Réseau

Le 04 mars 2018 , SVN-ID 451

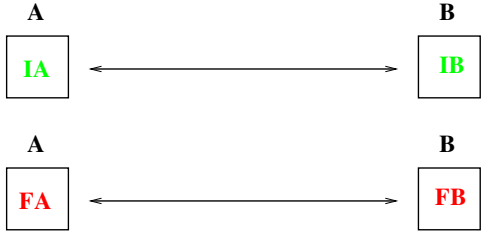
Contents

1 Protocole

1.1 Qu'est ce un protocole

fonction

- un ensemble d'entités dans un état
- les entités peuvent s'échanger des messages
- ensemble de règles et de conventions fixant les émissions de messages pour faire passer ces entités dans un autre état.



exemple 1: FTP

- Etats IA,IB: les SF des machines M1 et M2.
- Etats FA,FB: le SF de M1 contient le fichier Y qui est une copie du fichier X du SF de M2.

exemple 2: Robot d'appel

- Etats IA,IB: Paul et Marie ne sont pas en ligne.
- Etat FA,FB: Paul et Marie sont en ligne et Paul sait que c'est Marie et Marie sait que c'est Paul.

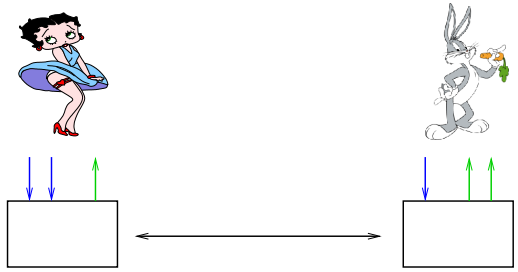
rôle Définir ces règles de manière très précise:

- implanter sur des supports de communication différents
- implanter dans des langages différents
- implanter par des personnes différentes

$\Longrightarrow$ que ça fonctionne
(EX: IE, firefox, chrome, ...)

1.2 Éléments d'un protocole

1.2.1 Service



Définition Les services sont les points d'entrées et de sorties du protocole. Un utilisateur interagit avec un protocole via ses services.

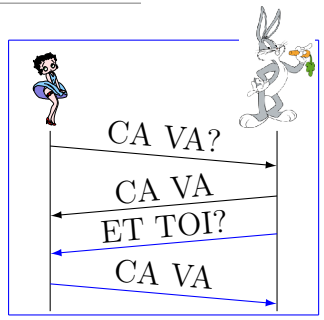
Exemple: FTP

- envoi d'un fichier
- récupération d'un fichier

1.2.2 Régles et conventions

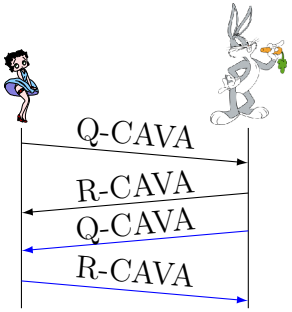
1.2.2.1 MSC (Message Sequence Chart)

Un MSC illustre un scénario d'un dialogue entre des entités
Exemple: Dialogue pour entamer une conversion avec confirmation



1.2.2.2 PDU (Protocol Data Unit)

Les entités s'échange des messages, ce sont les PDUs. Un protocole définit tous les types de PDUs.
Combien de PDU dans l'exemple?:



Les PDUs sont définis physiquement bit à bit ou octet par octet:

0	1	2-5	6-7	6-...
type	ident	CRC	taille	data

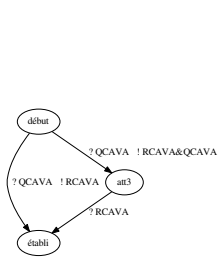
Pour les entiers l'endianness est spécifiée, le format des flottants, le charset pour les caractères et les chaînes de caractères, ...

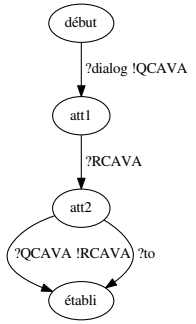
1.2.2.3 Automate

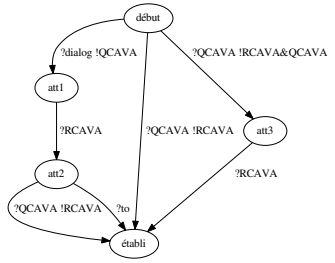
- MSC n'est qu'un scénario,
- définir le protocole $\Longrightarrow$ faire tous les scénarios possibles,

$\Longrightarrow$ souvent impossible.

$\Longrightarrow$ trop volumineux.







la fusion des 2

1.2.2.4 Résumé

Définir un protocole, c'est définir:

Les services: comment le protocole réagit avec le monde extérieur.

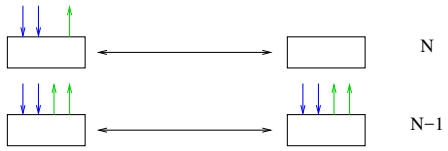
Les PDUs: Les messages que s'échangent les entités

L'automate (ou les automates): comment les échanges de PDUs et les services se déroulent.

1.3 Mise en oeuvre d'un protocole

La mise en oeuvre du protocole N nécessite:

1. Choisir la couche $N-1$ sur la quelle on s'appuie



2. Choisir l'API qui définit les services

```
// Version 1
Ts3 ToutEnUn(Tsap sap,
      Ts1 p1, Ts2 p2);
```

```
// Version 2
int myprot_init(Tsap sap);
int myprot_lose(int id);
int myprot_S1(int id, Ts1 p1);
int myprot_S2(int id, Ts2 p2);
Ts2 myprot_S3(int id);
```

3. Développer le code On utilise les service de la couche $N-1$ pour envoyer les PDUs au vis-à-vis.

1.4 Quiz

1. L'établissement de la liaison de communication avec l'entité vis-à-vis est une phase complexe des protocoles.

2. Un protocole de niveau N est défini indépendamment de la couche N-1 sur laquelle il s'appuie.

3. Un protocole peut être implanter sur n'importe quel autre protocole. N-1 sur laquelle il s'appuie.

4. Soit A et B 2 mises en œuvre d'une présentation API d'un protocole P, écrites par 2 personnes différentes. A et B peuvent elles communiquées?

5. Soit A une mise en oeuvre d'une présentation API1 d'un protocole P, et B une mise en oeuvre d'une présentation API2 d'un protocole P. A et B peuvent elles communiquées?

6. Dans l'automate qui spécifie le dialogue d'un protocole N apparait les services du protocole N-1.

7. La mise en oeuvre d'un protocole de niveau N utilise les services du protocole N-1.

8. La mise en oeuvre d'un protocole de niveau N-1 appelle les services du protocole N.

9. Un protocole peut-il ne pas avoir de service?

10. Une entité d'un protocole peut-elle ne pas avoir de service?

1.5 Illustration

1.5.1 Exercices

Exercice 1

Complétez la figure 1 et indiquez les différents éléments mis en œuvre (utilisateur, système de fichiers, pile réseau, service, PDU). Puis définissez ce qu'est un protocole.

Exercice 2

Donnez 2 définitions précises des services minimaux du protcol FTP et indiquez les sur la figure 1. Choisissez en une pour la suite.

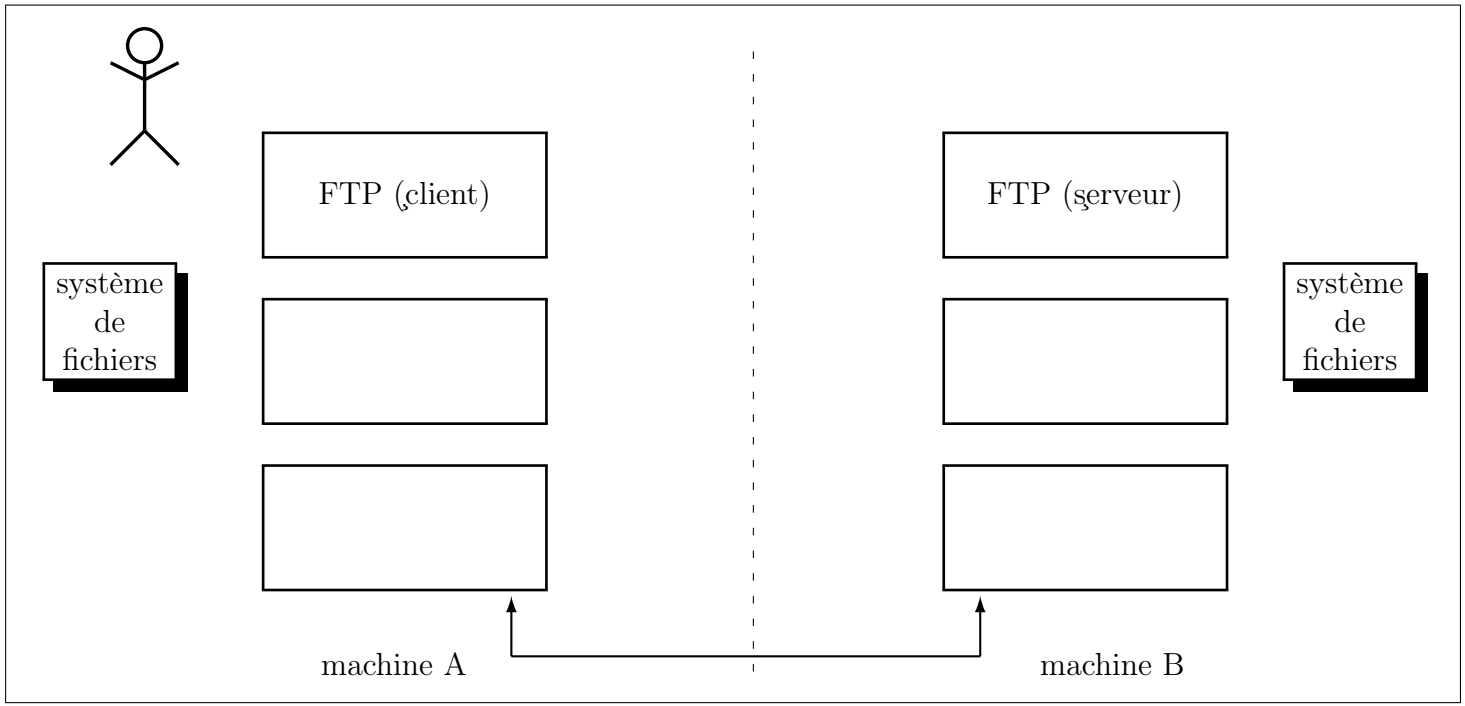


Figure 1: Éléments d'un protocole

Exercice 3

On considère un protocole de transport dont les caractéristiques sont les suivantes:

- Un paquet contient toujours N octets de données.
- Lorsqu'un paquet est envoyé, un paquet est reçu.
- Les données d'un paquet reçu ne correspondent pas forcément à celles émises.

Donnez les services de ce protocole.

Exercice 4

Élaboration d'un protocole FTP implantant les services définis à la question 2 pouvant fonctionner sur la couche transport précédente. On choisira un protocole simple où dans le cas de l'envoi d'un fichier, le serveur acquitte tous les paquets.

1. Donnez un MSC illustrant l'envoi d'un fichier sans problème.
2. Donnez un MSC illustrant l'envoi d'un fichier avec un paquet de données altéré.
3. Donnez un MSC illustrant l'envoi d'un fichier avec un paquet accusé de réception altéré.
4. Définissez les PDU du protocole
5. Écrivez l'automate du protocole (la partie envoi uniquement) du coté du client.
6. Écrivez l'automate du protocole (la partie envoie uniquement) du coté du serveur.
7. Donnez une description précise des PDU.

Exercice 5

Proposez 2 présentations précises des services définis à la question 2. Choisissez en une pour la suite.

Exercice 6

En considérant que la présentation de la couche transport de la question 3 est:

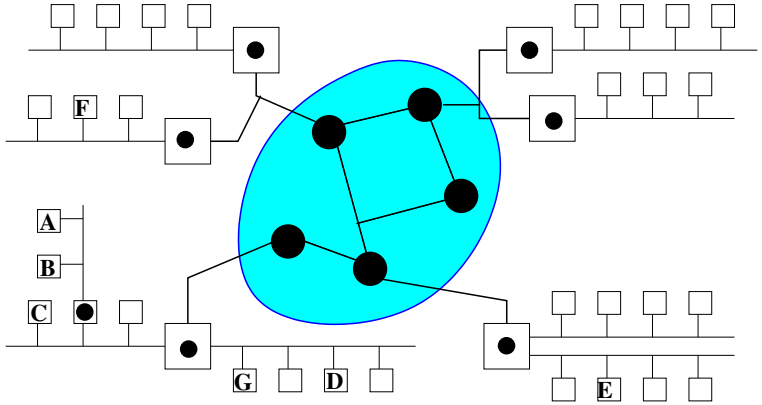
```
// envoie buf[0] à buf[N-1] octets à sap
// 0:OK ; -1:erreur ES
int  write(Tsap sap, char* buf)
// recoit buf[0] à buf[N-1] octets de sap
// 0:OK ; -1:erreur ES
int  read (Tsap*,char*)
```

donnez le pseudo-code en C de la fonction de présentation qui envoie un fichier.

2 Réseau Internet

2.1 Vue générale

2.1.1 Présentation



2.1.1.1 Définitions

Un hôte une machine.

Un réseau ensembles de machine pouvant communiquer directement.

Un routeur (1) une machine appartenant à plusieurs réseau, (2) programmée pour faire passer des messages venant d'un réseau sur un autre.

2.1.1.2 Communication

A $\longrightarrow$ **B** directe en utilisant le support physique qui les connecte.

A $\longrightarrow$ **C** indirecte en 1 saut:
- A $\longrightarrow$ R0 en utilisant le support qui les connecte.
- R0 $\longrightarrow$ B en utilisant le support qui les connecte.

A $\longrightarrow$ **F** indirecte en 6 ou en 8 sauts suivant le chemin suivi.

débit

latence

2.1.1.3 Problèmes relatifs à la communication

Réseau (local) en général des arbres $\Longrightarrow$ un seul chemin.

Réseaux des routeurs un graphe $\longrightarrow$ plusieurs chemins:

- 1 message peut tourner indéfiniment.
- Si A émet 2 messages d'abord M1 puis M2 pour F alors M2 peut arriver avant M1.
- Accroit la solidité du réseau.
- Nécessite d'algorithmes adaptatifs complexes.

2.1.1.4 Pile

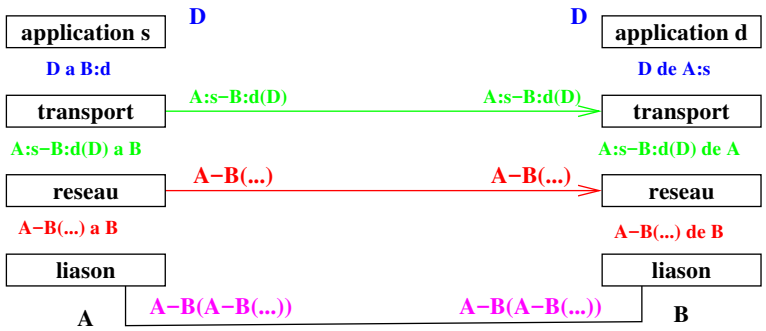
Liaison Communication sur le lien physique.

Réseau Acheminer les messages d'un bout à l'autre du réseau.

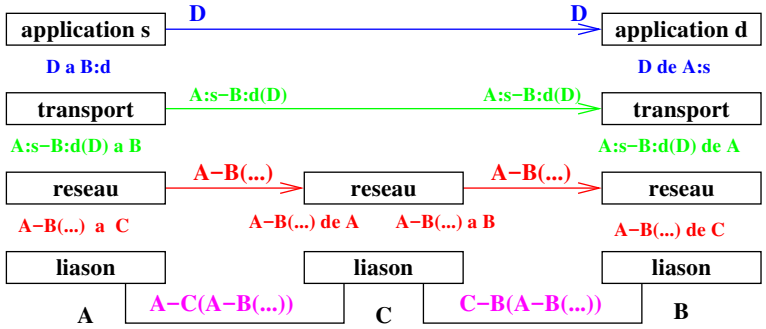
Transport Assurer que les messages transmis sont:

- Arrivés
- Intègres
- Dans le bon ordre

2.1.1.5 Fonctionnement de la pile



2.1.1.6 Routage



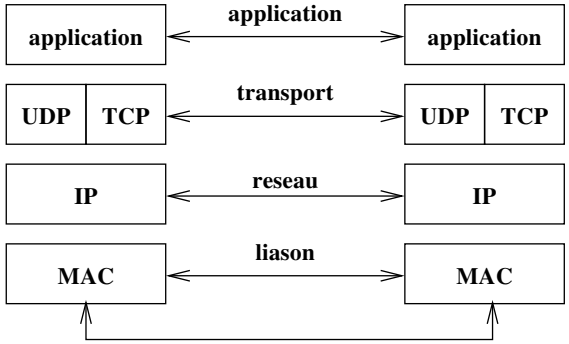
Un routeur remonte la pile jusqu'à la couche réseau,

- celle-ci extrait l'adresse destination du PDU réseau,
- renvoie le PDU à cette destination ou au routeur suivant.

Le PDU réseau n'est pas altéré.

2.1.2 Protocoles de l'Internet

2.1.2.1 Pile Réseau



2.2 Liaison MAC

2.2.1 Vue utilisateur

MAC : Medium Access Control
Rôle : communication directe entre 2 machines
SAP : adresse:port
adresse : 6 octets (notée: xx:xx:xx:xx:xx:xx)
port : 1 ou 2 octets (identifie la couche supérieure)
Service : envoyer_à(SAP,data) et reçu_de(SAP,data)

2.2.2 Supports physiques

Historique câble coaxial

- 10BASE5,10BASE2: 10 mb, quelques 100aine de mètres
- 100BASE2: 100 mb, quelques kilomètres

Filaire sur 2 paires torsadées RJ45

- 10/100/1000/10GBASE-T: 10, 100, 1G et 10G bits/s.
- switch simple: 100 mètres (5 niveaux)
- switch VLAN:

Hertzien

- 802.11b: 10/5 Mbits, $\sim$ mètres.
- 802.11a: 54/27 Mbits, $\sim$ mètres.

- bluetooth: $\sim$ 1 Mbits, 10 à 100 mètres.

CPL courant 220 volt: 10 Mbits, 100 mètres.

2.2.3 PDU

type 2: taille minimale 64 octets (prot>1500)

mac-des	mac-src	protocole	données	bourrage	CRC
6 octets	6 octets	2 octets			4 octets

type 3: taille minimale 64 octets (taille<=1500)

mac-des	mac-src	taille	llc	données	bourrage	CRC
6 octets	6 octets	2 octets	3 à 8			4 octets

2.2.4 Quelques caractéristiques

- Problème des collisions
- Explique les différences entre les débits des supports et réels
- Switch: communications full-duplex et simultanées, collisions quasi supprimées
  $\Longrightarrow$ débit supports $\sim$ débit réel
- Coaxial, Hub, CPL: collisions détectables $\Longrightarrow$ retransmission après un temps aléatoire
  $\Longrightarrow$ débit réel $\downarrow$ si le réseau est chargé
- Hertzien:
  - Débit support dépend de la distance
  - collisions non détectables $\Longrightarrow$ ordonnancement des échanges

  $\Longrightarrow$ débit réel $\downarrow$ dans tous les cas

2.3 Réseau IP

2.3.1 Vue utilisateur

IP	Internet Protocol
Rôle	acheminement de messages d'un bout à l'autre du réseau
SAP	adresse:port
adresse	32 bits en big-endian
port	1 octet (identifie la couche supérieure)
Service	envoyer_à(SAP,data) et reçu_de(SAP,data)

2.3.2 Adresse IP

2.3.2.1 Format

IP 32 bits en big-endian $\Longrightarrow$ 11000001001101101100001101111000

doted nnn.nnn.nnn.nnn avec chaque nnn dans [0,255]
$\Longrightarrow$ 120.193.54.195

usuel un nom de machine suivi d'un nom de domaine
$\Longrightarrow$ lunix120.ensiie.fr

conversion

- de IP ou "doted format" vers le format usuel $\Longrightarrow$ appel DNS. par table gérée par les DNS.

```
sh>  dig +short lunix120.ensiie.fr
193.54.195.120
sh>
```

- de IP vers "doted format" automatique
  `11000001001101101100001101111000` donne
  `11000001 00110110 11000011 01111000` donne
  `193 54 195 120` donne
  `193.54.195.120`

2.3.2.2 Sémantique

Une adresse IP se décompose en un numéro de réseau et un numéro d'hôte sur ce réseau. Par exemple cette adresse IP

`rrrrrrrrrrrrrrrrrrrrrhhhhhhhhhh` donne pour le réseau `rrrrrrrrrrrrrrrrrrrrr0000000000` et pour l'hôte `00000000000000000000hhhhhhhhhh`

Le point qui sépare les champs réseau et hôte n'est pas fixe, il est spécifié par un nombre de 32 bits contenant une suite de 1 suivi d'une suite de 0 (une des 2 suites peut être vide). Dans l'exemple précédent le masque réseau est:

`11111111111111111111110000000000`

Pour I une adresse IP et M un masque réseau, IP définit:
le réseau par I&M
l'hôte par I&∼M
l'adresse de broadcast du réseau par (I&M)|∼M

Par exemple en "doted format" si IP=193.54.195.120 et MR=255.255.255.0 alors Réseau=193.54.195.0, Hôte=120 et Broadcast=193.54.195.255.

Il existe aussi la notation ip/n qui est une abréviation de IP=ip et MR=<n bits à 1,32-n bits à 0>. Par exemple avec cette notation: IP=193.54.195.120/24 donne MR=255.255.255.0, Réseau=193.54.195.0/24, Hôte=120 et Broadcast=193.54.195.255.

2.3.2.3 Classes

A l'origine IP définissait les composantes réseau et hôte d'une adresse en fonction des premiers bits comme le montre la table ci-dessous:

classe A	0xxxx..	000 à 127.nnn.nnn.nnn	255.0.0.0	/8
classe B	10xxx..	126 à 191.nnn.nnn.nnn	255.255.0.0	/16
classe C	110xx..	192 à 223.nnn.nnn.nnn	255.255.255.0	/24
multicast	1110x..	224 à 239.nnn.nnn.nnn		
réservé	1111x..	240 à 255.nnn.nnn.nnn		

Depuis 2010 cette classification n'est plus valable sauf pour les réseaux alloués antérieurement.

2.3.2.4 Publiques/Privées

Une adresse IP est publique si le réseau IP la route sinon elle est dite locale.

adresses IP privées			
classe A	0xxxx..	010.nnn.nnn.nnn	/8
classe B	10xxx..	172.016.nnn.nnn à 172.031.nnn.nnn	/16
classe C	110xx..	192.168.000.nnn à 192.168.255.nnn	/24
plus	127.0.0.1 et 127.0.0.0 et 0.0.0.0		

2.3.2.5 Exemples

1. Donnez le /n du masque réseau, les numéros de réseau et d'hôte et l'adresse de broadcast pour IP=225.67.129.30 et MR=255.255.192.0.
2. Donnez le masque réseau, les numéros de réseau et d'hôte et l'adresse de broadcast pour IP=225.67.127.30/19.
3. Donnez le masque réseau et son /n puis les numéros de réseau et d'hôte et l'adresse de broadcast pour IP=193.54.193.12.
4. Donnez le masque réseau et son /n puis les numéros de réseau et d'hôte et l'adresse de broadcast pour IP=225.67.127.30.
5. Donnez le nombre d'hôtes du réseau IP=8.8.8.8 et MR=255.255.255.192
6. Donnez le nombre d'hôtes du réseau IP=8.8.8.8/22.
7. J'ai à disposition le réseau 192.168.5.64/28, je désire faire 3 sous-réseaux de 6, 4 et 3 hôtes. Est-ce possible?
8. J'ai à disposition le réseau 192.168.5.64/27, je désire faire 3 sous-réseaux de 6, 4 et 3 hôtes. Donnez un découpage possible (IP/MR de chaque sous réseau).

2.3.3 Communication

Table de routage

Fonction Elle indique ce que doit faire la machine des paquets IP qui sortent.

- A quelle machine.

- Sur quel lien physique.

Un lien physique est appelé interface. On les notera eth0, eth1, ...

Table de routage: hôte avec 1 interface

destination	passerelle	mask	...	interface
172.30.0.0	0.0.0.0	255.255.0.0	...	eth0
0.0.0.0	172.30.0.254	0.0.0.0	...	eth0

Table de routage: hôte avec plusieurs interfaces

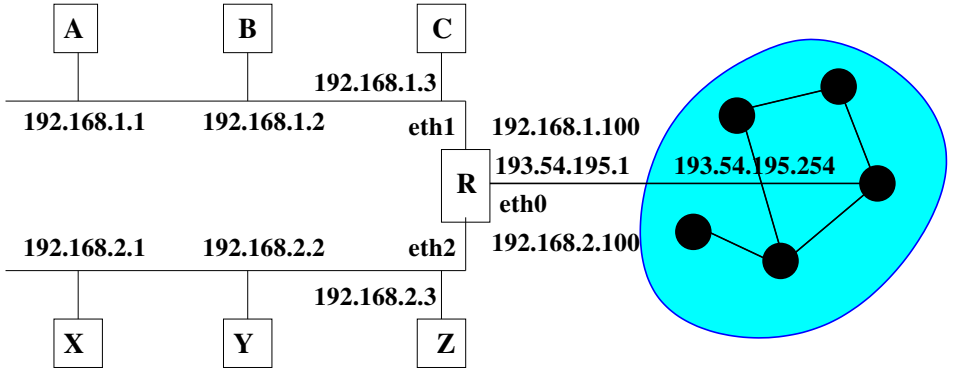
destination	passerelle	mask	...	interface
192.168.2.0	0.0.0.0	255.255.255.0	...	eth0
192.168.3.0	0.0.0.0	255.255.255.0	...	eth1
172.30.0.0	0.0.0.0	255.255.0.0	...	eth2
0.0.0.0	192.168.2.254	0.0.0.0	...	eth0

Hôte simple et routeur

Hôte	PDU		
	généré par une application interne	reçu du réseau destiné à l'hôte	reçu du réseau destiné à un autre hôte
non routeur	envoyé sur le réseau	transmis à l'application	poubelle
routeur	envoyé sur le réseau	transmis à l'application	réenvoyé sur le réseau

$\Longrightarrow$ Une table de routage n'indique pas si une machine est un routeur ou pas

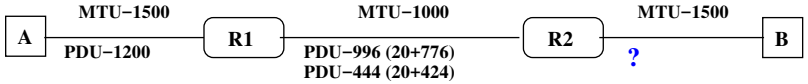
Exemples



1. De combien de réseaux est constitué notre réseau?
2. Donnez la table de routage de la machine A.
3. Donnez la table de routage de la machine Z.
4. Donnez la table de routage de la machine R.

2.3.4 Quelques caractéristiques

- Pour 1 PDU envoyé ne garantit pas:
  - Qu'il arrive à destination.
  - Si il arrive que les données sont correctes.
  - Si il tourne trop longtemps, le routeur qui le détruit envoie un message IMCP à l'émetteur (principe de traceroute et tracert).
- Pour PDU1, PDU2 envoyés en séquence à A:
  - A peut ne recevoir aucun des 2 PDU.
  - A peut ne recevoir que PDU1 ou que PDU2.
  - A peut recevoir PDU1 puis PDU2 ou PDU2 puis PDU1.
- S'adapte au MTU des liens parcourus:



- Options:
  - Options de QoS.
  - Options de routage.
  - ICMP (Internet Control Message Protocol).

2.3.5 PDU

31 28	27 24	23 16	15 13	12 0
version	taille entête	type de service	taille totale	
identifiant			flag	fragment offset
durée de vie		protocole	CRC entête	
adresse source				
adresse destination				
options [+ remplissage]				
données ...				

version IPv4

taille en-tête en mots de 32 bits options comprises.

type de service Qualité de service, ...

taille totale taille en octet du PDU.

identifiant Numéro identifiant le le PDU.

flags bit-15 non utilisé=0, bit-14=1 fragmentation interdite, bit-13=0 dernier fragment d'un PDU.

fragment offset Position du fragment dans le PDU initial en nombre de 8 octets.

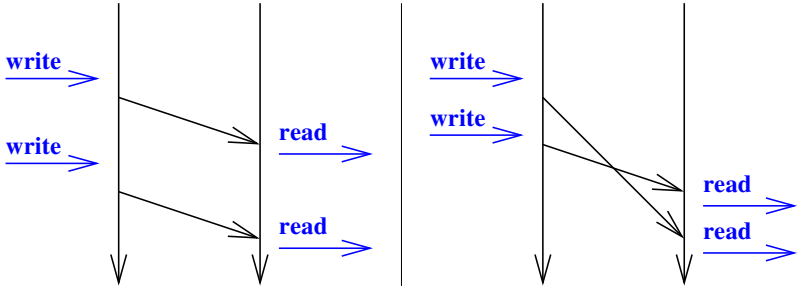
CRC entête IP uniquement.

options peut être vide. Exemple: découverte du MTU, routage par la source, enregistrement d'une route, ...

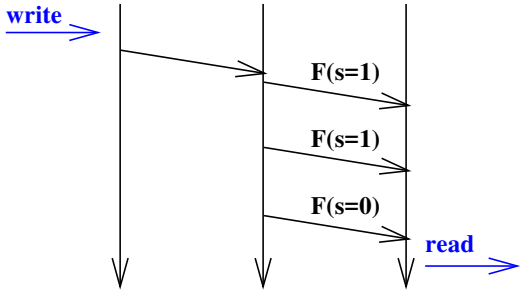
données peut être vide.

2.3.6 Protocole

Exemple sans fragmentation



Exemple avec fragmentation



2.4 Transport UDP/TCP

2.4.1 Transport UDP

2.4.1.1 Vue utilisateur

UDP : User Datagram Protocol
Rôle : communication entre applications d'un bout à l'autre du réseau en mode datagram
SAP : adresse-IP:port
port : 16 bits en big-endian, identifie l'application sur une machine
Service : envoyer_à(SAP,data,option) et reçu_de(SAP,data)

2.4.1.2 Caractéristiques

- Surcouche sur IP pour les applications.
  - flux de messages
  - aucune garantie qu'un message arrive
  - aucune garantie qu'un message reçu est correct
- options:
  - controle d'erreur de transmission
- Peu utilisé.

2.4.1.3 PDU

31 16	15 0
port-src	port-des
taille	CRC ou 0
data ...	

taille en octet entête comprise

data peut être vide (taille=4).

CRC 0 pas de CRC sinon PDU + entête IP

2.4.1.4 Protocole

```
        PDU
  |------------>|
```

2.4.2 Transport TCP

2.4.2.1 Vue utilisateur

TCP : Transmission Control Protocol
Rôle : communication entre applications d'un bout à l'autre du réseau en mode connecté
SAP : adresse-IP:port
port : 16 bits en big-endian, identifie l'application sur une machine
Service : écoute(SAP), nouvelle_connexion(SAP), connecte(SAP) envoyer(data), recevoir(data) et déconnecté()

2.4.2.2 Caractéristiques

- Mode connecté
  - flux octets bidirectionnel
  - contrôle d'erreur de transmission
  - contrôle de flux au niveau applicatif
  - contrôle de flux au niveau réseau
  - s'adapte au réseau physique:
    - si altération de PDU $\Longrightarrow$ diminue la taille des PDU
    - si perte de PDU (congestion du réseau) $\Longrightarrow$ diminue le débit
- Options:
  - envoi et réception de données hors flux
- Très utilisé.

2.4.3 PDU

31 28	27 22	21 16	15 0
port-src			port-des
numéro de séquence			
numéro d'acquittement			
taille en-tête	réservé	flags	taille fenêtre
CRC			pointeur données urgente
options			NOP
données ...			

taille en-tête en mots de 32 bits options comprises.

flags SYN, ACK, URG, ...

taille fenêtre taille du buffer de réception en octets.

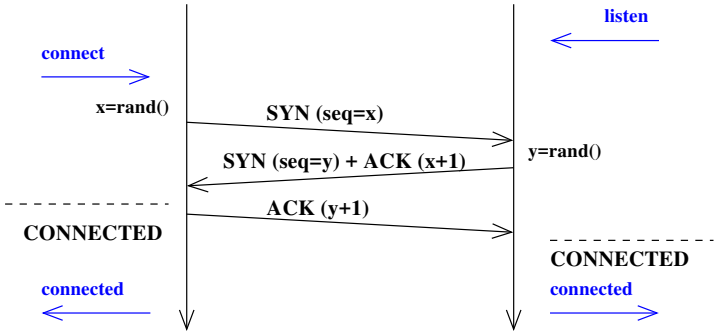
CRC PDU + entête IP.

options peut être vide. Exemple: Indiquer la taille maximale d'un segment, Indiquer une taille de fenêtre supérieur à 2^{16}, ...

données peut être vide.

2.4.3.1 Protocole

Connexion



Les 2 entités sont connectées et connaissent les numéros de séquence et d'acquittement de leurs vis-à-vis.
seq=x+1 ack=y+1 seq=y+1 ack=x+1

Échange de données

La figure 2.4.3.1 illustre le mécanisme de la fenêtre glissante pour un RTT égal à 3 fois le temps d'émission TE, dans ce cas:

Taille fenêtre 1 un segment tous les 3 TE

Taille fenêtre 2 deux segments tous les 3 TE

Taille fenêtre 3 3 segments tous les 3 TE qui est le débit maximal.

Déconnexion

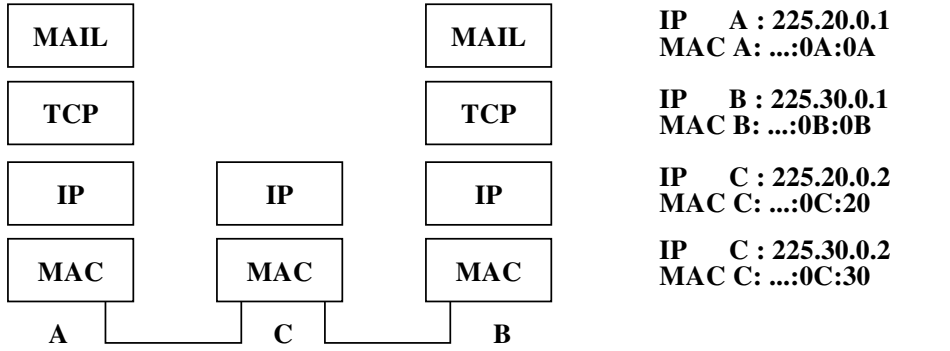
La figure 2.4.3.1 illustre la déconnexion du protocole TCP. On remarque que les échanges de données continuent dans un seul sens.

2.4.3.2 Conclusion

- Protocole robuste qui a fait ses preuves depuis 40 ans.
- Mais attention: coûte très cher si on n'envoie que quelques données (3 PDU pour la connexion + 4 PDU pour la déconnexion).

2.5 Exercices

Exercice 1



src−mac des−mac protocole	src−ip des−ip protocole	src−port des−port	MAIL DATA

src−mac des−mac protocole	src−ip des−ip protocole	src−port des−port	MAIL DATA

L'application cliente MAIL de A envoie un message SMTP à B, le port TCP du client MAIL est 2000 et le port SMTP est 25.

Q1 Complétez le PDU (MAC) qui transite de A à R.

Q2 Complétez le PDU (MAC) qui transite de R à B.

Q3 Quels champs de l'entête IP, R a il modifiés?

Q4 Sur un réseau IP:

- Qu'indique la partie réseau d'une adresse?
- Un routeur route il des IP de réseaux où d'hôtes?
- Quelle est la taille maximale de sa table de routage?

Q5 Sur un réseau téléphonique mobile:

- Une adresse est-elle composée d'une partie réseau et hôte?
- Qu'indique un numéro de téléphone mobile?

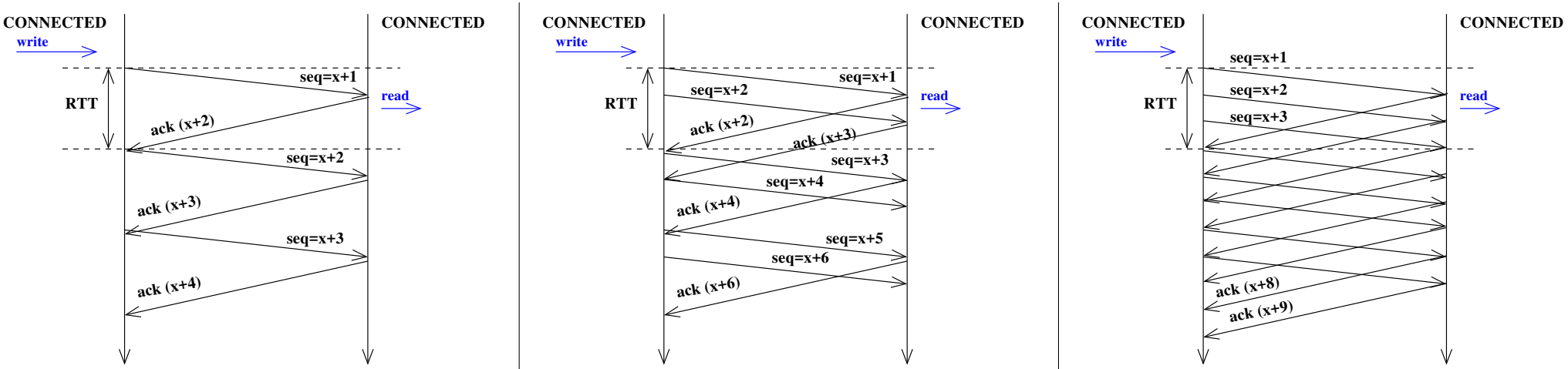


Figure 2: Fenêtre glissante de TCP

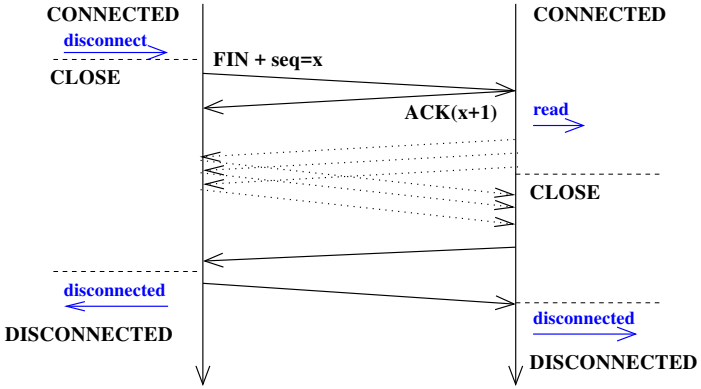


Figure 3: Déconnexion TCP

- Quelle est la taille maximale de la table de routage d'un routeur d'un réseau téléphonique mobile?

Exercice 2

Donnez le masque réseau, les numéros de réseau et d'hôte et l'adresse de broadcast pour

Q1 IP=225.67.127.30/24

Q2 IP=225.67.127.30/19.

Q3 IP=225.67.127.31 et MR=255.255.255.224

Q4 IP=8.80.80.1

Q5 IP=220.80.80.1

Exercice 3

Les quelles de ces adresses IP sont privées: 172.0.0.1, 8.8.8.8, 172.31.12.12, 172.15.12.12, 10.1.1.1, 192.168.0.12 192.169.0.12

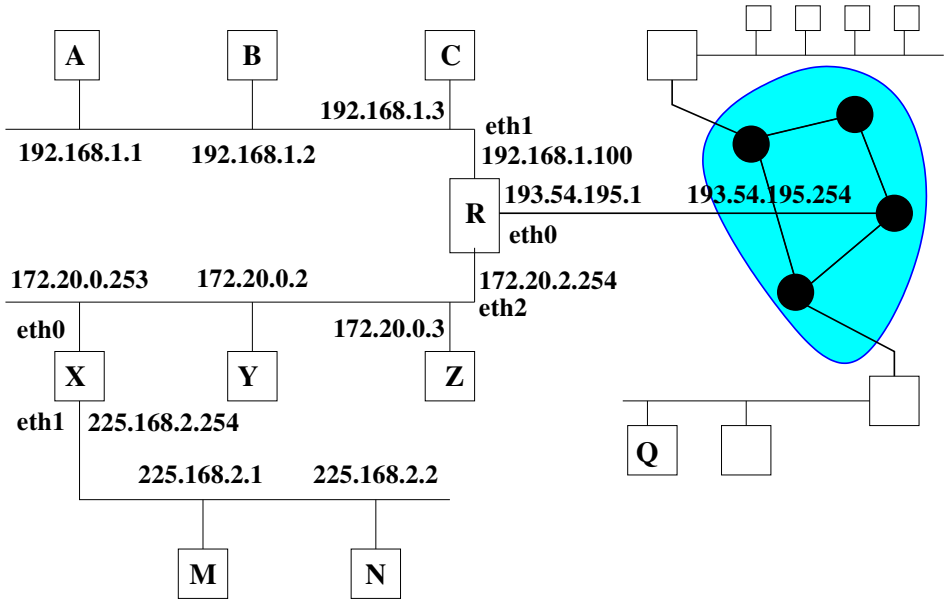
Exercice 4

Q1 J'ai le réseau 192.168.5.64/26, je désire faire 3 sous-réseaux de 35, 15 et 15 hôtes. Est-ce possible?

Q2 J'ai le réseau 192.168.5.64/26, je désire faire 3 sous-réseaux de 32, 10 et 8 hôtes. Est-ce possible?

Q3 J'ai le réseau 192.168.5.64/26, je désire faire 3 sous-réseaux de 20, 10 et 8 hôtes. Donnez un découpage possible (IP/MR de chaque sous réseau).

Exercice 5



Notre réseau est constitué des hôtes A, B, C, R, X, Y, Z, M et N.

Q1 De combien de réseaux est constitué notre réseau?

Q2 Quelles sont les machines de notre réseau qui peuvent communiquer avec Q?

Q3 Combien de sauts fera un PDU émis par N pour atteindre Q.

Q4 Donnez la table de routage de B.

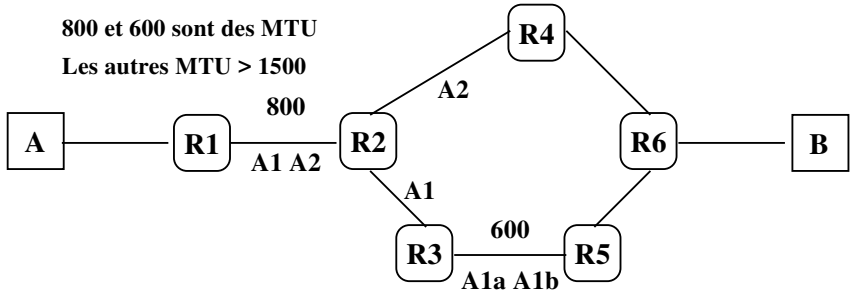
Q5 Donnez la table de routage de R.

Q6 Donnez la table de routage de N.

Q7 Donnez la table de routage de X.

Exercice 6

Sur le réseau ci-dessous, A envoie à B un PDU IP de 1000 octets



Q1 Complétez les champs du PDU émis par A (te et tp sont les tailles de l'entête et du PDU).

te=5	id=9103	ttl=5	tp=	flag=00 .	offset= . . . .

Q2 A arrive sur R1, complétez les champs des PDU émis par R1.

A1:	te=5	id=9103	ttl= .	tp=	flag=00 .	offset= . . . .
A2:	te=5	id=9103	ttl= .	tp=	flag=00 .	offset= . . . .

Q3 A arrive sur R2, R2 envoie le premier PDU sur R3 et le second sur R4. Complétez les champs des PDU émis par R3 et R4.

A1a:	te=5	id=9103	ttl= .	tp=	flag=00 .	offset= . . . .
A1b:	te=5	id=9103	ttl= .	tp=	flag=00 .	offset= . . . .
A2: :	te=5	id=9103	ttl= .	tp=	flag=00 .	offset= . . . .

Q4 Dans ce cas de figure, donnez une optimisation qu'aurait pu faire le routeur R1.

Q5 En supposant que le temps de transmission est proportionnel au nombre de routeurs traversés

- Donnez les TTL des PDU A1a, A1b et A2 en B.
- Indiquez dans quel ordre, les PDU A1a, A1b et A2 arrivent en B.
- Que se passera t-il si un des PDU (A1a, A1b, A2) s'est altéré entre R6 et B?
- Que se passera t-il si le PDU A2 s'est altéré entre R4 et R6?

3 Programmation

3.1 Algorithme entre 2 entités

```
1  // s'attacher au vis-à-vis
2  Tsap vav = ...
3
4  // Automate état-a0
5  PDU = ...
6  écrireRéseau(vav,PDU)
7  PDU,sap = lire_réseau(vav)
8
9  // Automate état-a1
10 PDU = ...
11 écrireRéseau(vav,PDU)
12 PDU,sap = lireRéseau(svr)
13 ...
14 // se détacher du vis-à-vis
```

```
1  // s'attacher au vis-à-vis
2  Tsap vav = ...
3
4  // Automate état-b0
5  PDU,sap=lireRéseau(vav)
6  PDU = ...
7  écrireRéseau(vav,PDU)
8
9  // Automate état-b1
10 PDU,sap=lireRéseau(vav)
11 PDU = ...
12 écrireRéseau(vav,PDU)
13 ...
14 // se détacher du vis-à-vis
```

Algorithme tout à fait correct et opérationnel.

3.2 Algorithme Clients/Serveur

3.2.1 naif

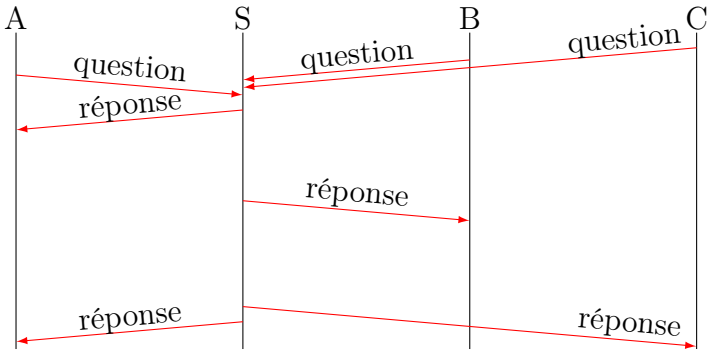
```
1  // s'attacher au serveur
2  Tsap svr = ...
3
4  // Automate état-c0
5  PDU = ...
6  écrireRéseau(svr,PDU)
7  PDU,vav = lire_réseau(svr)
8
9  // Automate état-c1
10 PDU = ...
11 écrireRéseau(svr,PDU)
12 PDU,vav = lireRéseau(svr)
13 ...
14 // se détacher du serveur
```

```
1  // s'attacher aux clients
2  TEsap clts= {sap0, ...}
3
4  // algorithme serveur
5  while (1) {
6    // Automate état-s0
7    PDU,clt=lireRéseau(clts)
8    PDU = ...
9    écrireRéseau(clt,PDU)
10
11   // Automate état-s1
12   PDU,clt=lireRéseau(clt)
13   PDU = ...
14   écrireRéseau(clt,PDU)
15   ...
16 }
17 // se détacher des clients
```

Attention: Ligne 12 clt et non clts.

Client tout à fait correct et opérationnel.
Serveur correct (il fonctionne) mais non performant et surtout non sûr.

3.2.2 Réactivité d'un serveur



$\Longrightarrow$ Grosse perte de réactivité.

3.2.3 Accroissement de la réactivité

3.2.3.1 Serveur à contexte

Le serveur doit conserver les contextes (état + sap) des dialogues en cours avec les clients dans un ensemble E.

Au début de la boucle, en fonction du SAP du client qui a envoyé le message, soit on récupère le contexte (appelé ctx d'un dialogue déjà commencé, soit on crée un nouveau contexte (début,sap). C'est le rôle de la fonction "context_de".

```
E = vide // ensemble de contexte
while (1) {
  PDU, clt=lireRéseau(clts)

  ctx = context_de(E, clt)

  // une étape de l'automate ctx
  if ( ctx->état==début ) {
      PDU = ... ;
      écrireRéseau(ctl,PDU)
      ctx->état = ...;
  } else if ( ctx->état==état_1 ) {
      PDU = ... ;
      écrireRéseau(ctl,PDU)
      ctx->état = ...;
  } else if ( ctx->état==état_2 ) {
  ...
  }
}
```

+ Le serveur fonctionne et gère les clients en parallèle.
+ Réactif pour les clients lents.
- Non réactif pour les traitements internes longs.
+ Économe en ressources (CPU, mémoire, ...).

3.2.3.2 Serveur multi-processus

Au lieu de créer un contexte par client, on crée un processus par client.

```
ile (1) {
PDU, clt=lireRéseau(clts)

if ( fork()==0 ) {
  // Automate état-s0
  PDU = ...
  écrireRéseau(clt,PDU)

  // Automate état-s1
  PDU, clt=lireRéseau(clt)
  PDU = ...
  écrireRéseau(clt,PDU)
  ...
  exit(0);
}
```

Le processus fils s'occupe du dialogue avec le client, le processus père se remet immédiatement à scruter un nouveau client.

Avantages

- Le serveur traite plusieurs clients en parallèle.
- Réactif pour les clients lents.
- Réactif pour les serveurs à traitements internes lents.
- Adapte la consommation de ressources à la demande.

Inconvénient

- Si le traitement du serveur est court
  la réactivité $\downarrow$
  car la création d'un processus est coûteuse.
- Si il a un grand nombre de clients
  la réactivité $\downarrow$
  car l'OS va faire beaucoup de commutation et dans les cas extrêmes même tomber.

- Dans cette version, impossible de calibrer les ressources utilisées par le serveur.

L'utilisation de processus léger (thread) atténue ces inconvénients.

3.2.3.3 *Serveur avec pool de processus*

L'algorithme est de créer N processus au départ. Chaque processus

1. Attend un nouveau client.
2. Traite le dialogue avec ce client.
3. Reprend en 1.

$\Longrightarrow$ Il répond bien au problème de la réactivité tant que l'on ne dépasse pas N clients en parallèle.

$\Longrightarrow$ Il permet de calibrer les ressources (mémoire, CPU, ...) utilisées par le serveur.

3.2.4 Conclusion

Protocole Question/Réponse avec traitements courts
Algorithme naif.

Protocole Question/Réponse avec traitements longs Serveur multi-processus ou pool de serveurs

Autres Protocoles avec traitements courts

- Algorithme avec contexte.
  $\Longrightarrow$ Mise en place un peu lourde, utilise très peu de ressources
- Pool de serveur.
  $\Longrightarrow$ Mise en place plus facile, utilise plus de ressources

Autres Protocoles avec traitement long Serveur multi-processus ou pool de serveurs.

3.3 Implémentation

3.3.1 Fonctions de base

Includes pour les primitives réseau

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
```

Gestion de l'endianess

Les protocoles de base de l'Internet sont en big-endian. Pour pouvoir écrire des programmes portables on a les fonctions:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

h signifie host et n signifie network.

Gestion des SAP

Un SAP est défini par le type sockaddr. Comme le montre sa définition ci-dessous, c'est un type opaque.

```
struct sockaddr {
  unsigned short sa_family;    // address family, AF_xxx
  char           sa_data[14]; // 14 bytes of protocol address
};
```

Dans ce cours, on utilise UDP ou TCP/IPv4 nos SAP sont définis par le type sockaddr_in:

```
struct in_addr sin_addr {
  unsigned int s_add;          // IP addr (big-endian)
};
struct sockaddr_in {
  unsigned short sin_family; // addr. family, AF_INET
  unsigned short sin_port;   // UDP/TCP port (big-endian)
  struct in_addr sin_addr;   // IP
```

```
  char              padding[8];
};
```

On peut passer d'un pointeur sur sockaddr à un pointeur sur sockaddr_in et vice-versa par simple cast.

La fonction getaddrinfo initialise une liste de addrinfo (et donc de SAP) en fonction de:

node le serveur: nom humain ou doted format.

service le port: un nombre (1200) ou un nom (http).

hints->ai_flags AI_PASSIVE pour serveur TCP, 0 pour les autres.

hints->ai_family spécifie la couche réseaui: AF_INET (IPv4).

hints->ai_socktype spécifie la couche transport: SOCK_STREAM (TCP), SOCK_DGRAM (UDP)

hints->ai_... 0

Si on obtient plusieurs SAP, on prend le premier.

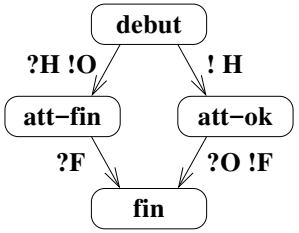
```
int getaddrinfo(
  const char *node,
  const char *service,
  const struct addrinfo
                  *hints,
  struct addrinfo **res);

void freeaddrinfo(
  struct addrinfo *res);

const char*gai_strerror(
  int errcode);
```

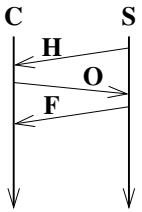
```
struct addrinfo {
  int       ai_flags;
  int       ai_family;
  int       ai_socktype;
  int       ai_protocol;
  socklen_t ai_addrlen;
  struct sockaddr
            *ai_addr; // le SAP
  char      *ai_canonname;
  struct addrinfo
            *ai_next;
};
```

Écrivez la fonction "`void getTCPsap(struct sockaddr* sap, socklen_t*saplen, const char *host, const char* port)`". Si host est non nul elle initialise *sap pour un client TCP/IPv4 sur le serveur host:port.
Si host est nul elle initialise *sap pour un serveur TCP/IPv4.
En cas d'échec, elle affiche un message d'erreur et termine le processus avec un statut de 1.

3.3.2 Le protocole "hello"

L'exemple TCP qui suit implémente le protocole présenté ci-dessous:





Le format des PDU en octet:

H :

H	E	L	L	0	\0

O :

O	K	\0

F :

F	I	N	\0

3.3.3 Client TCP

3.3.3.1 Initialisation

```
const char* prgname;

int main(int argc, char** argv)
{
  prgname = argv[0];
  int statut;

  /* vérification des arguments */
  if ( argc!=3 ) {
    fprintf(stderr,"%s:usage %s serveur port\n",
            prgname,prgname);
    exit(1);
  }
  char* namesvr = argv[1];
  char* service = argv[2];
```

3.3.3.2 Connexion au serveur

```
/* création du SAP du serveur */
struct sockaddr svrSAP;
socklen_t       svrSAPlen;
getTCPsap(.... , ..... , ..... , .....);

/* connexion au serveur */
int cx;
```

```
if ( (cx=socket(AF_INET,SOCK_STREAM,0))==-1 ) {
   fprintf(stderr,"%s: pb socket: %s\n",
           argv[0],strerror(errno));
   exit(1);
}
if ( connect(cx, ..... , ..... )==-1 ) {
   fprintf(stderr,"%s: pb connect: %s\n",
           argv[0],strerror(errno));
   exit(1);
}
```

cx est un double flux d'octets FIFO, un du client vers le serveur, l'autre du serveur vers le clients. Ils se comportent comme un pipe.

Le nombre de lecteurs est .... et le nombre d'écrivains est .....

Une lecture est ..... si la FIFO est vide et qu'il y a encore ....

Une lecture renvoie EOF si la FIFO est .... et qu'il n'y a plus ..... .

Une écriture est ..... si la FIFO est pleine et qu'il y a encore ....

Une écriture renvoie une erreur si il n'y a plus .... Dans ce cas le signal ..... est envoyé à ..... .

3.3.3.3 Dialogue

```
   /* Dialogue état début */
   char PDU[100];
   statut = lire_PDU(PDU,cx);
   if ( statut!='H' ) goto error;
printf("client: recu \"%s\" ;", PDU);
   sprintf(PDU,"OK");
   write( cx , .... , ...);
printf(" envoyé \"%s\"\n", PDU);

   /* Dialogue état att-fin */
   statut = lire_PDU(PDU,cx);
   if ( statut!='F' ) goto error;
printf("client: reeu \"%s\" ; quitte", PDU);
```

cx est un flux d'octets qui contient les PDU les uns derière les autres. De plus ils peuvent être mal formatés. Il faut donc une routine pour extraire un PDU du flux d'octets et les vérifier. C'est le rôle de "lire_PDU".

Définissez et écrivez la fonction lire_PDU.

3.3.3.4 Terminaison

```
 /* Terminaison état fin */
  close(cx);
  return 0;

error:
   fprintf(stderr,"%s: message de type %c est inattendu\n",
           prgname,statut);
  close(cx);
  return 1;
}
```

3.3.4 Serveur TCP (naif)

3.3.4.1 Initialisation

```
const char* prgname;

int main(int argc, char** argv)
{
  prgname = argv[0];
  int statut;

  /* vérification des arguments */
  if ( argc!=2 ) {
    fprintf(stderr,"%s: usage %s port\n",
            prgname,prgname);
    exit(1);
  }
  char* service = argv[1];
```

$\Longrightarrow$ Pas de nom d'hôte car il ne se connecte pas.

3.3.4.2 S'attacher aux clients

```c
/* création du SAP des clients */
struct sockaddr cltsSAP;
socklen_t       cltsSAPlen;
getTCPsap(.... , ..... , .... , .....);

/* activer l'automate de connexion */
int sock;
if ( (sock=socket(. . . . , . . . . . , . .))==-1 ) {
  fprintf(stderr,"%s: pb socket: %s\n",argv[0],...);
  exit(1);
}
if ( bind(sock,&cltsSAP,cltsSAPlen)<0 ) {
  fprintf(stderr,"%s: pb bind %s\n",argv[0],...);
  exit(1);
}
if ( listen(sock,100)!=0 ) {
   fprintf(stderr,"%s: pb listen %s\n",argv[0],...);
   exit(1);
}
```

3.3.4.3 La boucle infinie du serveur

Connexion à un client

```c
while (1) {
  int cx;
  struct sockaddr cltSAP;
  socklen_t       cltSAPlen = sizeof( cltSAP );

  /* creation du flux de communication (cx) */
  if ( (cx=accept(sock,&cltSAP,&cltSAPlen))==-1 ) {
    fprintf(stderr,"%s: pb accept %s\n",
                    argv[0],strerror(errno));
    exit(1);
  }
  /* Dialogue état début */
  ...
  /* Dialogue état att-ok */
  ...
  /* Terminaison du dialogue (état fin) */
  ...
}
```

Dialogue

```c
  /* Dialogue état début */
  char PDU[100];
  sprintf(PDU,"HELLO");
  write(cx,PDU,6);
printf("serveur: envoyé \"%s\"\n", PDU);

  /* Dialogue état att-ok */
  statut = lire_PDU(PDU,cx);
  if ( statut!='O' ) goto error;
printf("serveur: reeu \"%s\" ;", PDU);
  sprintf(PDU,"FIN");
  write(cx,PDU,4);
printf(" envoyé \"%s\" ; quitte\n", PDU);
```

Terminaison du dialogue

```c
while (1) {
  int cx;
  /* creation du flux de communication (cx) */
  if ( (cx=accept(sock,&cltSAP,&cltSAPlen))==-1 ) {
    fprintf(stderr,...); exit(1); }

  /* Dialogue état début */
  ...
  /* Terminaison du dialogue (état fin) */
  close(cx);
  continue;
error:
  close(cx);
  fprintf(stderr,"%s: message de type %c est inattendu\n",
```

```
15          prgname, statut);
16  }
```

3.3.4.4 Résumé des fonction

	clt	svr	fonction
s'attacher aux clients ou au serveur	*	*	`getaddrinfo(...) --> sap & saplen`
	cx=	s=	`socket(AF_INET,SOCK_STREAM,0)`
		*	`bind(s,sap,saplen)`
		*	`listen(s,100)`
connexion	*		`connect(cx, sap, saplen)`
		cx=	`accept (s ,&clt,&cltlen)`
dialogue	*	*	`read (cx,data,len)`
	*	*	`write(cx,data,len)`
fermer la connexion	*	*	`close(cx)`

bind: 1) réserve le SAP (un autre bind sur le même SAP échouera); 2) attache le SAP à la socket.

listen: active l'automate de connexion TCP pour le SAP attaché à la socket. Le N indique le nombre maximal d'automate de connexion qui peuvent tourner en même temps.

4 TP

4.1 Mise en place d'une application client/serveur

Exercice 1

Recopiez les fichiers "hello.h.tpl", "hello-svr.c.tpl" et "hello-clt.c.tpl" du répertoire /pub/ia/reseau dans votre répertoire de travail en enlevant l'extension ".tpl".

Q1 Complétez les trous de "hello.h" puis de "hello-svr.c" (ils sont marqués par "...").

(a) Compilez le serveur.

```
sh> gcc -o svr hello-svr.c
```

(b) Dans un nouveau terminal, lancez le serveur sur le port que vous voulez (on utilisera 2000 dans cet énoncé).

```
sh> ./svr 2000
```

Le serveur doit tenir

(c) Dans un nouveau terminal, relancez le serveur sur le même port.

```
sh> ./svr 2000
./svr: pb bind: Address already in use
sh>
```

En effet le port 2000 est déjà alloué et identifie le serveur tournant dans l'autre terminal.

- Pouvez vous relancez le serveur sur un autre port?

  OUI: sh> ./svr 2001
  à condition que le port 2001 soit libre.

- Pouvez vous lancez le serveur sur le port 2000 sur une autre machine?

  OUI à condition que le port 2000 soit libre sur cette machine.

Q2 Complétez les trous de "hello-clt.c" (ils sont aussi marqués par "...").

(a) Compilez le client.

```
sh> gcc -o clt hello-clt.c
```

(b) Dans le terminal, lancez le client sur le port de votre serveur (on utilisera 2000 dans cet énoncé).

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

Dans le terminal du serveur, on a:

```
sh> ./svr 2000
serveur: envoyé "HELLO" : reçu "OK" : envoyé "FIN" : quitte
```

(c) Dans le terminal, lancez le client en utilisant 127.0.0.1 qui est la conversion en doted format de localhost.

```
sh> ./clt 127.0.0.1 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

(d) Cherchez l'IP et le nom de votre machine sur le réseau de l'ENSIIE:

```
sh> hostname -i
```

```
sh> hostname -a
```

Utilisez ces adresses pour vous connecter à votre serveur.

Q3 Avec votre client connectez vous au serveur de vos voisins et vice-versa.

Exercice 2

Q1 Modifiez le serveur hello-svr.c pour qu'il arrête le dialogue avec le client dès qu'il a envoyé le PDU "HELLO".

aide

Il suffit d'ajouter la ligne "close(cx); `continue`;" après le ligne
"printf("serveur: envoyé \"HELLO\" : "); fflush(stdout);"

(a) Recompilez le serveur puis relancez le.

(b) Lancez un client sur ce serveur:

```
sh> ./clt 127.0.0.1 2000
client: reçu "HELLO" : sh>
```

Que s'est il passé?

Sur le write du PDU "OK", le client a reçu le signal SIGPIPE dont le traitement par défaut est la fin du processus.

(c) Corrigez le client pour qu'il affiche une erreur fatale dans ce cas.

```
sh> ./clt 127.0.0.1 2000
client: reçu "HELLO" :
clt: serveur déconnecté
sh>
```

aide

Il faut indiquer au système d'appeler le gestionnaire "gest_sigpipe" ci-dessous si le signal SIGPIPE arrive (appel système signal)

```
1 void gest_sigpipe(int sig)
2 {
3   fprintf(stderr,"\n%s: serveur déconnecté\n",prgname);
4   exit(1);
5 }
```

(d) Restaurez le serveur initial hello-svr.c.

Q2 Le même problème existe pour le serveur ce qui est plus grave car il ouvre la possibilité de créer un client malveillant qui permet de faire tomber le serveur.

(a) Modifiez hello-clt.c pour en faire un client qui fait tomber le serveur.

aide

En théorie, il suffit que le client se termine juste après la connexion (exit(0)) ainsi le premier write du serveur (le PDU HELLO) générera un SIGPIPE.
Ceci ne fonctionne pas trop car l'écriture du PDU HELLO est trop proche de accept et TCP n'a pas eu le temps de fermer la connexion. Le serveur se rendra compte que la connexion est fermée sur la lecture du PDU OK et là l'erreur est traitée.
Il faut donc essayer le second write (le PDU FIN) pour laisser à TCP le temps d'acheminer la déconnexion. D'où le code:
write(cx,"OK",3); close(cx); exit(0);

Essayez le jusqu'à ce que le serveur tombe. Dans la fenêtre du serveur on doit avoir:

```
sh> ./svr 2000
serveur: envoyé "HELLO" : serveur: reçu "OK" : envoyé "FIN" : quitte
serveur: envoyé "HELLO" : serveur: reçu "OK" : envoyé "FIN" : quitte
serveur: envoyé "HELLO" : serveur: reçu "OK" : sh>
```

(b) En utilisant votre client malveillant, tuez le serveur de votre voisin.

(c) Corrigez le serveur pour qu'au lieu de se terminer sur ce client malveillant, il écrive "client déconnecté".

```
sh> ./svr 2000
serveur: envoyé "HELLO" : serveur: reçu "OK" ; envoyé "FIN" ; quitte
serveur: envoyé "HELLO" : serveur: reçu "OK" ; envoyé "FIN" ; quitte
serveur: envoyé "HELLO" : serveur: reçu "OK" ; client déconnecté
serveur: envoyé "HELLO" : serveur: reçu "OK" ; envoyé "FIN" ; quitte
```

(d) Restaurez le client initial hello-clt.c.

Exercice 3

Pour pouvoir faire des expérimentations plus sérieuses sur notre implémentation du protocole hello de l'exercice précédent (hello-svr.c et hello-clt.c traitant les signaux SIGPIPE), il nous faut un client et un serveur qui simulent des temps de traitements plus ou moins longs.

Q1 Ajoutez un argument optionnel `tp` au client "hello-clt.c", il représente une durée en 1/10 de secondes. Si `tp` est donné, ce nouveau client simulera un traitement de `tp` dixième de seconde. Si il est omis, il vaut zéro et le comportement du nouveau client est identique à l'ancien.

aide

- Changez le test sur argc pour autoriser les valeurs 3 et 4.
- Ajoutez une variable "duree":
  `int duree = argc==3 ? 0: atoi(argv[3])*100000; // en µs`
- Vérifiez que la durée est positive ou nulle.
- Ajoutez "**if** (duree!=0) usleep(duree);"entre la réception du PDU "HELLO" et l'envoi du PDU "OK".

Compilez ce client.

```
sh> gcc -o clt hello-clt.c
```

(a) Lancez le sans durée, il doit fonctionner comme l'ancien client et se terminer quasi instantanément.

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

(b) Lancez le avec une durée de 5 secondes,

```
sh> ./clt localhost 2000 50
client: reçu "HELLO" :
```

puis 5 secondes plus tard

```
sh> ./clt localhost 2000 50
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

Q2 Faites la même modification pour le serveur hello-svr.c, compilez le nouveau serveur et testez le.

Q3 Testez la réactivité d'un serveur avec une durée nulle.

```
sh> ./svr 2000
```

(a) Dans une $1^{ère}$ fenêtre lancez un client rapide (durée 0),

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

la réponse doit être immédiate.

(b) Dans une autre fenêtre lancez un client lent.

```
sh> ./clt localhost 2000 100
client: reçu "HELLO" ;
```

Dans la $1^{ère}$ fenêtre relancez vite le client rapide

```
sh> ./clt localhost 2000
```

Pourquoi le dialogue n'apparait qu'au bout de 10 secondes.

```
Parce que le serveur ne s'occupera du client rapide que
quand il en aura fini avec le client lent (⤻ 10 secondes).
⟸ Pour ce client le serveur n'est pas très réactif.
```

(c) Comment faire un client méchant (qui bloque le serveur Ad vitam æternam).

```
Il suffit de remplacer tout le dialogue dans le fichier hello-
clt.c par "pause()".
```

Q4 Testez la réactivité d'un serveur avec une durée de 10 secondes

```
sh> ./svr 2000 100
```

(a) Dans une $1^{ère}$ fenêtre lancez un client rapide (durée 0),

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

Le texte bleu doit apparaitre au bout de 10 secondes.

(b) Dans une autre fenêtre lancez vite un second client rapide.

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

(c) Quand ce second client terminera son dialoque?

- 10 secondes pour attendre la fin du 1er client.
- 10 secondes pour son dialogue propre.

Au bout d'environ 20 secondes.

Exercice 4

Améliorons la réactivité de notre serveur hello-svr.c.

Q1 Serveur multi-processus.

(a) Copiez le fichier hello-svr.c dans le fichier hello-svr-mp.c puis modifiez ce dernier pour faire un serveur multi-processus.

(b) Compilez le, lancez le et vérifiez qu'il fonctionne bien avec 1 seul client.

(c) Refaites les expérimentations Q3 de l'exercice précédent. Il doit répondre instantanément au second client.

(d) Refaites les expérimentations Q4 de l'exercice précédent. Il doit répondre au bout de 10 secondes au second client au lieu de 20.

(e) Conclusion: On a un serveur réactif.

Q2 Serveur avec pool de 3 processus.

(a) Copiez le fichier hello-svr.c dans le fichier hello-svr-pool.c puis modifiez ce dernier pour faire un serveur avec un pool de 3 processus.

> aide
>
> Il suffit de créer 2 fils avant la boucle "while(1)".

(b) Compilez le, lancez le et vérifiez qu'il fonctionne bien avec 1 seul client.

(c) Refaites les expérimentations Q3 de l'exercice précédent. Il doit répondre instantanément au second client.

(d) Refaites les expérimentations Q4 de l'exercice précédent. Il doit répondre au bout de 10 secondes au second client au lieu de 20.

(e) Conclusion: On a un serveur réactif tant qu'on n'a pas .... de .... clients en .....

Conclusion: On a un serveur réactif tant qu'on n'a pas plus de 3. clients en parrallèle.

Exercice 5

Nos 3 serveurs (hello-svr.c, hello-svr-mp.c, hello-svr-pool.c) sont sensibles à l'attaque "deny of service".

Q1 Copiez hello-clt.c dans hello-clt-deny.c puis modifiez ce dernier pour en faire un client qui bloque le serveur.

> aide
>
> Remplacez tous le dialogue par "pause()".

Compilez le

```
sh> gcc -o clt-deny hello-clt-deny.c
```

Bloquons le serveur naif.

(a) Lancez le serveur naif dans une $1^{ère}$ fenêtre.

```
sh> ./svr 2000
```

(b) Vérifiez qu'il fonctionne bien en lançant dans votre fenêtre principale un client.

```
sh> ./clt localhost 2000
client: reçu "HELLO" : envoyé "OK" : reçu "FIN" : quitte
sh>
```

(c) Dans une 2^{nde} fenêtre lancez le client malveillant.

```
sh> ./clt-deny localhost 2000
```

(d) Vérifiez que le serveur est bloqué en relançant le client dans votre fenêtre principale.

```
sh> ./clt localhost 2000
```

Q2 Bloquez le serveur à pool de processus (svr-poll) de votre voisin. Combien de clt-deny faut il lancer ?

Il faut en lancer un pour chaque processus du pool, donc ici 3.

Q3 Que peut on faire avec ce client malveillant sur une machine où le serveur svr-mp tourne?

Une fork bombe :)

Q4 Pour se protéger d'une telle attaque, il faut sortir de lire_PDU après un timeout si aucun PDU n'est arrivé. Le choix de la valeur de ce

timeout n'est pas évident:

trop petit $\Longrightarrow$ les clients éloignés seront refusés.

trop grand $\Longrightarrow$ ne sert à rien.

Modifiez le serveur hello-svc.c pour qu'il abandonne le client courant si ce dernier ne répond pas au bout de 0.3 seconde.

(a) Regardez la page de manuel de l'appel système poll, puis copiez le fichier "hello-to.h.tpl" du répertoire /pub/ia/reseau dans votre répertoire sous le nom "hello-to.h", puis complètez les trous de la fonction lire_data (ils sont marqués par "...").

(b) Modifiez le serveur hello-svc.c

aide

- Ajoutez l'inclusion des headers de l'appel système poll.
- Incluez "hello-to.h" à la place de "hello-to.h".
- Initialisez la variable to à 0.3 secondes dans le main.

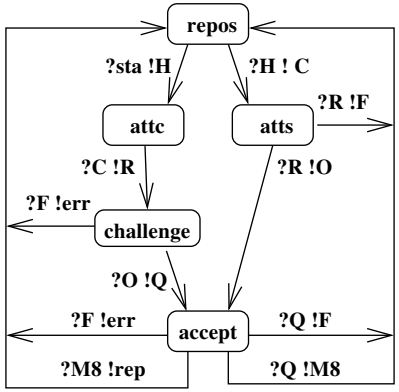
```
to = 300; // 0.3 s en ms
```

(c) Compilez le serveur hello-svc.c puis refaites les expérimentations de la question Q1 (il doit continuer à répondre aux clients "clt" même si on a lancé des clients "clt-deny").

4.2 Implémentation d'un protocole

Le protocole "QRAC" (Question/Réponse Avec Challenge) permet à un client de poser une question à un serveur. Le serveur ne délivre la réponse que si le client est autorisé. Le serveur identifie le client via un challenge.

L'automate du protocole "QRAC" est décrit ci-contre. La description des PDU est donnée ci-dessous:



Les PDU H (Hello), O (Ok) et F (Fin) ont une taille fixe de deux octets:

	0	1
H :	'H'	'\0'
O :	'O'	'\0'
F :	'F'	'\0'

Les PDU C (Challenge), R (Réponse challenge) et Q (Question) ont le même format et font 3 octets.

	0	1 2
C :	'C'	nc
R :	'R'	n
Q :	'Q'	n

Dans ces messages, n et nc sont en binaire naturel, non signés et en big-endian. nc est un nombre crypté, on le décrypte en faisant un xor avec la clé.

Le PDU M8 a une taille fixe de 10 octets. Il permet d'envoyer un message de 1 à 8 caractères.

	0	1 9	
M8 :	'8'	X X X X X X X X '\0'	mess. à 8 char.
M8 :	'8'	X X X X X X '\0' '\0' '\0'	mess. à 5 char.

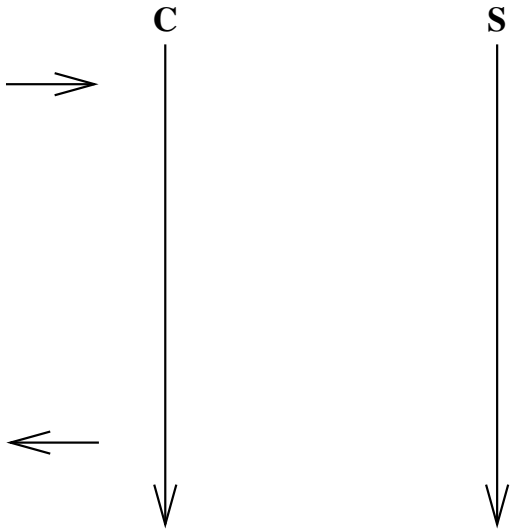
Dans ce message, X est un caractère ASCII non nul (différent de '\0'). L'octet 1 doit être non nul et l'octet 9 doit être nul.
Si le premier octet nul est i ($i \in [2-9]$), le message est constitué des octets 1 à $i-1$ et les octets de i à 9 doivent être nuls.

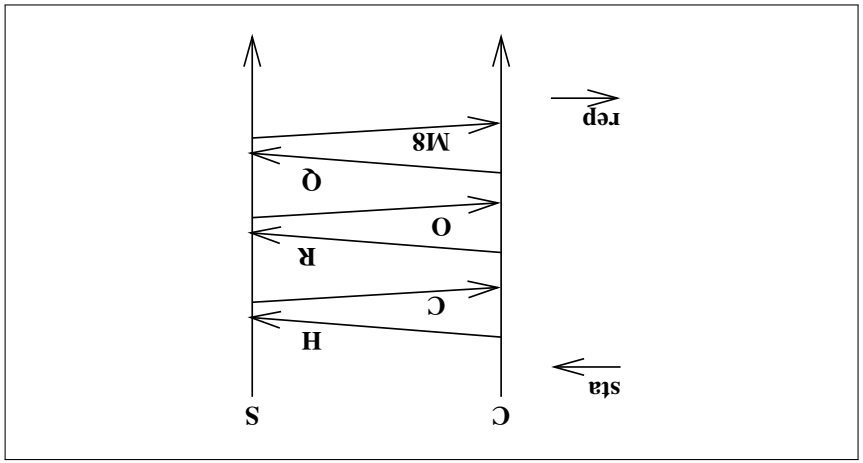
La phase d'identification est la suivante:

- Le serveur tire un nombre N au hasard. Le crypte en NC avec sa clé (NC=N xor clé).
- Le serveur envoie NC au client dans un PDU C (Challenge).
- A la réception du PDU C, le client extrait la valeur M du PDU, la décrypte en MD (MD=M xor clé).
- Le client envoie MD+1 au serveur dans un PDU R (Réponse challenge).
- A la réception du PDU R, le serveur extrait la valeur M' du PDU R.
  - Si M' est différent de N+1, le serveur considère que le client n'est pas autorisé. Il émet un PDU F (Fin) et quitte le dialogue.
  - Si M' est égal à N+1, le serveur considère que le client est autorisé et le dialogue continue.

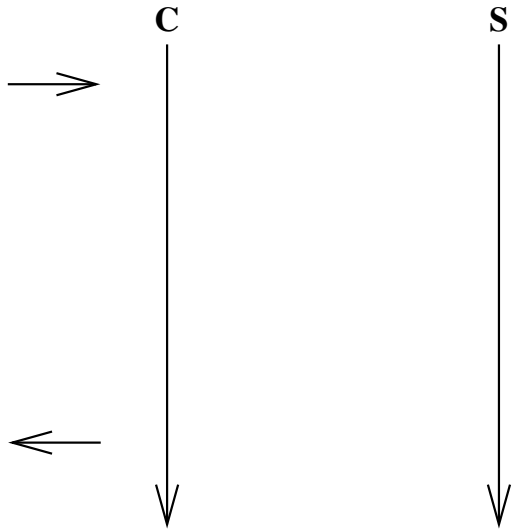
Exercice 1

Q1 Illustrez dans le MSC ci-dessous un échange sans erreur (pas de PDU F).





Q2 Illustrez dans le MSC ci-dessous un échange où le client échoue au challenge.



Exercice 2

L'objet de cet exercice est l'implémentation d'un client QRAC avec la clé 0xAA99.
Recopiez les fichiers "qrac.h.tpl" et "qrac-clt.c.tpl" du répertoire /pub/i-a/reseau dans votre répertoire de travail en enlevant l'extension ".tpl".

Q1 Dans "qrac.h" écrivez/complétez

(a) la macro CLE,
(b) la fonction getTCPsap,
(c) la fonction lire_data,
(d) la fonction lire_PDU (faites tous les PDU, vous en aurez besoin pour l'exercice suivant),
(e) la fonction gen_PDUcrq,

aide

- pdu[0] est type,
- pdu[1] est la partie haute de v soit: (v»8)&0xff,
- pdu[2] est la partie basse de v soit: v&0xff,

(f) la fonction extrait_N_de_PDUcrq (inspirez vous de l'aide précédente).

Vérifiez que votre client compile.

```
sh> gcc -o qrac-clt qrac-clt.c
sh>
```

Q2 Complétez le client qrac-clt.c, la compilation précédente vous indique où sont les trous.
Ses arguments sont dans l'ordre: la machine où réside le serveur, le port TCP du serveur et le numéro de la question à poser.
Pour l'état "attc", utilisez l'opérateur '^' du langage C qui est le xor bit à bit.

Q3 Testez votre client avec le serveur qrac-svr qui se trouve dans le répertoire /pub/ia/reseau. Utilisez l'option -h pour voir ses options.

(a) Lancez le serveur dans une $1^{ère}$ fenêtre.

```
sh> /pub/ia/reseau/qrac-svr 2000
serveur: 12 questions "Prénom de" ; clé=0xaa99.
```

(b) Dialogue complet (service rep)

```
sh> ./qrac-clt localhost 2000 5
réponse à la question 5: Brad
sh> ./qrac-clt localhost 2000 11
réponse à la question 11: Bruce
sh>
```

(c) Dialogue avec question inconnue (service err)

```
sh> ./qrac-clt localhost 2000 20
./qrac-clt: 20 est un numéro de question invalide
sh> ./qrac-clt localhost 2000 12
./qrac-clt: 12 est un numéro de question invalide
```

(d) Dialogue avec un échec au challenge (service err)
Dans une 2^{nde} fenêtre, lancez un serveur avec la clé 0xcafe

```
sh> /pub/ia/reseau/qrac-svr -k cafe 2001
serveur: 12 questions "Prénom de" ; clé=0xcafe.
```

Puis dans votre fenêtre principale

```
sh> ./qrac-clt localhost 2001 5
./qrac-clt: accès au serveur non autorisé (mauvaise clé)
sh> gcc -DCLE=0xcafe qrac-clt.c
sh> ./a.out localhost 2001 5
réponse à la question 5: Brad
sh>
```

Q4 Si le test précédent est concluant, testez votre client avec le serveur qrac-svr2 qui se trouve dans le répertoire /pub/ia/reseau. Il est identique au précédent mais il émet le PDU M8 par petit bout.

Exercice 3

L'objet de cet exercice est l'implémentation d'un serveur QRAC.
Recopiez le fichier "qrac-svr.c.tpl" du répertoire /pub/ia/reseau dans votre répertoire de travail en enlevant l'extension ".tpl".

Q1 Dans "qrac.h" complétez/écrivez

(a) la fonction lire_PDU (si vous ne l'avez pas déjà fait).
(b) la fonction gen_PDUm8,

Q2 Complétez les trous de "qrac-svr.c" (ils sont marqués par "...") et changer la table de questions/réponse ce qui permettra d'identifier votre serveur.

Q3 Compilez votre serveur et testez le avec votre client.

Q4 Vérifiez que votre client et votre serveur fonctionne aussi avec les serveurs et les clients de vos voisins.

4.3 Réalisation d'un proxy