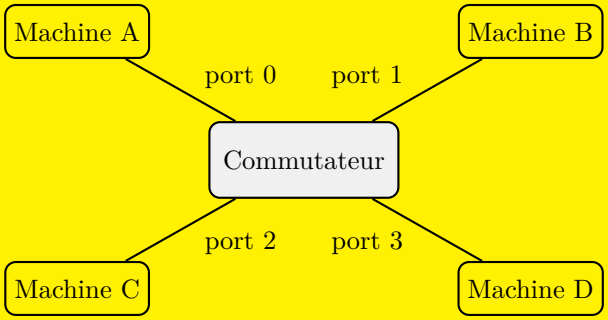
Solution. □

Devoir d'informatique — Ethernet, commutation et fragmentation IPv4

Un commutateur Ethernet qui apprend

Dans cette partie, nous simulons le fonctionnement d'un *commutateur Ethernet* (ou *switch*). Le commutateur relie plusieurs machines : chacune est branchée à l'un de ses *ports*, numérotés consécutivement à partir de zéro. Une *trame* est un message envoyé par une machine source à une machine destinataire. Chaque machine est reliée par un câble à un seul port du commutateur : toute trame qu'elle émet arrive au commutateur par ce port. Nous ne nous intéressons pas encore au contenu de la trame ni à sa représentation en mémoire : les machines sont simplement désignées par des lettres dans l'exemple, puis par des entiers dans le programme. Enfin, un port du commutateur peut être relié à une machine ou **à un autre commutateur**.



Le commutateur entretient une *table d'apprentissage* qui associe à chaque machine connue le port sur lequel elle a été observée. Cette table est initialement vide. Quand une trame de source s et de destination d arrive sur le port p, le commutateur effectue, dans cet ordre, les opérations suivantes :

1. **Apprentissage :** si s n'est pas dans la table, il ajoute l'association $s \mapsto p$; si s y figure déjà, il remplace son ancien numéro de port par p.
2. **Transmission :** si d figure dans la table, le commutateur transmet la trame uniquement sur le port associé à d, sauf si ce port est celui par lequel la trame vient d'arriver ; dans ce cas, il ne transmet rien. Sinon, il transmet la trame sur tous les ports sauf le port d'arrivée p : on parle de *diffusion*.

Voici une exécution de cet algorithme sur trois trames successives :

1. A envoie une trame à C. Elle arrive par le port 0. Le commutateur apprend que A est sur le port 0. Il ne connaît pas encore C : il transmet la trame sur les ports 1, 2 et 3.
2. C répond à A. La trame arrive par le port 2. Le commutateur apprend que C est sur le port 2 et transmet la réponse uniquement sur le port 0.
3. B envoie une trame à C. Elle arrive par le port 1. Le commutateur apprend que B est sur le port 1 et transmet la trame uniquement sur le port 2.

La table d'apprentissage est initialement vide. Ses états successifs dans cet exemple sont :

$$\{\} \longrightarrow \{A \mapsto 0\} \longrightarrow \{A \mapsto 0, C \mapsto 2\} \longrightarrow \{A \mapsto 0, C \mapsto 2, B \mapsto 1\}\,.$$

Dans la suite, nous supposerons qu'une machine n'envoie pas de trame à elle-même et que chaque trame reçue comporte un numéro de port d'arrivée valide. Les capacités des tableaux seront fixées à la compilation ; aucune allocation dynamique ne sera nécessaire.

1 Généralités

Dans cette partie, on utilise le réseau à quatre machines représenté dans l'introduction. Les ports sont numérotés de 0 à 3. La table d'apprentissage est initialement vide.

Question 1.

Les trames suivantes arrivent successivement au commutateur : D envoie à B, puis B envoie à C, puis C envoie à D. Pour chacune d'elles, indiquer le port d'arrivée, les ports sur lesquels cette trame est transmise et le contenu de la table d'apprentissage après son traitement.

Solution. La trame de D arrive sur le port 3 ; la table devient $\{\mathrm{D} \mapsto 3\}$ et la trame sort sur les ports 0, 1 et 2. La trame de B arrive sur le port 1 ; la table devient $\{\mathrm{D} \mapsto 3, \mathrm{B} \mapsto 1\}$ et la trame sort sur les ports 0, 2 et 3. La trame de C arrive sur le port 2 ; la table devient $\{\mathrm{D} \mapsto 3, \mathrm{B} \mapsto 1, \mathrm{C} \mapsto 2\}$ et la trame sort sur le port 3. □

Question 2.

Après ces trois trames, A envoie une trame à B. Indiquer les ports de sortie et la nouvelle table d'apprentissage. Que se passe-t-il ensuite si B envoie une trame à A ?

Solution. La trame de A arrive sur le port 0 : le commutateur apprend $\mathrm{A} \mapsto 0$ et la transmet sur le port 1, où se trouve B. La table contient alors D sur le port 3, B sur le port 1, C sur le port 2 et A sur le port 0. La trame suivante de B à A est transmise uniquement sur le port 0. □

Pour la suite du devoir, chaque machine est désignée par un entier positif ou nul.

En réalité, une trame Ethernet transporte, en plus des informations permettant d'identifier sa source et sa destination, des données. Dans cette première partie, nous ne nous intéressons qu'au fonctionnement du commutateur : nous modélisons donc provisoirement une trame par une structure ne comportant que les identifiants de sa source et de sa destination.

Le contenu transporté par la trame sera introduit et étudié dans une partie ultérieure.

Question 3.

Définir une structure contenant exactement deux champs entiers nommés `source` et `destination`, puis créer l'alias de type `trame` correspondant.

Déclarer ensuite une variable `t` représentant une trame envoyée par la machine 7 à la machine 12.

Solution. On peut écrire :

```
typedef struct {
    int source;
    int destination;
} trame;

trame t = {7, 12};
```

□

Question 4.

La table d'apprentissage peut contenir au plus `CAP` entrées, où `CAP` est une constante définie plus loin. Chaque entrée contient l'identifiant d'une machine et le numéro du port sur lequel elle a été observée.

1. Pour représenter une entrée de la table, définir une structure comportant les champs entiers `machine` et `port`, et lui donner l'alias de type `entree`.
2. Pour représenter le commutateur, définir une structure contenant : un tableau statique `table` d'éléments de type `entree`, un entier `taille` donnant le nombre d'entrées effectivement utilisées, et un entier `nb_ports` donnant le nombre de ports du commutateur. Lui donner l'alias de type `commutateur`.

Solution. Les deux types sont définis ainsi :

```c
typedef struct {
    int machine;
    int port;
} entree;

typedef struct {
    entree table[CAP];
    int taille;
    int nb_ports;
} commutateur;
```

Seules les entrées d'indices 0 à `taille - 1` sont utilisées. □

2 Routage

On utilise les types et les noms de champs définis dans la partie Généralités. Les fonctions reçoivent des pointeurs sur les structures. Tous les pointeurs fournis sont valides. La table contient au plus une entrée par machine et seules ses `taille` premières entrées sont utilisées.

La constante `NB_PORTS` désigne le nombre maximal de ports ; on suppose $1 \leq$ `nb_ports` $\leq$ `NB_PORTS`. Les ports actifs sont numérotés de 0 à `nb_ports - 1`. Tous les numéros de ports fournis ou mémorisés sont valides. On suppose que `stdbool.h` est inclus.

Question 5.

Pour la suite du devoir, on choisit des capacités inspirées d'un commutateur Ethernet courant : 24 ports et une table d'apprentissage pouvant contenir 8192 entrées.

Définir les constantes de préprocesseur `CAP` et `NB_PORTS` correspondant respectivement à la capacité de la table et au nombre maximal de ports.

Solution. Les deux constantes sont définies ainsi :

```c
#define CAP 8192
#define NB_PORTS 24
```

□

Remarque. `CAP` limite le nombre de machines connues ; `NB_PORTS` limite le nombre de ports d'un commutateur (rappelons qu'il peut y avoir plusieurs commutateurs).

Question 6.

Écrire la fonction suivante, qui renvoie l'indice de l'entrée associée à `machine` dans la table du commutateur, ou -1 si cette machine est inconnue. La table ne doit pas être modifiée.

```c
int chercher(commutateur *c, int machine);
```

Solution. On parcourt les entrées utilisées :

```c
int chercher(commutateur *c, int machine)
{
    for (int i = 0; i < c->taille; i++) {
        if (c->table[i].machine == machine)
            return i;
    }
    return -1;
}
```

□

Question 7.

Si une nouvelle entrée doit être ajoutée alors que la table est pleine, le programme doit déclencher une erreur d'assertion. En revanche, si une machine déjà présente dans la table est observée sur un autre port, par exemple après avoir été débranchée puis reconnectée ailleurs, le port qui lui est associé doit être mis à jour. Cette modification reste possible même si la table est pleine, puisqu'elle ne nécessite pas l'ajout d'une nouvelle entrée.

1. Écrire la directive d'inclusion de l'en-tête nécessaire à l'utilisation de `assert`.
2. Écrire la fonction d'apprentissage ci-dessous. Si la machine est déjà connue, son numéro de port est remplacé par `port`. Sinon, une entrée est ajoutée à la fin de la partie utilisée de la table et sa taille est mise à jour.

```c
void apprendre(commutateur *c, int machine, int port);
```

Solution. Une machine connue conserve son entrée :

```c
#include <assert.h>

void apprendre(commutateur *c, int machine, int port)
{
    int i = chercher(c, machine);
    if (i == -1) {
        assert(c->taille < CAP);
        i = c->taille;
        c->table[i].machine = machine;
        c->taille++;
    }
    c->table[i].port = port;
}
```

□

On suppose défini le type `sorties`, représentant les ports sur lesquels une trame doit être transmise.

Il contient un unique champ `port`, tableau statique de `NB_PORTS` booléens. La valeur `port[p]` indique si la trame doit être transmise sur le port p.

Les cases correspondant aux ports inactifs valent toujours `false`.

Solution. On définit le type ainsi :

```c
typedef struct {
    bool port[NB_PORTS];
} sorties;
```

□

Question 8.

Écrire la fonction `transmettre` suivante, qui calcule les ports de sortie sans modifier la table. Si le destinataire est connu, la trame est transmise uniquement sur son port. Si le destinataire est inconnu, la trame est diffusée sur tous les ports actifs sauf le port d'arrivée dans le commutateur (si la trame va de A à B et que A est connecté sur le port 0, le commutateur n'envoie pas de message sur le port 0). Toutes les cases du résultat retourné sont fausses sauf celles correspondants aux ports de sortie.

```
sorties transmettre(commutateur *c, const trame *t, int port_arrivee);
```

Solution. On distingue trois comportements possibles d'un commutateur lorsqu'il reçoit une trame :

- **Diffusion (*flooding*) :** si la destination est inconnue, la trame est transmise sur tous les ports actifs, sauf le port d'arrivée.
- **Transmission unicast :** si la destination est connue et associée à un port différent du port d'arrivée, la trame est transmise uniquement sur ce port.
- **Filtrage (*filtering*) :** si la destination est connue et associée au port d'arrivée, la trame n'est transmise sur aucun port.

On initialise le résultat, puis on distingue les cas :

```
sorties transmettre(commutateur *c, trame *t, int port_arrivee)
{
    sorties s;
    for (int p = 0; p < NB_PORTS; p++)
        s.port[p] = false;

    int i = chercher(c, t->destination);
    if (i != -1) {
        int p = c->table[i].port;
        if (p != port_arrivee)
            s.port[p] = true;
    } else {
        for (int p = 0; p < c->nb_ports; p++)
            s.port[p] = (p != port_arrivee);
    }
    return s;
}
```

□

Question 9.

Écrire la fonction suivante, qui traite une trame venant d'arriver à un commutateur : elle enrichie d'abord la connaissance de la table, puis renvoie les ports de sortie en utilisant des fonctions précédentes. La trame ne doit pas être modifiée.

```
sorties recevoir(commutateur *c, const trame *t, int port_arrivee);
```

Solution. L'ordre des deux opérations est imposé :

```
sorties recevoir(commutateur *c, const trame *t, int port_arrivee)
{
    apprendre(c, t->source, port_arrivee);
    return transmettre(c, t, port_arrivee);
}
```

□

Question 10.

1. Pourquoi `apprendre` reçoit-elle un pointeur sur le commutateur ?
2. Dans `transmettre`, le résultat est une structure locale contenant un tableau. Pourquoi peut-on renvoyer cette structure par valeur sans obtenir un pointeur vers un objet disparu ?

Solution. a) Le pointeur permet de modifier le commutateur de l'appelant, donc de faire des effets de bords.

b) Le retour par valeur copie la structure entière, y compris les éléments de son tableau. Le tableau du résultat reçu appartient à cette structure ; il ne désigne pas le tableau local de la fonction. Il n'y a pas de problème de pointeur fantôme.

□

Question 11.

On souhaite simuler la réception successive de `n` trames par un commutateur.

Les tableaux `trames` et `ports_entree` contiennent chacun au moins `n` éléments. Pour chaque indice i, `ports_entree[i]` donne le port d'entrée de la trame `trames[i]`.

Le tableau bidimensionnel `historique` possède au moins `n` lignes, chacune contenant exactement `NB_PORTS` booléens.

Écrire la fonction suivante, qui traite les trames dans l'ordre des indices à l'aide de `recevoir`. Après le traitement de la trame d'indice i, la case `historique[i][p]` doit indiquer si cette trame est transmise sur le port de sortie p. Les cases correspondant aux ports inactifs doivent valoir `false`. La table d'apprentissage doit évoluer au fil des réceptions.

On suppose $n \geq 0$ et tous les ports d'entrée valides. Si un apprentissage nécessite l'ajout d'une entrée alors que la table est pleine, l'assertion de `apprendre` doit faire échouer le programme. Les trames et leurs ports d'entrée ne doivent pas être modifiés.

```
void simuler(commutateur *c, const trame trames[],
             const int ports_entree[], int n,
             bool historique[][NB_PORTS]);
```

Solution. On traite chaque trame, puis on mémorise ses ports de sortie :

```
void simuler(commutateur *c, const trame trames[],
             const int ports_entree[], int n,
             bool historique[][NB_PORTS])
{
    for (int i = 0; i < n; i++) {
        sorties s = recevoir(c, &trames[i], ports_entree[i]);
        for (int p = 0; p < NB_PORTS; p++)
            historique[i][p] = s.port[p];
    }
}
```

□

2.1 Deux commutateurs reliés

On considère deux commutateurs S_1 et S_2, possédant chacun trois ports actifs, numérotés de 0 à 2 :

— A et B sont reliées respectivement aux ports 0 et 2 de S_1 ;

— C et D sont reliées respectivement aux ports 0 et 1 de S_2 ;

— un câble relie le port 1 de S_1 au port 2 de S_2.

Une trame transmise sur le port de sortie 1 de S_1 est reçue sur le port d'entrée 2 de S_2. Réciproquement, une trame transmise sur le port de sortie 2 de S_2 est reçue sur le port d'entrée 1 de S_1.

Une machine qui reçoit une trame ne la retransmet pas. Les deux tables d'apprentissage sont initialement vides.

Question 12.

Les trames suivantes sont émises successivement : A vers C, C vers A, D vers B, puis B vers D. Chaque trame est entièrement traitée avant l'émission de la suivante.

Pour chacune, indiquer le port d'entrée et les ports de sortie de chaque commutateur qui la reçoit, puis donner les deux tables d'apprentissage après son traitement.

Solution. On applique l'apprentissage avant de déterminer les ports de sortie.

1. **A vers C.** S_1 reçoit la trame sur le port d'entrée 0, apprend $A \mapsto 0$ et la diffuse sur les ports de sortie 1 et 2. S_2 la reçoit sur le port d'entrée 2, apprend $A \mapsto 2$ et la diffuse sur les ports de sortie 0 et 1.
   Les tables sont :
   $$T_1 = \{A \mapsto 0\}, \qquad T_2 = \{A \mapsto 2\}.$$
2. **C vers A.** S_2 reçoit la trame sur le port d'entrée 0, apprend $C \mapsto 0$ et la transmet sur le port de sortie 2. S_1 la reçoit sur le port d'entrée 1, apprend $C \mapsto 1$ et la transmet sur le port de sortie 0.
   Les tables sont :
   $$T_1 = \{A \mapsto 0, C \mapsto 1\}, \qquad T_2 = \{A \mapsto 2, C \mapsto 0\}.$$
3. **D vers B.** S_2 reçoit la trame sur le port d'entrée 1, apprend $D \mapsto 1$ et la diffuse sur les ports de sortie 0 et 2. S_1 la reçoit sur le port d'entrée 1, apprend $D \mapsto 1$ et la diffuse sur les ports de sortie 0 et 2.
   Les tables sont :
   $$T_1 = \{A \mapsto 0, C \mapsto 1, D \mapsto 1\}, \qquad T_2 = \{A \mapsto 2, C \mapsto 0, D \mapsto 1\}.$$
4. **B vers D.** S_1 reçoit la trame sur le port d'entrée 2, apprend $B \mapsto 2$ et la transmet sur le port de sortie 1. S_2 la reçoit sur le port d'entrée 2, apprend $B \mapsto 2$ et la transmet sur le port de sortie 1.
   Les tables sont :
   $$T_1 = \{A \mapsto 0, B \mapsto 2, C \mapsto 1, D \mapsto 1\},$$
   $$T_2 = \{A \mapsto 2, B \mapsto 2, C \mapsto 0, D \mapsto 1\}.$$

□

Question 13.

Après cette séquence, A émet une trame destinée à une machine E dont l'identifiant ne figure dans aucune des deux tables.

Indiquer les ports de sortie de chaque commutateur. Pourquoi S_2 ne renvoie-t-il pas la trame vers S_1 ? La machine E reçoit-elle nécessairement la trame ? Expliquer également pourquoi la table d'un commutateur peut contenir davantage d'entrées qu'il ne possède de ports.

Solution. S_1 reçoit la trame sur le port d'entrée 0 et la diffuse sur les ports de sortie 1 et 2. S_2 la reçoit sur le port d'entrée 2 et la diffuse sur les ports de sortie 0 et 1.

Les tables restent inchangées : l'apprentissage de la source A conserve les associations déjà mémorisées.

S_2 ne renvoie pas la trame vers S_1, car une diffusion exclut toujours le port d'entrée, ici le port 2.

La diffusion ne garantit pas la réception par E : dans le réseau décrit, aucune machine E n'est branchée.

Plusieurs machines peuvent être accessibles par un même port relié à un autre commutateur. Ici, S_1 possède trois ports actifs mais sa table contient quatre entrées ; C et D sont toutes deux accessibles par son port 1. □

Question 14.

Définir une structure et lui donner l'alias de type `sorties_reseau`. Elle doit contenir exactement deux champs, nommés `s1` et `s2`, tous deux de type `sorties`. Ils représentent respectivement les ports de sortie de S_1 et de S_2 pour une même trame.

Solution. On définit le type ainsi :

```
typedef struct {
    sorties s1;
    sorties s2;
} sorties_reseau;
```

□

Question 15.

On considère deux commutateurs S_1 et S_2, désignés respectivement par les pointeurs `c1` et `c2`.

Le port `port_c1_c2` de S_1 est relié au port `port_c2_c1` de S_2. Ces deux ports sont actifs et ce câble constitue l'unique liaison entre les deux commutateurs.

Écrire la fonction suivante, qui traite une trame entrant dans S_1 par le port `port_entree`. Ce port est actif.

La fonction doit effectuer les apprentissages nécessaires et donner les ports de sortie de chacun des deux commutateurs pour cette trame.

Si S_1 ne transmet pas la trame vers S_2, aucun port de sortie ne doit être sélectionné pour S_2.

Utiliser `recevoir`. La trame ne doit pas être modifiée.

```
sorties_reseau recevoir_reseau(commutateur *c1, commutateur *c2,
                               const trame *t, int port_entree,
                               int port_c1_c2, int port_c2_c1);
```

Solution. On traite d'abord la réception de la trame par S_1.

```
sorties_reseau recevoir_reseau(commutateur *c1, commutateur *c2,
                               const trame *t, int port_entree,
                               int port_c1_c2, int port_c2_c1)
{
    sorties_reseau r;

    r.s1 = recevoir(c1, t, port_entree);

    if (r.s1.port[port_c1_c2]) {
        r.s2 = recevoir(c2, t, port_c2_c1);
    }
    else {
        for (int p = 0; p < NB_PORTS; p++)
            r.s2.port[p] = false;
    }

    return r;
}
```

Aucun nouvel appel à `recevoir` pour S_1 n'est nécessaire : lorsque S_2 reçoit la trame par le port `port_c2_c1`, il ne la retransmet jamais sur ce même port.

Si la trame entre dans S_1 précisément par le port qui relie S_1 à S_2, alors `recevoir(c1, t, port_entree)` ne sélectionnera jamais ce même port comme port de sortie. Ainsi `r.s1.port[port_c1_c2] == false` et l'appel à `recevoir(c2, ...)` ne sera pas effectué. Donc notre code gère naturellement ce cas. □

Remarque. La fonction `recevoir_reseau` appartient à notre programme de simulation, exécuté sur un ordinateur : elle représente le traitement effectué par les deux commutateurs et le transport de la trame par leur câble de liaison.

Dans un réseau réel, chaque commutateur traite localement les trames qu'il reçoit. Le câble transporte ensuite les trames transmises vers l'autre commutateur.

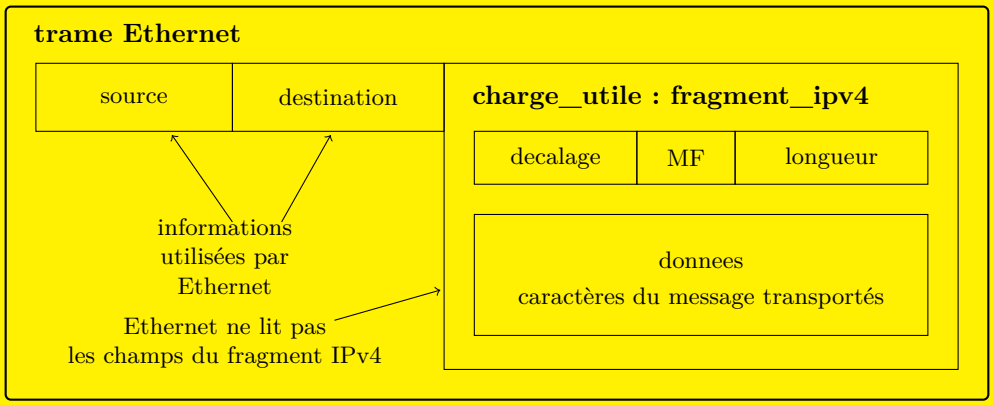
Certains commutateurs possèdent un processeur et un système d'exploitation embarqué. Cependant, la transmission rapide des trames est généralement assurée par des circuits électroniques spécialisés, appelés *ASIC*, que le logiciel configure. Notre programme C modélise donc leur comportement, sans reproduire leur réalisation matérielle.

3 Fragmentation et reconstitution : un modèle inspiré d'IPv4

Une machine souhaite transmettre un *message*, c'est-à-dire une suite de caractères. Dans notre modèle, ce message constitue les données d'un unique *datagramme IPv4*.

Un datagramme IPv4 possède un en-tête et des données. Lorsqu'il est trop grand pour être transporté dans une seule trame Ethernet, IPv4 peut le découper en plusieurs *fragments IPv4*. Chacun de ces fragments possède son propre en-tête IPv4 et est transporté dans une trame Ethernet distincte.

Ethernet transporte ensuite chacun de ces fragments dans une trame distincte. On parle d'*encapsulation* : le fragment IPv4 constitue la charge utile de la trame Ethernet.



Ainsi, on distingue trois objets :
- le *message*, constitué des caractères que la machine veut envoyer ;
- le *fragment IPv4*, qui transporte une partie de ces caractères ainsi que les informations nécessaires à la fragmentation ;
- la *trame Ethernet*, qui transporte un fragment IPv4 sur la liaison.

Ethernet n'interprète pas les informations de fragmentation : pour un commutateur, le fragment IPv4 n'est que la charge utile de la trame.

Remarque. Dans un véritable datagramme IPv4, l'en-tête contient de nombreux champs, notamment les adresses IP source et destination. Nous ne représentons ici que les informations utiles à la fragmentation et à la reconstitution. Nous supposons également qu'un seul datagramme fragmenté est en cours de traitement à la fois. Cette dernière simplification sera discutée à la fin.

Le champ `decalage` contient l'indice, dans le message initial, du premier caractère transporté par le fragment. Comme chaque caractère occupe un octet, ce décalage correspond ici directement à un nombre d'octets [1].

Le booléen `MF` correspond à l'indicateur IPv4 *More Fragments*. Il vaut `true` si d'autres fragments suivent dans le datagramme initial et `false` pour le dernier fragment. Aucun champ ne donne le nombre total de fragments.

IPv4 ne garantit pas que tous les fragments arrivent à destination. Pour la reconstitution, nous supposerons qu'ils sont tous reçus, une unique fois, mais éventuellement dans un ordre différent de celui de leur émission.

Pour faciliter les exemples, on choisit des capacités volontairement petites. On suppose définies les constantes de préprocesseur suivantes :
- `CONTENU_MAX` vaut 16 : un fragment IPv4 transporte au plus 16 caractères du message ;
- `MESSAGE_MAX` vaut 256 : le message initial contient au plus 256 caractères ;
- `TRAMES_MAX` vaut 16 : un datagramme peut être transporté dans au plus 16 trames Ethernet.

Il n'est pas demandé de donner le code de ces déclarations.

Les types utilisés dans cette partie sont déjà déclarés. Il n'est demandé d'écrire aucune déclaration de structure.

Le type `message` possède deux champs :
- `donnees` : tableau statique de caractères de capacité `MESSAGE_MAX` ;
- `longueur` : nombre de caractères utilisés dans ce tableau.

Le type `fragment_ipv4` représente la partie du datagramme IPv4 qui nous intéresse. Il possède quatre champs :
- `donnees` : tableau statique de caractères de capacité `CONTENU_MAX` ;
- `longueur` : nombre de caractères transportés par ce fragment ;

1. En IPv4 réel, le décalage est exprimé en unités de huit octets.

— `decalage` : indice, dans le message initial, du premier caractère transporté ;
— `MF` : indicateur *More Fragments* décrit ci-dessus.

Le type `trame` possède trois champs :
— `source` et `destination` : identifiants Ethernet des machines source et destinataire ;
— `charge_utile` : fragment IPv4 transporté par la trame, de type `fragment_ipv4`.

Enfin, le type `sequence_trames` possède un tableau statique `elements` de capacité `TRAMES_MAX` contenant des trames, et un entier `taille` donnant le nombre de trames utilisées.

Les tableaux de caractères ne sont pas nécessairement terminés par le caractère `'\0'` : seules les cases d'indice strictement inférieur au champ `longueur` sont utilisées.

Les fonctions de routage étudiées précédemment restent utilisables : elles n'accèdent qu'aux champs `source` et `destination` de la trame et n'interprètent pas sa charge utile.

Question 16.

Écrire la fonction suivante, qui affiche les caractères transportés par le fragment IPv4 contenu dans une trame, sans ajouter de saut de ligne. La trame ne doit pas être modifiée.

Peut-on utiliser directement le format `"%s"` ?

```
void afficher_contenu(const trame *t);
```

Solution. On accède aux données du fragment contenu dans la charge utile de la trame :

```
void afficher_contenu(const trame *t)
{
    for (int i = 0; i < t->charge_utile.longueur; i++)
        printf("%c", t->charge_utile.donnees[i]);
}
```

Le format `"%s"` attend une chaîne terminée par `'\0'`. Cette condition n'est pas garantie ici. □

On découpe désormais chaque message non vide en fragments transportant au plus `CONTENU_MAX` caractères. Tous les fragments, sauf éventuellement le dernier, en transportent exactement `CONTENU_MAX`. Par convention pédagogique, un message vide ne produit aucune trame.

Question 17.

Pour chacun des messages de longueurs 0, 16 et 35 caractères, indiquer le nombre de fragments IPv4 nécessaires.

Pour chaque fragment obtenu, donner les valeurs des champs `longueur`, `decalage` et `MF`.

Solution. Avec `CONTENU_MAX` égal à 16, on obtient :
— pour un message de longueur 0 : aucun fragment ;
— pour un message de longueur 16 : un seul fragment,

`longueur` = 16 `decalage` = 0 `MF` = `false`

— pour un message de longueur 35 : trois fragments,

	`longueur`	`decalage`	`MF`
fragment 1	16	0	`true`
fragment 2	16	16	`true`
fragment 3	3	32	`false`

□

Question 18.

Écrire la fonction suivante, qui renvoie le nombre de fragments nécessaires pour transporter un message de longueur `longueur`. Une assertion doit vérifier que cette longueur est comprise entre 0 et `MESSAGE_MAX`.

```
int nombre_fragments(int longueur);
```

Solution.
```
int nombre_fragments(int longueur)
{
    assert(0 <= longueur && longueur <= MESSAGE_MAX);

    int n = longueur / CONTENU_MAX;
    if (longueur % CONTENU_MAX != 0)
        n++;

    return n;
}
```

□

Question 19.

Écrire la fonction suivante, qui construit et renvoie la trame Ethernet transportant le fragment de numéro `numero` du message `m`, envoyé par `source` à `destination`.

Le paramètre `numero` désigne le rang du fragment dans le découpage, à partir de 0 ; ce rang n'est pas transmis. Les champs `decalage` et `MF` du fragment IPv4 doivent être renseignés. Une assertion doit vérifier que le rang demandé est valide.

Les cases inutilisées du tableau `donnees` du fragment devront être initialisées à `'\0'`. Le message ne doit pas être modifié.

```
trame construire_trame(const message *m, int source,
                       int destination, int numero);
```

Solution. Le fragment commence au caractère d'indice `numero * CONTENU_MAX` du message.

```
trame construire_trame(const message *m, int source,
                       int destination, int numero)
{
    int n = nombre_fragments(m->longueur);
    assert(0 <= numero && numero < n);

    trame t;
    t.source = source;
    t.destination = destination;

    int debut = numero * CONTENU_MAX;
    t.charge_utile.decalage = debut;
    t.charge_utile.MF = (numero < n - 1);

    t.charge_utile.longueur = m->longueur - debut;
    if (t.charge_utile.longueur > CONTENU_MAX)
        t.charge_utile.longueur = CONTENU_MAX;

    for (int i = 0; i < CONTENU_MAX; i++)
        t.charge_utile.donnees[i] = '\0';

    for (int i = 0; i < t.charge_utile.longueur; i++)
        t.charge_utile.donnees[i] = m->donnees[debut + i];

    return t;
}
```

Même avec cette initialisation, un fragment plein ne contient pas nécessairement de caractère nul parmi les cases utilisées. □

Question 20.

Écrire la fonction suivante, qui découpe un message et renvoie les trames Ethernet obtenues, rangées par décalages croissants. Utiliser les fonctions précédentes et vérifier par une assertion que le résultat tient dans le tableau. Le message ne doit pas être modifié.

```
sequence_trames decouper(const message *m, int source,
                         int destination);
```

Solution. Voici

```
sequence_trames decouper(const message *m, int source,
                         int destination)
{
    sequence_trames s = {0};
    s.taille = nombre_fragments(m->longueur);
    assert(s.taille <= TRAMES_MAX);

    for (int i = 0; i < s.taille; i++)
        s.elements[i] =
            construire_trame(m, source, destination, i);

    return s;
}
```

Avec les constantes choisies, un message de longueur maximale produit exactement 16 trames. □

Question 21.

Les trames Ethernet correspondant aux fragments d'un même datagramme peuvent être reçues dans un ordre différent de celui de leur émission. La machine de destination doit pouvoir retrouver le message initial.

Écrire la fonction suivante, qui construit le message reconstitué à partir d'une séquence de trames et le fournit comme valeur de retour. La séquence ne doit pas être modifiée.

```
message reconstituer(const sequence_trames *s);
```

On suppose que la séquence contient exactement les trames transportant les fragments d'un seul datagramme découpé selon notre convention :

— les fragments couvrent tout le message, sans trou ni recouvrement, et chacun apparaît une unique fois ;
— leurs décalages sont positifs ou nuls ;
— un unique fragment possède le champ `MF` égal à `false` si la séquence n'est pas vide ;
— tous les fragments sauf le dernier transportent `CONTENU_MAX` caractères ;
— le dernier en transporte entre 1 et `CONTENU_MAX` ;
— une séquence vide représente un message vide.

Pour chaque fragment, vérifier par une assertion que les caractères à copier tiennent dans le tableau du message reconstitué. Les cases inutilisées du résultat doivent être initialisées à `'\0'`.

Solution. Code

```
message reconstituer(const sequence_trames *s)
{
    message m = {0};

    for (int i = 0; i < s->taille; i++) {
        fragment_ipv4 f = s->elements[i].charge_utile;
        int debut = f.decalage;

        assert(debut + f.longueur <= MESSAGE_MAX);

        for (int j = 0; j < f.longueur; j++)
            m.donnees[debut + j] = f.donnees[j];

        if (!f.MF)
            m.longueur = debut + f.longueur;
    }
```

```
17
18      return m;
19  }
```

Les hypothèses garantissent que toutes les positions utilisées du message sont remplies exactement une fois. Le fragment dont le champ `MF` vaut `false` donne la longueur totale, quel que soit l'ordre d'arrivée des trames.

Remarque. Nous supposons que tous les fragments sont reçus et sont sans chevauchement. Sous ces hypothèses, le champ `MF` n'est pas indispensable pour reconstituer le message initial.

Toutefois, ce n'est pas toujours le cas en pratique : certains fragments peuvent être perdus.

Le fragment dont le champ `MF` vaut `false` permet, grâce à son décalage et à sa longueur, de déterminer la longueur totale du message initial. On peut alors vérifier si toutes les données ont été reçues et identifier, le cas échéant, les plages de données manquantes.

De plus, si le fragment ayant `MF = false` est lui-même perdu, le destinataire ne connaît pas la longueur totale attendue. Il sait néanmoins, puisque les fragments reçus ont `MF = true`, qu'il attend encore des données.

□

Question 22.

La machine A envoie plusieurs messages à B, chacun constituant les données d'un datagramme IPv4 distinct. Les trames transportant leurs fragments peuvent arriver entrelacées.

Pourquoi les champs `decalage` et `MF` ne suffisent-ils plus à reconstituer les messages ? Quelle information supplémentaire faudrait-il ajouter à notre modèle ?

Solution. Deux fragments appartenant à des datagrammes différents peuvent avoir le même décalage et la même valeur de `MF`. Les informations présentes dans notre modèle ne permettent donc pas de savoir quels fragments doivent être regroupés.

Il faut ajouter à chaque fragment un identifiant commun à tous les fragments issus d'un même datagramme et permettant de les distinguer de ceux des autres datagrammes en cours de réception.

IPv4 possède précisément un champ d'*identification* utilisé lors de la reconstitution des fragments. Dans le protocole réel, l'identification est interprétée avec d'autres informations de l'en-tête IPv4, notamment les adresses IP source et destination et le protocole transporté. □

FIN DU DEVOIR