
BeforeROS

Bruno Gas

Jan 24, 2022

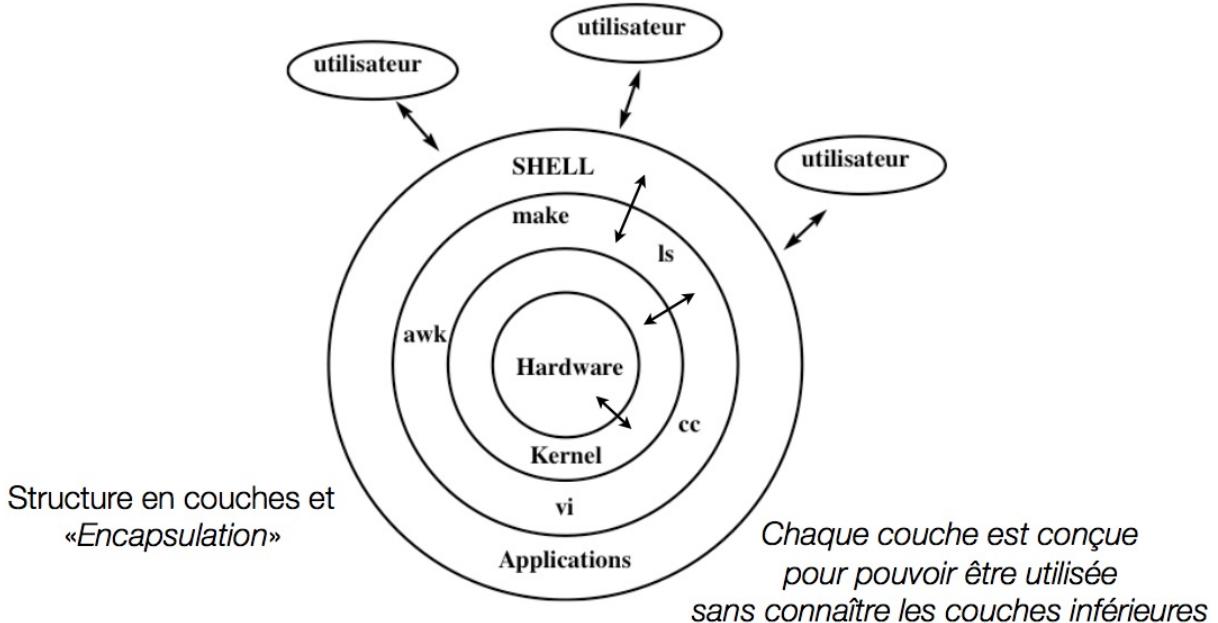
CONTENTS:

1	Shell unix	1
1.1	Commandes Shell	1
1.2	Communication inter-processus	1
1.3	Synchronisation	7
1.4	Multi-threading	7
2	Réseaux	13
2.1	Les modèles OSI et TCP/IP	13
3	Architecture TCP/IP	17
3.1	Sockets et connexion	19
4	Protocoles TCPROS et UDPROS	23
5	Architecture Client-Serveur	25
5.1	Example: navigateur et serveur web	25
5.2	Example: client python et serveur web	27
5.3	Exemple: client navigateur et serveur Python	27
5.4	Exemple: système client/serveur sous Python	29
5.5	Exemple d'architecture client/serveur sous ROS	31
6	Programmation évènementielle	35
6.1	Callback	35
6.2	Programmation asynchrone	38
6.3	Programmation évenementielle	39
6.4	Exemple sous tkinter	40
6.5	Lier une fonction à un évènement	40
6.6	Exemple sous ROS	41
7	Indices and tables	43

SHELL UNIX

1.1 Commandes Shell

Point de vue utilisateur



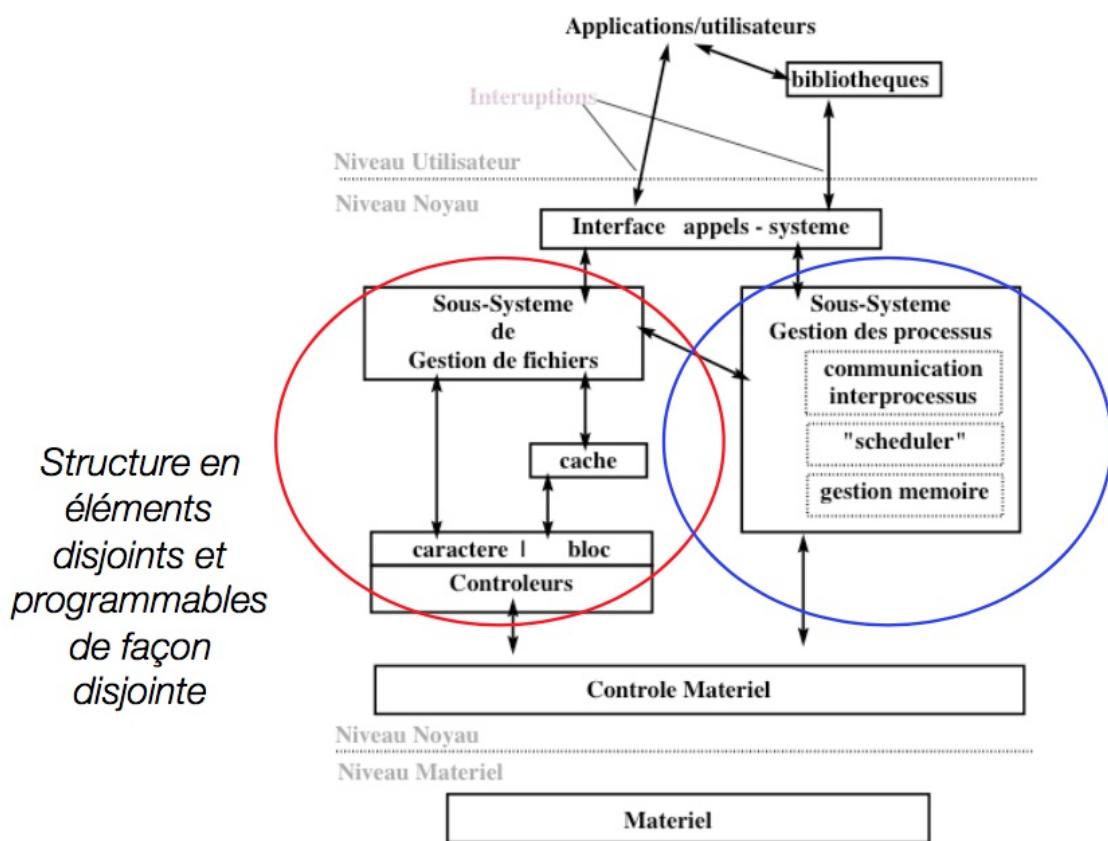
1.1.1 Les commandes de base

1.1.2 Mémento

1.2 Communication inter-processus

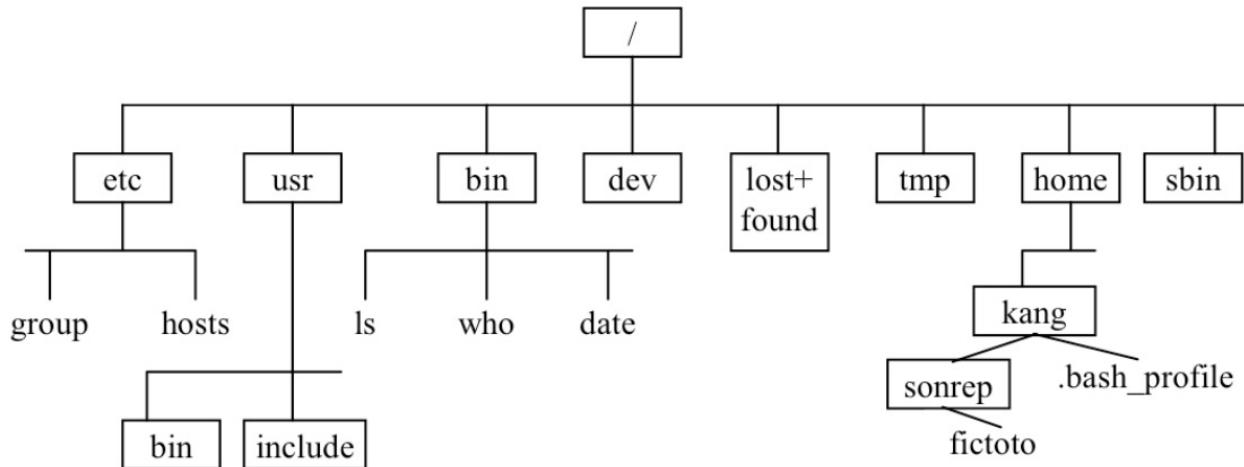
Afin de modulariser les gros programmes on subdivise le plus souvent les tâches à exécuter qui sont alors réalisées par plusieurs programmes s'exécutant en parallèle.

Architecture du noyau



Arborescence

- Une seule racine
- Une structure dynamique
- Une grande puissance d'expression
- Un graphe acyclique



Arborescence sous linux

- / Racine
- /bin Les fichiers exécutables (en binaire)
- /boot Le noyau et les fichiers de démarrage
- /dev Fichiers spéciaux de communication avec les périphériques
- /etc Fichiers de configuration du système
- /etc/rc.d scripts de démarrage du système
- /etc/X11 scripts de configuration du serveur X
- /etc/sysconfig configuration des périphériques
- /etc/cron description des tâches périodiques à effectuer
- /home Racine des répertoires personnels des utilisateurs
- /lib Bibliothèques et modules du noyau
- /mnt Racine des points de montage des systèmes de fichier périphériques
- /opt Applications supplémentaires
- /root Répertoire personnel de l'utilisateur *root*
- /sbin Fichiers exécutables pour l'administration du système
- /tmp Stockage des fichiers temporaires
- /usr Programmes accessibles à tout utilisateur (*structure id. à la racine*)
- /var Données variables liées à la machine (*ex: fichiers d'impression*)
- /proc Pseudo répertoire contenant une «image» du système

Droits d'accès aux fichiers et répertoires

- Pour chaque fichier, 3 classes d'utilisateurs :

U	g	O
user	group	others

- Pour chaque classe d'utilisateur, 3 types d'accès :

r	W	X	-
read	write	execute	none

- Exemples :

```
-rw-r--r-- 1 gas staff 29 13 sep 17:45 test.txt
```

```
lrwxr-xr-x 1 gas staff 26 7 aoû 23:33 www -> /Applications/MAMP/htdocs/
```

Une architecture type pour cela est l'architecture *client-serveur*. Un programme *maître* appelé *serveur* tourne en permanence tandis qu'un ou plusieurs programmes appelés *clients* s'exécutent au fur et à mesure des besoins. Les programmes clients effectuent des *requêtes* auprès du programme serveur qui envoi ses réponses en retour aux programmes *clients*.

Cette architecture est vue en détail dans la section *architecture clients-serveurs* plus loin.

Ce qui nous intéresse ici, c'est que pour mettre en oeuvre ce type d'architecture, les programmes doivent pouvoir communiquer entre eux. Or sous un système unix il existe plusieurs moyens de faire communiquer entre eux des programmes tournant sur la même machine.

- **Les signaux:** c'est le moyen de communication le plus rudimentaire. Un programme émet un signal parmi 256 possibles vers un autre programme qui réagit selon le signal reçu. Certains de ces signaux sont réservés aux interruptions matérielles. Par exemple un *Ctrl C* en provenance du clavier;
- **Les fichiers:** les données à échanger sont enregistrées dans un fichier par le processus émetteur, lisible par le processus receveur;
- **Les tubes ou pipes:** un canal de transmission unidirectionnel est ouvert entre un programme et un autre, permettant au programme émetteur d'envoyer des données vers le programme récepteur. Ce système de communication est lent car il utilise le système de fichiers pour stocker les informations transmises. Il est tout à fait possible de créer un *pipe* entre deux programmes à partir du shell comme on l'a vu plus haut.
- **Les IPC (Inter Process Communication):** permettent d'échanger, de partager des données et de synchroniser des processus. Trois systèmes sont proposés qui sont les files de messages, la mémoire partagée et les sémaphores. La communication IPC utilise la mémoire centrale de l'ordinateur et est donc rapide.
- **Les sockets réseau:** les données à échanger sont envoyées via une interface réseau. Contrairement aux solutions précédentes, les échangeant peuvent se faire entre deux processus s'exécutant sur deux machines différentes.

Modifier les droits

- Pour modifier les droits, il faut être propriétaire du fichier:
- `chmod <mode> <fichier> <CR>`

opérateurs :
+ : ajoute un droit
- : retire un droit
= : droit absolu

opérandes:
u : droits du propriétaire
g : droits du groupe du propriétaire
o : droits des autres utilisateurs
a : droits de tous les utilisateurs

- Exemples :

`chmod u+rw toto.txt`
`chmod o-rwx toto.txt`
`chmod a+r toto.txt`
`chmod go=r toto.txt`

Redirections

- Une commande unix :
 - lit ses données sur le **fichier d'entrée standard** (le clavier)
 - écrit ses résultats sur le **fichier standard de sortie** (l'écran)
 - écrit ses erreurs sur le **fichier standard d'erreur** (l'écran)
- A chaque fichier standard correspond un **descripteur** (ou numéro) :

0 : stdinput 1 : stdoutput 2 : stderror

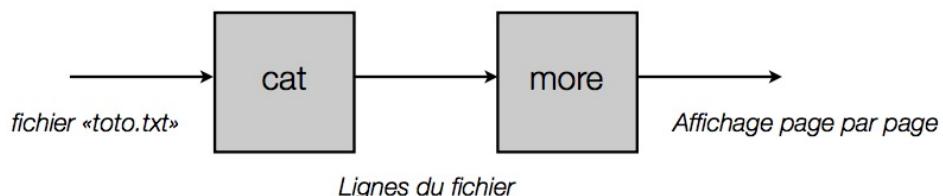
- Les entrées/sorties standard peuvent être redirigées :

```
cat toto.txt > fichier.txt
ls >> fichier.txt
ls 2> messages.txt
mail nom@upmc.fr < fichier.txt
```

Tubes (Pipes)

- Système d'échange d'informations entre deux commandes exécutées en parallèle
- Exemple :

- `cat toto.txt | more <CR>`



- `ls | grep 'toto' <CR>`

1.3 Synchronisation

Dans l'échange et le partage de données entre processus, la synchronisation est très importante. Deux processus qui s'attendent mutuellement conduisent à un blocage de la communication par exemple.

1.4 Multi-threading

pwd (affiche le chemin absolu du répertoire courant)

ls (list, affiche les répertoires et les fichiers du répertoire actif)

ls (affiche seulement les noms)

ls toto* (affiche les fichiers commençant par toto)

ls -l (affiche le format long : types + droits + Nbre de liens +)

cd (change directory)

cp chemin (vers le répertoire dont le chemin absolu est donné)

cd .. (répertoire parent)

cd ~ (répertoire de base)

cd - (répertoire précédent)

cd / (répertoire racine)

cp (copie)

cp rapport*.txt sauvegarde

cp * dossier (copie)

mv (move, renomme et déplace un fichier)

mv source destination

mv * dossier (déplace tous les fichiers du répertoire actif vers le répertoire dossier)

mkdir (créer un répertoire)

mkdir répertoire

rmdir (effacer un répertoire)

rmdir dossier (supprime un répertoire vide)

rm (remove, efface!!!)

rm -R (enlèvement récursif!!!)

rm fichier

rm -i fichier (interactivement, avec demande de confirmation)

rm -f fichier (avec force, sans demande de confirmation)

rm -r fichier (avec récursivité, avec les sous répertoires)

rm -rf dossier (supprime le répertoire et tou son contenu, sans confirmation)

Aide sur les commandes		Imprimer	
man ls	Appel de l'aide pour la commande ls	lpr -Phpv prog.ps	Impression du fichier prog.ps sur hpv
-h ou --help	Demande d'aide pour une commande	lpq	File d'attente sur l'imprimante par défaut
ls --help	Demande d'aide pour la commande ls	lpq -Plp	File d'attente sur l'imprimante lp
		lprm -Plp 367	Effacement du job 367 dans la file d'attente lp
Manipulation des fichiers		a2ps fichier.txt	Impression du fichier texte fichier.txt sur une imprimante Postscript
ls	Liste des fichiers du répertoire		
ls -l	Liste détaillée des fichiers du répertoire	Combinaisons de commandes	
cd	Déplacement dans l'arborescence des fichiers	echo "Bonjour" > fich.txt	Ecriture de "Bonjour" dans le fichier fich.txt
cd /etc	Positionnement sur le répertoire etc	ls > liste.txt	Envoi de la liste des fichiers du répertoire dans liste.txt
pwd	Nom du répertoire courant	ls >> liste.txt	Idem mais c'est copié au bout de liste.txt
cd ..	Positionnement sur le répertoire précédent	l	Envoi de la sortie d'une commande dans l'entrée de la suivante
mkdir prog	Création du répertoire prog	ls wc -l	Nombre de fichiers du répertoire en cours (wc -l compte les lignes affichées par ls)
cd prog	Positionnement dans le répertoire prog		
rmdir prog	Effacement du répertoire prog	Gestion de la session	
cp prog1.c prog2.c	Copie du fichier prog1.c dans prog2.c	passwd	Changement du mot de passe
rm prog1.c	Effacement du fichier prog1.c	who	Utilisateurs connectés
mv prog1.c prog2.c	Renommage ou déplacement du fichier prog1.c en prog2.c	w	Utilisateurs connectés et action en cours
file prog.c	Type du fichier prog.c	whoami	Userid de la session en cours
wc prog.c	Nombre de lignes, de mots, de caractères, du fichier prog.c	id	uid et gid (numéro d'utilisateur et de groupe)
cat prog.c	Liste du contenu du fichier prog.c	h	Historique des commandes
cat a.txt >> b.txt	Copie du fichier a.txt au bout du fichier b.txt	"	Commande précédente
more prog.c	Liste du contenu du fichier prog.c, arrêt en bas d'écran	echo "Bonjour"	Affichage d'une chaîne de caractères
less prog.c	Liste du contenu du fichier prog.c, amélioration de more	echo \$PATH	Affichage du chemin d'accès aux commandes
grep "main" prog.c	Affiche toutes les lignes du fichier prog.c contenant main	printenv	Affichage des variables d'environnement
vi prog.c	édition du fichier prog.c	alias	Liste des alias
emacs prog.c	édition du fichier prog.c	tty	Nom du terminal
chmod a+r fich.htm	Permission de lecture pour tous du fichier fich.htm	export LANG=fr_FR	Diagnostic en français
sort fich.txt	Tri du fichier fich.txt	export LANG=C	Diagnostic en anglais
cmp a.txt b.txt	Compare deux fichiers	locale	Affiche les options locales de langue
diff a.txt b.txt	Affiche les différences entre les deux fichiers	exit	Quitte le shell (ou la session)
touch fich.txt	Crée un fichier vide de ce nom s'il n'existe pas, sinon change la date de dernière modif. du fichier	logout	Idem
		Ctrl-d	Idem (Ctrl = touche contrôle)
Compression et archivage			
tar tzvf prog.tar.gz	Liste (v) de la table (t) des fichiers de l'archive prog.tar.gz	Temps	
tar czf prog.tar.gz	Création (c) d'un fichier archive (f) prog.tar.gz comprimé (z) à partir de tous les fichiers de l'arborescence prog	date	Date et heure
tar xzf prog.tar.gz	Extraction (x) des fichiers de l'archive prog.tar.gz	cal	Calendrier du mois en cours
gzip fich.txt	Compression du fichier fich.txt en fich.txt.gz	cal 6 1994	Calendrier du mois de juin 1994
gunzip fich.txt.gz	Décompression du fichier fich.txt.gz en fich.txt	calendar	Gestion d'agenda
gzip -d fich.txt.gz	Idem		

Calculette		Communication réseau	
dc	Calculateur en notation polonaise inversée	ping auger.c-strasbourg.fr	Test de l'accessibilité de la machine auger.c-strasbourg.fr
echo "10 5 * p" dc	Calcul de 10×5	host auger.c-strasbourg.fr	Demande au serveur DNS l'adresse IP de auger.c-strasbourg.fr
bc	Calculette de bureau à précision quelconque	mail -s bonjour dupond@truc.fr	Envoi d'un mail à dupond@truc.fr ayant pour sujet "bonjour". Terminer le message par '.' en début de ligne
echo "10*5" bc	Calcul de 10×5	mail	Lecture de sa boîte aux lettres par la commande mail BSD
echo "4*a(1)" bc -l	Calcul du nombre Pi ($4 * \arctg(1)$)	mutt	Gestion de boîte aux lettres par mutt
		lynx url	Navigateur Web non graphique
Gestion des processus		wget -r url	Déchargement récursif de pages Web à partir de url
ps auxr	Liste des process en cours d'exécution	slogin auger.c-strasbourg.fr	Connexion sécurisée sur la machine Unix auger.c-strasbourg.fr
ps auximore	Liste de tous les process	scp prog.c auger :/tmp	Copie du fichier prog.c dans /tmp sur la machine auger
top	Suivi de l'activité de la machine	scp -r auger:prog .	Copie récursive des fichiers du répertoire prog de la machine auger dans le répertoire courant de la machine locale
&	Mise en arrière plan d'un processus	ftp ftp.u-strasbg.fr	Déchargement de fichier par ftp, userid anonymous, password votre e-mail, commandes cd, get
prog &	Lancement de prog en arrière plan	ncftp ftp.u-strasbg.fr	ftp amélioré (login automatique sur serveur ftp anonymous)
fg	Mise en avant plan d'un processus stoppé		
jobs	Liste des jobs en arrière plan		
kill %1	Tue le job d'arrière plan [1]		
kill 1492	Tue le processus de PID 1492		
free	Espace mémoire disponible		
Gestion de l'espace disque			
df	Espace occupé/disponible sur les disques montés		
mount	Liste des disques montés		
mount /cdrom	Montage d'un cd-rom		
Compilation			
gcc -o prog prog.c	Compilation C du fichier prog.c, exécutable dans le fichier prog		
gcc -o prog prog.c -lm	Idem avec recherche de fonctions dans la librairie mathématique		
prog	Exécution du programme prog		
gcc prog.c	Compilation C du fichier prog.c, code objet dans le fichier prog.o		
g++ -o hello hello.C	Compilation C++ du fichier hello.C		
ar crv libamoi.a sub1.o	Rangement d'un code objet dans une librairie personnelle		
t crv libamoi.a	Liste des fichiers objet d'une librairie personnelle		
gcc -o prog prog.c -L -lamoi	Compilation C avec recherche de sous-programmes dans la librairie libamoi.a du répertoire courant		
make	Exécution des commandes du fichier Makefile		
ldd prog	Librairies partagées appelées par le programme exécutable prog		
nm prog	Symboles du programme exécutable prog		
gdb prog	Recherche des erreurs du programme exécutable prog		
strace date	Trace des appels systèmes de la commande date		
strip prog	Enlève les symboles du programme exécutable prog		

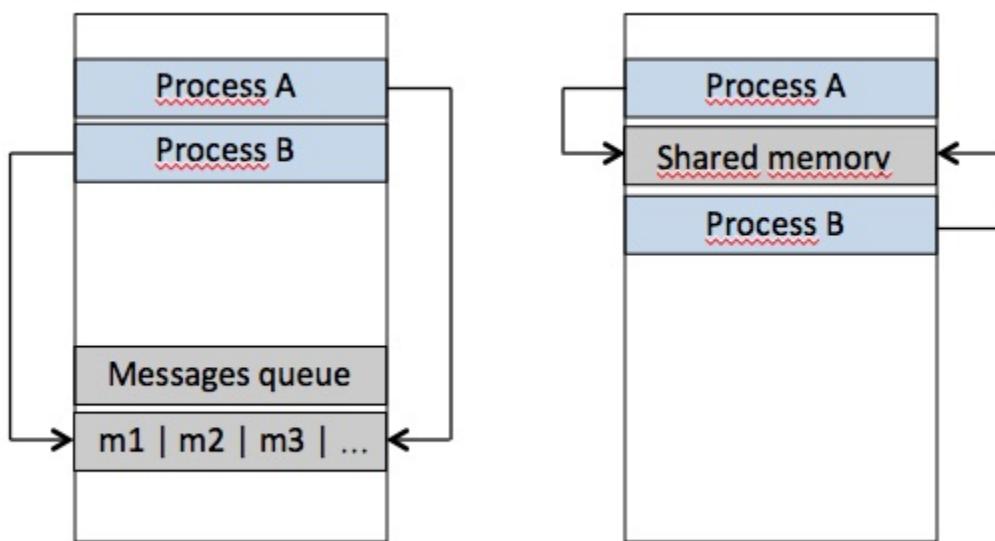


Fig. 1: Files de messages et mémoire partagée

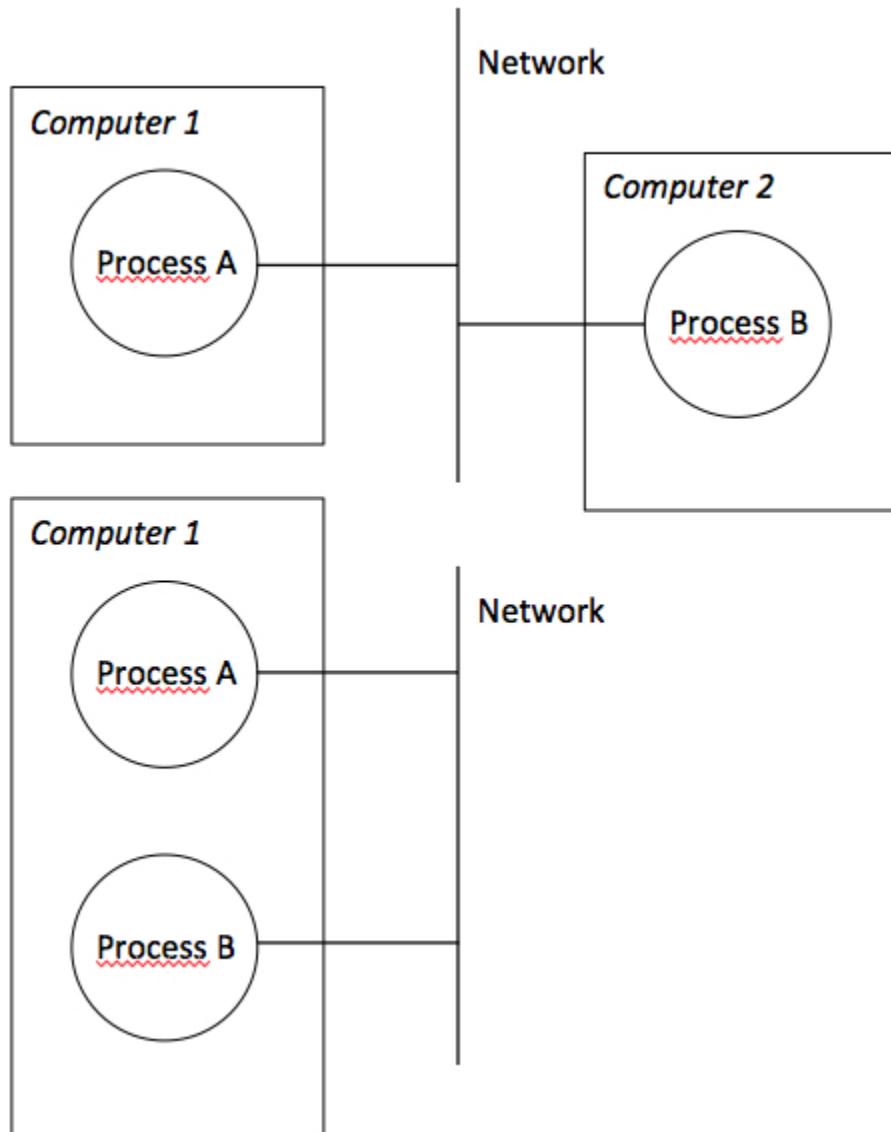


Fig. 2: Sockets réseau: même programmation, pour des processus sur un même ordinateur ou sur deux ordinateurs différents

RÉSEAUX

2.1 Les modèles OSI et TCP/IP

L'internet est un réseau de réseaux. Il obéit à un modèle qu'on appelle le modèle **OSI** (de l'anglais *Open Systems Interconnection*) et qui est une norme de communication, en réseau, de tous les systèmes informatiques. C'est un modèle de communications entre ordinateurs proposé par l'**ISO** (*International Organization for Standardization*) qui décrit les fonctionnalités nécessaires à la communication et l'organisation de ces fonctions.

Il est possible de réaliser sur le même modèle des réseaux uniquement locaux, c'est à dire non connectés à l'internet.



Fig. 1: Deux machines connectées sur un réseau local

Voir d'envisager un réseau de réseaux locaux. les sous-réseaux sont alors interconnectés par un *routeur*

Si le réseau local est connecté à l'internet, le routeur jouera le rôle de passerelle entre le réseau des réseaux et le réseau local.

Le modèle d'organisation OSI que l'on peut voir ci-dessous est un modèle abstrait qui permet d'obtenir une description rigoureuse des réseaux.

C'est le modèle plus simple appelé *modèle TCP/IP* (qui date de 1976) qui va l'emporter sur l'internet. Il comporte 4 couches:

- **Application:** FTP, WWW, telnet, SMTP (emails), tous les programmes qui communiquent sur internet;
- **Transport:** TCP (Transport Control protocol), UDP (User Datagram Protocol), assure la liaison entre deux processus (indépendamment des processus);
- **Réseau:** IP (Internet Protocol), assure le routage des données

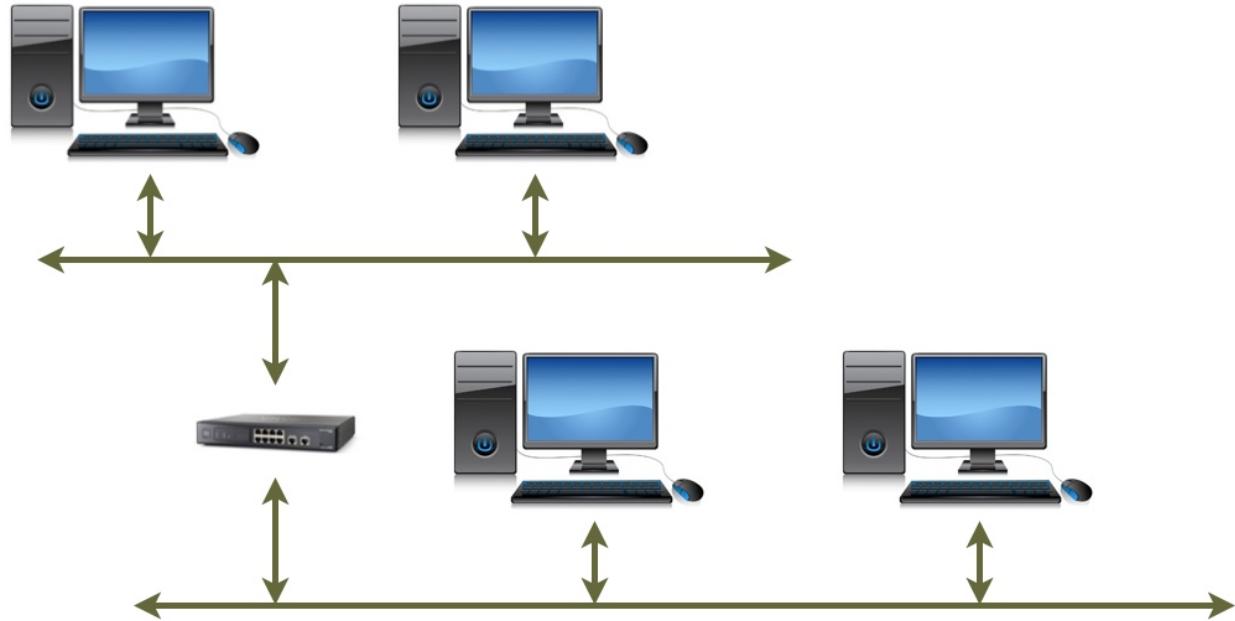


Fig. 2: Un réseau de réseaux locaux

- **Physique:** assure la transmission entre deux sites (RJ45, cables Cat 5-6, fibre, liaisons satellite, etc.)

Les deux protocoles TCP et UDP assurent la liaison entre deux processus. Ce sont des protocoles de transport de bout en bout. Ils se distinguent de la façon suivante:

- TCP: assure le transport fiable des segments, il fonctionne en mode connecté, il gère et corrige les erreurs, garanti le séquencement;
- UDP: Fonctionne en transmission directe des données, sans négociation, sans procédure de connexion. La livraison des datagrammes est non garantie, ni leur ordre d'arrivée, ni leur nombre d'exemplaire.

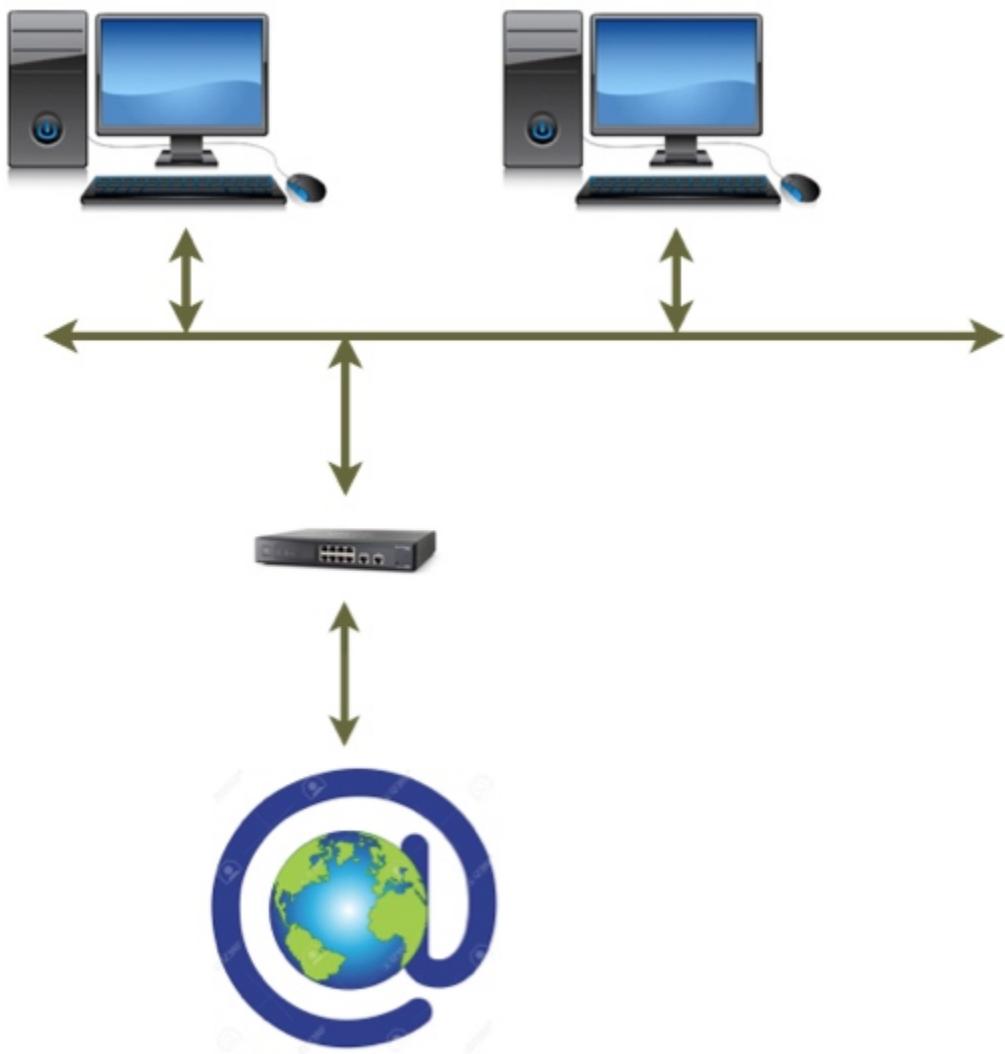


Fig. 3: Réseau local connecté à l'Internet

Modèle OSI			
	Type de données	Couche	Fonction
Couches hautes	Données	Application	Services standards
		Présentation	Codage des informations sous forme standard
		Session	Gestion d'une session de communication entre 2 utilisateurs
	Segments	Transport	Transport de l'information entre deux processus (multiplexage)
Couches matérielles	Paquets	Réseau	Transport entre deux points du réseau (routage)
	Trames	Liaison	Transport sans erreurs entre deux points (trames)
	Bits	Physique	Transport de l'information sous forme binaire

Fig. 4: Le modèle OSI en sept couches

ARCHITECTURE TCP/IP

Du point de vue des données transmises, l'architecture des données en TCP/IP se présente comme suit:

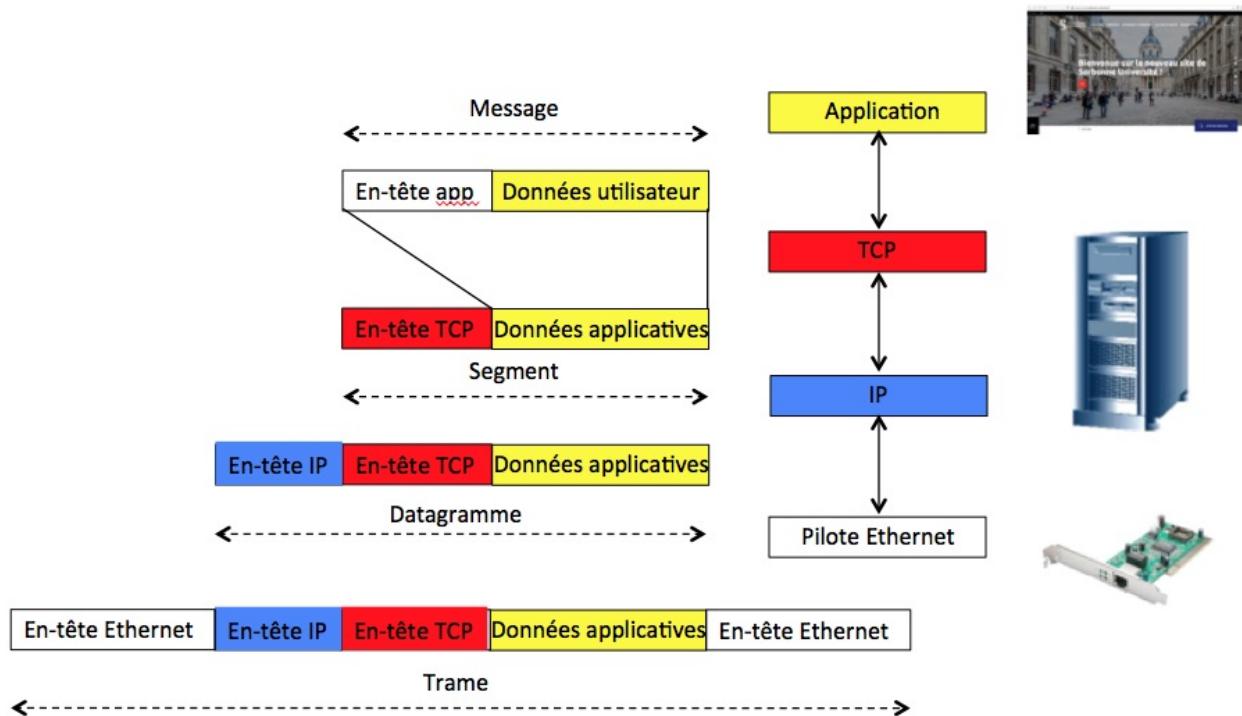


Fig. 1: Architecture de TCP/IP

La couche réseau (IP) assure le routage des messages au travers du réseau. Il fonctionne en mode non connecté, n'offre pas de garantie, mais il est simple et robuste. En cas de défaillance d'un noeud intermédiaire, le mécanisme de routage propose un nouveau chemin opérationnel.

Couches réseau et couche transport assurent le transport de bout en bout des données, donc d'une application à une autre:

Chaque paquet transmis par le protocole IP contient l'adresse du destinataire, l'adresse de l'émetteur. Ces adresses permettent de localiser les machines sur le réseau. Il s'agit d'adresses *logique* dont la définition est gérée par le système d'exploitation des machines. Lorsque l'on connecte un ordinateur au réseau via une carte réseau, la carte dispose d'une adresse *physique* qui elle est immuable et unique. C'est l'adresse *MAC*. Le système d'exploitation associe à l'ordinateur une adresse logique qui celle que définie par le protocole IP c'est l'adresse *IP* de la machine:

Dans la norme IPv4, le dernier numéro représente le numéro de la machine dans le réseau local. Le numéro précédent le numéro du réseau local. Le numéro encore précédent le numéro du réseau de réseaux locaux dans les réseaux de

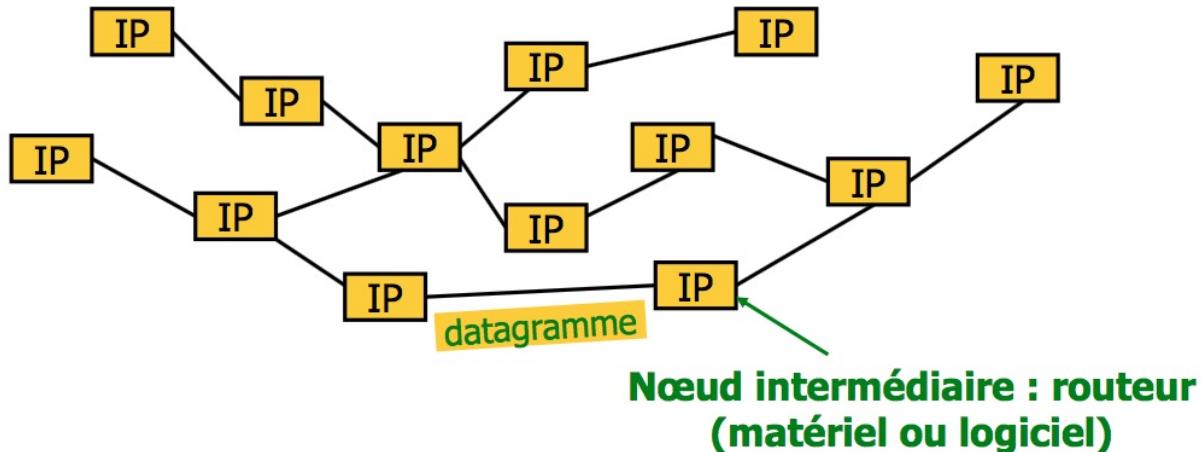


Fig. 2: Routage IP sur Internet

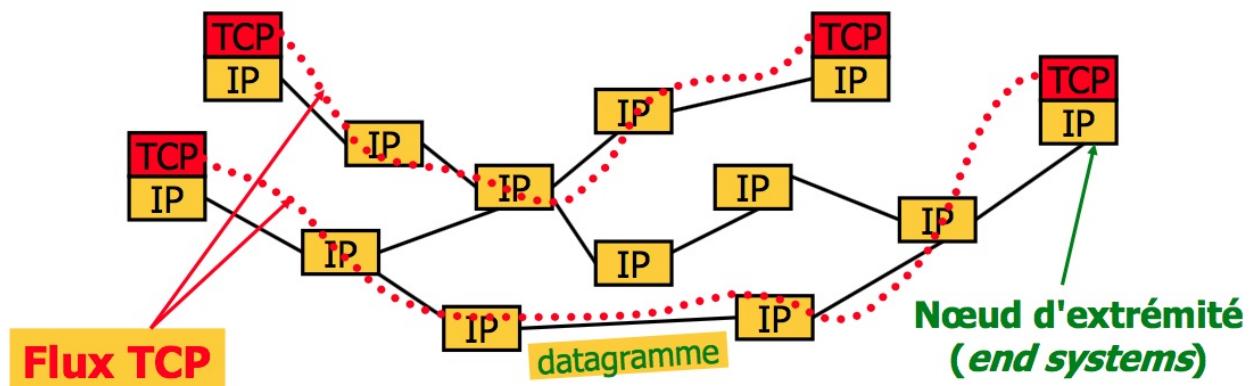


Fig. 3: Transport TCP/IP sur Internet

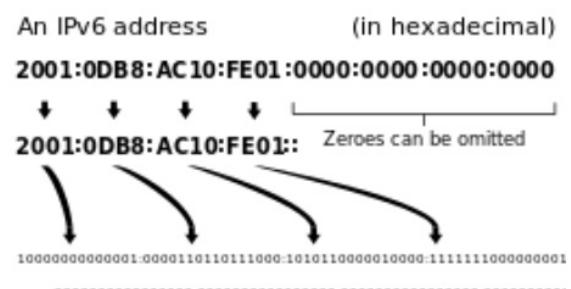
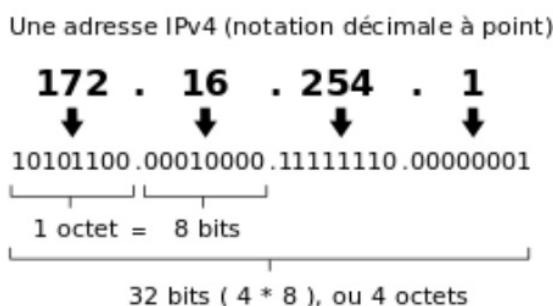


Fig. 4: Définition d'une adresse IP: norme IPV4 et norme IPV6

l'internet Le premier numéro, le numéro du réseau dans le réseau des réseaux.

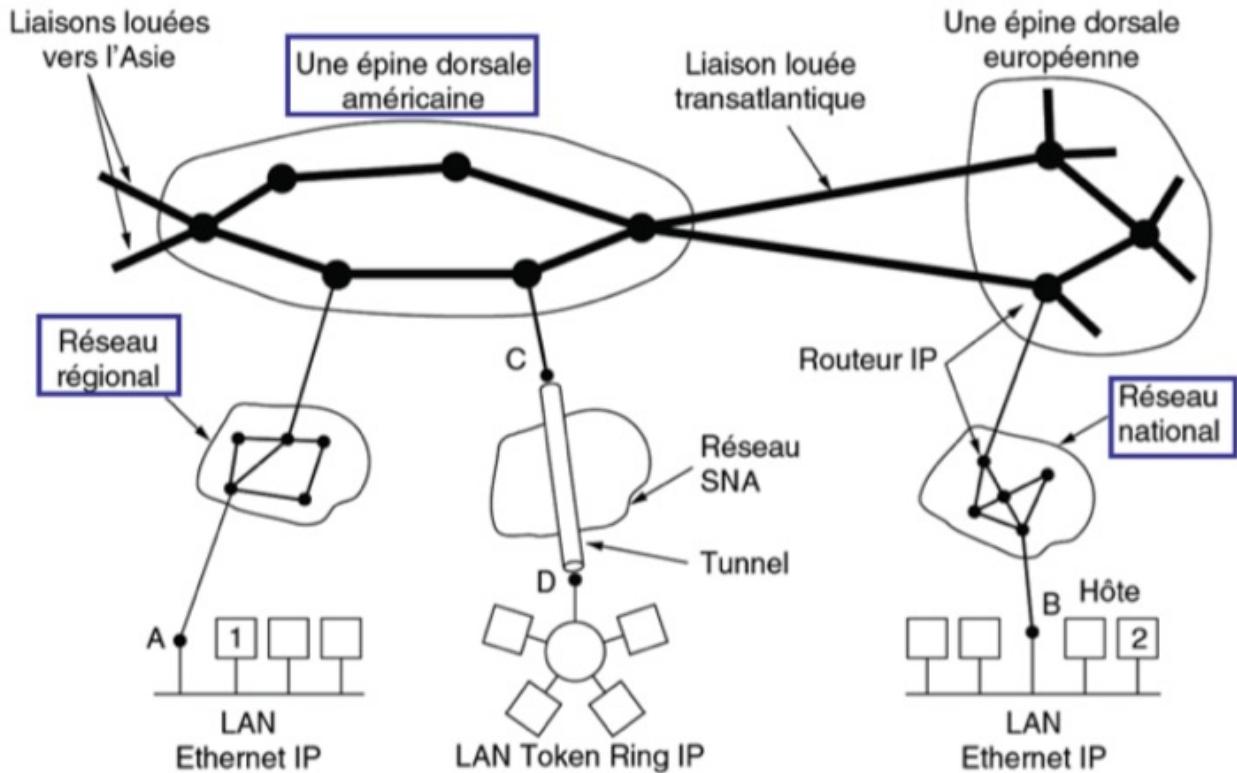


Fig. 5: Structuration des adresses dans le réseau des réseaux (LAN = Local Area Network ou Réseau Local. Token Ring = topologie de réseau Anneau à Jetons).

Pour faire plus simple on associe à un réseau un *nom de domaine* ou *domain name*. Les programmes qui gèrent l'association des numéros IP aux noms de doamine et de machines sont appelés *serveurs de nom de domaine* ou *domain name server* (DNS).

3.1 Sockets et connexion

Lors d'une communication entre deux machines, la connaissance de leurs deux adresse IP ne suffit pas à acheminer les données. En effet les données doivent être véhiculées de programme à programme et non pas seulement de machine à machine. Il faut donc repérer le ou les programmes qui utilisent le réseau sur une même machine.

Une connexion s'établit entre une *socket source* et une *socket destinataire*. C'est un quintuplet:

- Protocole
- Adresse source
- Port source
- Adresse destination
- Port destination

Le port désigne la porte d'entrée/sortie de la machine, étant entendu qu'il existe plusieurs portes d'entrée/sortie pour une même machine. Les processus communiquent sur le réseau en choisissant un *port*. C'est avec ce mécanisme de multiplexage que les applications peuvent donc communiquer entre elles.

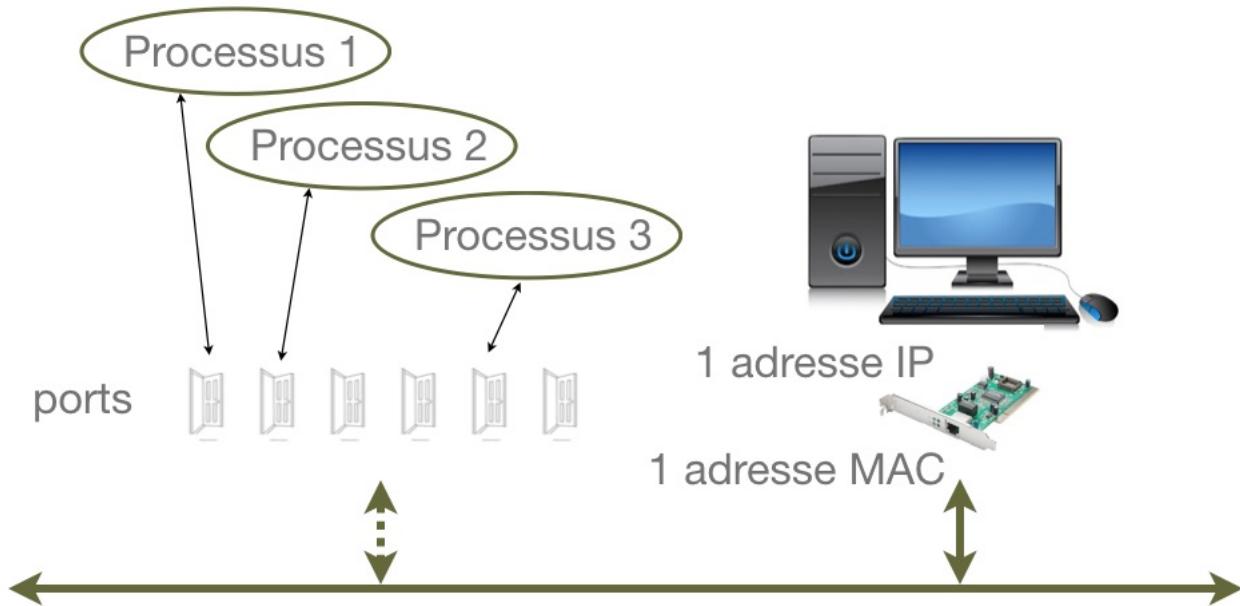


Fig. 6: Les numéros de port permettent de préciser avec quel programme on souhaite dialoguer par le réseau

Les numéros de ports inférieurs à 1024 sont réservés aux services standards. Par exemple:

- 21 FTP
- 22 telnet
- 25 SMTP
- 53 DNS
- 80: HTTP
- 443 HTTPS

Exemple: appel de la page du site *sorbonne-universite.fr* par le port 22:

```
https://www.sorbonne-universite.fr:22
```

Exemple: appel de la page du site *sorbonne-universite.fr* par le port 80:

```
https://www.sorbonne-universite.fr:80
```

La connexion suivante fonctionne, par le port 443 dédié aux accès HTTP sécurisés, fonctionne:

```
https://www.sorbonne-universite.fr:443
```

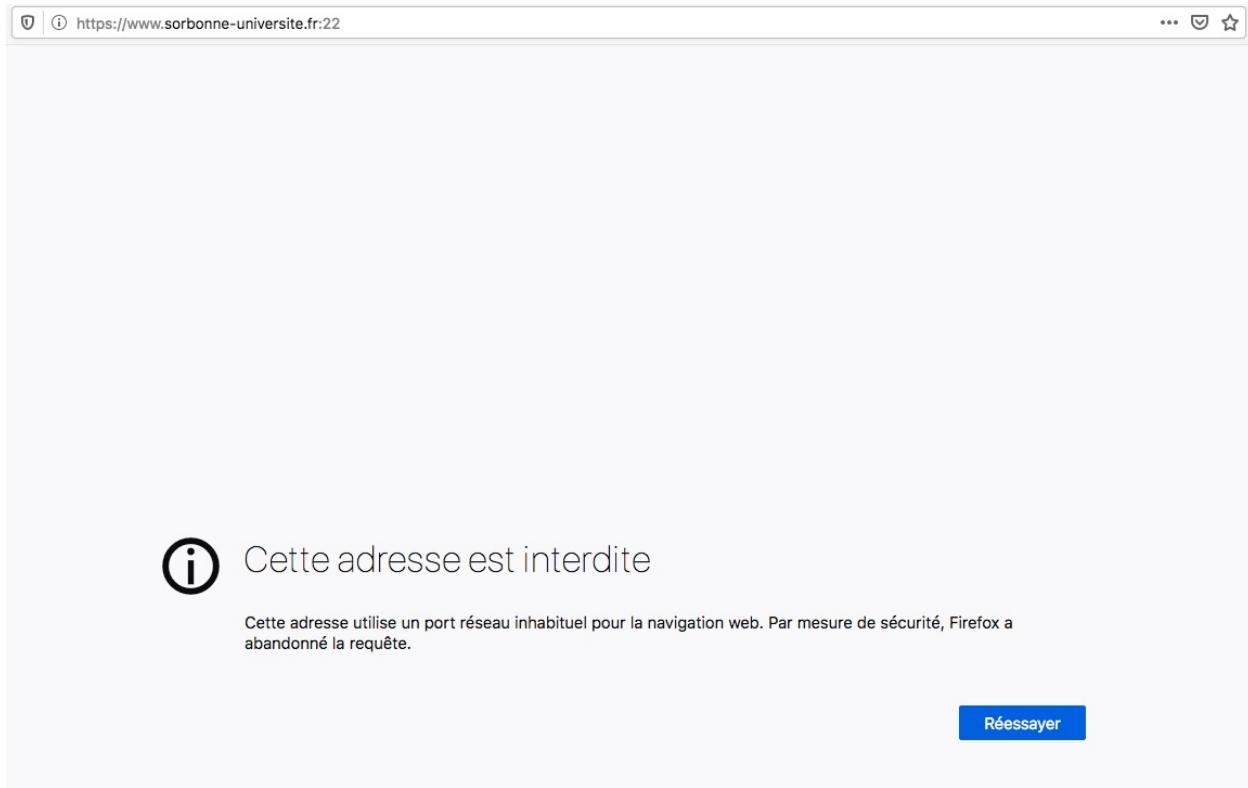


Fig. 7: Tentative de connexion HTTP à l'adresse <http://www.sorbonne-universite.fr> par le port 22

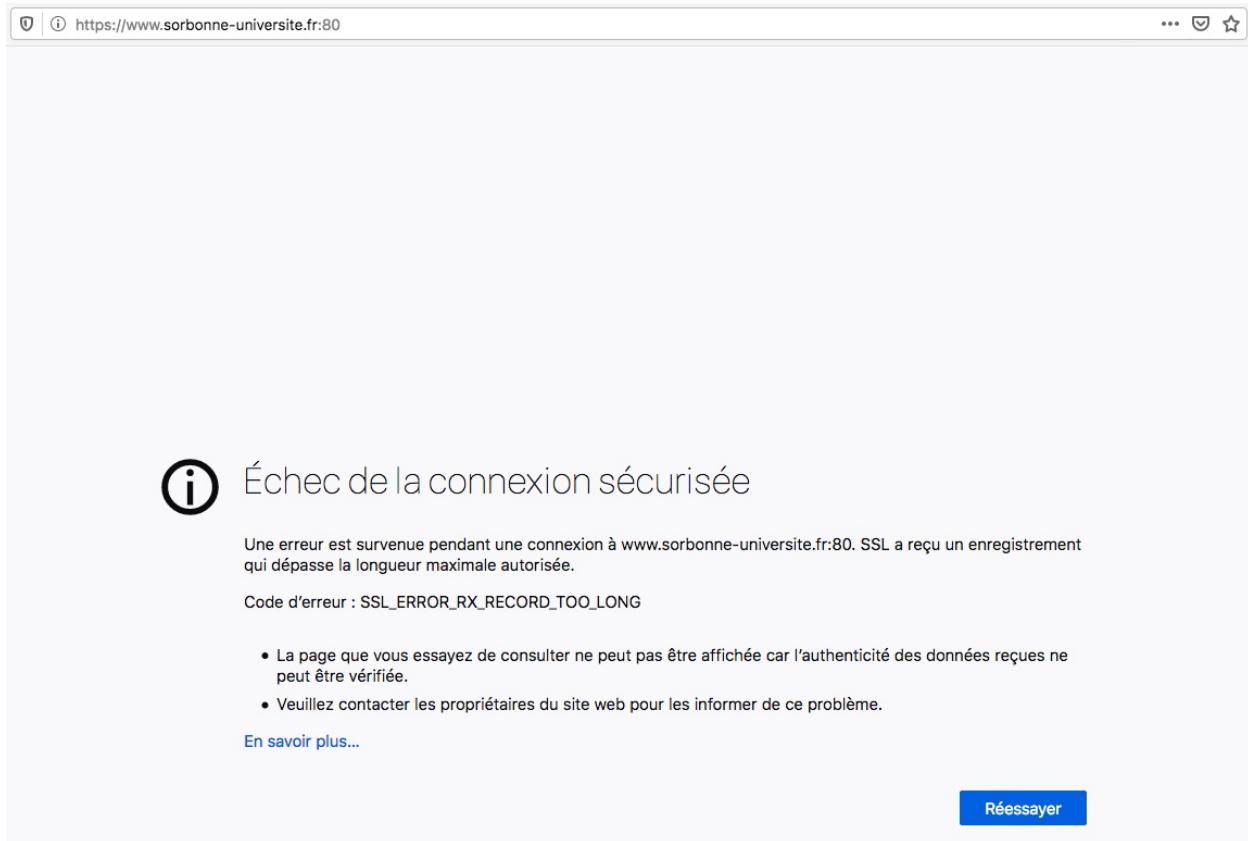
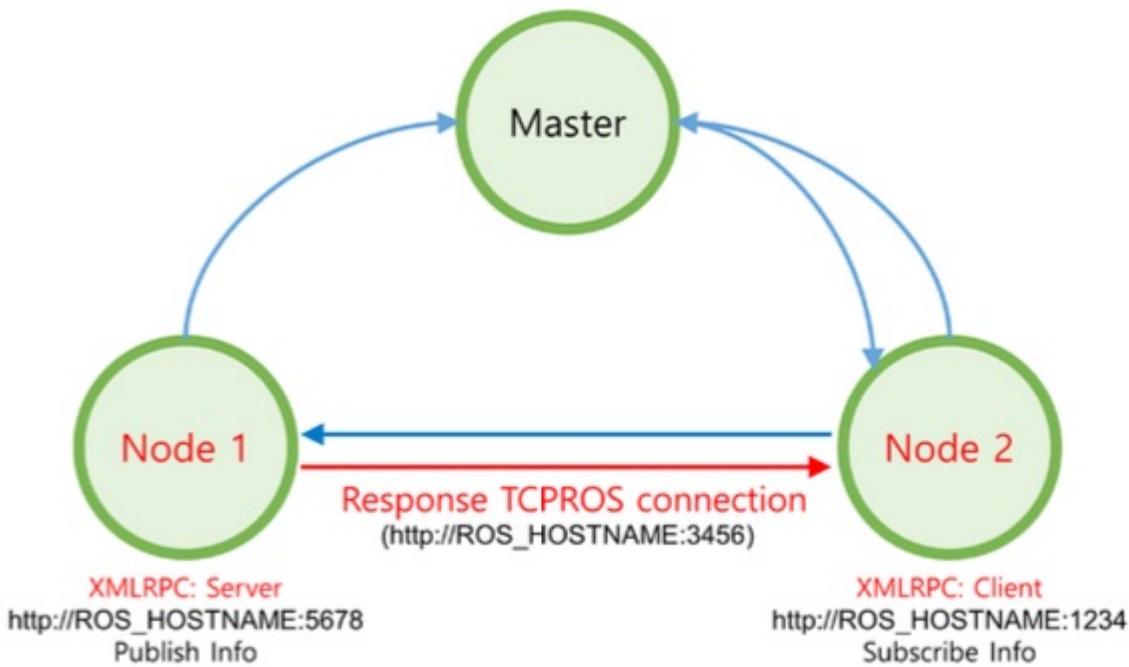


Fig. 8: Tentative de connexion HTTP à l'adresse <https://www.sorbonne-universite.fr> par le port 80

PROTOCOLES TCPROS ET UDPROS

Le système ROS utilise des *noeuds* (nodes) pour désigner la plus petite entité d'exécution. Il faut voir un noeud est un programme exécutable. La communication entre les noeuds s'effectue par le réseau (réseau local) et les protocoles utilisés sont *TCPROS* (le plus souvent) ou *UDPROS*, directement inspirés de TCP et UDP vus plus haut.



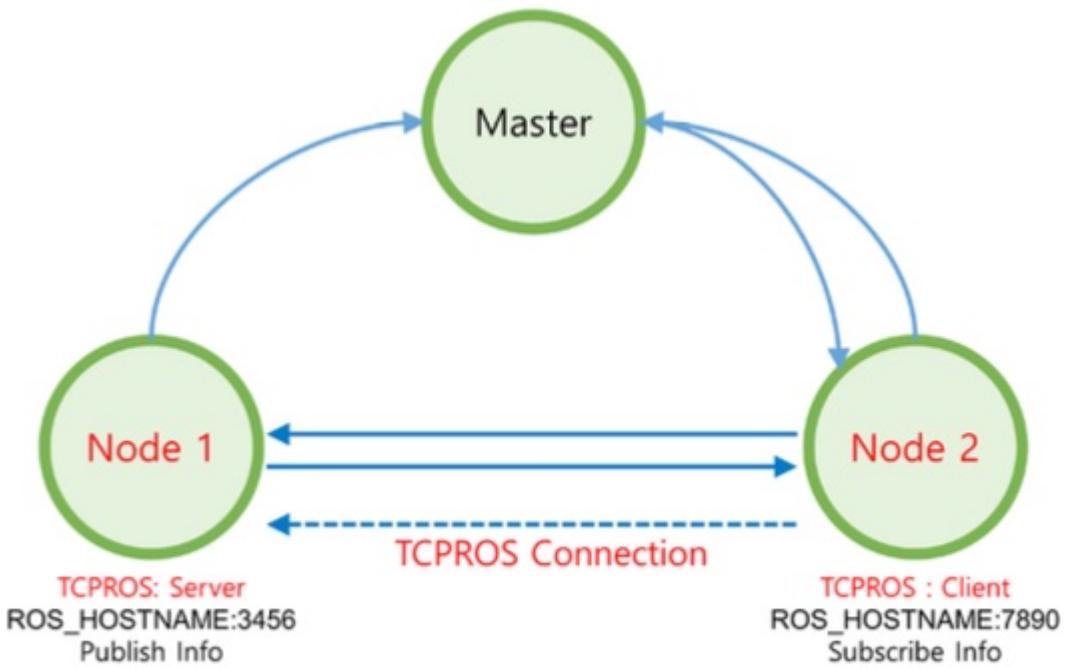


Fig. 1: Tiré de la documentation ROS (p. 57). Notez l'adressage des noeuds sur le réseau local (ports utilisés).

ARCHITECTURE CLIENT-SERVEUR

L'architecture client/serveur est un moyen de répartir des applications sur plusieurs machines. Les services internet sont conçus selon cette architecture.

Le serveur est un programme qui offre un service sur le réseau.

Le client est un programme qui utilise le service offert par un serveur.

Le serveur

- Le serveur accepte des requêtes, les traite et renvoie le résultat au demandeur.
 - Une requête est un appel de fonction, la réponse éventuelle pouvant être synchrone ou asynchrone (le client peut émettre d'autres requêtes sans attendre)
 - Les arguments et les réponses sont énoncés dans un protocole. Le terme serveur s'applique à la machine sur lequel s'exécute le logiciel serveur.
- Pour pouvoir offrir ces services en permanence, le serveur doit être sur un site avec accès permanent
- Il doit s'exécuter en permanence (daemon - suffixe d pour le nom du logiciel ex. ftpd, httpd, ...).

Le client

- le client envoie une requête à un serveur et reçoit sa réponse

5.1 Example: navigateur et serveur web

La pratique la plus répandue aujourd'hui sur internet est la "navigation". Deux programmes sont nécessaires pour cela: le *navigateur* (programme client) et le *site internet consulté* (programme serveur).

Lors de la consultation de la page <http://www.sorbonne-universite.fr>, le navigateur émet une requête vers le serveur des pages web de l'université et reçoit en réponse la page HTML générée par le serveur...

...qu'il peut alors afficher:

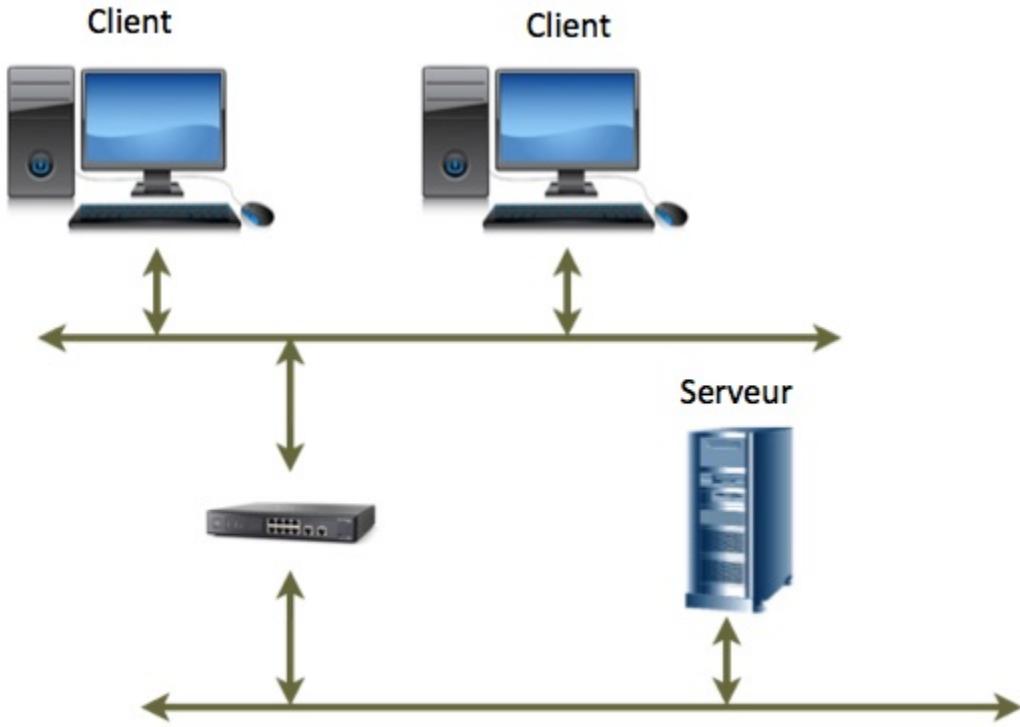


Fig. 1: Clients et serveur sur des réseaux différents

```
1 <!DOCTYPE html>
2 <html lang="fr" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns..>
3 <head>
4   <meta charset="utf-8" />
5   <script>var _paq = _paq || [];(function(){var u(("https:" == document.location.protocol) ? "https://cmsstat.ent.upmc.fr/piwik/" : "htt..>
6   <meta name="title" content="Sorbonne Université" />
7   <link rel="shortcut icon" href="https://www.sorbonne-universite.fr/" />
8   <link rel="canonical" href="https://www.sorbonne-universite.fr/" />
9   <meta name="Generator" content="Drupal 8 (https://www.drupal.org)" />
10  <meta name="MobileOptimized" content="width" />
11  <meta name="Handheldfriendly" content="true" />
12  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
13 <script>function euCookieComplianceLoadScripts(category){if (category === "stats") {var scriptTag = document.createElement("script");scriptTag.type="text/javascript";scriptTag.src="https://piwik.pvt.sorbonne-universite.fr/piwik.php?category=stats&..>
14  <link rel="shortcut icon" href="/themes/custom/boots/favicon.ico" type="image/vnd.microsoft.icon" />
15  <script>window.a2a_config=window.a2a_config||{};a2a_config.callbacks=[];a2a_config.overlays=[];a2a_config.templates=[];a2a_config.callbacks.share= function(data) {
16    jQuery.ajax({
17      type: 'POST',
18      cache: false,
19      url: drupalSettings.statisticsShare.url,
20      data: drupalSettings.statisticsShare.data
21    });
22  };</script>
23 };</script>
24 </head>
25 <title>Sorbonne Université</title>
26 <link rel="stylesheet" media="all" href="/sites/default/files/css/css_hra973uZzzEjCvzDj9ixzlr0CLE_BuAmLsEwxUSR5HE.css?q4vmqc" />
27 <link rel="stylesheet" media="all" href="/sites/default/files/css/css_DVDJ9kc8Co-bUbjXlSSEAV8zhhOA3chXVwKgiBUhpQ.css?q4vmqc" />
28 <!--[if lt IE 8]>
29 <script src="/sites/default/files/js/js_VtafjXmRvoUgAzqzYTA3Wrjkx9wcWhjP0G4ZnnqRamA.js"></script>
30 <![endif]-->
31 <script src="/core/assets/vendor/modernizr/modernizr.min.js?v=3.3.1"></script>
32 </head>
33 <body class="page--front">
34   <div class="page-wrap">
35     <ul class="tabulation">
36       <li>
37         <button id="tab-nav" class="visually-hidden focusable">
38           Skip to main navigation
39         </button>
40       </li>
41       <li>
42         <button id="tab-content" class="visually-hidden focusable">
43           Aller au contenu principal
44         </button>
45       </li>
46     </ul>
47   </div>
48 </body>
49 </html>
```

Fig. 2: Page HTML reçue à la requête du client <http://www.sorbonne-universite.fr>

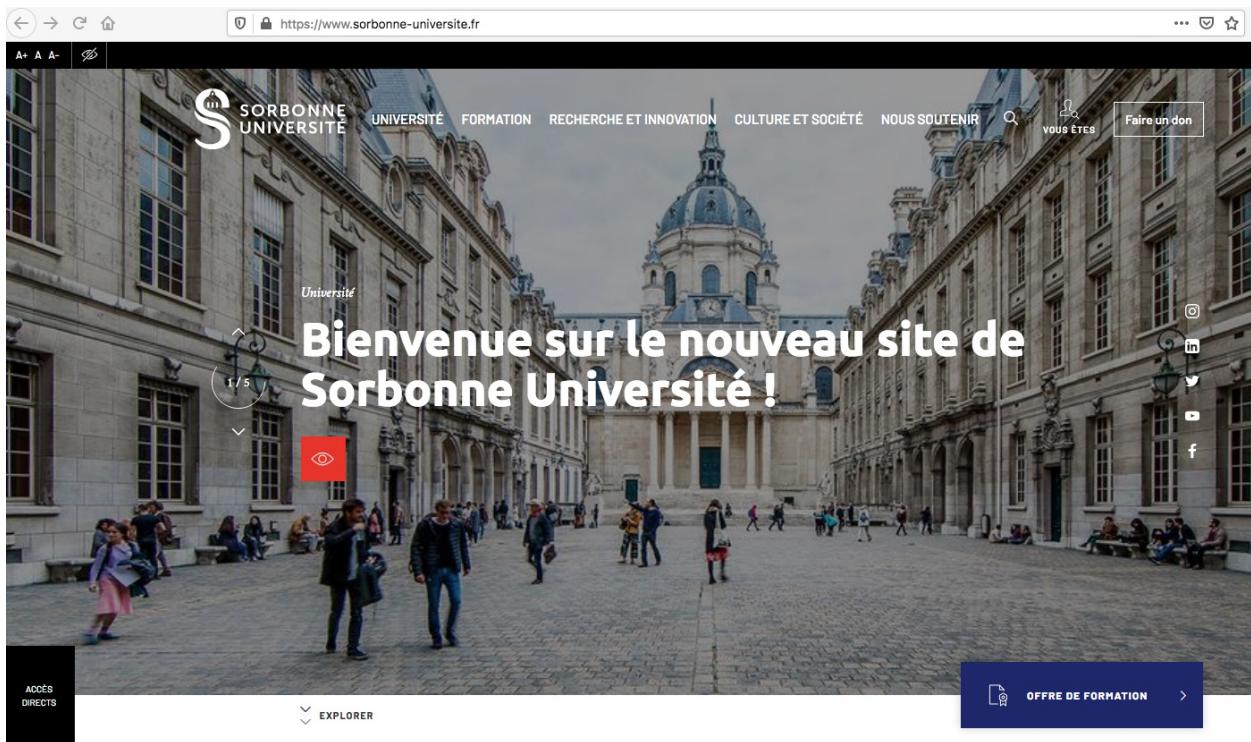


Fig. 3: Page affichée en réponse à la requête <http://www.sorbonne-universite.fr>

5.2 Example: client python et serveur web

Il est possible de créer soi-même en Python un client pour lancer une requête vers le même serveur et obtenir la page HTML:

```
import requests

x = requests.get('https://www.sorbonne-universite.fr')
print( x.text )
```

Résultat obtenu:

Le module *requests* sous Python permet d'envoyer des requêtes à des serveurs HTTP et de récupérer la réponse de ces serveurs. Il est ainsi possible de construire un client en Python.

5.3 Exemple: client navigateur et serveur Python

Créer un serveur en Python utilisant le service HTTP, comme n'importe quel serveur web, est également possible.

Le programme doit dans ce cas se mettre en attente, à l'écoute du port HTTP (80), et répondre aux requêtes.

server.py:

```
import http.server
PORT = 8887
```

(continues on next page)

BeforeROS

```
Entrée [1]: import requests
x = requests.get('https://www.sorbonne-universite.fr')
print(x.text)

<!DOCTYPE html>
<html lang="fr" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/content/ dc: http://purl.org/dc/terms/
foaf: http://xmlns.com/foaf/0.1/ og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema: ht
p://schema.org/ sioc: http://rdfs.org/sioc/ns# sioc: http://rdfs.org/sioc/types# skos: http://www.w3.org/2004/0
2/skos/core# xsd: http://www.w3.org/2001/XMLSchema# ">
<head>
    <meta charset="utf-8" />
    <script>var _paq = _paq || [];(function(){var u=("https:" == document.location.protocol) ? "https://cmsstat.ent.up
mc.fr/piwik/" : "http://cmsstat.ent.upmc.fr/piwik/";_paq.push(["setSiteId", "101"]);_paq.push(["setTrackerUrl",
u+"piwik.php"]);_paq.push(["setDoNotTrack", 1]);_paq.push(["trackPageView"]);_paq.push(["setIgnoreClasses", ["no-tr
acking","colorbox"]]);_paq.push(["enableLinkTracking"]);var d=document,g=d.createElement("script"),s=d.getElementsByTagName("script")[0];g.type="text/javascript";g.async=true;g.src=u+"piwik.js";s.parentNode.insertBefore(g,s);})();</script>
<meta name="title" content="Sorbonne Université" />
<link rel="shortlink" href="https://www.sorbonne-universite.fr/" />
<link rel="canonical" href="https://www.sorbonne-universite.fr/" />
<meta name="Generator" content="Drupal 8 (https://www.drupal.org)" />
<meta name="Mobileoptimized" content="width" />
<meta name="HandheldFriendly" content="true" />
```

Fig. 4: Page affichée en réponse à la requête [http://www.sorbonne-universite.fr](https://www.sorbonne-universite.fr)

(continued from previous page)

```
server_address = ("", PORT)

server = http.server.HTTPServer
handler = http.server.CGIHTTPRequestHandler
handler.cgi_directories = ["/"]
print("Serveur actif sur le port :", PORT)

httpd = server(server_address, handler)
httpd.serve_forever()
```

Dans cet exemple, le serveur exécute les scripts Python placés dans le répertoire mentionné par `cgi_directories`.

L'hôte est local, c'est à dire que le serveur tourne sur la même machine que le client dont l'adresse est *localhost*.

Nous décidons d'affecter le port 8887 au serveur.

Toute requête prenant la forme suivante:

```
http://localhost:8887/mypage.py
```

conduira à faire exécuter par le serveur le programme *mypage.py* dont la réponse sera retournée au client.

Voici un exemple très simple consistant à retourner le code HTML d'une page comportant un titre et un sous titre:

mypage.py:

```
#!/usr/local/bin/python3

print('Content-type: text/html; charset=utf-8\n')

htmlcontent = """
    <!DOCTYPE html>
        <head>
            <title>Mon programme</title>
        </head>
        <body>
```

(continues on next page)

(continued from previous page)

```

<h1>My Page title)</h1>
<h2>My Page (sub title)</h2>

    <p>This is my page (paragraph)...</p>
</body>
</html>
.....
print( htmlcontent )

```

Lancement du serveur:

```

$ > python3 server.py
$ Serveur actif sur le port : 8887

```

résultat obtenu en effectuant la requête par un client navigateur (firefox) à l'adresse:

```
http://localhost:8887/mypage.py
```



Fig. 5: Page récupérée et affichée par le serveur

5.4 Exemple: système client/serveur sous Python

L'utilisation conjointe du serveur ci-dessus avec le client créé sous Python plus haut nous permet d'obtenir le texte HTML généré par le programme *mypage.py*:

```

import requests

x = requests.get('http://localhost:8887/mypage.py')
print( x.text )

```

Plutôt que de générer une page HTML, on peut imaginer le programme *mypage.py* devenant *tourner_a_gauche.py*, dédié à l'exécution de commandes motrices d'un robot pour lui faire engager un virage à gauche.

C'est à peu de choses près ce qu'il se passe sous ROS lorsque des noeuds communiquent entre eux.

```
Entrée [1]: import requests  
  
x = requests.get('http://localhost:8887/mypage.py')  
print( x.text )
```

```
<!DOCTYPE html>  
  <head>  
    <title>Mon programme</title>  
  </head>  
  <body>  
    <h1>My Page (title)</h1>  
    <h2>My Page (sub title)</h2>  
  
    <p>This is my page (paragraph)...</p>  
  </body>  
</html>
```

Fig. 6: Système client/serveur intégralement sous Python

Le protocole utilisé est le protocole *XMLRPC* XML Remote Procedure Call. Le service utilisé est le même que celui des serveurs web, à savoir HTTP (mais sur des ports quelconques choisis, autres que 80). En revanche le code de retour des requêtes n'est plus du code HTML mais du code XML.

XMLRPC

XML-RPC est un protocole RPC (remote procedure call), une spécification simple et un ensemble de codes qui permettent à des processus s'exécutant dans des environnements différents de faire des appels de méthodes à travers un réseau.

XML-RPC permet d'appeler une fonction sur un serveur distant à partir de n'importe quel système (Windows, Mac OS X, GNU/Linux) et avec n'importe quel langage de programmation. Le serveur est lui-même sur n'importe quel système et est programmé dans n'importe quel langage.

Si l'on écrit une nouvelle fonction sous Python générant cette fois du XML (par exemple un email) :

myemail.py

```
#!/usr/local/bin/python3  
  
print( 'Content-type: application/xml; charset=utf-8\n' )  
  
htmlcontent = """<?xml version="1.0" encoding="ISO-8859-1"?>  
<email>  
  <entete>  
    <date type="JJMMAAAA">28102003</date>  
    <heure type="24" local="(GMT+01 :00)">14:01:01</heure>  
    <expediteur>  
      <adresse mail="marcel@ici.fr">Marcel</adresse>  
    </expediteur>  
    <recepteur>  
      <adresse mail="robert@labas.fr">Robert</adresse>  
    </recepteur>  
    <sujet>Hirondelle</sujet>  
  </entete>
```

(continues on next page)

(continued from previous page)

```

<corps>
    <salutation>Salut,</salutation>
    <paragraphe>Pourrais-tu m'indiquer quelle est la vitesse de vol d'une hirondelle
        transportant une noix de coco ?</paragraphe>
    <politesse>A très bientôt,</politesse>
    <signature>Marcel</signature>
</corps>
</email>
"""

print( htmlcontent )

```

Les navigateurs web pouvant décoder des textes en XML, on obtient :

The screenshot shows a Firefox browser window with the URL "localhost:8887/myemail.py". The page content is a plain text representation of an XML document. It starts with an XML declaration, followed by an email element containing an envelope, header, and body sections. The body section includes a salutation, a paragraph asking about a swallows' flight speed, a politeness phrase, and a signature. The XML uses color-coded syntax highlighting for tags and attributes.

```

<!DOCTYPE html>
<html>
<head>
<title>localhost:8887/myemail.py</title>
</head>
<body>
<pre>
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="mailto:marcel@ici.fr"?>
<?xml-stylesheet type="text/xsl" href="mailto:robert@labas.fr"?>
<?xml-stylesheet type="text/xsl" href="mailto:Hirondelle"?>
<?xml-stylesheet type="text/xsl" href="mailto:Marcel"?>
<?xml-stylesheet type="text/xsl" href="mailto:Pourrais-tu m'indiquer quelle est la vitesse de vol d'une hirondelle
        transportant une noix de coco ?"?>
<?xml-stylesheet type="text/xsl" href="mailto:A très bientôt,"?>
<?xml-stylesheet type="text/xsl" href="mailto;Marcel"?>

<email>
    <envelope>
        <date type="JMMMAAAA">28102003</date>
        <heure type="24" local="(GMT+01:00)">14:01:01</heure>
    </envelope>
    <header>
        <expediteur>
            <adresse mail="marcel@ici.fr">Marcel</adresse>
        </expediteur>
        <recepteur>
            <adresse mail="robert@labas.fr">Robert</adresse>
        </recepteur>
        <sujet>Hirondelle</sujet>
    </header>
    <body>
        <corps>
            <salutation>Salut,</salutation>
            <paragraphe>Pourrais-tu m'indiquer quelle est la vitesse de vol d'une hirondelle transportant une noix de coco ?</paragraphe>
            <politesse>A très bientôt,</politesse>
            <signature>Marcel</signature>
        </corps>
    </body>
</email>
</pre>
</body>
</html>

```

Fig. 7: Rendu par un client navigateur du type *firefox*

5.5 Exemple d'architecture client/serveur sous ROS

L'architecture client/serveur sous ROS s'appuie sur la notion de service, le service étant le dispositif permettant d'envoyer des données à un client exclusivement sur requête de ce dernier.

Voir et tester l'exemple simple de création et d'utilisation d'un service par un client donné dans la documentation ROS: [Writing-A-Simple-Py-Service-And-Client](#)

```
Entrée [1]: import requests

x = requests.get('http://localhost:8887/mymail.py')
print( x.text )

<?xml version="1.0" encoding="ISO-8859-1"?>
<email>
    <entete>
        <date type="JJMMAAAA">28102003</date>
        <heure type="24" local="(GMT+01 :00)">14:01:01</heure>
        <expediteur>
            <adresse mail="marcel@ici.fr">Marcel</adresse>
        </expediteur>
        <recepteur>
            <adresse mail="robert@labas.fr">Robert</adresse>
        </recepteur>
        <sujet>Hirondelle</sujet>
    </entete>
    <corps>
        <salutation>Salut,</salutation>
        <paragraphe>Pourrais-tu m'indiquer quelle est la vitesse de vol d'une hirondelle
            transportant une noix de coco ?</paragraphe>
        <politesse>A très bientôt,</politesse>
        <signature>Marcel</signature>
    </corps>
</email>
```

Fig. 8: Rendu par le client Python écrit plus haut

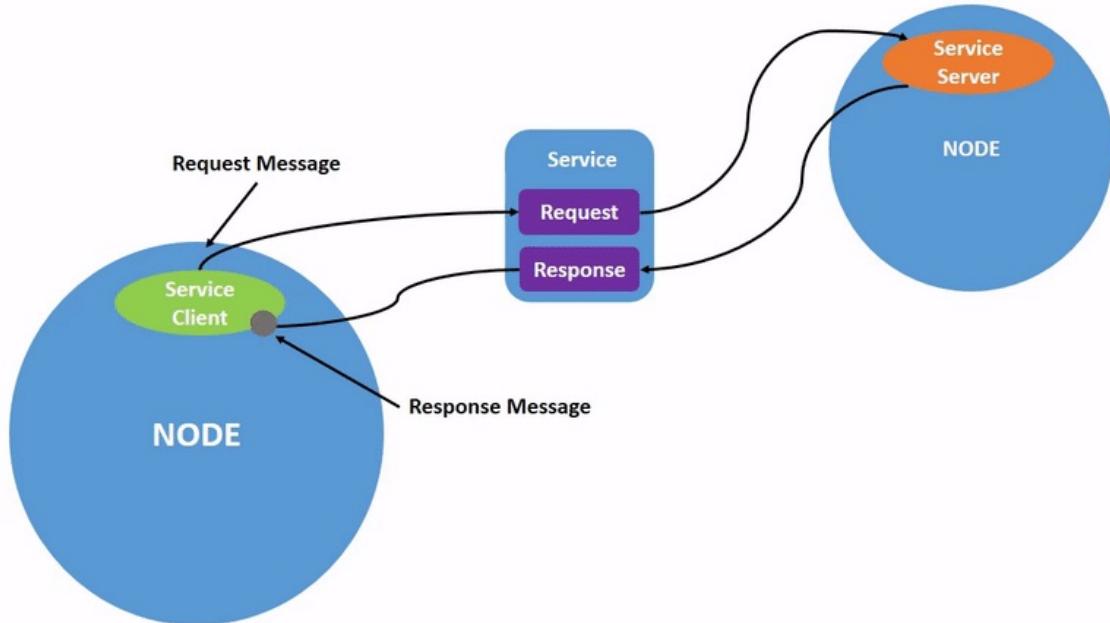


Fig. 9: Architecture client/serveur sous ROS

There can be many service clients using the same service. But there can only be one service server for a service.

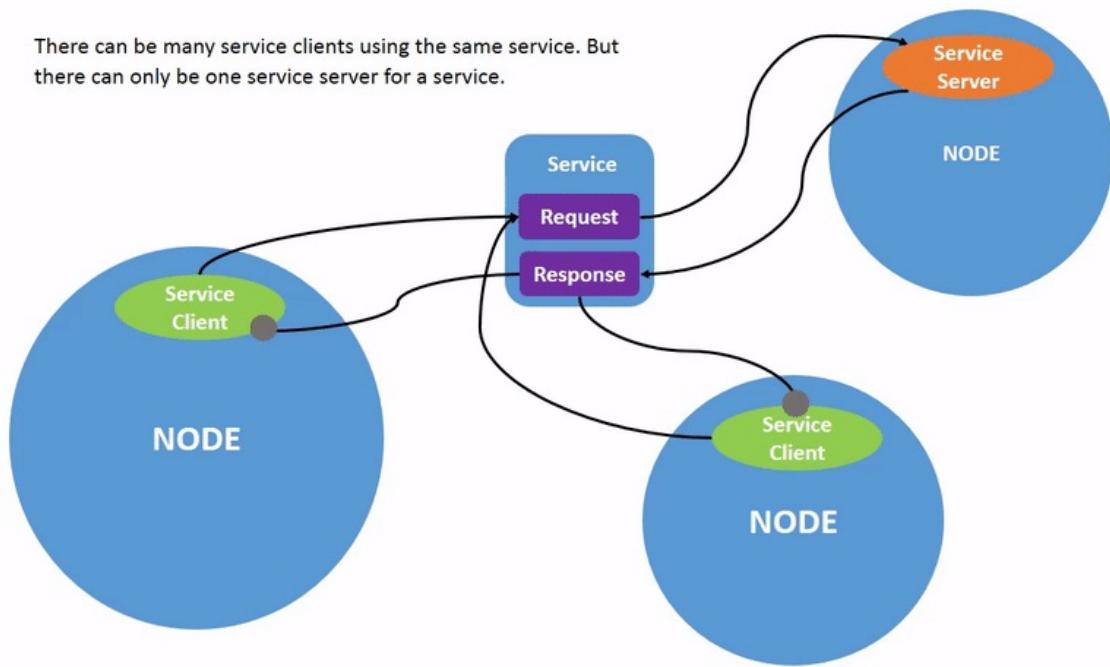


Fig. 10: Architecture client/serveur sous ROS: plusieurs clients

PROGRAMMATION ÉVÈNEMENTIELLE

6.1 Callback

On parle de *callback function* (fonction de rappel) quand on passe une fonction en paramètre d'une autre fonction.

Alors qu'une fonction normale est appelée directement, la fonction de rappel est simplement définie au départ et n'est appelée et exécutée que lorsqu'un certain événement se produit.

Les fonctions *callback* sont très souvent utilisées dans des bibliothèques et des frameworks tels que les applications écrites en JavaScript jQuery, Angular ou node.js, mais également dans d'autres langages comme C, php, java et python.

Dans l'exemple ci-dessous, deux fonctions callback sont définies, `power2()` et `power3()`, et une fonction *caller* (`compute()`):

```
def power2(val):
    print( 'resultat=' + str(val**2) )

def power3(val):
    print( 'resultat=' + str(val*val*val))

# caller
def compute(func, val):
    func(val)

def main():
    compute(power2, 2)
    compute(power3, 2)

main()
```

Résultat:

```
resultat=4
resultat=8
```

Quelle différence avec le programme suivant ?

```
def main2():
    power2(2)
    power3(2)

main2()
```

Aucune:

```
resultat=4
resultat=8
```

Il est vrai qu'il s'agit simplement d'appeler les fonctions via une autre fonction, mais l'idée clé des fonctions *callback* est qu'elles sont considérées comme des **objets**. Il devient possible de choisir quelle fonction appeler selon l'état du système.

Dans l'exemple ci-dessous l'action est changée selon la réponse de l'utilisateur (exemple de fonction d'appel à 2 arguments):

```
def add(x, y):
    print( 'resultat=' + str(x + y) )

def mul(x, y):
    print( 'resultat=' + str(x * y) )

def show(func, x, y):
    func(x, y)

def main():
    x = str(input('Enter an operator '))
    if x == '+':
        show(add, 2, 3)
    elif x == '*':
        show(mul, 2, 3)
    else:
        print( 'unknown operator <%s>' % (x))

main()
```

6.1.1 1er intérêt

La fonction appelante (*caller*) peut appeler la fonction *callback* au sein d'un programme plus complexe dans lequel elle apparaît comme un objet pouvant être utilisé autant de fois que nécessaire. Le traitement exécuté se trouve ainsi "factorisé" ce qui favorise la modularité et l'extensibilité du programme.

Dans l'exemple ci-dessous les callbacks définis sont appelés 2 fois dans le *caller*:

```
import random

def print3numbers( gennumber ):
    """
    La fonction de génération est un paramètre du programme:
    on l'appelle autant de fois que nécessaire...
    """
    x1 = gennumber()
    x2 = gennumber()
    x3 = gennumber()
    print( "numbers are %f, %f and %f" % (x1, x2, x3))

def genOneNumber():
```

(continues on next page)

(continued from previous page)

```

return 23

def genNumberOver5000():
    return 5321

def main():
    print3numbers(genOneNumber)
    print3numbers(genNumberOver5000)
    """
    La fonction callback appartient ici à une librairie externe:
    """
    print3numbers(random.random)

main()

```

Résultat:

```

numbers are 23.000000, 23.000000 and 23.000000
numbers are 5321.000000, 5321.000000 and 5321.000000
numbers are 0.923829, 0.325309 and 0.985599

```

6.1.2 2ème intérêt

La fonction appelante (*caller*) peut appeler les fonctions callback avec les valeurs de paramètres qu'elle souhaite. Ces valeurs peuvent donc ne pas être connues du code initial appelant le *caller*.

Ci-dessous, le paramètre est le nom. C'est la fonction appelante qui gère le paramètre par un appel à l'utilisateur, pas le code initiateur de l'appel:

```

def greeting(name):
    print("Hello %s!" % name)

def processUserInput(callback):
    name = str(input('Enter your name: '))
    callback(name)

def main():
    """
    le paramètre name n'est pas connu ici:
    """
    processUserInput(greeting)

```

Résultat:

```

Enter your name: Gerard
Hello Gerard!

```

6.2 Programmation asynchrone

En programmation *synchronous* (le mode le plus conventionnel), les instructions du programme sont exécutées les unes après les autres, et surtout, aucune commande ne peut s'exécuter sans que la précédente ne soit terminée.

Avec la venue des systèmes multi-processeurs la programmation parallèle s'est imposée avec deux implémentations possibles: le *multi-processing* pour lequel plusieurs processus s'exécutent en parallèle et le *multi-threading* pour lequel plusieurs fonctions s'exécutent en parallèle au sein d'un même processus.

Le modèle de **programmation asynchrone** est un troisième procédé par lequel des commandes peuvent être exécutées avant que les précédentes ne soient terminées.

Synchronous model



Threaded model



Asynchronous model



Task 1

Task 2

Task 3

Fig. 1: Modèles de programmation

On rencontre ce modèle de programmation dans le domaine des interfaces, de la programmation web ou encore de la programmation de processus dont on ne sait à l'avance lorsqu'ils prendront fin. Le framework ROS et la programmation robotique en sont un exemple.

La programmation asynchrone permet d'éviter à un programme d'attendre sans rien faire qu'un évènement se produise. Il s'agit donc d'optimiser le programme de sorte qu'il puisse réaliser des tâches pendant les temps d'attente.

Par exemple lorsqu'il s'agit d'un serveur dans une architecture client/serveur, le serveur exécutera des tâches pour d'autres clients lorsqu'il est en attente de la fin d'exécution d'une tâche pour un premier client.

Quand surviennent les temps d'attente? Réponse: principalement pendant les opérations d'entrée/sortie. Une entrée/sortie (IO), au sens large, est une tâche pendant laquelle le programme attend un résultat qui vient de l'extérieur. L'exemple le plus simple est celui d'une entrée à partir du clavier. Lorsqu'il est sollicité, l'utilisateur peut mettre

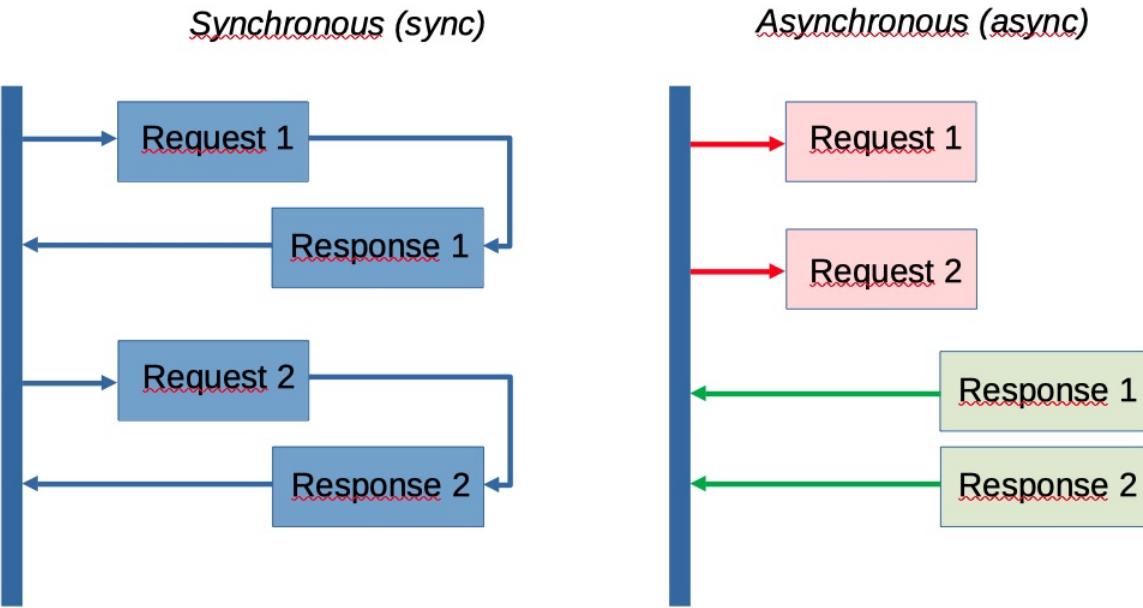


Fig. 2: Programmation synchrone/asynchrone

plusieurs secondes avant de répondre. Pendant ce temps là, en programmation synchrone, le processus est bloqué en attente. En programmation asynchrone il peut poursuivre son exécution pour réaliser d'autres tâches.

Une opération de communication via le réseau entre deux processus (deux noeuds d'un réseau ROS par exemple) est une opération d'entrée/sortie. Ainsi la programmation asynchrone est très présente dans les applications réseau, en particulier dans les architectures clients/serveur.

En python la librairie privilégiée pour réaliser des programmes asynchrones est la librairie *asyncio*, fondée principalement sur la notion de *coroutines*. Les *coroutines* sont une généralisation des fonctions. Tandis que les fonctions s'exécutent en un seul tenant du début à la fin, les coroutines s'exécutent par morceaux, au fur et à mesure qu'elles sont appelées (forte utilisation de *yield* en Python). Entre deux exécutions elles peuvent également recevoir des données susceptibles de modifier le cours de leur exécution ultérieure.

6.3 Programmation événementielle

Selon le paradigme de la *programmation séquentielle*, les instructions se déroulent toujours selon le même ordre prévu à l'avance. C'est le programme qui a le contrôle: l'utilisateur fait ce que le programme lui demande.

Un programme écrit selon le paradigme *évenementiel* régit à de évènements provenants de l'utilisateur ou du système. C'est l'utilisateur qui a le contrôle du programme.

Les interfaces graphiques fonctionnent selon le principe de la programmation événementielle. Mais La programmation événementielle est une technique très générale qui ne se limite pas aux interfaces graphiques.

Dans le principe, une boucle infinie prenant la forme suivante est programmée:

```
while true:
    event = getEvent()
    processEvent(event)
```

- Une boucle infinie “en arrière-plan” permet de détecter un événement (*getEvent()*) et de le traiter (*processEvent()*).

- Le déroulement des instructions dépend de l'ordre des événements déclenchés par l'utilisateur.

La programmation événementielle repose sur l'exécution de fonctions lorsqu'un événement se produit sur élément cible. On parle *d'abonnement* de la fonction à l'élément pour l'événement. L'abonnement d'une fonction à un événement est réalisé par la « mise sur écoute » d'un élément : on parle *d'event listener*.

6.4 Exemple sous tkinter

Le module *tkinter* permet de générer des interfaces graphiques sous Python. La programmation avec *tkinter* est événementielle.

L'exemple suivant ouvre une fenêtre vide:

```
import tkinter as tk

app = tk.Tk()
app.mainloop()

print("Le programme est terminé.")
```

Une fenêtre vide apparaît à l'écran. Lorsque l'on clique sur quitter, le message *Le programme est terminé.* apparaît sur la console.

La méthode *mainloop()* contient la boucle infinie avec laquelle on ne peut intéragir que de deux manières:

- En définissant une fonction (*callback*) que tkinter devra appeler lorsqu'un événement donné se produira. Exemple: lorsque la souris se déplace, appelle la fonction qui va afficher la position de la souris. On lie un callback à un événement avec la méthode *bind*.
- En créant un compte à rebours (*timer*) qui exécutera une fonction (*callback*) après un temps donné. Exemple: dans 30 secondes, appelle la fonction qui vérifie si j'ai un nouvel email.

A chaque tour de boucle, voici les opérations qui sont effectuées:

- Pour chaque événement détecté depuis le dernier tour de boucle (comme l'appui d'une touche de clavier, le déplacement de la souris, ...) tkinter exécute tout callback lié à cet événement.
- Si le temps qui s'est écoulé est supérieur au temps du compte à rebours, le callback est exécuté.

6.5 Lier une fonction à un évènement

Pour que tout cela fonctionne, il faut lier les évènement à des fonctions qui sont executées lorsqu'ils surviennent. Sous tkinter, c'est le rôle de la méthode **bind(event, callback)**.

Le premier argument est un évènement codé sous la forme d'une chaîne de caractères qui respecte un formatage particulier

le deuxième argument est le *callback*, c'est à dire la fonction qui sera exécutée lors de la survenue de l'évènement.

```
import tkinter as tk

def on_double_click(event):
    print("Position de la souris:", event.x, event.y)

app = tk.Tk()
```

(continues on next page)

(continued from previous page)

```
app.bind("<Double-Button-1>", on_double_click)
app.mainloop()

print( "Le programme est terminé.")
```

Exécution du programme:

```
$ > python3 graph.py
Position de la souris: 41 73
Position de la souris: 119 123
Position de la souris: 161 90
Le programme est terminé.
```

Une bonne méthode de programmation consiste à s'attacher à ce que les fonctions callback soient le plus simple possible et d'exécution rapide.

6.6 Exemple sous ROS

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
import numpy as np

from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

# We publish a Twist message to mybot input topic
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
message = Twist()
message.linear.x = 0.0
message.angular.z = 0.0

def read_laserscan_callback(msg):
    distance = np.mean(msg.ranges[240:300])
    global pub, message, flag

    if distance < 0.5 and distance > 0.25:
        print("WARNING: the robot is approaching a wall")

    if distance < 0.25:
        print("STOP : emergency stop triggered")
        pub.publish(message)

if __name__ == '__main__':
    try:
        rospy.init_node('emergency_stop', anonymous=True,
                        log_level=rospy.INFO, disable_signals=False)
```

(continues on next page)

(continued from previous page)

```
# We subscribe to turtle1 output Pose topic
sub = rospy.Subscriber('/mybot/scan', LaserScan, read_laserscan_callback)

print("emergency_stop node")
print("-----")
print("This node will send a null Twist message when facing a wall")

rospy.spin()
# while not rospy.is_shutdown():

except rospy.ROSInterruptException:
    pass
```

**CHAPTER
SEVEN**

INDICES AND TABLES

- genindex
- modindex