physique** (par exemple $N_P = 16383$, $N_T = 16$ et $N_S = 63$, pour un disque de 8GB)

Ordonnancement des Accès au Disque

© 2006 Marc Feeley

- Dans un contexte **multitasking**, plusieurs processus peuvent être en attente d'un accès au disque

- Puisque le disque est relativement lent il est avantageux d'**ordonnancer les accès au disque**

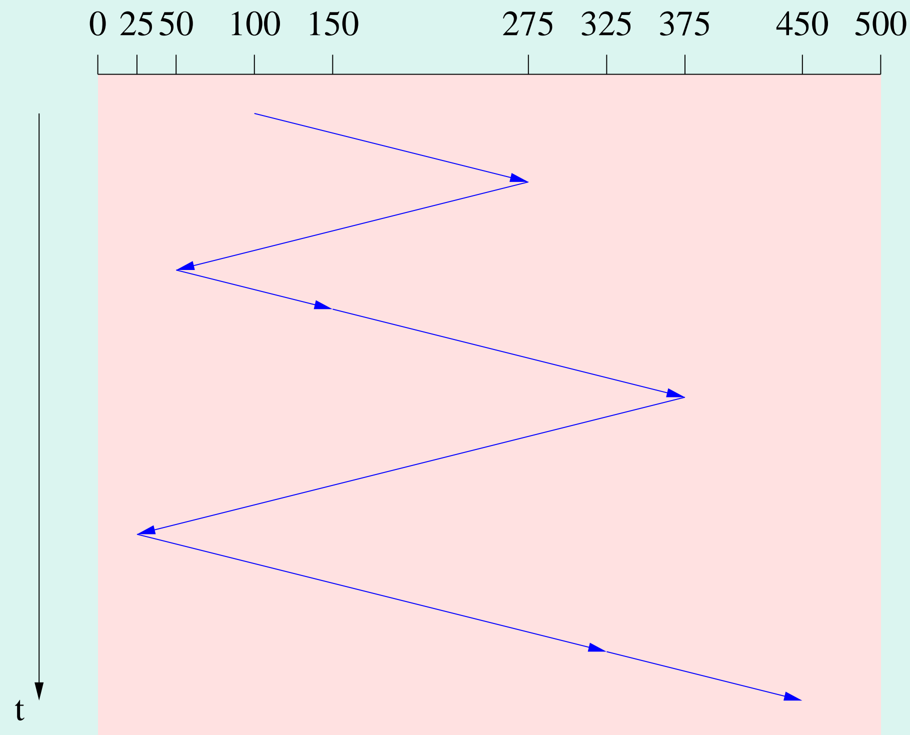
- L'ordonnanceur d'accès au disque peut s'implanter comme un **processus serveur**; les processus clients envoient des requêtes d'accès qui sont mis dans une **file d'attente**

- L'ordonnanceur choisit **l'ordre dans lequel ces requêtes sont traitées** de façon à **maximiser le débit du disque**

Ordonnancement “FCFS”

© 2006 Marc Feeley

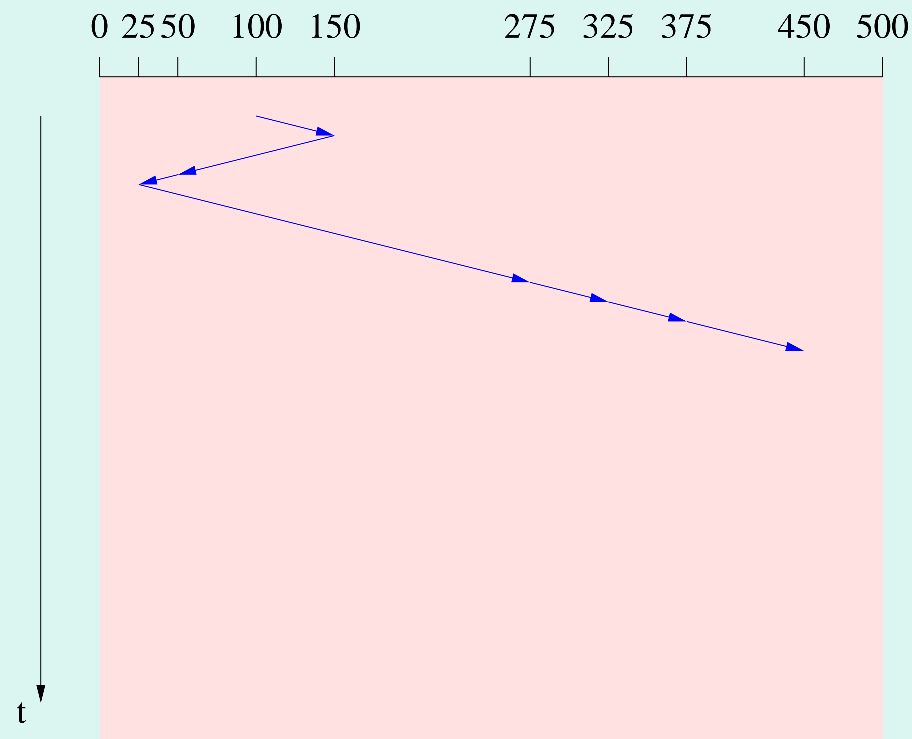
- Une approche simple consiste à traiter les requêtes dans l’ordre d’arrivée; premier arrivé premier servi (“FCFS”)
- Exemple : des requêtes arrivent dans l’ordre de numéro de piste 100, 275, 50, 150, 375, 25, 325, 450



Ordonnancement “SSTF”

© 2006 Marc Feeley

- L’approche SSTF (Shortest Seek Time First) tente de réduire ST en traitant les accès proches en premier

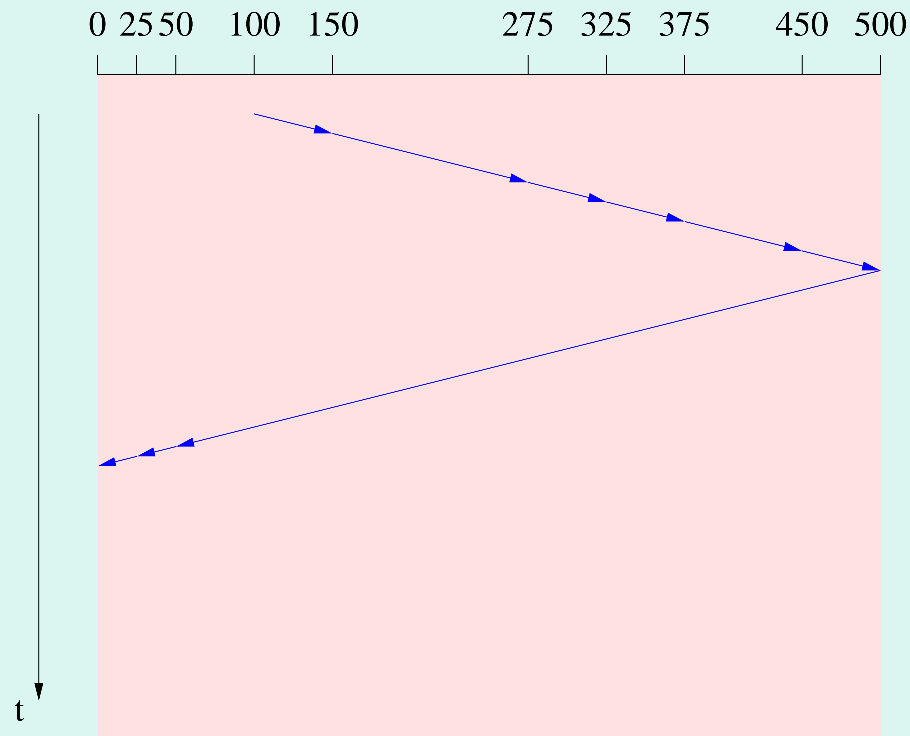


- Il y a un danger de “famine”, si des accès proches arrivent sans arrêt

Ordonnancement “SCAN”

© 2006 Marc Feeley

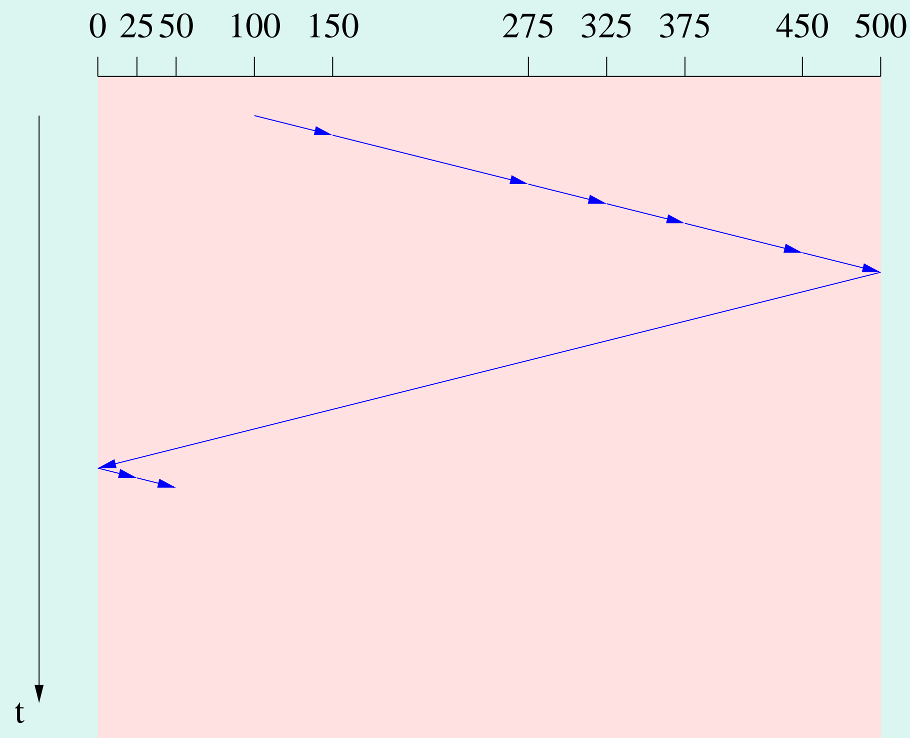
- L’approche SCAN est aussi connu sous le nom d’**algorithme de l’ascenseur**: les têtes changent de direction seulement aux extrémités du disque



Ordonnancement “C-SCAN”

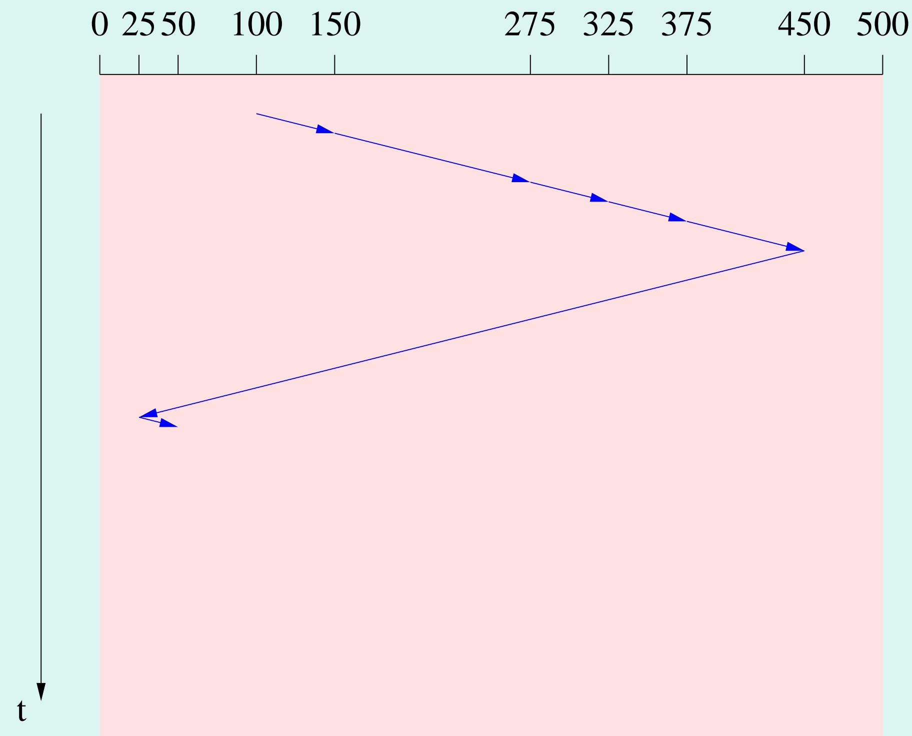
© 2006 Marc Feeley

- L’approche SCAN cyclique est comme SCAN mais les accès se font seulement pendant la phase montante, ce qui donne une distribution plus uniforme du temps d’attente



Ordonnancement “C-LOOK”

- L’approche C-LOOK est comme C-SCAN mais les déplacements de tête sont limités à la piste minimale et maximale



Comparaison

© 2006 Marc Feeley

- SSTF est **facile à implanter** mais il y a danger de famine
- SCAN et C-SCAN ont des bonnes performances lorsque la charge du disque est élevée
- Le patron et fréquence d'accès au disque, l'organisation des données sur disque ("file system"), et le type d'application sont tous à considérer dans le choix d'un algorithme d'ordonnancement
- SSTF et C-LOOK ont des bonnes performances en général
- Ces algorithmes peuvent aussi être modifiés pour tenir compte de la **priorité des processus**