MASTER EN SCIENCES INFORMATIQUES

Conception d'une liaison X.25

1e partie

Art, Joseph; Lambion, Patrick

Award date:
1979

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

CONCEPTION D'UNE LIAISON X.25 : 1e partie

--- Présentation générale

--- Analyse et réalisation du niveau trame

Joseph ART

Mémoire présenté
en vue de l'obtention du grade
de Licencié et Maître en Informatique.

Année académique 1978-1979

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

CONCEPTION D'UNE LIAISON X.25 : 1e partie

--- Présentation générale

--- Analyse et réalisation du niveau trame

Joseph ART

Mémoire présenté
en vue de l'obtention du grade
de Licencié et Maître en Informatique.

Année académique 1978-1979

Je tiens à adresser mes remerciements à toutes les personnes qui ont collaboré, de près ou de loin, à mener à bien ce projet.

Je m'adresse plus particulièrement à :

Monsieur Brunin, promoteur de ce travail, qui nous permit de démarrer cette étude.

Monsieur Van Bastelaer qui s'est chargé de la coordination des travaux du groupe XYZ. Ses services et ses conseils m'ont été très précieux.

Monsieur Milgrom qui a mis à notre disposition l'ensemble des outils informatiques requis pour le développement et la mise au point des programmes.

Messieurs Laloux et Lobelle qui ont coordonné les activités de l'unité ELHY au sein du groupe XYZ, apportant ainsi une solution pour le développement d'un matériel approprié aux besoins de cette étude.

Messieurs Beudin et Zone qui ont consacré leur mémoire à la réalisation de ce matériel (basé sur l'emploi d'un micro-processeur) et l'étude du niveau physique du protocole X.25 .

Monsieur Lambion qui a consacré son mémoire à l'implémentation du niveau paquet et de la liaison avec les utilisateurs.

J. ART

TABLE DES MATIERES

I. Introduction

I.I Avant-propos

Le mémoire traite d'une partie d'un projet réalisé en équipe par les membres du groupe XYZ.
Ce groupe résulte d'une collaboration entre l'unité d'informatique et l'unité ELHY de l'Université Catholique de Louvain et l'Institut d'Informatique des Facultés Notre Dame de la Paix de Namur.
Il s'occupe de l'étude des problèmes relatifs à l'interconnexion d'ordinateurs.
Cette année l'étude fut consacrée à la conception d'une liaison X.25 et aux problèmes relatifs à l'utilisation d'une telle liaison pour interconnecter deux ordinateurs.
Quatre étudiants ont consacré leur mémoire à cette réalisation, bien que chaque partie fut développée séparément, les réunions régulières du groupe XYZ ont permis un suivi des travaux et la coordination des activités.
Dans ce mémoire, je me suis efforcé de présenter une analyse du problème qui soit le moins possible lié aux caractéristiques du matériel sur lequel seront exécuté les programmes. De cette façon, ce travail pourra être utile pour d'autres applications qui pourraient être développées par la suite par le groupe XYZ.
Quelques extraits de programme sont donnés dans le chapitre 4 uniquement dans le but d'illustrer la réalisation. Les personnes intéressées par la version complète des programmes pourront obtenir de plus amples renseignements en me contactant ou en s'adressant au laboratoire de l'Unité d'Informatique de l'UCL.(Bâtiments du Cyclotron, Louvain-La-Neuve.).

I.2 But du travail

Le but de notre travail se définit comme suit : concevoir et réaliser une liaison X.25 afin de se familiariser avec les concepts relatifs aux protocoles d'échanges de données et en particulier le protocole standard d'accès aux réseaux de commutation de paquet qu'est X.25 .

Dans un premier temps, nous n'avons pas voulu une réalisation qui soit avant tout performante, mais plutôt une réalisation bien structurée.
C'est pourquoi une démarche modulaire a été suivie avec des définitions fonctionnelles aussi précises que possible pour chaque module ainsi que pour les interfaces qui les relient. Il est important de noter que notre objectif est de développer une réalisation pratique en utilisant comme référence l'avis X.25 du CCITT (2).
En aucun cas nous n'avons voulu critiquer l'ensemble des principes qui y sont exposés, ce genre de travail sortant du cadre de nos préoccupations.

I.3 Plan commenté du mémoire

Après l'introduction du chapitre I nous avons :

Le chapitre 2 a pour but de situer l'avis X.25 .
Les notions de réseaux, architecture, commutation et paquet sont introduites dans le paragraphe 2.I .
Le rôle des organismes de standardisation est donné en 2.2 ainsi qu' un commentaire sur les plus connus d'entre eux.
Le paragraphe 2.3 introduit la notion de protocoles, leur application dans le domaine des transmissions de données et en particulier pour l'accès aux réseaux.
Enfin, en 2.4, l'avis X.25 est situé vis à vis de ce qui a été exposé précédemment.

Le chapitre 3 est consacré à l'exposé général du problème qui nous était posé. La première partie (3.I) donne un exposé de l'avis X.25 . Cet exposé à pour but de dégager les principes fondamentaux relatifs à l'échange des données et de les illustrer par quelques exemples.

Après un exposé dans le paragraphe 3.II de la structure de X.25, les 3 niveaux du protocole sont décrits séparément. Pour les niveaux physiques et paquets, un exposé plus détaillé est disponible dans les mémoires de G. Zone et Y. Beudin pour le niveau physique et P. Lambion pour le niveau paquet.
Le niveau trame quand à lui est suffisamment détaillé que pour pouvoir faire le lien avec le chapitre 4 consacré à son implémentation.
La seconde partie du chapitre 3 donne un aperçu des fonctions qui peuvent être remplies par un niveau supérieur au protocole X.25.
Des propositions pratiques sont données au sujet de ce niveau dans le mémoire de P. Lambion.
La troisième partie du chapitre 3 est consacrée à la découpe du travail et à sa répartition entre les différentes personnes qui ont participé à cette réalisation.
On y donne les spécifications des tâches imparties à chacun.
On y expose les contraintes de réalisation, les moyens dont on dispose et les solutions envisagées pour résoudre ces différents problèmes.

Le chapitre 4 traite de l'implémentation du niveau trame.
La première partie (4.I) donne la structure générale du logiciel développé pour remplir les fonctions du niveau trame et assurer les échanges avec le niveau paquet.
Le paragraphe 4.2 donne les spécifications fonctionnelles des interfaces.
Tout d'abord pour la communication trame-paquet et ensuite pour les échanges entre trame 2 et trame I.
La méthode d'analyse utilisée pour définir l'automate qui gère la procédure du niveau trame est exposée au paragraphe 4.3 .
Le reste du chapitre 4 expose la programmation des différents modules et la démarche suivie pour la mise au point.

Le chapitre 5 donne en guise de conclusions l'état d'avancement du travail, les difficultés rencontrées et comment envisager une suite à ce qui a été fait.

2. Situation de l'avis X.25

2.I Les réseaux de transmission de données

Un réseau se compose de deux types d'éléments, qui sont : d'une part, les éléments manipulateurs de données tels que les ordinateurs, les terminaux, les concentrateurs, les multiplexeurs, etc.; et, d'autre part, les moyens de transmission de données à distance tels que : les lignes téléphoniques, les faisceaux Hertziens, les fibres optiques... Les réseaux de transmission de données ont deux sortes d'utilisateurs : les grandes sociétés qui, si c'est nécessaire, mettent en place leur propre réseau, le gèrent et le financent elles-même; et, les utilisateurs plus modestes qui n'y voient qu'un service externe auquel ils désirent faire appel.

Les premiers sont concernés avec l'ensemble des problèmes de conception matérielle et logicielle, car tout est à faire et à construire, tandis que les seconds peuvent à la limite ne se préoccuper que des règles d'accès au réseau et considérer celui-ci comme une boîte noire offrant un ensemble de services bien définis.

2.I.I. L'architecture des réseaux

Une caractéristique importante d'un réseau est son architecture.

Celle-ci définit le rôle de chaque élément, la répartition des fonctions de gestion, d'acheminement et de contrôle.

La figure 2.I nous montre trois architectures typiques.

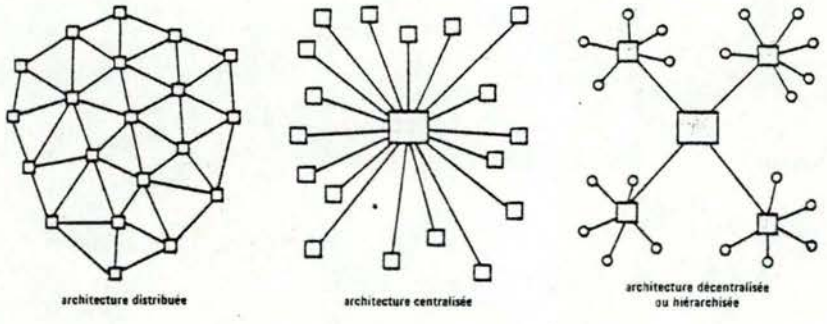


Fig. 2.I - Architectures de réseaux .

L'organisation centralisée

Le contrôle du réseau est entièrement assuré par le noeud central. Toutes les communications passe par celui-ci.
Exemple : un ordinateur et l'ensemble de ses terminaux.

L'organisation décentralisée ou hiérarchisée

L'intelligence est ici distribuée sur plusieurs niveaux. Ceci permet de décharger l'ordinateur central d'un certain nombre de tâches pouvant être résolues à un niveau inférieur.
Exemple : un ordinateur central relié à plusieurs mini- ordinateurs capables de supporter chacun un ensemble de terminaux. Chaque mini effectue un pré-traitement des informations saisies au niveau terminal avant de les envoyer vers l'ordinateur central.

L'organisation distribuée

Dans un réseau distribué, la notion de hiérarchie disparaît. Tous les noeuds remplissent des fonctions semblables. Les noeuds sont interconnectés de façon à pouvoir pallier à la défaillance de l'un ou l'autre élément de transmission

La plupart des réseaux d'une certaine importance ont une architecture combinée faisant appel aux trois formes d'organisation décrites ci-dessus.

2.I.2. La fonction commutation

Les ressources du réseau sont partagées entre les divers utilisateurs. La façon dont sont allouées ces ressources est définie par la fonction commutation.
Il existe trois façons de faire la commutation et elles sont illustrées à la figure 2.2.

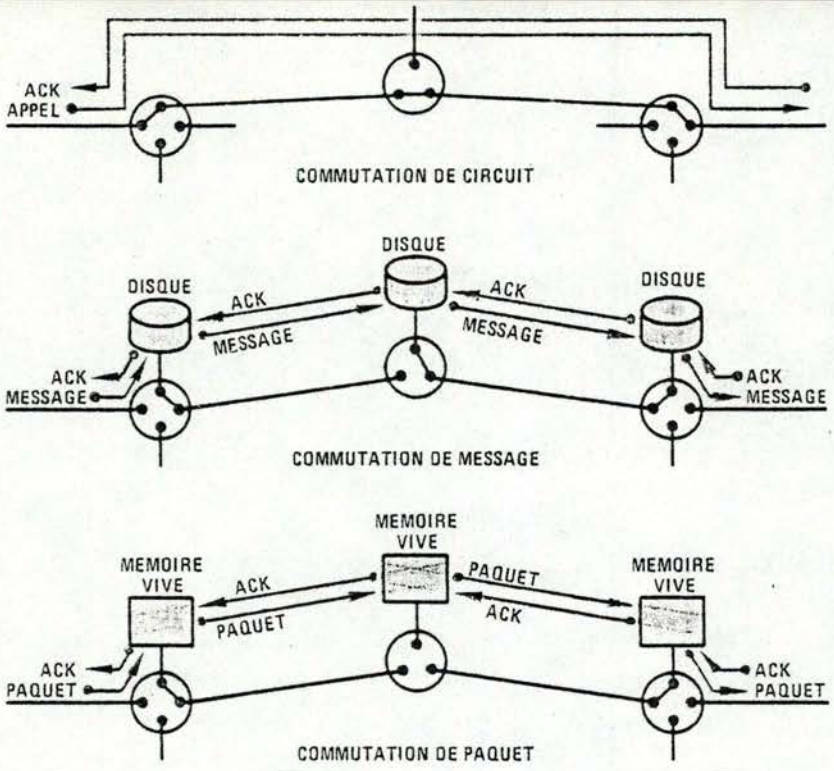


Fig. 2.2 - Les modes de commutation .

La commutation de circuits

Un réseau qui met en oeuvre la commutation de circuits relie deux utilisateurs d'une manière physique et réelle pendant toute la durée des échanges. Ceci s'explique par la lenteur d'établissement d'une communication qui peut nécessiter plusieurs secondes. L'inconvénient majeur de la commutation de circuit réside dans le sous-emploi de l'infrastructure du réseau.

Les temps morts dans l'utilisation d'un circuit physique ne pouvant pas être récupérés pour être affectés à un autre utilisateur. Cette technique ancienne et relativement simple est utilisée dans les réseaux téléphoniques classiques également exploités pour les transmissions de données à basse vitesse.

La commutation de messages

Avec ce mode de transmission, l'unité véhiculée est le message; c'est-à-dire, l'unité logique d'information du point de vue de l'utilisateur.

Par exemple : un fichier, un programme, une ligne ou une page d'impression.

Le message traverse le réseau en passant par différents points, les noeuds, où il est stocké temporairement en attendant qu'un circuit devienne libre.

Le trajet émetteur-récepteur est donc découpé en étapes élémentaires, au terme desquelles, le message est stocké sur mémoire de masse.
C'est le principe du "Store and Forward". Dans ce cas-ci, les circuits sont partagés entre les messages des différents utilisateurs.
La longueur, parfois importante, des messages nuit à la fluidité du transfert des informations ce qui peut introduire des délais assez importants.
Le réseau télex est un exemple de réseau de dimension mondiale utilisant la commutation de messages.

La commutation de paquets

Les données transmises sont découpées en paquets de longueur fixe ou variable, mais dont la taille maximum est fixée par l'administration du réseau (I00 à 250 caractères environ).
Le paquet comporte une entête qui identifie l'adresse du destinataire, celle de l'émetteur et la nature des services demandés ainsi qu'un champ contenant des informations de contrôle.
Le principe du "Store and Forward" est également appliqué. Les paquets sont transmis de noeud en noeud avec détection et récupération des erreurs à chaque étape.

La méthode d'acheminement des paquets à travers le réseau est appelée routage. Il en existe deux familles : le routage non adaptable et le routage adaptable.

Dans la première méthode, les règles d'acheminement sont fixes.

Exemples : -Le routage fixe: les paquets d'une même communication suivent tous le même chemin.
-Le routage aléatoire: la direction prise par un paquet lorsqu'il quitte un noeud est choisie au hasard et ce jusqu'au moment où il atteint sa destination.

Ces méthodes qui ne tiennent pas compte de l'évolution de l'état du réseau sont simples mais peu efficaces dans

un environnement évolutif (variations de la charge, modifications dans la structure du réseau, variations des capacités de transfert, etc.).

Le routage adaptable est, par contre, beaucoup plus sophistiqué. Le chemin suivi par un paquet est déterminé selon le volume du trafic et les conditions du réseau à un moment donné. Ceci nécessite une tenue dynamique des tables de routage soit par un système centralisé, soit par un système distribué sur l'ensemble des noeuds.
De telles méthodes permettent de répondre aux évolutions d'état du réseau et de sa charge. La gestion est cependant plus lourde et constitue un supplément de charge pour les ressources du réseau. Le routage adaptable se justifie lorsque performances et utilisation maximum des ressources sont visés et que le réseau se veut ouvert à une grande diversité d'utilisateurs.

Deux types de service sont offerts par les réseaux à commutation de paquets :

- Le circuit virtuel qui nécessite une procédure d'appel pour l'établissement d'un circuit logique réalisant une communication virtuelle entre deux utilisateurs. L'ordre des paquets est respecté et un contrôle de flux est assuré de bout en bout sur chaque circuit virtuel. De plus, les débits émission-réception sont adoptés. Un tel service nécessite une grande diversité des informations de contrôle portée par le paquet.

- Le datagramme, par contre, est un simple service de transmission de paquets. Chacun d'entre eux est acheminé sans tenir compte de l'existence des autres. Ils peuvent donc être reçus dans n'importe quel ordre. Le datagramme ne permet pas le contrôle de flux sélectif de bout en bout.
  En résumé, un service simple mais qui risque d'être insuffisant pour beaucoup d'utilisateurs.

2.I.3. Les besoins des utilisateurs

Un réseau peut servir à mettre en communication des utilisateurs dont les besoins et les moyens sont très diversifiés.

-Les volumes à transmettre sont très variables, de quelques milliers de caractères à plusieurs millions lorsqu'il s'agit de gros fichiers.

-Le délai de transmission, c'est-à-dire l'intervalle de temps qui sépare l'émission d'un caractère de sa réception, doit être minimal.

-Le taux d'erreurs doit être aussi réduit que possible. Les erreurs de transmission qui ne peuvent pas être corrigées par le réseau, doivent être au moins détectées et signalées aux utilisateurs.

-N'importe quel type d'équipement terminal ou ordinateur hôte, doit pouvoir être connecté.
Le réseau doit être transparent sur le plan des codes afin de permettre la transmission de n'importe quelle suite binaire.
Il doit assurer la liaison entre les équipements qui travaillent à des vitesses différentes.

-Les ressources du réseau doivent être partagées afin d'en assurer une utilisation maximum malgré le caractère intermittent des communications.
Le coût d'utilisation doit être le plus faible possible.

2.I.4. Les réseaux de commutation de paquets

Quelques exemples de réalisation :

- I968 : lancement du réseau Arpa aux Etats-Unis (réseau privé universitaire).
- I97I : Arpa permet le raccordement d'autres utilisateurs.
- I975 : en France, RCP (réseau expérimental PTT) et Cyclades (IRIA) sont opérationnels.
- I977 : ouverture officielle du réseau public Datapac au Canada.

- 1978 : inauguration du réseau public français Transpac.

2.2. Les organismes de standardisation

Le besoin de normalisation est rapidement apparu dans le domaine des réseaux et des transmissions de données en général.
Les équipements qui composent un réseau sont multiples: terminaux, unités centrales, multiplexeurs, concentrateurs, modems, etc.
Les supports de transmission ne se limitent pas aux lignes téléphoniques mais comprennent aussi les liaisons par satellites, les fibres optiques, etc.

L'architecture des réseaux est très diversifiée (centralisée, distribuée, en étoile, en anneau, etc.).

Les utilisateurs ont des besoins très divers.

Un certain nombre d'organismes sont apparus et ont établi des normes.
Les plus connus sont :

- Le CCITT (Consultative Commitee for International telegraphy and telephony).
  Il traite du transport des données, émet des avis de normalisation auquels se réfèrent les administrateurs des télécommunications (PTT) des pays membres du comité.

- L'ISO (International Organization for Standardisation); groupement des organismes de normalisation de nombreux pays tels que :
  ANSI aux Etats-Unis
  AFNOR en France
  IBN en Belgique, etc.

- L'EIA (Electronique Industries Association)

qui se préoccupe essentiellement de normes physiques.

2.3. Les protocoles standards

La normalisation des réseaux s'applique à quatre niveaux distincts.
Les différents niveaux sont illustrés à la figure 2.3.

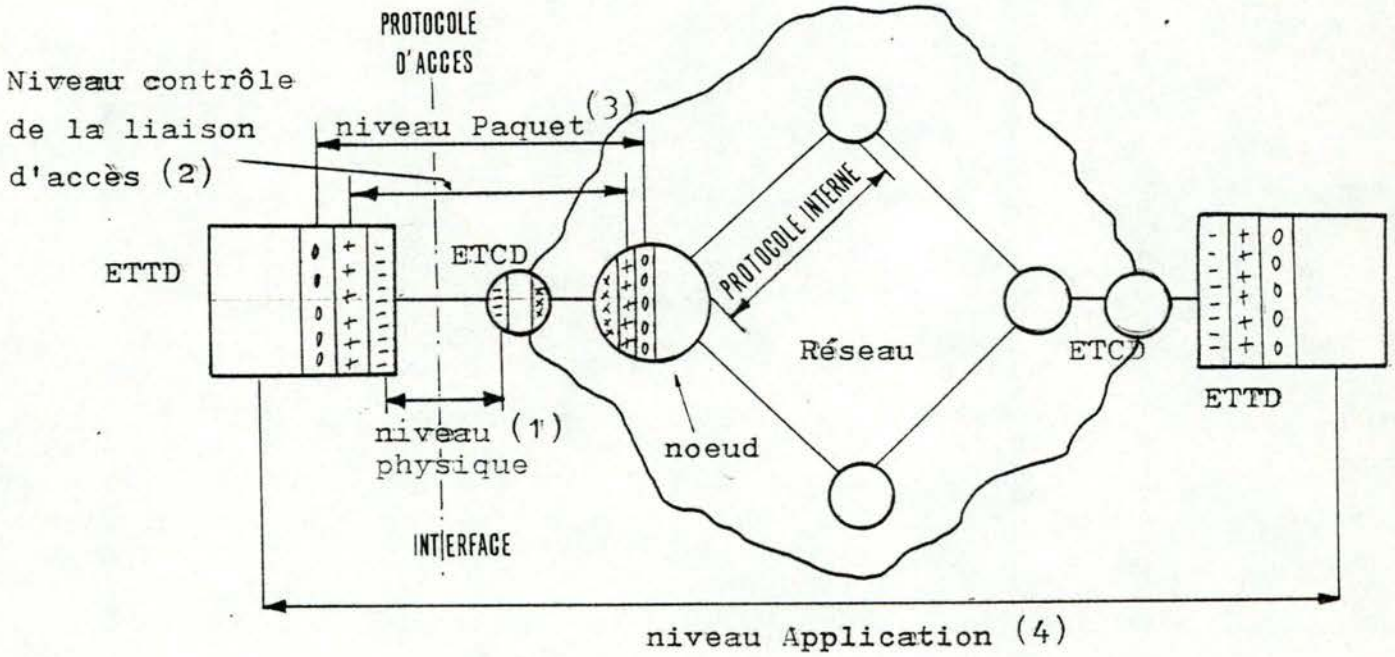


Fig. 2.3 - Les différents niveaux de protocole .

2.3.I. Les protocoles physiques de connexion définissent comment les équipements tels que : équipement terminal de données, équipement de connexion, modem, multiplexeur, concentrateur, etc. ;sont liés entre eux et au réseau.
Ces règles physiques de connexion sont produites par le CCITT et l' EIA.
Les avis du CCITT sont répartis en deux séries:

- la série V. pour la transmission de données sur le réseau téléphonique (I);
- la série X. pour les réseaux publics de transmission de données (2).

2.3.2. Les protocoles de contrôle de lien définissent un ensemble de règles et de conventions ayant pour but de se comprendre correctement entre interlocuteurs distants (utilisateurs du lien).
Ils assurent différentes fonctions telles que :

- la synchronisation de l'émetteur et du récepteur.
- la détection et la récupération des erreurs de transmission.
- la régulation du débit de l'émetteur par le récepteur
- une procédure d'adressage pour les liaisons multipoint.
- assurer la transparence afin de permettre le transfert de n'importe quelle séquence binaire.

Les procédures les plus connues sont :

- BSC (Binary Synchonous Communications) d'IBM, date de I960, prévue pour le multipoint en mode Half-duplex.
  Elle est orientée caractère et permet le transfert de blocs de longueur variable.

- SDLC (Synchonous Date Link Communication) annoncé en I973 par IBM, est une procédure orientée bit, half ou full-duplex.
  Plus s ouple et plus efficace que BSC, elle est un élément de base des réseaux SNA d'IBM.

- HDLC (High level Data Link Control) proposé par l'ISO en I974 et similaire à SDLC tout en étant plus général.
  La structure de trame est décrite dans l'avis ISO/ DIS 3309.2 (5); les éléments de procédure dans ISO/ DIS 4335 (6).

- ADCCP (Advanced Data Communications Control Procedure); version américaine de HDLC développée par l'ANSI .

- DDCMP (Digital Data Communications Message Processor), protocole propre à DEC et utilisé pour les réseaux développés par ce constructeur.

- D'autres protocoles sont aussi proposés par les autres constructeurs; ils sont tous plus ou moins ressemblants à BSC ou HDLC .

La liaison d'accès à un réseau pour un équipement terminal de traitement de données sera gérée par une procédure semblable à celle décrite ci-dessus.

2.3.3. Les protocoles au niveau paquet

Les protocoles au niveau paquet sont une extention du niveau précédent, ils précisent la méthode par laquelle les messages sont découpés en paquets et acheminés d'un point à un autre d'un réseau.
Pour l'utilisateur du réseau, c'est le protocole paquet au niveau du lien d'accès qui est important.
Ce protocole fournit des fonctions spécifiques à la commutation de paquet.
Les protocoles différent suivant le type de services offert par le réseau.
Un protocole de datagramme n'est pas le même qu'un protocole de circuits virtuels. Les fonctions requises pour la gestation de circuits virtuels seront étudiées en détails dans le chapitre 3.

2.3.4. Les protocoles au niveau "application"

Le dernier niveau de normalisation est celui qui pose le plus de problèmes. En fait, ce niveau concerne les utilisateurs du réseau et non le réseau qui est transparent vis à vis des informations transmises.
Si pour des applications privées, il n'est pas nécessaire

d'imposer de normalisation, il n'en est plus de même pour des applications à vocation de service publique, tel que : banques de données où centres de traitement de l'information.

Dans ce domaine, tout reste à faire.

2.4. L'avis X. 25

Emis en septembre I976 par le CCITT, X. 25 (2) résulte de l'expérience accumulée sur les différents réseaux utilisant la communication de paquets (Arpa aux USA, Cyclades en France, EPSS en Angleterre).

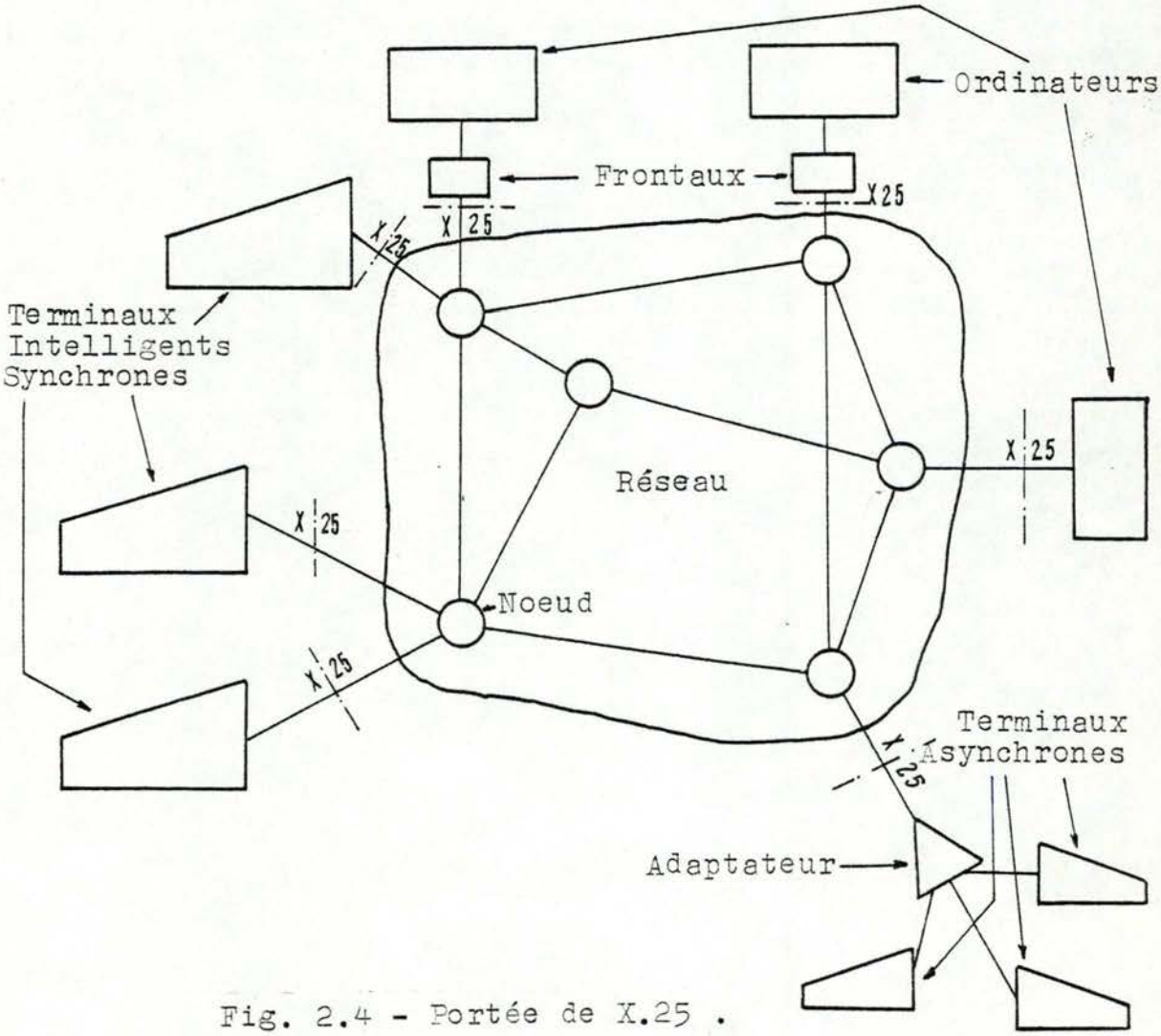


Fig. 2.4 - Portée de X.25 .

L'avis X. 25 (2) est intitulé :
"Interface entre équipement terminal de traitement de données (ETTD) et équipement de terminaison du circuit de données (ETCD) pour terminaux fonctionnant en mode paquet, raccordés à un réseau public de transmission de données".

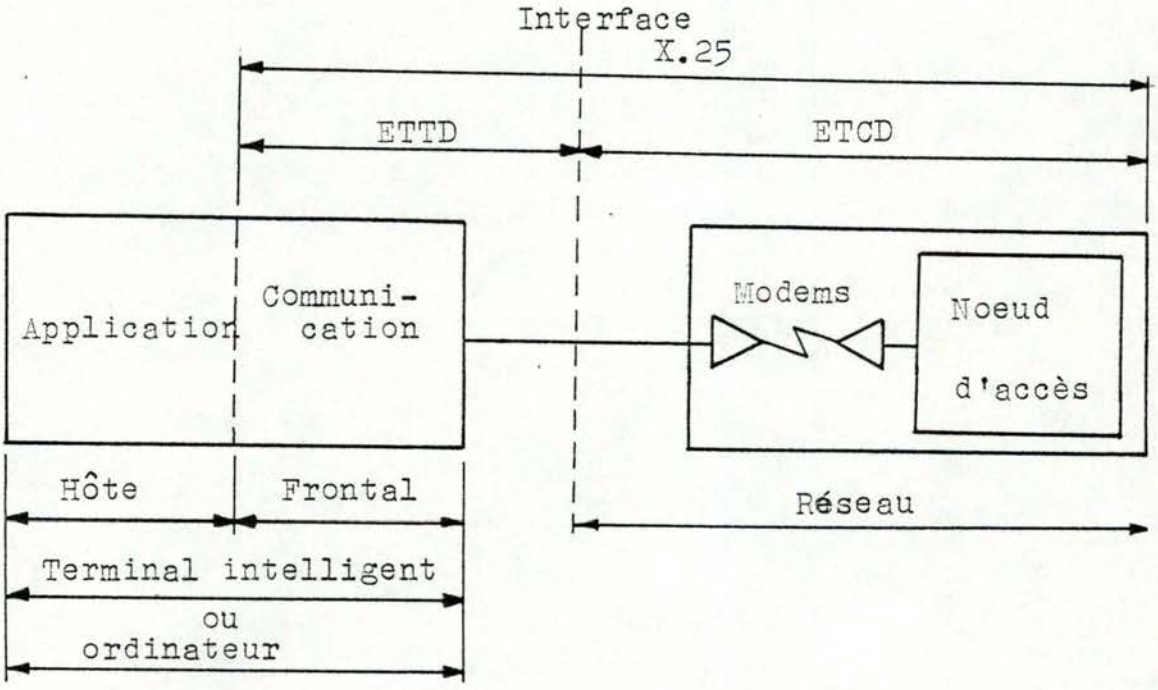


Fig. 2.5 - Jonction X.25 .

La figure 2.4 nous montre ces différents éléments.
L'ETTD est l'équipement de l'abonné qui produit les paquets, les envoie vers le réseau, reçoit les paquets provenant du réseau et en extrait l'information. Il s'agit physiquement d'un ordinateur hôte, d'un processeur frontal ou d'un terminal intelligent.
L'ETCD est l'équipement du réseau auquel est connecté l'ETTD. X.25 définit donc l'interface local entre l'utilisateur et le réseau.
Les fonctions offertes par le réseau sont décrites mais pas leur implémentation.

La normalisation porte sur trois niveaux :

- définition des éléments de jonction physique
- description d'une procédure de gestion de la liaison d'accès
- définition d'un protocole de niveau paquet dans le cadre d'un service de type circuits virtuels.

Aucune hypothèse n'est faite sur la réalisation interne du réseau qu'il faut considérer ici comme une boîte noire.

L'avis X.25 a fait l'objet d'une révision en I977. Les compléments apportés par cette révision seront exposés avec l'avis original de I976 dans le chapitre 3.

= = = = = = = = = = =

3. Exposé général du problème

3.I Exposé de l'avis X.25

3.I.I Structure de X.25

L'avis X.25 définit de façon indépendante trois niveaux distincts pour gérer l'échange d'information entre un ETTD et un ETCD.
Cet échange doit permettre l'utilisation des services de transmission de données en commutation par paquets fournis par certains réseaux.

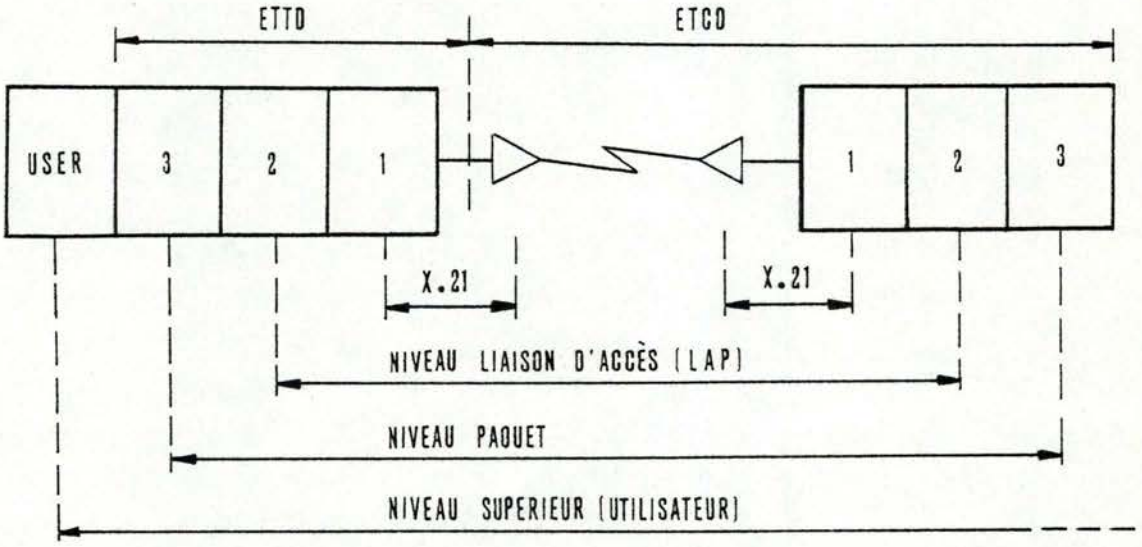


Fig. 3.I - Niveaux de protocole à l'interface ETTD/ETCD .

Ces trois niveaux sont les suivants :

Niveau I -Les caractéristiques physiques, électriques, fonctionnelles et de procédure pour établir, maintenir et déconnecter la liaison physique entre l'ETTD et l'ETCD.

Ce niveau est également appelé "niveau physique" ou "niveau bit".
Il donne les spécifications de la jonction

physique qui sont :

-les caractéristiques électriques des circuits de jonction telles que : les niveaux de tension, les impédances de charge, etc.

-les caractéristiques mécaniques des connecteurs et des câbles.

-les conventions de câblage

-les procédures de connection, déconnection et de transmission.

Vu du niveau directement supérieur, il présente un circuit de transmission de signaux digitaux de type série, synchrone, bidirectionnel simultané et de point à point.

Niveau 2 -La procédure d'accès à la liaison pour l'échange de données sur la liaison entre l'ETTD et l'ETCD.

Ce niveau est également appelé "niveau trame". Il spécifie une procédure de contrôle d'une liaison qui, d'une part, assure la synchronisation de l'ETTD et de l'ETCD; d'autre part, fixe les règles de détection et de récupération des erreurs de transmission.

Des méthodes permettant la régulation des débits sur la liaison pendant le transfert de données sont également fournies à ce niveau. Cette procédure utilise les principes et la terminologie de la procédure de commande de chaînon à haut niveau (HDLC).

Le circuit physique offert par le niveau I (sujet à des erreurs de transmission) est géré par le niveau 2 de façon à réaliser une liaison ETTD-ETCD libre d'erreur de transmission pouvant être exploitée par le niveau supérieur.

Niveau 3 - Le format de paquet et les procédures de commande pour l'échange des paquets contenant des informations de supervision et des données de l'usager entre l'ETTD et l'ETCD.

Ce niveau appelé "niveau paquet" est le plus haut niveau décrit dans l'avis X.25. Il spécifie la façon dont les données de l'utilisateur sont mises sous forme de paquets qui seront transmis sur la liaison offerte par le niveau trame.
Le niveau paquet réalise un multiplexage de la liaison trame en un certain nombre de voies logiques.
Pour le réseau, établir une communication entre deux ETTD revient à associer une voie logique de l'un avec une voie logique de l'autre. La liaison logique ainsi établie est appelée "circuit virtuel" (virtuel car les paquets empruntent des circuits physiques partagés).
Cette association peut être permanente (circuit virtuel permanent) ou temporaire (circuit virtuel commuté).

Sur chaque circuit virtuel est assuré un contrôle de flux c'est-à-dire :

- -d'une part les paquets sont délivrés dans l'ordre avec lequel ils ont été émis;
- -d'autre part chacun des deux ETTD communiquant par un circuit virtuel peut régler le débit d'émission de l'autre.

Des paquets de service, ne transportant pas d'information utilisateur sont utilisés notamment pour transmettre les acquittements ou établir un circuit virtuel commuté.

Fig. 3.2 - Cheminement de l'information à travers les niveaux du protocole X.25 .

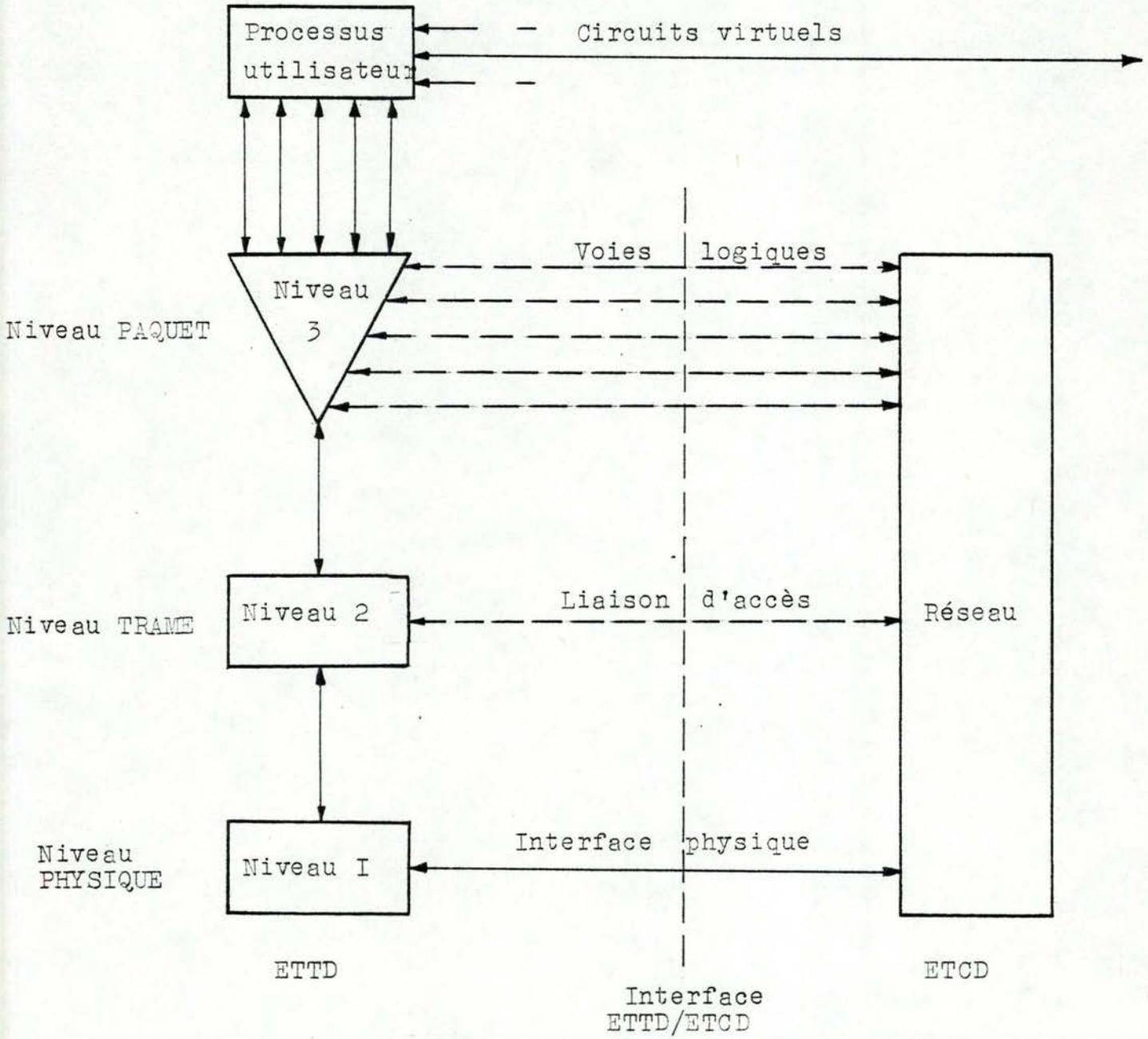


Fig. 3.3 - Structure logique de l'interface X.25 .

Chaque niveau utilise les services offerts par le niveau directement inférieur, ce qui constitue la seule forme de dépendance d'un niveau par rapport à un autre. Les spécifications des fonctions réalisées par un niveau sont bien définies, la façon de les implémenter ne doit pas être précisée ici et est indépendante pour chaque niveau.

Le cheminement de l'information à travers ces différents niveaux est représenté par la figure 3.2. La figure 3.3 donne la structure logique de X.25.
Le niveau utilisateur qui apparaît dans ces figures n'est pas abordé dans l'avis X.25 et sera discuté dans le chapitre 3.2 .

3.I.2 Le niveau physique

Les caractéristiques de l'interface ETTD-ETCD doivent être conformes à l'avis X.2I ou (à titre provisoire) à l'avis X.2I bis.
X.2I définit un interface standard entre l'ETTD et l'ETCD qui tout comme X.25 se limite aux spécifications fonctionnelles et ne présume en rien de l'implémentation ou du moyen de transmission utilisé.
Le but est de présenter un interface commun pour l'accès aussi bien aux réseaux de commutation par paquet que aux réseaux de commutation par circuit.

Les caractéristiques physiques de la jonction telles que le type de connecteur utilisé et l'allocation de ses broches, sont décrites dans l'avis X.24 .

Les caractéristiques électriques sont celles définies par X.26 et X.27 .

La transmission des bits se fait en série et dans le mode synchrone; l'échange est bidirectionnel et simultané (full-duplex) et la liaison est point à point.

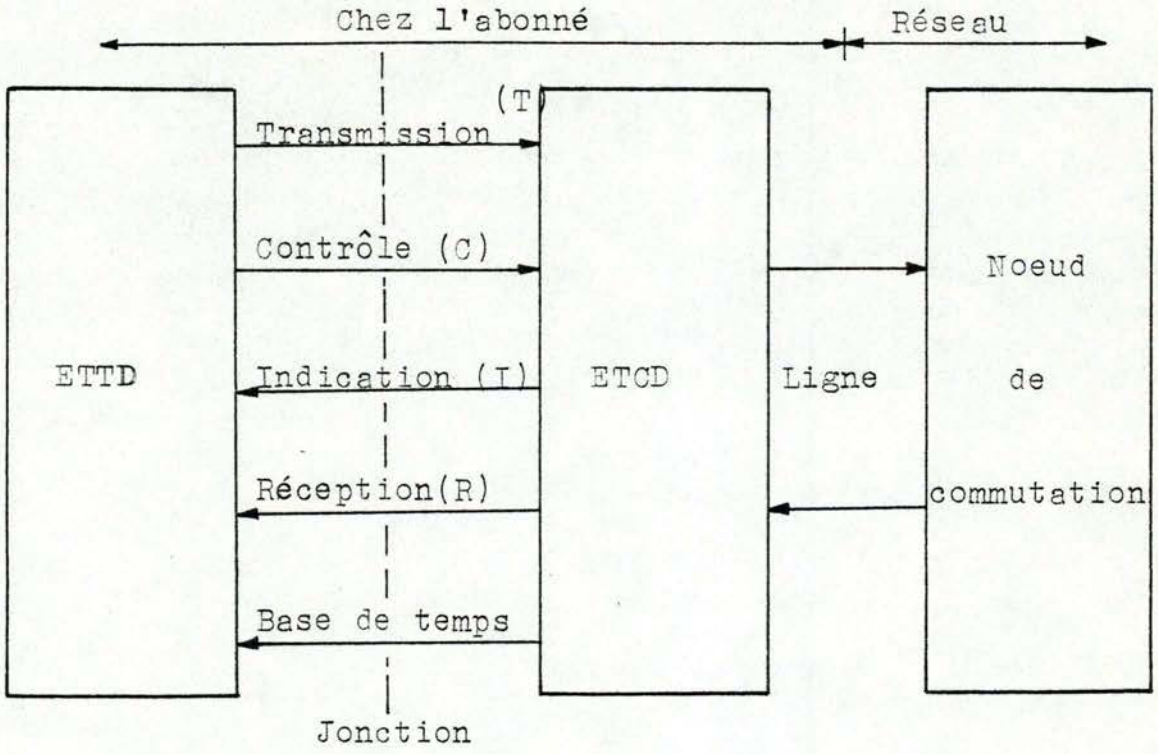


Fig. 3.4 - Jonction X.2I .

La figure 3.4 montre les circuits de jonction de l'interface ETTD - ETCD défini par l'avis X.2I .

Les lignes C et I assurent la transparence pour le transfert des données sur les lignes T et R en indiquant si les signaux présents sur T et R sont des données ou des informations de contrôle.

Les informations de contrôle servent à l'établissement de la communication ainsi qu'à sa libération.

Pour lesservices à commutation de circuit, une phase d'établissement de la communication est nécessaire.
Cette phase comprend l'échange des signaux d'appel et l'identification de l'ETTD appelé.
Les principes sont les mêmes que pour l'établissement d'une communication téléphonique.

Pour les services sur circuit loués à la demande, l'établissement de la communication se limite à l'établissement du transfert de données.

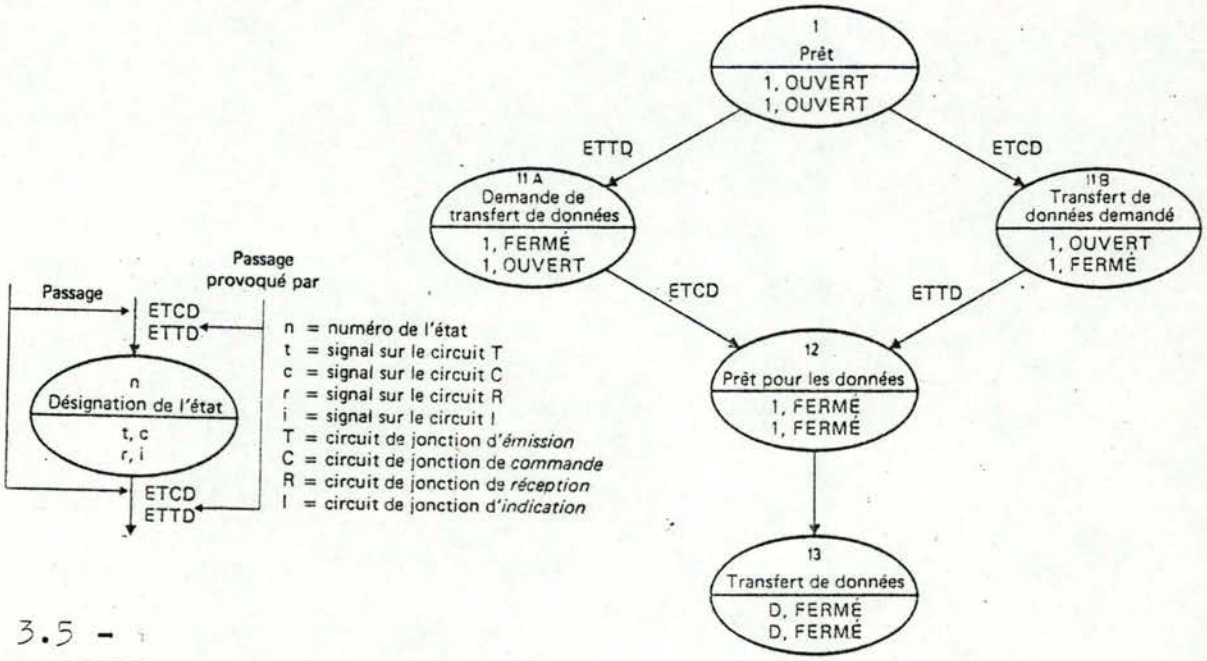


Fig. 3.5 -

Etablissement de la phase de transfert des données (service de poste à poste sur circuits loués à la demande)

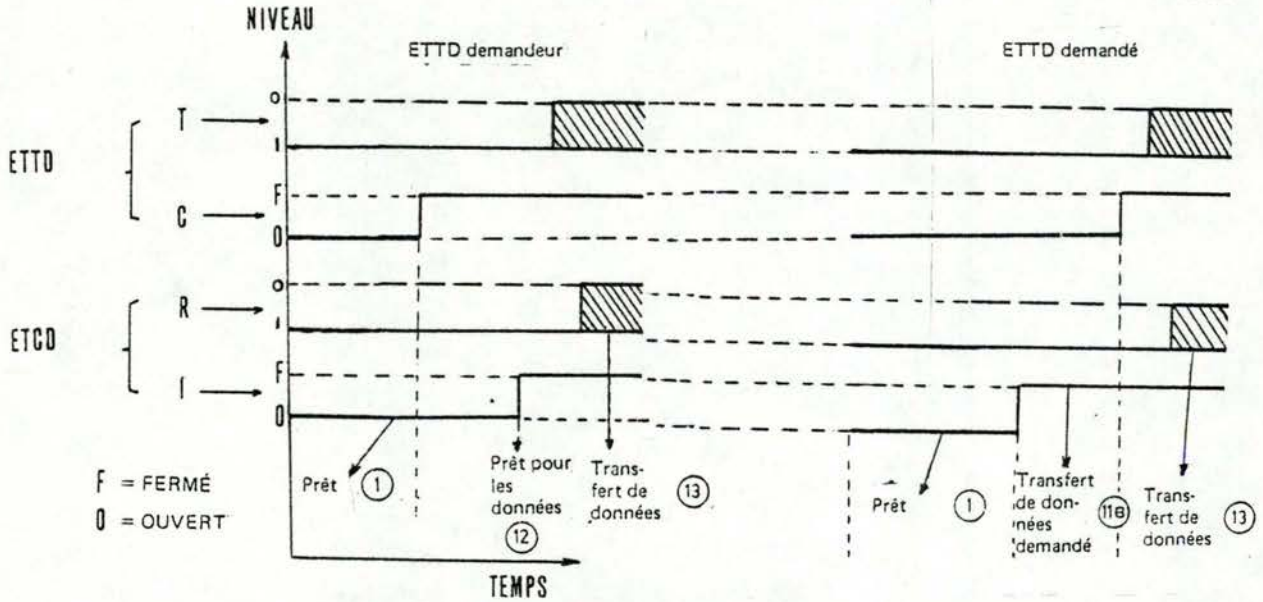


Fig. 3.6 - Diagramme temporel de l'établissement.

La figure 3.5 montre l'utilisation des circuits de jonction dans la phase d'établissement de la phase de transfert de données dans le cas de circuits loués.

Au départ, l'ETTD et l'ETCD sont dans l'état prêt.

Ils l'indiquent par un signal "I" sur T et R.
Pour indiquer une demande de transfert, l'ETTD positionne C dans l'état fermé.
L'ETCD répond en fermant le circuit I.
Lorsque C et I sont tous deux fermés, le transfert de données peut commencer sur les lignes T et R.

Pour plus de détails, se référer à l'avis X.2I du CCITT (2).
Afin d'assurer la continuité avec du matériel déjà existant pour le transfert de données dans le mode synchrone, X.25 propose une procédure compatible avec la norme V.24 .
Cette procédure décrite dans l'avis X.2I bis est proposée à titre provisoire.

Avis V.24 Circuit de jonction n°	Désignation du circuit de jonction
102	Terre de signalisation ou retour commun
103	Emission des données
104	Réception des données
105	Demande pour émettre
106	Prêt à émettre
107	Poste de données prêt
108/1	Connectez le poste de données sur la ligne
109	Détecteur du signal de ligne reçu sur la voie de données
114	Base de temps pour les éléments de signal à l'émission (ETCD)
115	Base de temps pour les éléments de signal à la réception (ETCD)
142	Indicateur d'essai (ETCD)

Fig. 3.7 - Circuits de la jonction X.2I bis .

3.I.3 Le niveau trame

Le niveau trame sert à transporter sur la liaison ETTD - ETCD les paquets élaborés par le niveau paquet. Il assure la correction des erreurs de transmission ainsi que les procédures d'établissement et de libération du lien.

L'analyse nous montre qu'il peut être décomposé en deux sous-niveaux relativement indépendants.
Cette découpe permet d'isoler deux ensembles de fonctions qui réalisent, l'un le transfert des trames, et l'autre la gestion des trames transférées.

3.I.3.I Le sous-niveau enveloppe de trame

Ce sous-niveau permet de délimiter les blocs d'information utile dans le flot de bits transmis par le niveau I.
Il permet également de détecter les erreurs éventuelles introduites par le niveau physique dans les blocs d' informations.
Il permet enfin d'effectuer une signalisation de l'état actif ou inactif pour chaque sens de transmission.

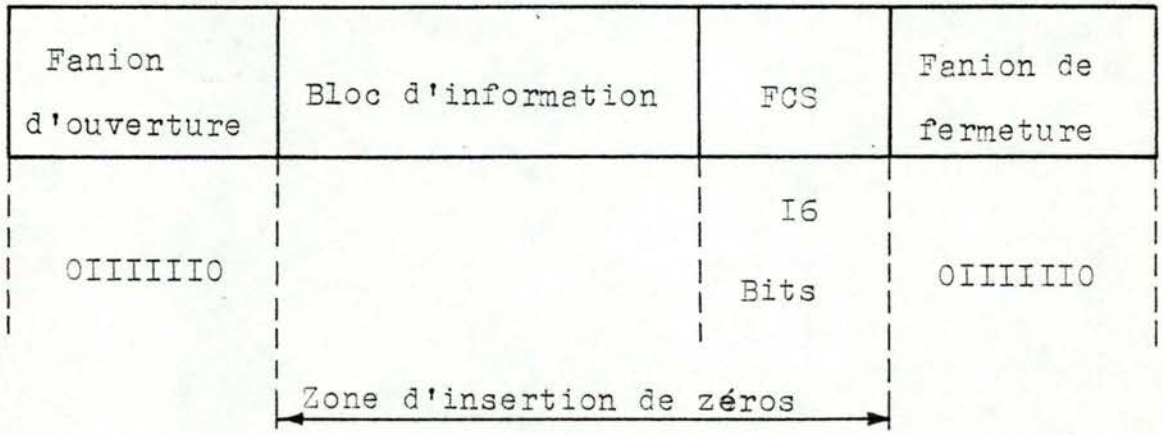


Fig 3.8 - Structure de l'enveloppe de trame .

3.I.3.I.I Structure de trame

Toutes les transmissions se font sous forme de trames dont la structure est donnée à la figure 3.8 .

Les éléments sont :

A. Le fanion

Toutes les trames doivent commencer et se terminer par une séquence fixe et réservée de huit bits appelées "fanion".
Le récepteur doit rechercher cette séquence afin de déter-

miner les limites de la trame.
La synchronisation est assurée par l'envoi permanent de fanions entre les trames. Un même fanion peut-être utilisé à la fois comme fanion de fermeture pour une trame et fanion d'ouverture pour la trame suivante.

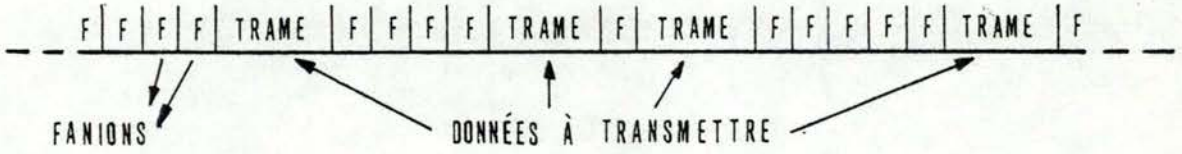


Fig. 3.9 - Remplissage entre trames .

B. Le bloc d'information
C'est la séquence de bits à délimiter et à protéger contre les erreurs et dont le contenu sera exploité par le niveau trame 2 .

C. Séquence de contrôle de trame (FCS)
La FCS est une séquence de seize éléments binaires.
Elle est calculée à partir du contenu du bloc d'information afin d'obtenir un code redondant qui servira à la détection des erreurs.
A la réception, ce code est recalculé et comparé à la FCS reçue. Les deux valeurs doivent coïncider en l'absence d'erreur de transmission.
Un exposé détaillé du calcul de la FCS est donné au paragraphe 2.2.7 de l'avis X.25 .
Les performances obtenues sur le plan de la détection des erreurs à l'aide de la FCS ne seront pas discutés dans cet exposé.

La transparence du contenu de la trame entre les deux fanions est assurée par l'insertion d'un élément "0" après toute séquence de 5 éléments "I" consécutifs.
Cette insertion faite à l'émission permet d'éviter qu'une séquence binaire identique à celle d'un fanion ne soit transmise entre deux fanions réels.

Les "0" insérés seront éliminés à la réception de la trame. Une trame est considérée comme non valable lorsqu'elle n'est pas limitée par deux fanions ou lorsqu'elle comprend moins de 32 bits entre les deux fanions.
L'émetteur peut signaler l'abandon d'une trame en cours en transmettant sept éléments binaires "I" consécutifs.
Toute trame non valable ou donnant un FCS erroné est abandonnée au niveau enveloppe de trame.
Le bloc d'information des trames correctement transmises est passé au niveau gestion de la procédure.

3.I.3.2 Le sous-niveau gestion de la procédure

Le sous-niveau enveloppe de trame a permis de transformer le service offert au niveau bit (transport de bits avec un certain taux d'erreurs) en un service de transport de blocs d'information, sans erreur, mais avec possibilité de perte.
Le second sous-niveau du niveau de trame permet de corriger ces pertes de blocs d'information à l'aide d'une procédure.
Il permet de transporter sans erreur et en conservant l'ordre, sur une liaison d'accès, les paquets qui seront manipulés par le niveau paquet.

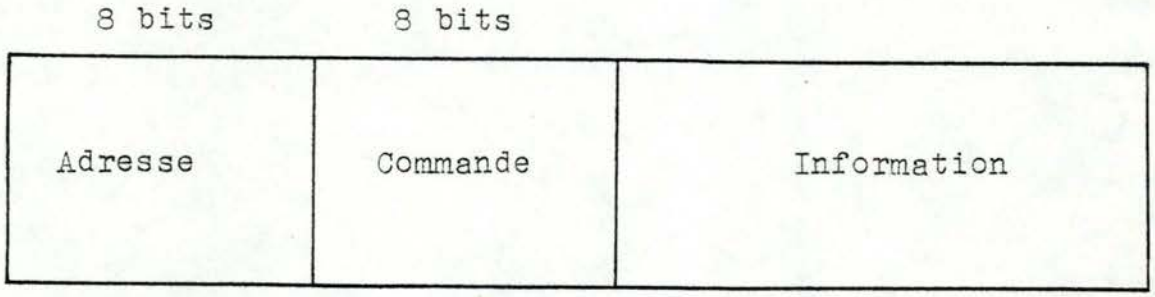


Fig. 3.IO - Format interne de la trame .

3.I.3.2.I Structure logique de la liaison au niveau trame

Les deux sens de transmission sont contrôlés de manière symétrique et séparée, chacune des deux stations (ETTD et ETCD) étant maîtresse du flux de données qu'elle émet.

Pour cela, chaque station dispose de deux fonctions :

- -une fonction primaire qui émet des trames de commande et assure l'envoi des informations.
- -une fonction secondaire qui émet des réponses aux commandes de primaire de l'autre station.

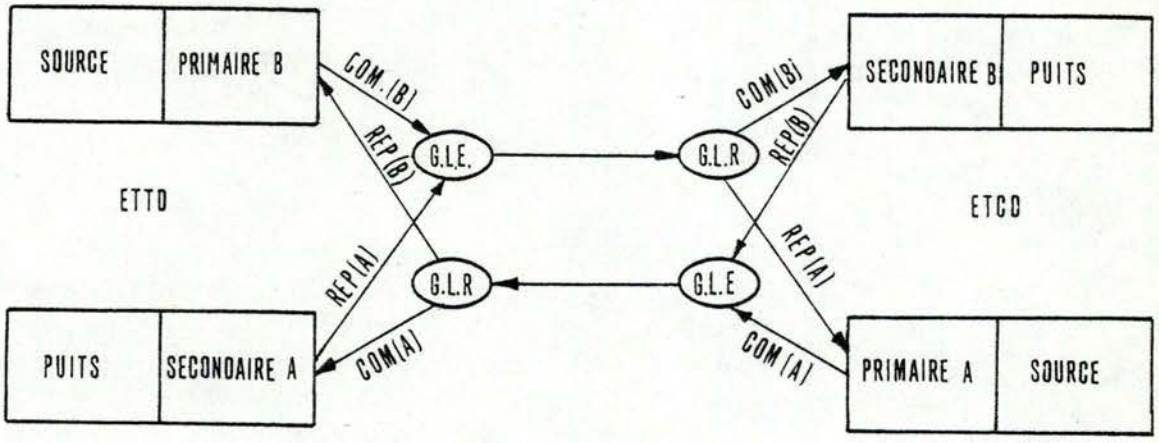


Fig. 3.II - Configuration symétrique (L.A.P.) .

Cette configuration est appelée configuration symétrique (fig. 3.II). La liaison bidirectionnelle peut être logiquement décomposée en deux liaisons unidirectionnelles quasi-indépendantes. Le primaire est responsable de l'émission des trames d'information dans le sens qu'il contrôle.
Le secondaire correspondant lui envoie les acquittements pour confirmer le bon déroulement des échanges. Le mode de réponse est asynchrone, c'est-à-dire que le secondaire peut envoyer une réponse au primaire sans que celle-ci ne soit explicitement permise par le primaire.
Une telle réponse peut servir à indiquer un changement d'état dans le secondaire (par exemple, le numéro de pro-

chaine trame attendue, la transition d'un état "prêt" à un état "occupé" ou vice-versa...).

3.I.3.2.2 La procédure d'adressage (A)

Toutes les trames sont munies d'un champ d'adresse de 8 bits. Cette adresse indique entre quel couple primaire-secondaire la trame est échangée.

Les commandes et réponses échangées entre le primaire de l'ETCD et le secondaire de l'ETTD portent l'adresse A. Celles échangées entre le primaire de l'ETTD et le secondaire de l'ETCD portent l'adresse B.

Les valeurs de A et B sont :

bits	8	7	6	5	4	3	2	I		
A	0	0	0	0	0	0	I	I	=	3
B	0	0	0	0	0	0	0	I	=	I

3.I.3.2.2 Le champ de commande (C)

Ce champ permet de déterminer le type de la trame et contient s'il y a lieu des numéros de séquence.

Bits	8 7 6	5	4 3 2	I
Trame I	N(R)	P/F	N(S)	0
Trame S	N(R)	P/F	SUF 0	I
Trame U	MOD	P/F	MOD I	I

Fig. 3.I2 - Codification du champ de commande .

La figure 3.I2 montre le format du champ de commande pour les 3 types de trame utilisés.

-Les trames d'information (I) sont numérotées et servent au transfert des paquets du niveau 3.

-Les trames de supervision (S) sont numérotées et servent à la supervision du transfert des données.

-Les trames non numérotées (U) utilisées pour les fonctions de supervision complémentaires (connexion, déconnexion, réinitialisation).

Le bit P est mis à "I" par le primaire dans une trame de commande pour solliciter une réponse de secondaire. Cette réponse devant contenir un bit F mis à "I" pour indiquer qu'elle correspond bien à la sollicitation du primaire.

3.I.3.2.3 Commandes et réponses

La figure 3.I3 nous montre la codification du champ de commande pour l'ensemble des commandes et réponses définies dans l'avis X.25.
Les parties claires du tableau correspondent à la version I976 de X.25, les parties sombres proviennent des ajoutes apportées par les amendements de I977 (3).
Les modifications de procédure apportées par la version I977 sont abordées à la fin du chapitre 3.

Format	Commandes	Réponses	Codification			
Transfert d'information	I -Information		N(R)	P	N(S)	0
Supervision	RR-Receive Ready	RR-Receive Ready	N(R)	P/F	0 0 0 I	
	RNR-Receive Not Ready	RNR-Receive Not Ready	N(R)	P/F	0 I 0 I	
	REJ-Reject	REJ-Reject	N(R)	P/F	I 0 0 I	
U - Trames non numérotées	SARM-Set Asynchronous Response Mode	DM-Disconnected Mode	0 0 0	P/F	I I I I	
	SABM-Set Asynchronous Balanced Mode		0 0 I	P	I I I I	
	DISC-Disconnect		0 I 0	P	0 0 I I	
		UA -Unnumbered Acknowledgement	0 I I	F	0 0 I I	
		CMDR-Command Reject	I 0 0	F	0 I I I	
		FRMR-Frame Reject				

Fig. 3.I3 - Commandes et réponses .

Le rôle des différentes commandes et réponses est donné au fur et à mesure que sont exposés les principes de la procédure du niveau trame.
Cette procédure est appelée LAP (Link Acess Procedure) et correspond à la configuration symétrique de la figure 3.8 .

3.I.3.2.4 Le temporisateur

Quand un primaire émet une commande il attend une réponse à celle-ci. Or, il se peut que la commande soit perdue (au sous-niveau enveloppe de trame) et n'atteigne pas le secondaire ou que la réponse du secondaire soit mal transmise et ne parvienne pas au primaire. Pour éviter d'attendre indéfiniment, le primaire est équipé d'un temporisateur TI qui est armé lors de l'envoi de la commande et désarmé à la réception de la réponse attendue. Si le temporisateur arrive en fin de course, la commande est réémise et le temporisa-

teur est réarmé. Après N2 retransmissions infructueuses, la ligne est considérée comme en panne au niveau trame, et, le niveau paquet en est averti.
La durée de TI et le nombre maximum de réarmement N2 sont des paramètres de la liaison.

3.I.3.2.5 L'établissement de la liaison

Chaque primaire d'une station est responsable de l'initialisation du secondaire de l'autre station.
La procédure est la suivante :
Le primaire émet la commande SARM (mise en mode de réponse asynchrone) et arme son temporisateur.
Quand le secondaire de l'autre station a reçu correctement le SARM, il envoie la réponse UA (accusé de réponse non numéroté).
Le primaire de la même station commence alors son initialisation (il émet SARM et attend UA de la même manière que l'autre station).
Chaque primaire utilise son temporisateur comme décrit en 3.I.3.2.4 .
Lorsque les deux sens de transmission sont initialisés, le transfert de données peut commencer.

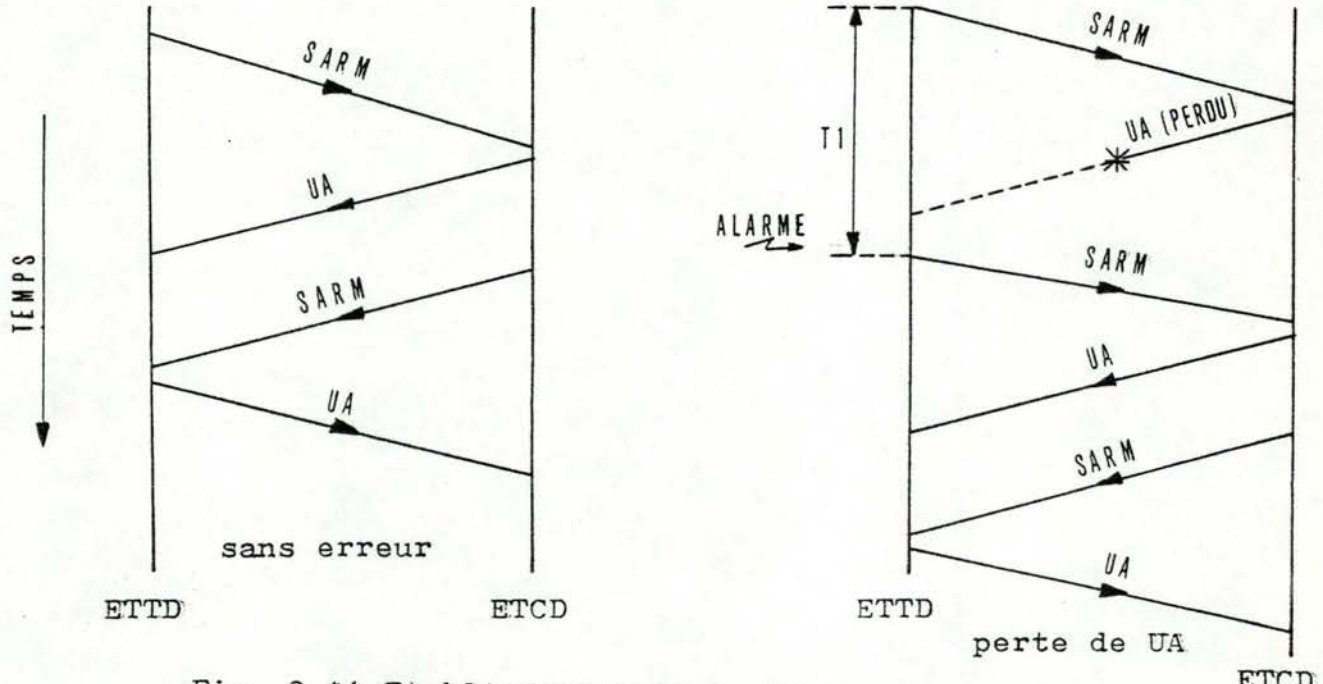


Fig. 3.14 Etablissement de la liaison .

3.I.3.2.6 Déconnexion de la liaison

Quand la phase de transfert de données est terminée, il faut procéder à la déconnexion.
Cette procédure est analogue à celle pour l'établissement de la liaison mais on utilise la trame DISC (commande de déconnexion) au lieu de SARM .

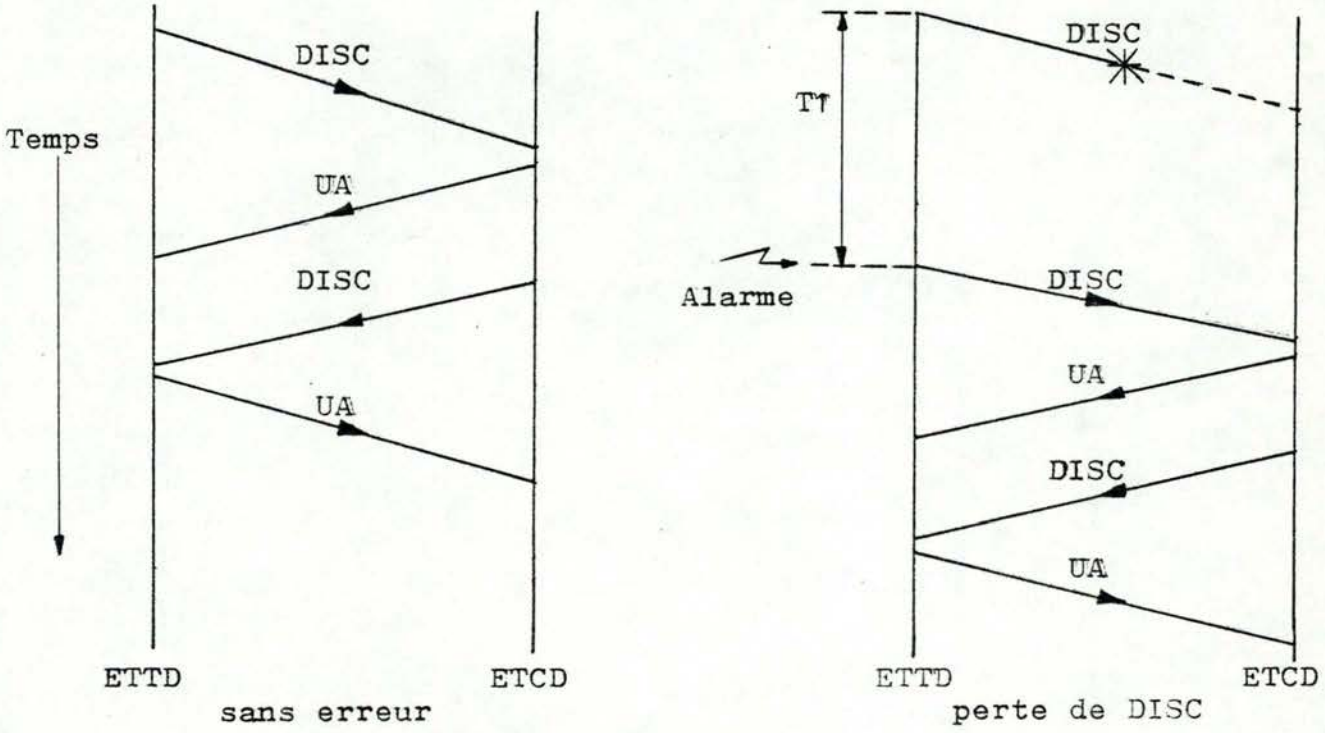


Fig. 3.15 - Deconnexion de la liaison .

3.I.3.2.7 Le transfert de données

Le primaire de chaque station contrôle le transfert de l'information vers le secondaire de l'autre station. Les deux sens sont contrôlés indépendament.

Pour gérer son émission, le primaire utilise deux variables d'état.

-V(S) indique le numéro de séquence de la prochaine trame d'information devant être émise.
La numérotation est cyclique de 0 à 7.

-V(N) indique le numéro de la première trame émise

mais non encore acquittée.

Le secondaire en utilise une pour gérer la réception des trames d'information.

-V(R) indique le numéro de la prochaine trame d'information attendue en réception.

Ces variables sont mises à zéro au début de la phase de transfert de données.

3.I.3.2.7 Emission des trames d'information

Le primaire émet les trames d'information en séquence. Pour cela, il met la valeur de V(S) dans le champ N(S) (numéro de séquence en émission) des trames I qu'il émet. Il peut émettre un certain nombre K de trame d'information en anticipation, c'est-à-dire sans attendre l'acquittement des précédentes. Ce nombre K est un paramètre de la liaison, sa valeur doit être comprise entre I et 7 inclus. A la fin de l'émission de la trame d'information, le primaire incrémente sa variable d'état en émission (V(S)) d'une unité.

3.I.3.2.8. Acquittement des trames d'information

Le secondaire acquitte les trames d'information reçues en transmettant la valeur de sa variable V(R) au primaire qui les a émises. Cette valeur est transmise au moyen du champ N(R) des trames de supervision (RR, RNR, REJ) ou des trames I émises par le primaire de la même station.

Lors de la réception d'une trame I, le secondaire doit vérifier le numéro de séquence N(S) de la trame reçue, deux cas peuvent se présenter :

I- N(S) = V(R); la trame est reçue en séquence, V(R) est incrémenté de I (modulo 8), la nouvelle valeur de V(R) est le plus rapidement possible transmise à l'autre station.
Pour cela le champ N(R) de la prochaine trame I à émettre est utilisé.

Si il n'y a pas de trame I à émettre à ce moment, une trame RR (Receive Ready) est envoyée.
Le N(R) ainsi transmis accuse réception des trames I reçues par le secondaire jusqu'à N(R)-I.

2- N(S) ≠ V(R); la trame reçue est hors séquence, ce qui indique que les trames manquantes dans la séquence ont été perdues.
La trame reçue est abandonnée et le secondaire demande la retransmission des trames I à partir de celle qui était attendue et dont le numéro est mémorisé par V(R). Ceci est réalisé par l'envoi d'une trame REJ (Reject) avec un N(R) égal à V(R).

Remarque: L'avis X.25 de 1976 indique que dans le cas ou N(S) = V(R) un contrôle de dépassement de l'anticipation est à effectuer.
Si N(S) = (dernier N(R) émis + K) alors le champ de commande est considéré comme erroné et une procédure de réinitialisation est demandée par le secondaire (voir procédure de réinitialisation).

3.I.3.2.9 Etat occupé

La trame RNR est utilisée par le secondaire pour indiquer un état occupé, c'est-à-dire une incapacité temporaire à accepter des trames d'information supplémentaires. Elle accuse également réception des trames I reçues par le secondaire jusqu'à N(R) - I.

Le primaire recevant un RNR peut néanmoins émettre la trame d'information numérotée N(R) mais il doit attendre une trame RR ou REJ avant d'émettre d'autres trames d'information.

Le secondaire ayant émis RNR peut ignorer les trames I. Si la trame I reçue avait son bit P à I, le secondaire émet RNR avec F=I.
Par contre il traite les trames de supervision.
Il signale la fin de son état occupé en émettant une trame RR dont le N(R) accuse réception, le cas échéant,

des trames d'information qu'il avait prises en compte avant l'envoi du RR ou par une trame REJ s'il a perdu des trames pendant son état occupé.

3.I.3.3.0 Reprise sur temporisateur

Lorsqu'il émet une trame I, le primaire arme son temporisateur TI si celui-ci ne l'est pas encore.
TI est désarmé lors de la réception d'un N(R) acquittant toutes les trames émises. Si le N(R) acquitte effectivement certaine trame mais ne les acquitte pas toutes, le temporisateur est réarmé.
Si TI arrive en fin de course, la trame de numéro égal à V(N) (première trame non encore acquittée) est réémise avec le bit P mis à I.
Le temporisateur est réarmé et le compteur de répétition est incrémenté de I. La condition de reprise sur temporisateur est annulée lorsque le primaire reçoit une trame correcte de supervision avec le bit F mis à I.

3.I.3.3.I Procédure de réinitialisation

Cette procédure est utilisée pour réinitialiser un sens de transmission de l'information entre un primaire et le secondaire qui lui correspond.
Cette réinitialisation est faite par le primaire.
Celui-ci envoie une trame SARM au secondaire de l'autre station qui lui répond par un UA. Le temporisateur est utilisé comme pour l'établissement de la liaison.
Les variables V(S), V(N) du primaire et V(R) du secondaire sont remises à 0.

Cette procédureest enclenchée par le primaire s'il reçoit une trame qui lui est adressée et qui remplit les conditions suivantes :

-Le type de la trame n'appartient pas à l'ensemble des types utilisables pour une réponse;

-le champ d'information n'est pas valable;

-le numéro N(R) contenu dans le champ de

commande n'est pas correct;

-la réponse contient un élément binaire F mis à1 sauf s'il y a reprise sur temporisateur.

Le secondaire peut demander au primaire de l'autre station de procéder à une réinitialisation. Pour cela le secondaire lui envoie une trame CMDR. Cette trame possède un champ d'information qui est garni d'un diagnostic d'erreur. Elle est envoyée si le secondaire a reçu une trame qui lui est adressée et qui remplit les conditions suivantes :

-le type de trame n'appartient pas à l'ensemble des types utilisables pour une commande;

-le champ d'information est incorrect ou plus long que permis;

-le numéro N(S) contenu dans le champ de commande est incorrect.

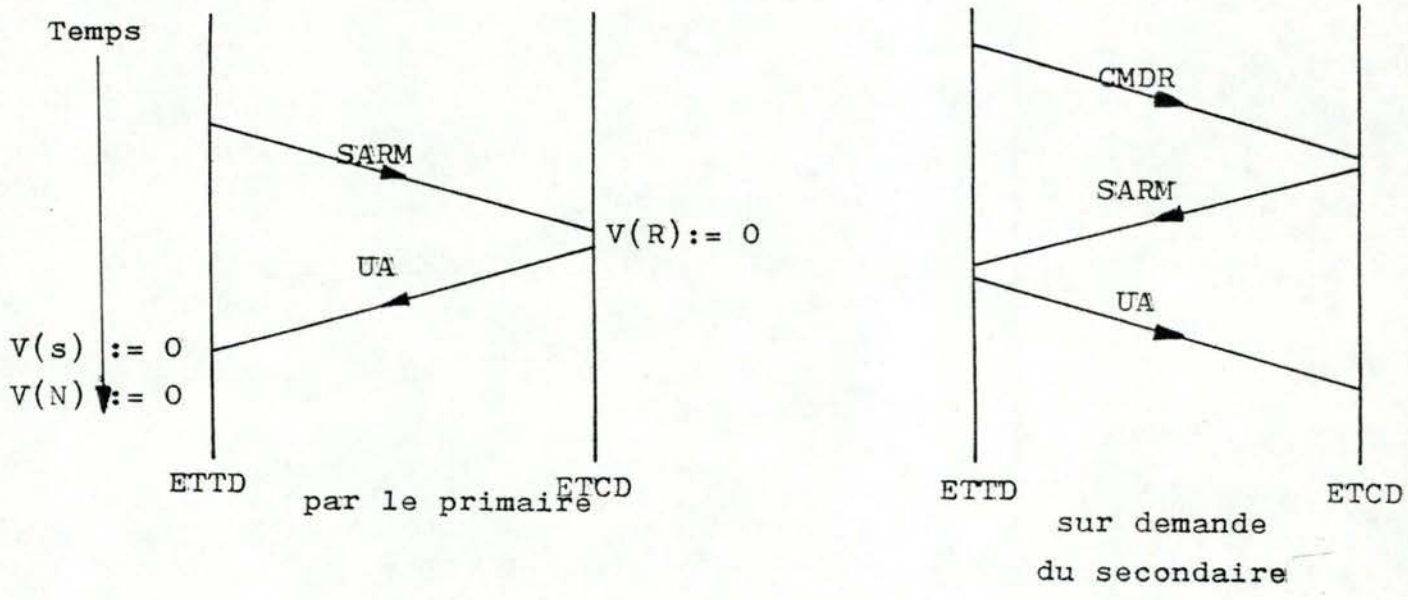


Fig. 3.16 - Reinitialisation d'un sens de transmission de l'information .

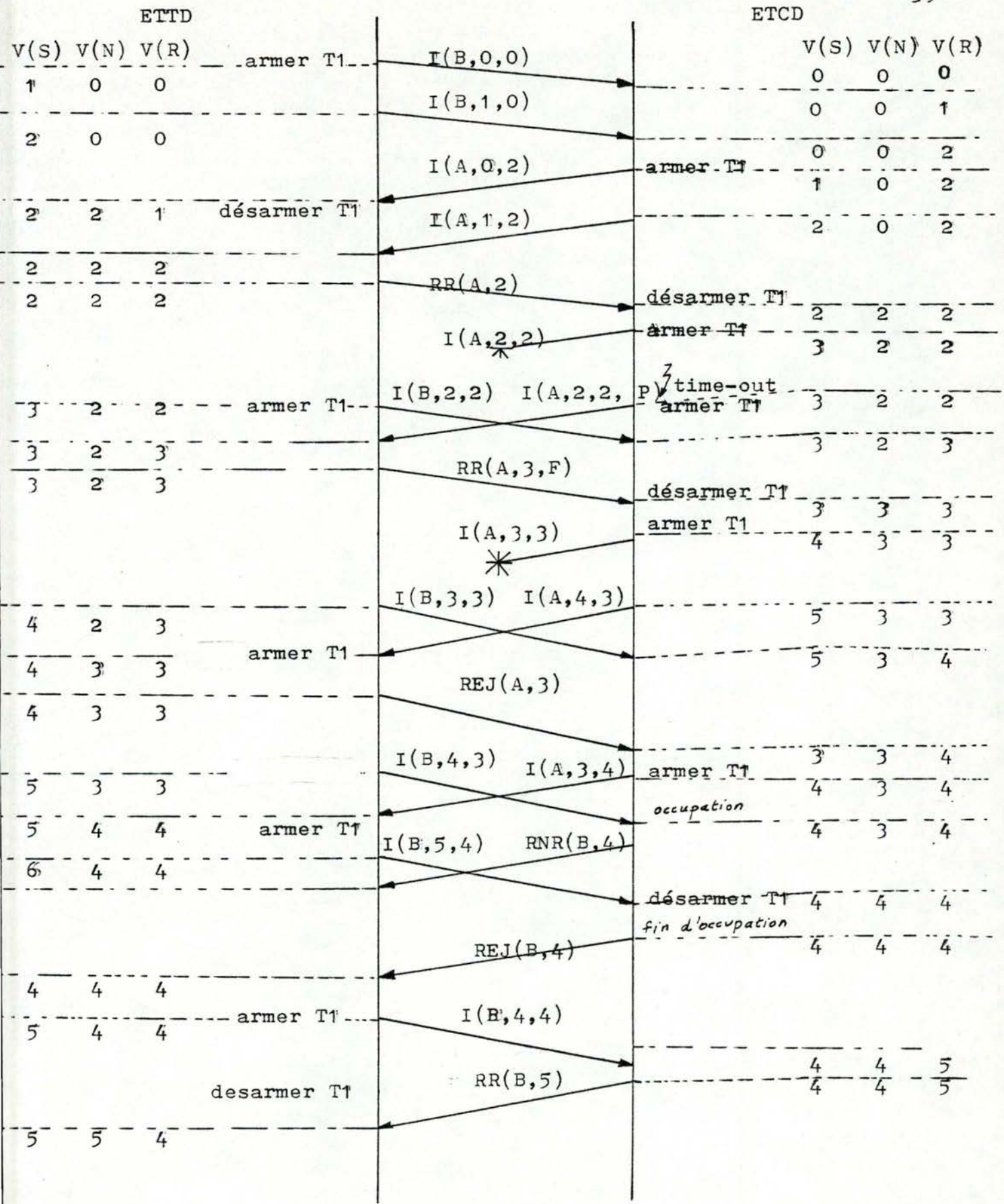


Fig. 3.17 - Exemple d'échanges sur la liaison ETTD/ETCD .

3.I.3.3.2 Version révisée du niveau 2 (I977)

L'avis X.25 a fait l'objet d'une révision approuvée en octobre I977, les spécifications du niveau 2 ont été complétées dans le but d'améliorer la compatibilité avec HDLC de l'ISO.
Une nouvelle procédure a été introduite sous le nom de LAPB (Balanced mode Link Access procedure) et se différencie de LAP (I976) par les points suivants :

Etablissement de la liaison

L'ETTD initialise la transmission en envoyant une commande SABM (Set Asynchronous Balanced mode) à l'ETCD qui doit lui envoyer une réponse UA. Les deux sens de transmission d'information sont établis en une fois et le transfert d'information peut commencer dès que les variables d'état ont été remises à 0.
(L'utilisation de la commande SABM par l'ETCD est pour une étude ultérieure.)

Déconnexion de la liaison

Identique à la phase d'établissement mais avec DISC au lieu de SABM.

Etat déconnecté

Après avoir reçu un DISC de l'ETTD et renvoyé un UA, l'ETCD entre dans l'état déconnecté.
Dans cet état l'ETCD répond par une trame DM (Disconnected Mode) à toute commande DISC.
Et par une trame DM avec bit F mis à I à toute commande dont le bit P est mis à I. Les autres trames reçues sont ignorées.
L'ETCD peut également atteindre l'état déconnecté sans certaines conditions d'erreur.
(Réception d'un UA ou DM pendant la phase de transfert de données ou erreur locale de fonctionnement) il utilise alors la trame DM pour requérir un SABM de l'ETTD.

Réinitialisation

La réinitialisation déclenchée par l'ETTD en transmettant un SABM l'ETCD lui répond par un UA.
Les deux sens de transmission de l'information sont réinitialisées en même temps, les variables d'état sont remises à 0. L'ETCD peut demander une réinitialisation à l'ETTD en lui transmettant une trame FRMR ou DM.

En plus du mode LAPB, la version I977 prévoit :

I- L'utilisation par l'ETTD les trames RR, RNR, REJ en tant que commandes pour requérir une information sur l'état de l'ETCD.

2- La suppression du test sur le N(S) reçu dans une trame I en ce qui concerne le dépassement de l'anticipation.

Remarque : il faut noter que ces ajoutes introduisent une certaine forme d'assymétrie entre l'ETTD et l'ETCD.
Les nouvelles commandes introduites étant réservées à l'ETTD et les nouvelles réponses à l'ETCD.
C'est en fait l'ETTD qui choisit le mode symétrique ou équilibré lors de l'établissement de la liaison.
L'ETCD doit pouvoir accepter ces deux modes et il utilise une variable B qu'il positionne à I dans le mode équilibré (Balanced) et à 0 dans le mode symétrique afin de mémoriser le type de procédure à utiliser par la suite.

Le niveau 2 est repris plus en détails dans le chapitre 4 qui en propose une implémentation.

3.I.4 Le niveau paquet

Le niveau 3 de l'avis X.25 définit les différents types de paquet échangés sur la liaison ETTD-ETCD.
Il donne pour chacun d'entre eux le format du paquet et en décrit la fonction.
Afin de permettre la coexistence à un instant donné de plusieurs communications virtuelles entre un ETTD et un ou plusieurs autres ETTD, le lien d'accès fourni par le niveau 2 est décomposé en un certain nombre de voies logiques.
Chaque voie logique porte un numéro composé d'un numéro de groupe de voies logiques codé sur quatre bits (0-I5) et un numéro de voie logique codé sur huit bits (0-255).
Une communication virtuelle entre deux ETTD correspond à une association logique faite par le réseau entre une voie logique d'un ETTD et une voie logique de l'autre. Cette association peut être momentanée et réalisée pendant une phase d'établissement de la communication virtuelle. Elle peut également être permanente et définie au moment de l'abonnement au service.
Dans le premier cas on parle de circuit virtuel temporaire ou commuté (CVC) dans le second de circuit virtuel permanent (CVP).
X.25 décrit également l'ensemble des états dans lesquel peut se trouver une voie logique ainsi que les transitions d'état causées par la réception ou l'émission d'un paquet sur la voie logique par l'ETCD. Il s'agit donc du comportement de l'ETCD, celui de l'ETTD devant en être déduit car il n'est pas détaillé dans l'avis.
La figure 3.20 illustre le principe des communications virtuelles. On peut y voir la fonction de multiplexage réalisée au niveau paquet par le partage temporel de la liaison d'accès entre les paquets des différentes voies logiques.

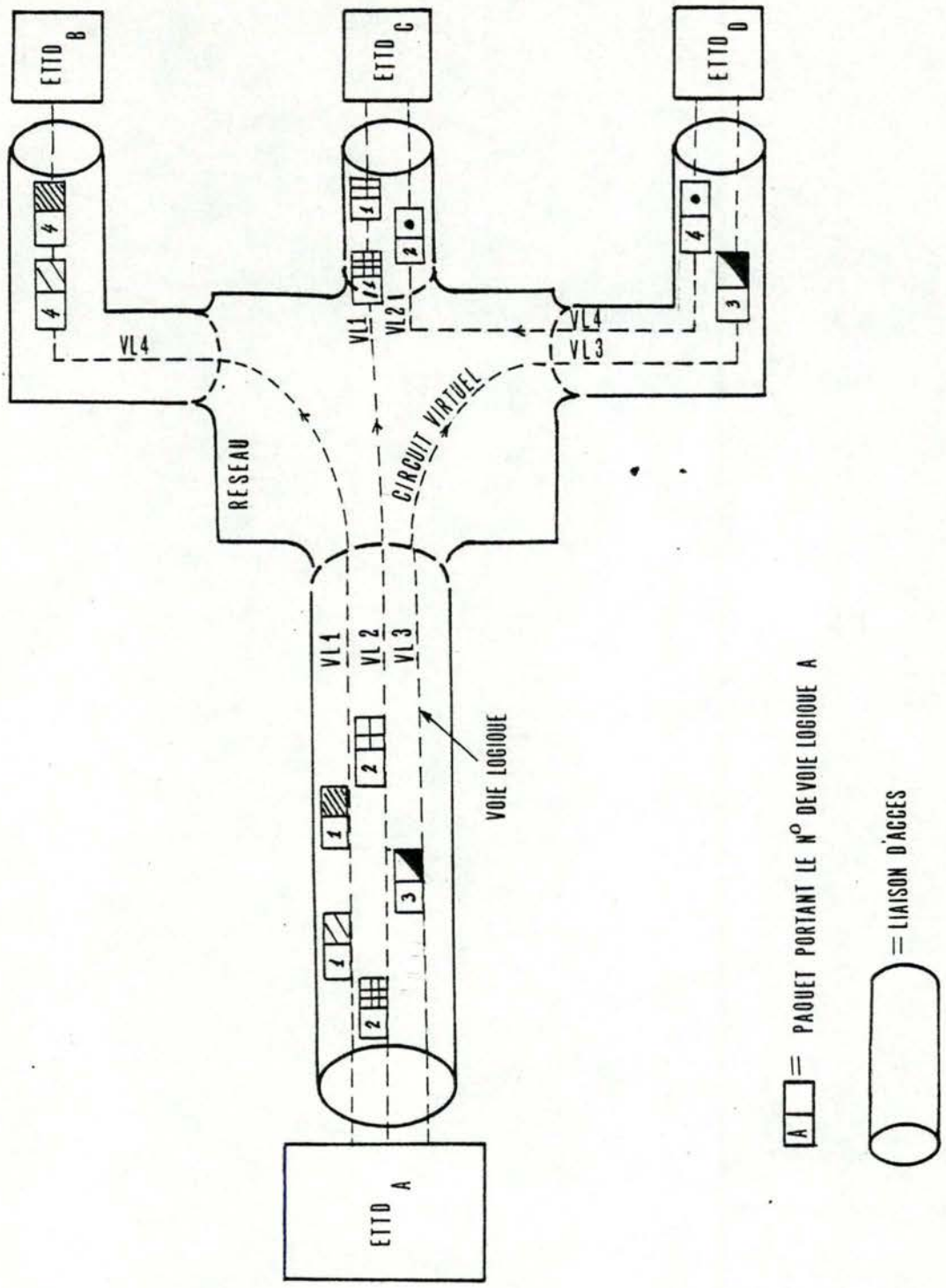


Fig. 3.20 - Principe des communications virtuelles .

L'ETTD A communique à la fois avec les ETTD B, C et D . Le circuit virtuel entre l'ETTD A et l'ETTD B emprunte la voie logique n°1 sur la liaison d'accés de l'ETTD A et la voie logique n°4 sur la liaison d'accés de l'ETTD B .L'ETTD C et l'ETTD D communiquent également .

Les différentes procédures du niveau paquet sont décrites dans l'avis X.25 uniquement au niveau local de l'interface ETTD-ETCD.
Si le comportement de l'ETCD est bien décrit au niveau local, la signification au niveau de la communication de bout en bout entre deux ETTD, qu'il faut attribuer à l'information de contrôle portée par les paquets n'est pas donnée dans l'avis X.25.
En fait, cela dépend de la façon dont le niveau 3 est implémenté au niveau du réseau pour offrir un contrôle de bout en bout ou un contrôle local à l'interface ETTD-ETCD. Les exemples présentés pour illustrer les différentes procédures correspondent à l'implémentation du réseau public français TRANSPAC.

3.I.4.I Etablissement d'une communication virtuelle

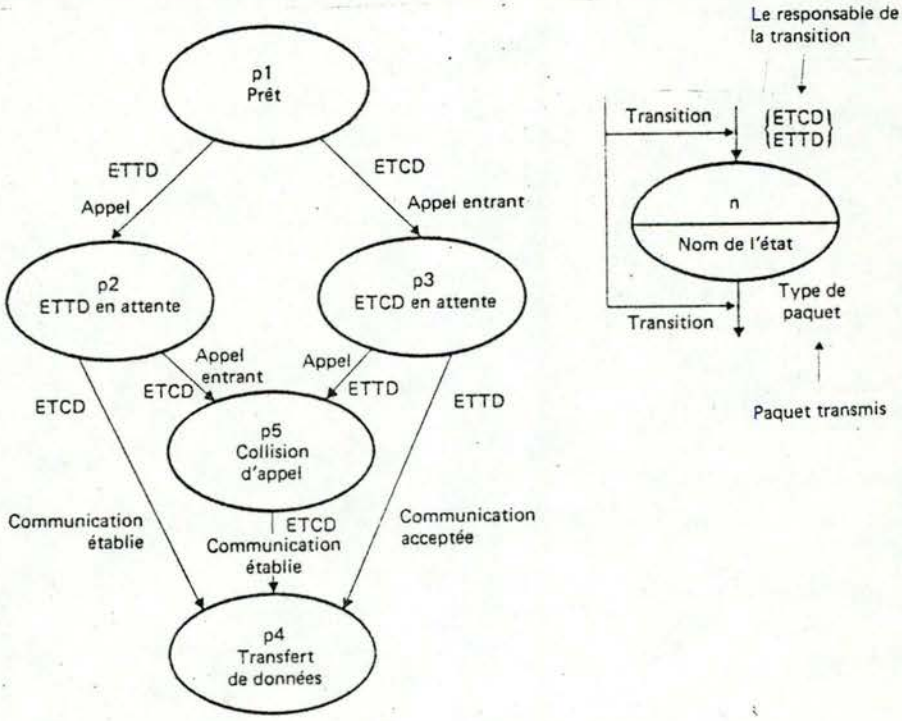


Fig. 3.21' - Phase d'établissement de la communication sur une voie logique .

On dit qu'une voie logique est dans l'état prêt lorsqu'il n'y a aucune communication en cours sur cette voie logique.

La figure 3.2.I donne le diagramme des états dans lesquels peut se trouver une voie logique pendant l'établissement de la communication.
L'ETTD appelant transmet un paquet d'appel sur la voie logique X.
Ce paquet contient l'adresse de l'ETTD appelé.
Le réseau essaye d'établir la communication en transmettant un paquet d'appel entrant sur la voie logique'y' de l'ETTD appelé.
Ce paquet contient l'adresse (mise par le réseau) de l'ETTD appelant.
Si l'ETTD appelé accepte l'appel, il transmet sur la voie logique 'y' un paquet communication acceptée, ce qui place cette voie logique dans l'état transfert de données. L'acceptation est alors répercutée chez l'ETTD appelant par le réseau qui lui transmet un paquet communication établie sur la voie logique 'X'.
L'association logique des voies 'X' et 'y' constitue la communication virtuelle entre les deux ETTD.
Une collision d'appel peut se produire lorsque l'ETTD et l'ETCD transmettent simultanément un paquet d'appel et un paquet d'appel entrant sur une même voie logique.
Dans ce cas le paquet d'appel à la priorité et est traité par l'ETCD qui abandonne l'appel entrant.

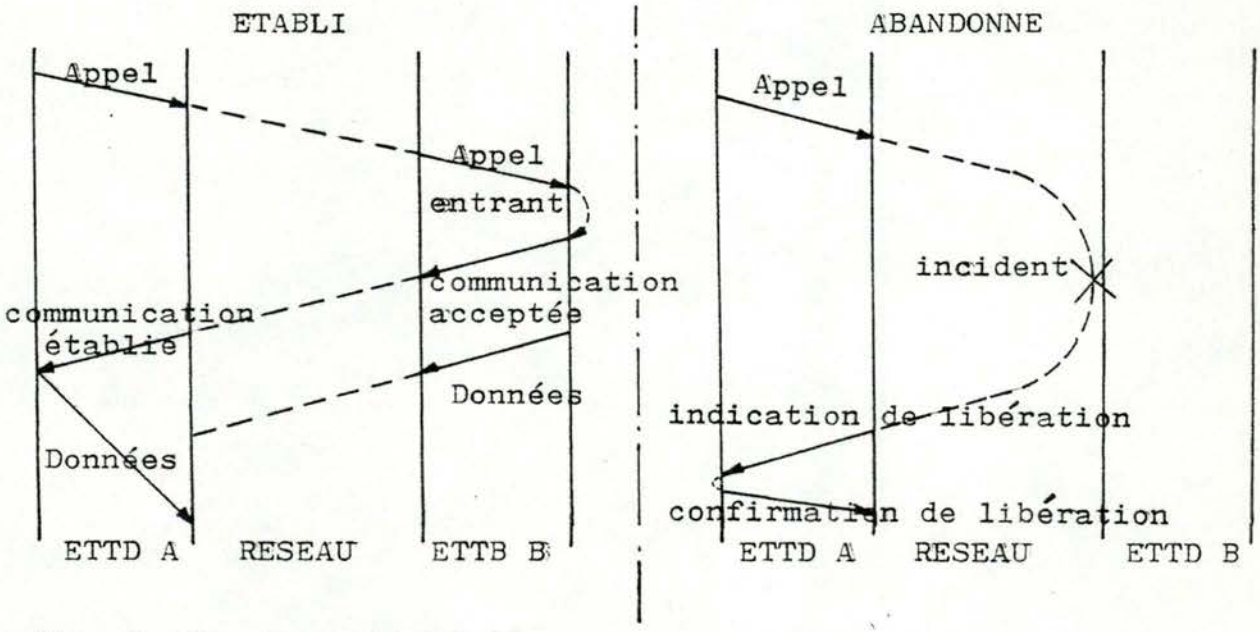
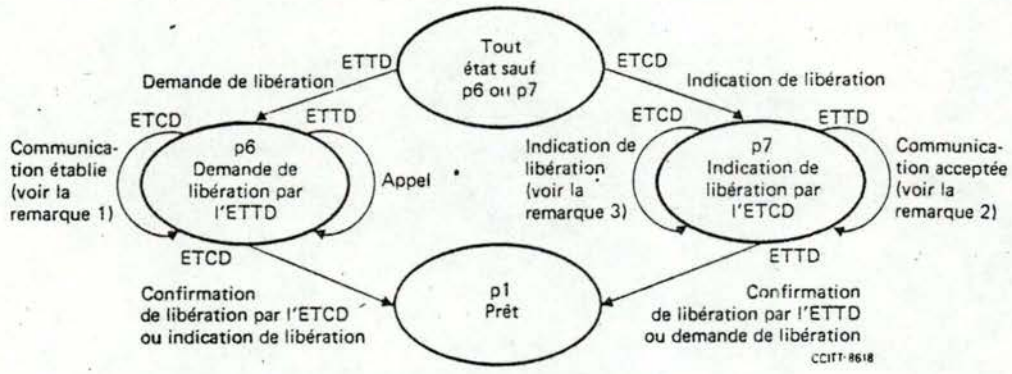


Fig. 3.22 - Exemples de phases d'etablissement d'une communication virtuelle entre 2 ETTD .

3.I.4.2 Libération d'une communication virtuelle



b) Phase de libération de la communication

Remarque 1. – Cette transition n'est possible que si l'état précédent était *ETTD en attente* (p2).
Remarque 2. – Cette transition n'est possible que si l'état précédent était *ETCD en attente* (p3).
Remarque 3. – Cette transition se produit à l'arrivée en fin de course d'un temporisateur dans le réseau.

Fig. 3.23 - Phase de libération de la communication.

L'ETTD peut indiquer la fin de la communication en transmettant un paquet de demande de libération, l'ETCD libère alors la voie logique et envoie un paquet confirmation de libération.

L'ETCD peut également demander une libération en transmettant un paquet d'indication de libération à l'ETTD qui lui répond par un paquet de confirmation de libération.

Une collision de libération intervient lorsqu'un ETTD et un ETCD transmettent simultanément un paquet de demande de libération précisant une même voie logique.

Dans une telle situation, la demande de l'unest considérée comme confirmation par l'autre et la voie logique passe à l'état prêt. La procédure de libération est également utilisée pour indiquer que la communication ne peut être établie (voir figure 3.22 et 3.24).

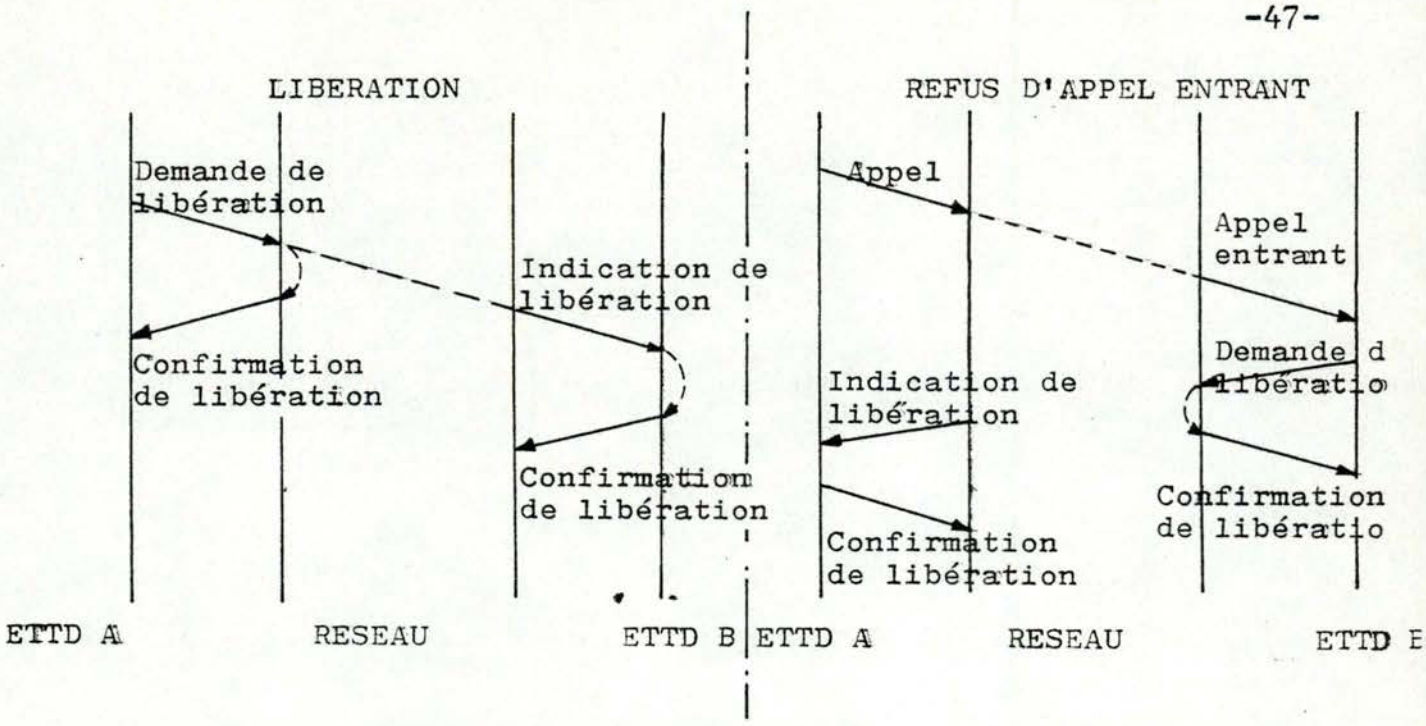


Fig. 3.24 - Exemples de phases de libération d'une communication virtuelle entre 2 ETTD .

Note : Les phases d'établissement et de libération ne s'appliquent qu'aux circuits virtuels commutés. Les voies logiques aloués à des circuits virtuels permanents restent constamment dans l'état transfert de données.

3.I.4.3 Phase de transfert de données

La procédure de transfert de données décrite ci-dessous s'applique indépendamment à chaque voie logique existant à l'interface ETTD-ETCD.

3.I.4.3.I Numérotation des paquets de données

La numérotation des paquets de données est réalisée modulo 8. Les numéros de séquence décrivent le cycle complet de 0 à 7 (la possibilité de numéroter modulo I28 est également prévue en tant que service complémentaire). Seuls les paquets de données contiennent le numéro de séquence en émission P(S). Le premier paquet de données transmis dans un sens donné porte un numéro P(S) = 0.

3.I.4.3.2 Procédure de contrôle de flux

A l'interface ETTD/ETCD d'une voie logique utilisée pour une communication virtuelle, la transmission des paquets de données est contrôlée séparément dans chaque sens et est fondée sur des autorisations venant de l'extrémité réceptrice.

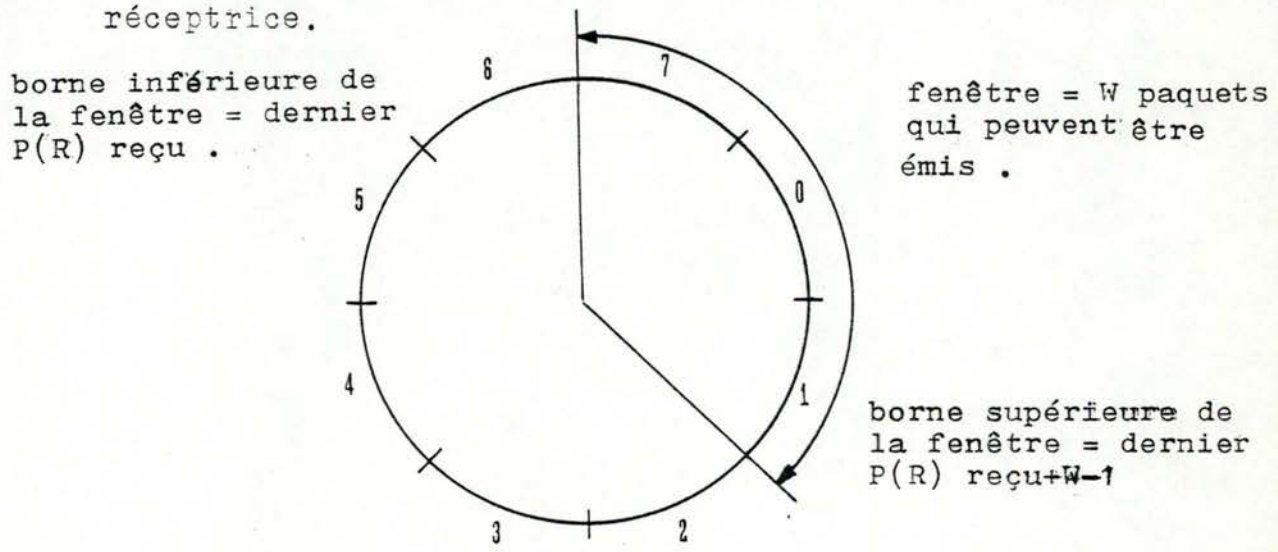


Fig. 3.25 - Description de la fenêtre .

La fenêtre est définie comme l'ensemble des w numéros P(S) consécutifs des paquets de données autorisés à traverser l'interface. Le plus petit P(S) de la fenêtre est appelé limite inférieure de la fenêtre.

Le premier paquet non autorisé à traverser l'interface porte un P(S) égal à la limite inférieure de la fenêtre augmentée de la taille de la fenêtre x (valeur I à 7).

L'émetteur peut donc transmettre tous les paquets dont le numéro est à l'intérieur de la fenêtre. Le récepteur peut faire évoluer la limite inférieure de cette fenêtre en transmettant à l'émetteur un numéro P(R) appelé numéro de séquence en réception. De cette façon il autorise l'émission de paquets de données supplémentaires.

Le numéro P(R) est acheminé par les paquets prêts à recevoir (RR) et non prêts à recevoir (RNR).

En général les P(R) reçus par l'émetteur doivent être interprétés comme mise à jour locale de la fenêtre.

Cependant, certains réseaux peuvent implémenter un contrôle de bout en bout d'un circuit virtuel et les P(R) reçus

indiquent la bonne réception des paquets de données par l'ETTD distant . La valeur d'un P(R) reçu doit rester dans l'intervalle entre le dernier P(R) reçu et le P(S) du prochain paquet de données à émettre.
Dans le cas contraire il s'agit d'une erreur de procédure et il faut réinitialiser la communication virtuelle.

3.I.4.3.3 Paquets prêts à recevoir (RR)

Les paquets PR sont utilisés par le récepteur pour indiquer qu'il est prêt à recevoir les w paquets de données qui sont à l'intérieur de la fenêtre dont la limite inférieure est égale au P(R) transmis dans le paquet RR.

3.I.4.3.4 Paquets non prêts à recevoir (RNR)

Ces paquets sont utilisés par le récepteur pour indiquer une incapacité temporaire à accepter des paquets de données supplémentaires. Lorsqu'il reçoit un paquet RNR, un ETCD ou un ETTD cesse de transmettre des paquets de données sur la voie logique concernée. L'état non prêt à recevoir est annulé soit par la transmission dans le même sens d'un paquet RR, soit par une procédure de réinitialisation.

3.I.4.3.5 Procédure d'interruption

Cette procédure permet à un ETTD de transmettre des données à un ETTD distant sans suivre la procédure de contrôle de flux.

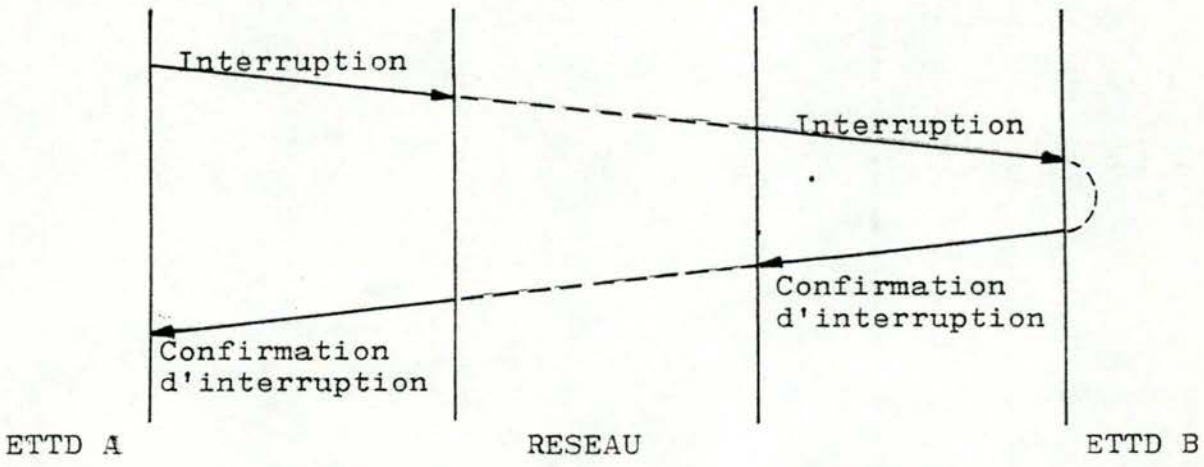
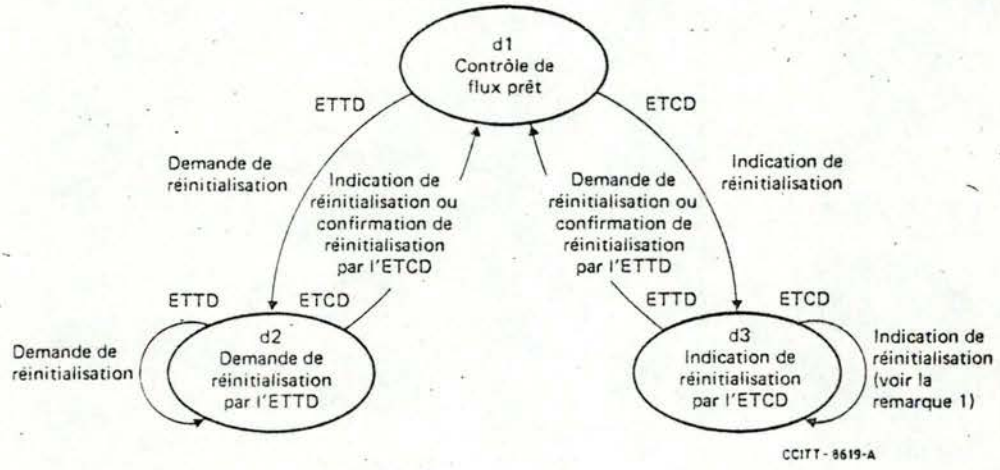


Fig. 3.26 - Procédure d'interruption dans une implémentation de bout en bout.

L'ETTD transmet un paquet d'interruption à l'ETCD qui confirme la réception par un paquet de confirmation d'interruption.
L'interruption est transmise à l'ETTD distant qu'il l'acquitte de la même façon.
Le champ d'information d'un paquet d'interruption ne contient qu'un seul octet de données mais son transfert peut être beaucoup plus rapide car il n'est pas soumis au contrôle de flux.
Pour transmettre un paquet d'interruption il faut que le précédent ait été acquitté par une confirmation d'interruption.

3.I.4.3.6 Procédure de réinitialisation

Cette procédure est utilisée pour réinitialiser une communication virtuelle. Pour cela, elle supprime, dans les deux sens, tous les paquets de données ou d'interruption relatifs à ce circuit virtuel qui peuvent se trouver dans le réseau. Cette procédure n'est applicable que dans l'état transfert de données. La réinitialisation remet à 0 la limite inférieure de la fenêtre (dans les 2 sens) et la numérotation des paquets en émission reprend à partir de 0.



Remarque 1. – Cette transition se produit à l'arrivée en fin de course d'un temporisateur dans le réseau.

Fig. 3.27 - Procédure de réinitialisation : Diagramme des états pour une voie logique .

L'ETTD peut demander une réinitialisation en transmettant un paquet de demande de réinitialisation sur la voie logique concernée.
L'ETCD confirme par un paquet confirmation de réinitialisatio
L'ETCD peut également demander une réinitialisation en transmettant un paquet d'indication de réinitialisation à l'ETTD qui répond par un paquet confirmation de réinitialisation.

Les collisions de réinitialisation sont résolues de la même manière que les collisions de libération.

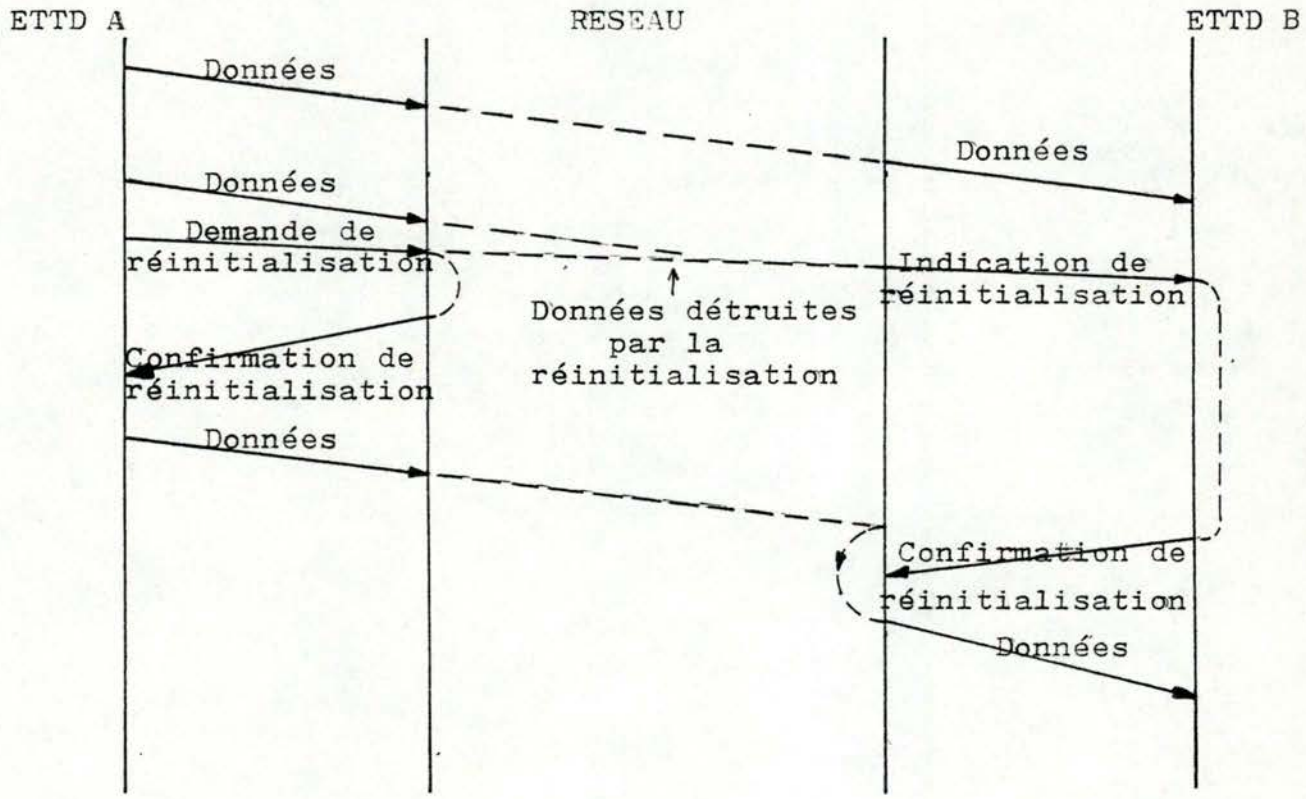
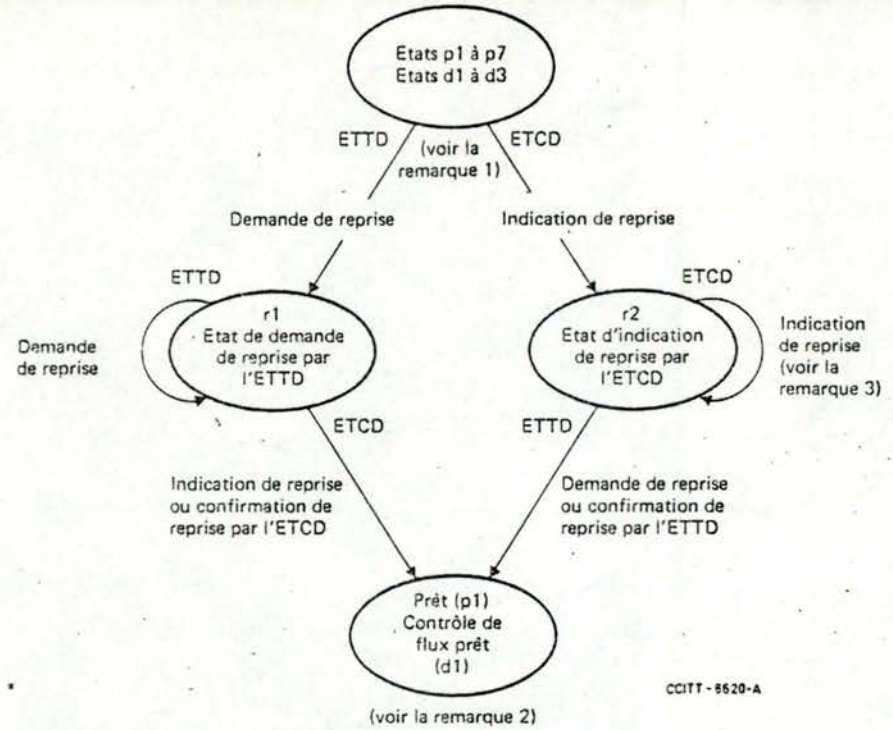


Fig. 3.28 - Exemple de réinitialisation par l'ETTD A .

3.I.4.4 Procédure de reprise

La reprise est une remise à zéro générale sur l'ensemble des voies logiques de l'interface ETTD-ETCD. Les voies logiques utilisées pour des circuits virtuels commutés sont placées dans l'état prêt (ce qui correspond à une libération) et celles utilisées pour les circuits virtuels permanents sont placées dans l'état contrôle de flux prêt (ce qui correspond à une réinitialisation).



Remarque 1. – Etats p1 à p7 pour les communications virtuelles ou états d1 à d3 pour les circuits virtuels permanents.
Remarque 2. – Etat p1 pour les communications virtuelles ou état d1 pour les circuits virtuels permanents.
Remarque 3. – Cette transition se produit à l'arrivée en fin de course d'un temporisateur dans le réseau.

Fig. 3.29 - Procédure de reprise :
Diagramme des états pour une voie logique .

Bien que cela ne soit pas dit explicitement dans l'avis X.25, il est évident qu'une reprise doit être répercutée sous forme de libération et de réinitialisation à l'autre extrémité de chaque circuit virtuel rattaché à l'interface concerné.

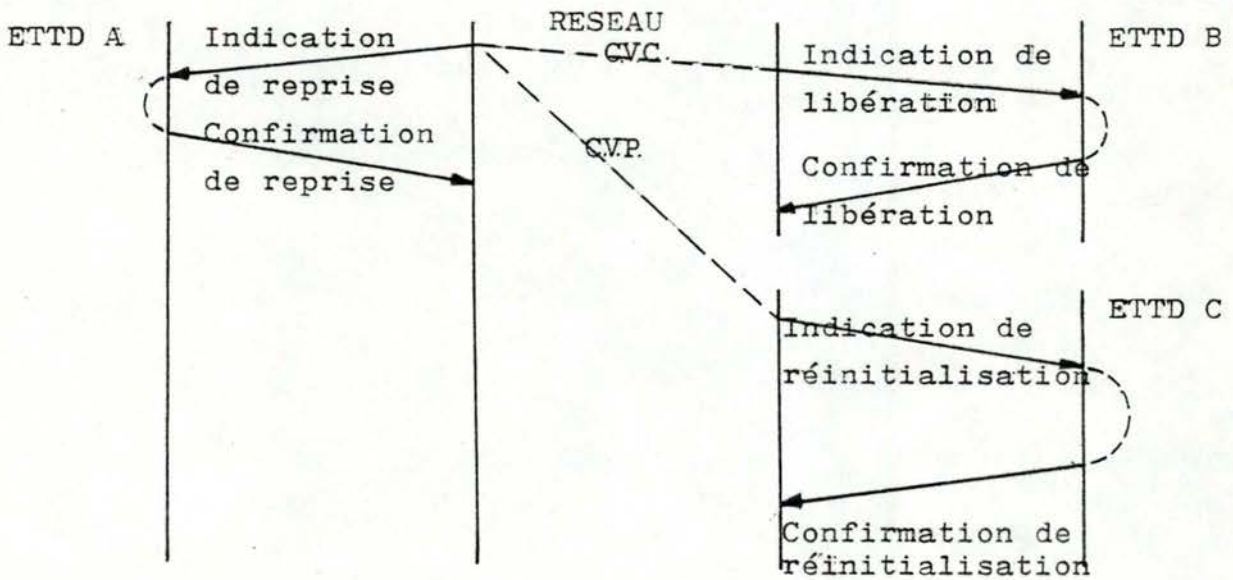


Fig. 3.30 - Exemple de reprise sur la liaison de l'ETTD A qui a :-un circuit virtuel commuté avec l'ETTD B,
-un circuit virtuel permanent avec l'ETTD C.

3.I.4.5 Temporisateur

X.25 ne définit aucun temporisateur pour le niveau 3. Celui-ci semble pourtant nécessaire pour contrôler les détails de réponse au niveau de la transmission des paquets de données, des procédures d'établissement, de réinitialisation et libération, etc.
La durée de temporisation dépend du caractère local ou de bout en bout des réponses attendues.

3.I.4.6 Services complémentaires d'usager

X.25 définit un certain nombre de services complémentaires qui peuvent être proposés par certains réseaux à leurs abonnés.
Certains sont souscrits à l'abonnement et pour une période déterminée, les autres sont demandés lors de l'établissement d'un circuit virtuel portant sur la durée de celui-ci.

Exemples : -Taxation au demandé.

-Sélection des paramètres de contrôle de flux (taille de la fenêtre, largeur maximum des paquets, etc.).

-Retransmission de paquets par l'ETCD.

-Voies logiques spécialisées.

3.I.4.7 Conclusions

Le niveau 3 de X.25 est beaucoup moins spécifique que le niveau 2.
L'ensemble des services offerts par un réseau doivent être plus amplement définis par l'administration responsable en fonction des choix qui ont été faits au niveau de l'implémentation. Ces problèmes sont étudiés plus en détail dans le mémoire de P. Lambion (8) consacré à l'implémentation du niveau paquet et de son interface avec les utilisateurs.

3.2 Le niveau supérieur ou protocole

Si la communication virtuelle définie par X.25 permet le transfert de données entre les ETTD distants.
Pour que ceux-ci puissent communiquer il faut encore qu'ils parlent le même langage, d'où la nécessité de définir des règles d'échange à un niveau supérieur au protocole X.25. Il s'agit de conventions entre les interlocuteurs qui peuvent avoir un caractère tout à fait privé (par exemple : le siège central d'une entreprise qui communique via un réseau public avec l'ensemble de ses succursales suivant un protocole interne à l'entreprise), ou bien, un caractère public comme dans le cas de services publics de base de données ou de centre de calcul.
Pour les applications nécessitant une grande sécurité au niveau des échanges (tels que les applications banquaires) il faut prévoir un contrôle de bout en bout supérieur à X.25 car celui-ci est rarement fourni par le réseau.
Certaine de ces fonctions peuvent être réalisées directement à partir d'éléments proposés au niveau paquet tels que :

- -les procédures d'interruption pour le transfert de bout en bout de données de contrôle;
- -le champ de données de l'utilisateur situé dans le paquet d'appel pour spécifier le protocole supérieur à utiliser;
- -le champ cause de libération dans les paquets de libération pour l'échange de diagnostics d'erreur au niveau supérieur;
- -etc.

Une étude de ce niveau dans le cadre d'une communication entre deux ordinateurs est faite dans le mémoire de P. Lambion (8).

3.3 Découpe et répartition du travail

3.3.I Les objectifs, l'existant

Nous voulions concevoir et réaliser une liaison X.25 entre 2 ordinateurs de façon à permettre l'échange de données entre les deux machines et qu'au moins une des deux se comporte comme un ETTD entièrement compatible avec l'avis X.25 (à tous les niveaux).
Afin de ne pas alourdir la réalisation, il a été décidé que les deux ordinateurs seraient du même type.

Les moyens dont nous disposons au départ sont :

I. L'ordinateur pdp II/45 de l'unité d'Informatique muni du système d'exploitation multi-utilisateurs UNIX (IO).

2. Un ordinateur pdp II/IO connecté au II/45 par un interface parallèle.
Des programmes peuvent être chargés dans le II/IO à partir du II/45.

3. Un système de développement basé sur un microprocesseur Motorola 6800 et disponible à l'unité ELHY.

L'unité d'informatique nous offre l'outil nécessaire au développement du logiciel, le laboratoire de l'unité ELHY est équipé pour développer le matériel qui sera requis dans ce projet.

3.3.2 Evolution du problème en vue de l'implémentation

3.3.2.I Problème initial

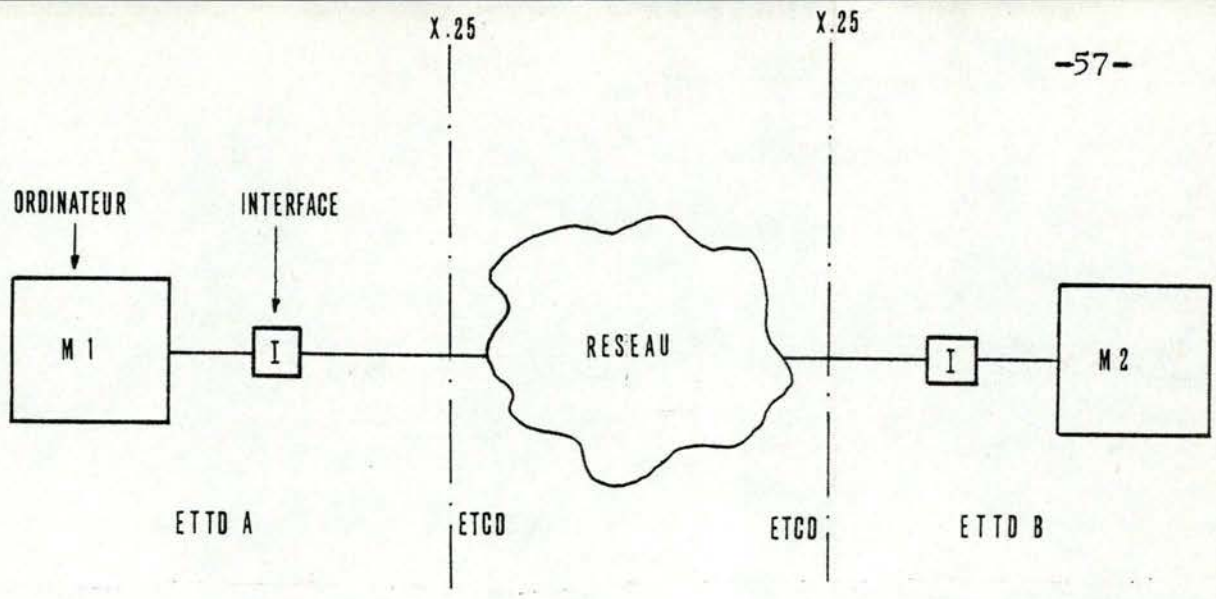


Fig. 3.31 - Communication via un réseau entre 2 ordinateurs .

La figure 3.3I nous montre une configuration dans laquelle deux ordinateurs (MI et M2) sont connectés via un réseau. Chaque ordinateur joue un rôle d'ETTD.
Le réseau doit remplir au moins les fonctions de deux ETCD. Notre étude portant sur le protocole d'accès ETTD-ETCD, la structure interne du réseau sort de nos préoccupations.
Cette solution nécessite la réalisation de deux jonctions X.25 ETTD-ETCD, plus l'étude des fonctions nécessaires pour simuler le comportement du réseau.

3.3.2.2 Simplifications

Afin d'éviter les problèmes liés à la simulation du réseau, nous avons limité la réalisation à une seule jonction X.25.

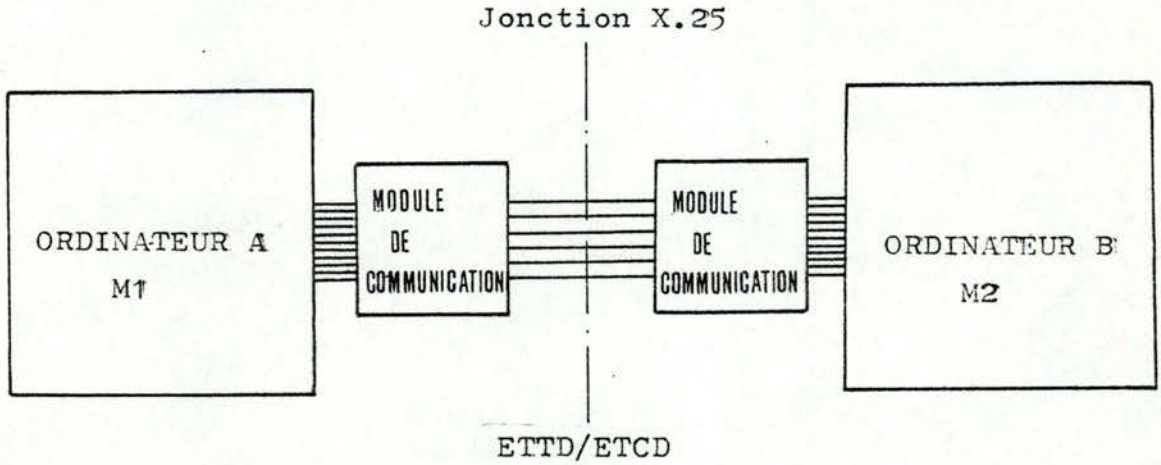


Fig. 3.2 - Liaison X.25 entre 2 ordinateurs , l'un considéré comme un ETTD et l'autre simulant l'ETCD .

L'ordinateur M1 joue le rôle de l'ETTD et est l'élement de base de notre étude, l'ordinateur M2 simule l'ETCD. Cette configuration est plus simple à tester et peut être réalisée avec moins de matériel.

3.3.3 Répartition du travail

3.3.3.I Découpe logique de l'ensemble des fonctions X.25

Les critères de découpe sont les suivants :

-permettre un développement en parallèle des différents modules;

-avoir le moins possible de dépendance entre modules;

-pouvoir tester progressivement les fonctions réalisées.

Les fonctions de X.25 étant réparties sur trois niveaux quasi indépendants, nous avons gardé cette structure pour notre découpe.
De plus, le niveau trame peut être décomposé en deux sous-niveaux (voir 3.I.3) trame I et trame 2.
Il faut encore y ajouter un niveau supérieur à X.25 qui fait l'interface entre les utilisateurs et le niveau paquet.
Ce qui nous donne les modules suivants :

a) Niveau physique: toutes les fonctions requises par X.2I ou X.2I bis.
Adaptation aux caractéristiques électriques (X.26,X.27).
Etablissement et libération de la communication physique.

b) Niveau trame I:

I) Gestion de la ligne en émission:

-prise en charge des trames à émettre pour le primaire (commandes) et le secondaire (réponses).

-calcul du FCS.

-insertion de "0" pour la transparence
-ajouter des fanions pour la synchronisation
-avorter les trames inutiles (dépassées par une demande plus prioritaire)
-envoi des bits en série sur la ligne d'émission
-remplissage entre trames.

2) Gestion de la ligne en réception

-détecter les bits en série sur la ligne de réception
-supprimer des fanions de synchronisation
-enlever les 0 de transparence
-contrôler le FCS reçu
-éliminer les trames invalides
-passer les champs A,C,I des trames correctes au niveau trame 2.

c) Niveau trame 2

-analyser les trames correctement reçues par trame I
-contrôler l'adresse, séparer les commandes et les réponses.

Fonctions du primaire

-prendre en charge les paquets à émettre
-générerles trames de commandes (SARM, DISC, INFO)
-gérer le temporisateur, V(S), V(N)
-traiter les réponses.

Fonctions du secondaire

-traiter les commandes
-gérer V(R)
-générer les réponses -(RR, RNR, REJ, UA, CMDR)
-extraire les paquets des trames I reçues en séquence et les passer au niveau paquet.

d) Niveau paquet

-prendre en charge le transfert des blocs d'information du niveau utilisateur

-établir les communications virtuelles
-contrôler le flux par gestion de la fenêtre en émission et en réception (pour chaque voie logique)
-détecter les erreurs de procédure au niveau paquet
-entreprendre les actions de recouvrement de ces erreurs

e) Niveau utilisateur

-définir l'ensemble des primitives nécessaires pour que les utilisateurs au niveau application puissent utiliser le service de communication virtuelle du niveau paquet
-gérer : -l'allocation des circuits virtuels
-les échanges avec les processus utilisateurs
-l'utilisation des paquets d'interruption
-les diagnostics d'erreurs.

3.3.3.2 Architecture finale du projet

I. L'ETTD

Le rôle de l'ETTD est rempli par un ordinateur pdp II/45.
Afin d'éviter une surcharge trop importante pour le système d'exploitation de cette machine, un micro-ordinateur lui est adjoint. Celui-ci joue le rôle de processeur frontal et gère une partie du protocole X.25.

2. L'ETCD

Sa structure est semblable à celle de l'ETTD, le micro-ordinateur étant étudié pour simuler le comportement de l'ETCD au niveau de la jonction physique.

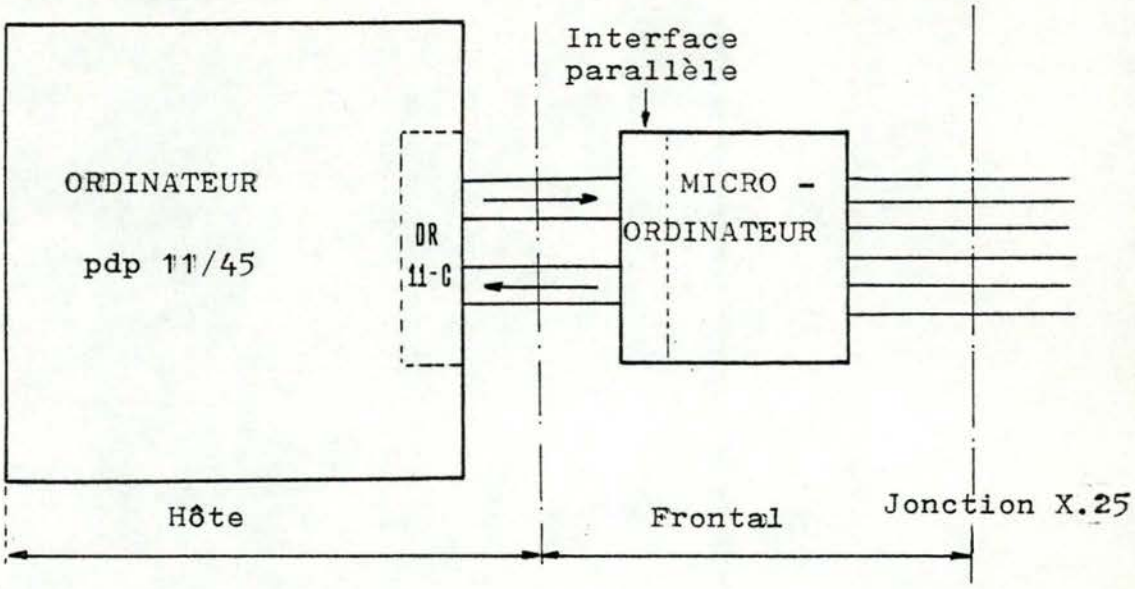


Fig 3.33 - Structure de l'ETTD (ou de l'ETCD) .

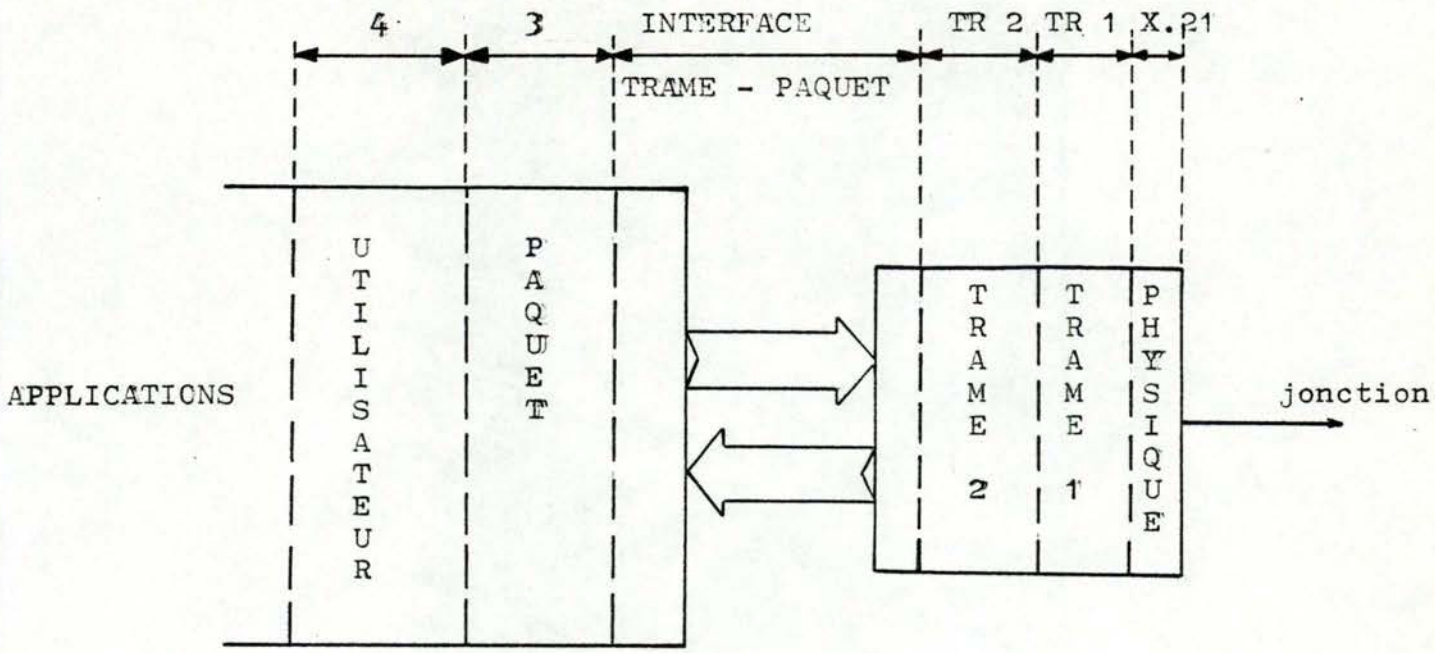
Pour connecter le micro-ordinateur sur la pdp II/45 on utilise du côté pdp un "General device interface" de type DR II-C (I5).
Le micro-ordinateur sera équipé d'un interface compatible avec le DR II-C dont les spécifications seront données au chapitre 4.2 .

3.3.3.3 Affectation des fonctions X.25 au matériel

Le niveau physique et le niveau trame (trame I + trame 2) sont implémentés sur le micro-ordinateur.

Le niveau paquet et le niveau utilisateur sont implémentés sur la pdp II/45.

Les paquets sont échangés sur l'interface entre la pdp et le micro-ordinateur ainsi qu'un certain nombre de messages de contrôle échangés ou le niveau paquet et le niveau trame.



Le niveau utilisateur est en fait l'interface entre les applications exécutées sous le contrôle du système d'exploitation UNIX et les fonctions de communication offertes par le niveau PAQUET .

Fig. 3.34 - Répartition des niveaux dans l'architecture .

3.3.4 Planification

Après une étude générale réalisée par l'ensemble du groupe XYZ , le travail a été divisé comme suit :

-G.Zone et Y. Beudin sont chargés de l'étude et de la réalisation du micro-ordinateur, de l'implémentation du protocole physique (X.21) des routines de gestion du matériel réalisant les fonctions de trame 1 et la communication avec la pdp.

-J. Art est chargé de l'étude et l'implémentation du niveau trame 2 (automate gérant la procédure trame) et d'élaborer les spécifications de l'interface trame 2 - trame I.

- P. Lambion s'occupe de l'implémentation du niveau paquet et du niveau utilisateur sur le pdp II/45, y compris les répercutions de cette implémentation au niveau du système d'exploitation UNIX.

Les trois parties étant développées simultanément, la mise au point du niveau trame 2 doit se faire sur une autre machine en attendant que le micro-ordinateur soit opérationnel.
Le mini-ordinateur pdp II/IO déjà connecté par un DR II-C sur le pdp II/45 vient à point pour ce travail.
Les programmes sont écrits en langage C qui est un langage de haut niveau, ce qui permet d'être peu dépendant du matériel sur lequel les programmes doivent être exécutés.
L'intégration des travaux exécutés séparément se fera en fonction de l'état d'avancement de ceux-ci.

= = = = = = = = = =

4. Analyse et réalisation du niveau trame

Ce chapitre est consacré à l'implémentation du niveau trame, aussi bien pour l'ETTD que pour l'ETCD.
Pendant la phase d'analyse les principes suivants ont été appliqués.

1) La compatibilité avec l'avis X.25 doit être assurée.

2) Le comportement de l'ETTD est inspiré de celui de l'ETCD sauf spécifications contraires dans l'avis X.25 .

3) Dans le but d'obtenir une réalisation aussi symétrique que possible de la jonction X.25, la procédure LAP version I976 a été retenue comme protocole au niveau trame.

4) Un protocole a été défini pour l'échange des paquets et des messages de contrôle entre le niveau trame implémenté sur un processeur frontal et le niveau paquet implémenté sur l'ordinateur hôte.

4.I Structure générale du niveau trame

Cette structure est illustrée par la figure 4.I .
L'élément central est l'automate de gestion de la procédure trame 2.
Son rôle est de traiter les trames reçues sans erreur de transmission et les messages de contrôle venant du niveau paquet.
Pour chacun de ces éléments, il entreprend un certain nombre d'actions telles que :

- -la mise à jour des variables de la transmission
- -la génération des commandes et des réponses à envoyer sur la ligne
- -la mise en marche ou l'arrêt du temporisateur.

L'arrivée en fin de course du temporisateur est également traitée par cet automate.

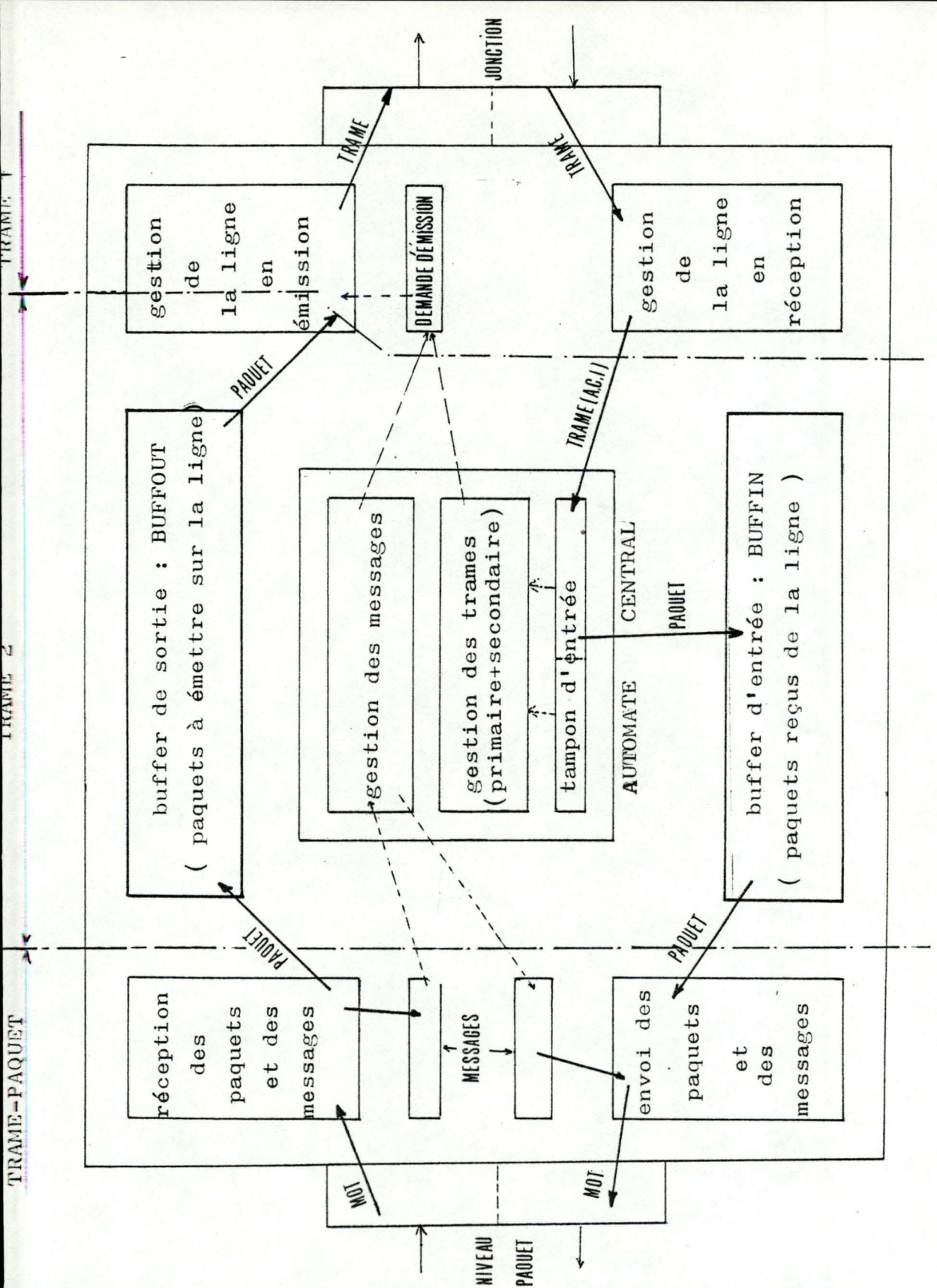


Fig. 4.1 - Sructure générale du niveau trame .

En plus de l'automate central, on trouve quatre modules chargés de la gestion des entrées - sorties.

Gestion de la réception des paquets et messages

Ce module gère la partie réception de l'interface trame-paquet. Il s'occupe également de ranger dans le buffer de sortie les paquets à émettre dans les trames d'information. Les messages reçus du niveau paquet sont rangés dans une zone d'attente avant d'être traités par l'automate central.

Gestion de l'envoi des paquets et messages

Ce module gère la partie émission de l'interface trame-paquet. Il s'occupe d'extraire du buffer d'entrée les paquets reçus dans les trames d'information.
Les messages déposés par l'automate dans une zone d'attente sont transmis au niveau paquet.

Gestion de la ligne en émission

Ce module gère l'émission des trames. Pour cela il exécute toutes les fonctions du niveau trame 1 en émission.
Il reçoit des demandes d'émission de l'automate central et les exécute.
De plus, il s'occupe de l'envoi des trames d'information dans lesquelles il place les paquets à émettre.
Les ordres d'émission, de répétition et de reprise sur temporisateur lui sont fournis par l'automate central.

Gestion de la ligne en réception

Ce module gère la réception des trames.
Pour cela, il exécute toutes les fonctions du niveau trame 1 en réception. Les trames reçues sont rangées dans un double tampon, les trames invalides ou ayant subi une erreur de transmission (FCS erroné) sont éliminées.
Les trames correctes sont analysées par l'automate central qui libère ainsi le tampon de réception.

4.2 Spécification des interfaces

4.2.I Interface trame-paquet

Le niveau trame (dans l'ordinateur frontal) et le niveau paquet (dans l'ordinateur hôte) communiquent via un interface permettant un échange bidirectionnel simultané de mots de I6 bits.
Chaque côté fournit 2 signaux pour le contrôle des échanges (Hand - Shaking).

A. Structure physique

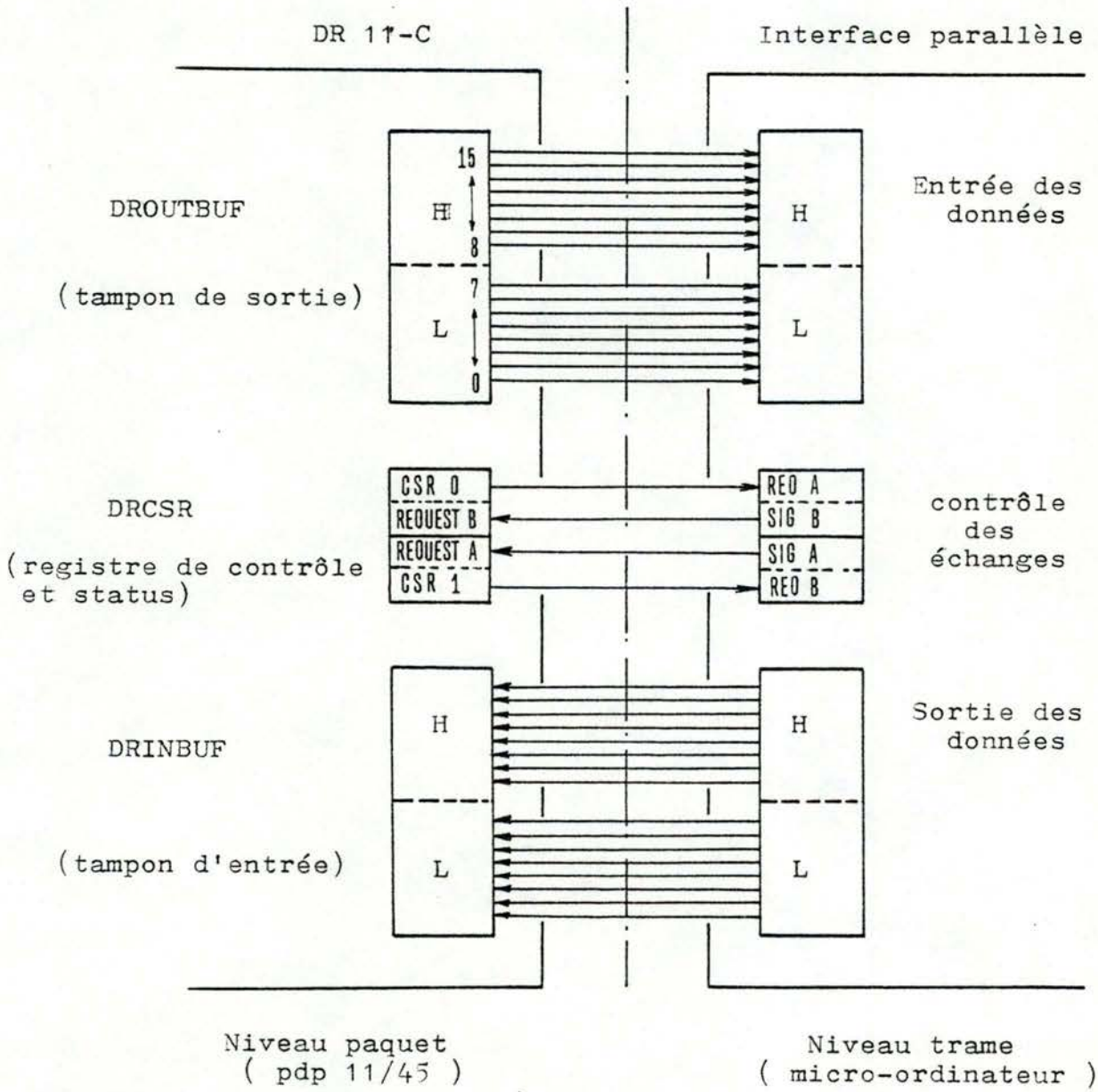


Fig. 4.2 - Structure physique de l'interface trame-paquet .

La figure 4.2 montre les différents éléments de l'interface trame paquet au niveau physique.
Chaque côté possède :

-une entrée de données (I6 bits)

-une sortie de données (I6 bits)

-deux signaux de contrôle (SIG A et SIG B)

-deux entrées de requête (REQ A et REQ B)

SIG A (CSR 0) : utilisé pour signaler qu'un nouveau mot de I6 bits est présent à la sortie des données et peut être lu par l'autre station (demande d'envoi).

SIG B (CSR I) : utilisé pour signaler que les données en entrée sont lues, et, qu'un nouveau mot de I6 bits peut être envoyé par l'autre station (accusé de réception).

REQ A (REQUEST A) : permet de recevoir le SIG A de l'autre station.

REQ B (REQUEST B) : permet de recevoir le SIG B de l'autre station.

B. Protocole d'échange des paquets et messages

I. Messages

Les 2 niveaux dialoguent au moyen de messages dont voici la liste :

NIVEAU PAQUET		NIVEAU TRAME
-demande de connexion	→	
	←	indication de connexion (ou de réinitialisation)
-demande de déconnexion	→	
	←	indication de déconnexion

-demande de status →
← indication de status
← demande de status
-indication de status →
← indication d'erreur "GRAVE"
-entête de paquet →
← entête de paquet

Chaque message est en fait un mot de I6 bits transmis sur l'interface physique et dont le format est donné à la figure 4.3 .

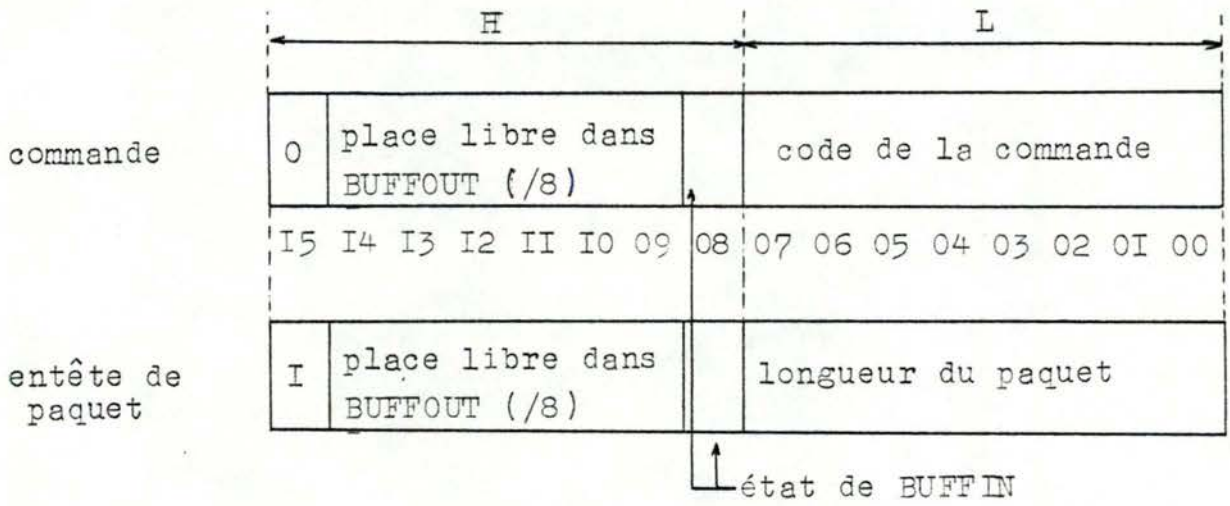


FIG. 4.3 - Format des messages .

2. Codification des messages

a. type commande

bit I5 ---> mis à 0 pour une commande
bit I4-9 ---> indique l'état de buffout (nombre de places libres / 8)
bit 8 ---> mis à I si buffin est rempli
bit 7-0 ---> donne le code de la commande

codification des commandes

bit 7 ---> erreur grave entraînant une procédure de maintenance

bit 6-4 ---> type de l'erreur grave
bit 3 ---> indication de status
bit 2 ---> demande de status
bit I ---> déconnexion (demande ou indication)
bit 0 ---> connexion (demande ou indication) ou indication de réinitialisation

b. type entête

bit I5 ---> mis à I pour une entête
bit I4-9 ---> indique l'état du buffout (nombre de places libres / 8)
bit 8 ---> mis à I si buffin est rempli
bit 7-0 ---> donne la longueur réelle (en bytes) du paquet qui va suivre

Remarques : I) Les erreurs graves détectées au niveau trame sont signalées au niveau paquet pour indiquer un mauvais fonctionnement de la liaison trame qui devient indisponible pour le transport des paquets.

Les erreurs graves sont codées comme suit
bit 6 ---> non réponse à SARM répété N2 fois
bit 5 ---> non réponse à DISC répété N2 fois
bit 4 ---> bit de réserve, pas utilisé jusqu'à nouvel ordre.

2) La demande de connexion venant du niveau paquet est ignorée par le niveau trame si celui-ci n'est pas dans l'état déconnecté (0).

3) La demande de déconnexion est ignorée si le niveau trame n'est pas dans l'état transfert de données (7).

4) Pendant la phase de transfert de données, le niveau trame envoi un message indication de réinitialisation chaque fois que la liaison trame est réinitialisée dans un sens ou dans l'autre.
Ceci afin de signaler que certains paquets ont pu être perdus.

5) Le bit 8 permet au niveau trame d'indiquer que son buffer contenant les paquets reçus dans les trames d'information est rempli. Ce qui place le secondaire dans l'impossibilité d'accepter de nouvelles trames d'information.
Le niveau paquet doit alors si possible augmenter le rythme de traitement des paquets reçus.

3. Principe de l'échange des paquets

Les paquets sont échangés mot par mot sur l'interface.
Chaque paquet est précédé d'un message "entête de paquet" qui contient la longueur réelle (en octets) du paquet suivant cette entête.
Les paquets contenant un nombre impair d'octets sont complétés par un octet quelconque pour obtenir un nombre entier de mots.

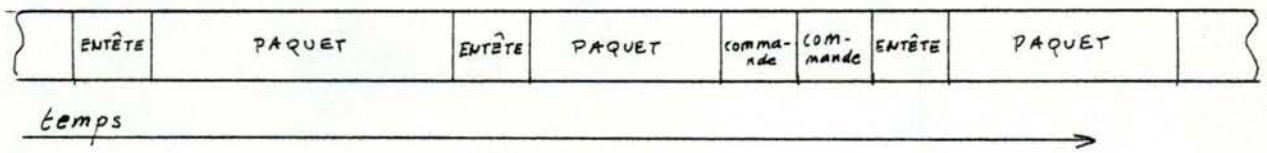


Fig. 4.4 - Enchaînement des messages et des paquets sur l'interface trame-paquet .

Chaque côté possède un buffer d'entrée (BUFFIN) et un buffer de sortie (BUFFOUT).
Avant d'émettre un paquet, l'émetteur doit s'assurer qu'il reste suffisamment de place dans le buffer de réception de l'autre côté. Il utilise pour cela un compteur de place libre (en octets) qui est régulièrement remis à jour grâce aux messages venant dans l'autre sens (bits I4-9).
Ceci permet d'éviter de rester bloqué pendant le transfert d'un paquet, ce qui empêcherait la transmission de messages plus prioritaires.
Le message demande de status permet à l'émetteur de demander un rafraîchissement de son compteur en l'absence de trafic dans l'autre sens de transmission. Le message indication de status est utilisé pour transmettre l'état des buffer quand il n'y a pas d'autre message à envoyer.

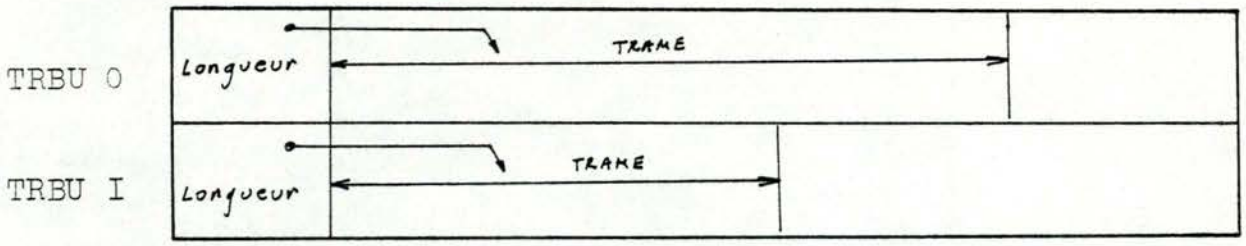
4. Synchronisation ou démarrage des machines

Chaque machine indique son activité en positionnant son SIG B à I. La machine lancée la première attend l'autre. Lorsque les 2 machines sont actives, le niveau paquet et le niveau trame entre en fonctionnement.

4.2.2 Interface trame2 - trame I

A. Réception

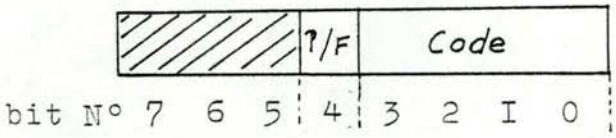
Les trames reçues de la ligne sont rangées alternativement dans un double tampon d'entrée. Une moitié servant à ranger la trame qui est reçue de la ligne, l'autre contenant la trame traitée par l'automate. Une zone est réservée pour ranger la longueur de la trame contenue dans un tampon. Le champ est garni par trame I dès que la trame peut être traitée par trame 2. Il est remis à 0 dès que la trame a été traitée ce qui indique que le tampon est vide et peut à nouveau être rempli par trame 1.



Double tampon d'entrée des trames en réception

B. Emission

L'automate contrôle le fonctionnement de l'émetteur à l'aide de deux zones. La première permet à l'automate de demander l'émission d'une trame de type S ou U.
L'automate dépose un code correspondant à la trame à émettre dans cette zone.
L'émetteur remet la zone à 0 dès que la demande est prise en charge.



zone de demande d'émission .

Codification

TRAME	OCTAL
RR	0I
RNR	02
REJ	03
SARM	05
DISC	06
UA	07
CMDR	0I0

La seconde zone permet à trame 2 de contrôler le transfert des trames d'information effectué par trame 1.

Codification : 0 --> l'émission des trames I n'est pas permise.

I --> les trames I doivent être émises en séquence en respectant la gestion de la fenêtre.

2 --> relancer l'émission à partir de la première trame non encore acquittée (N(S) := V(N)).
Dès que cette commande est effectuée, la zone de contrôle doit être remise à I.

3 --> indique une reprise sur temporisateur, il faut réémettre la première trame non encore acquittée avec le bit P mis à I.
Dès que cette commande est effectuée, la zone de contrôle doit être remise à 0.

4.3 Elaboration des tables séquentielles

Les règles d'évolution de l'automate qui gère la procédure du niveau trame peuvent être définies au moyen de tables séquentielles. Ce mode de représentation est très utile pour l'analyse. Il permet de vérifier si tous les transitions sont définies.
De plus, la conversion sous forme de programme est aisée si les actions sont bien décrites dans la table.
Ces tables ont été réalisées à partir des définitions de procédure présentes dans l'avis X.25. Elles sont valables pour un ETTD.
Lorsque, dans X.25, une action n'est pas explicitement décrite pour l'ETTD, le comportement de l'ETCD est pris comme modèle.
Tout ce qui n'est pas défini dans X.25 a fait l'objet de choix d'implémentation.

Ces choix portent sur les points suivants :

I) La définition du protocole entre le niveau trame et le niveau paquet est propre à l'implémentation.

2) Actions entreprises après N2 retransmissions infructueuses des trames SARM et DISC.

a) Non réponse à SARM : l'erreur est signalée au niveau paquet et une procédure de déconnexion de la liaison est engagée au niveau trame.

b) Non réponse à DISC : l'erreur est signalée au niveau paquet et la liaison est considérée comme étant dans l'état déconnecté.

3) Le bit P n'est jamais mis à I dans les trames SARM ou DISC émises. La réception d'une trame UA avec le bit F mis à I correspond donc à une réponse invalide.

4) En cas de réinitialisation de la liaison, tous les paquets qui ont été émis sont éliminés du buffer de sortie (même s'ils n'ont pas été acquittés).
Ceci est fait dans le but d'éviter de retransmettre 2 fois le même paquet. La perte d'un ou plusieurs paquets reste possible, c'est pourquoi la réinitialisation est toujours signalée au niveau paquet.

5) Le niveau trame n'offre son service de transport des paquets que pendant la phase de transfert de l'information (état 7 de la table I).
En dehors de cette phase, les paquets reçus du niveau paquet et ceux restants dans les buffers du niveau trame sont éliminés.

6) Les trames contenant une erreur dans le champ d'adresse sont éliminées de la même façon que celles ayant subit une erreur de transmission (FCS).

Pour la description de l'automate, il est préférable de séparer la phase d'établissement ou de libération de la liaison et la phase de transfert de données.

a) Phase d'établissement ou de libération

Dans cette phase, seules les trames SARM, DISC et UA sont utilisées. Les autres trames sont ignorées et n'ont pas d' influence sur le comportement de l'automate.
Il est inutile de faire la distinction entre primaire et secondaire, car les deux sens de la liaison doivent absolument être établis pour atteindre l'état transfert de données.
La figure 4.5 donne un diagramme d'état de l'automate pendant la phase d'établissement et de libération de la liaisor
Ce mode de représentattion illustre mieux la dynamique de l'automate mais ne convient pas pour une représentation détaillée des actions entreprises par l'automate (le diagramme deviendrait illisible).
La table séquentielle numéro I reprend en détail la phase décrite dans le diagramme de la figure 4.5 .

b) Phase de transfert de données

Pendant cette phase (état 7 de la table I), les deux sens de transmission de l'information sont contrôlés séparément. Il convient donc de distinguer le primaire et le secondaire. L'état global de l'ETTD (ou de l'ETCD) sera en fait représenter par deux sous-états :
-l'état du primaire (voir table 2)
-l'état du secondaire (voir table 3).
La réception d'une réponse (CMDR, UA, RR, RNR, REJ) influence l'état du primaire, celle d'une commande (SARM, Information) influence l'état du secondaire.
La réception d'une trame DISC ou d'une demande de déconnexion venant du niveau paquet fait sortir l'automate de l'état transfert de données.

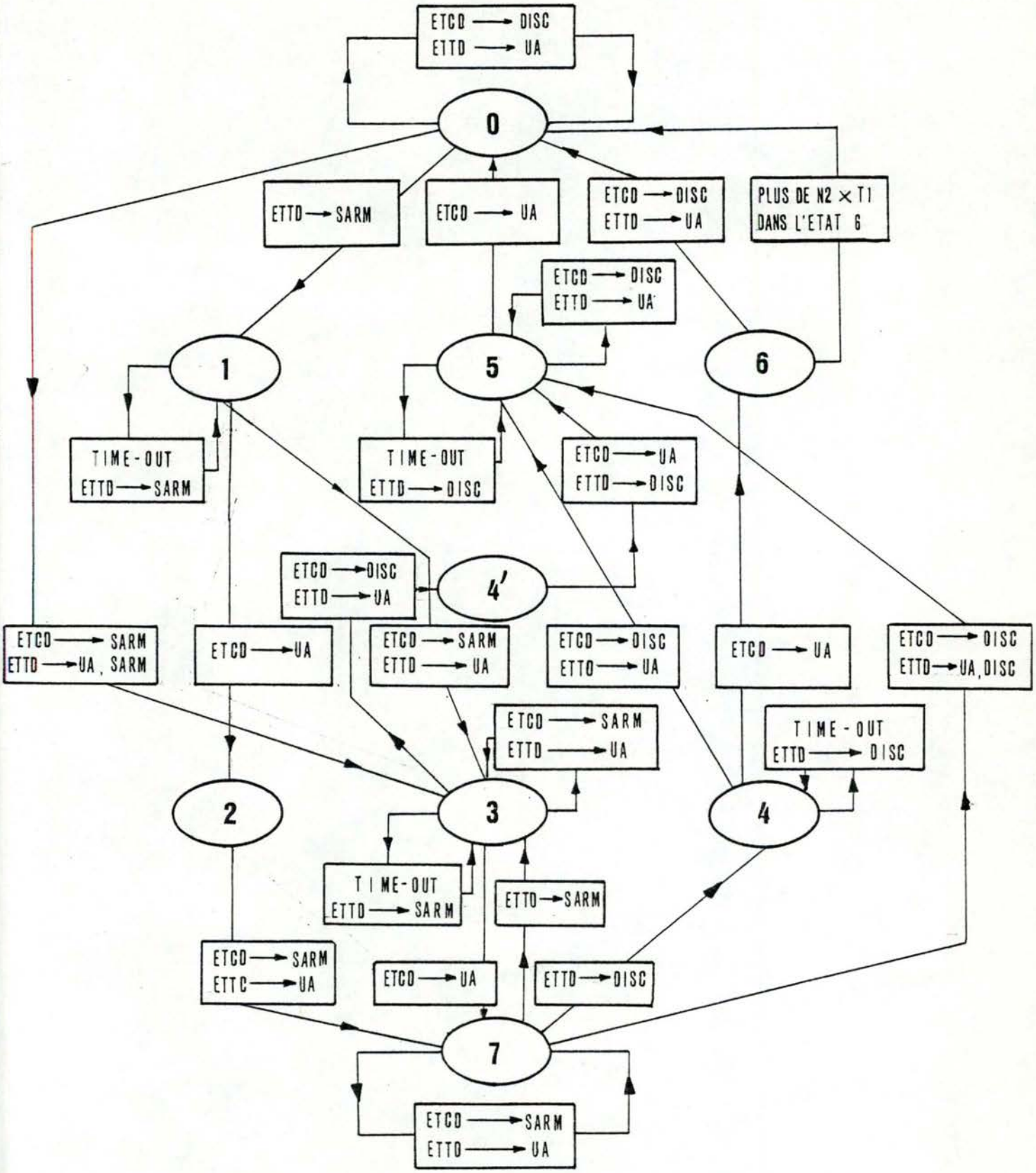


Fig. 4.5 - diagramme d'état de l'automate trame 2 pour l'établissement et la libération de la liaison d'accès .

Table I - Etablissement et libération de la liaison .

ENTREES / ETS	SARM	DISC	UA	TIME-OUT	CR > N2	CONNEXION	DECONNEXION	CMDR. RR,RNR,REJ, REPONSE INV.	I, COMMANDE INV.
0 DECONNECTE	UA (3) SARM CR=0 ARMER T1	UA	IGN.	ERREUR	ERREUR	SARM (1) ARMER T1 CR=0	MES.D.	IGN.	IGN.
1	UA (3)	UA (4')	CR=0 (2) ARMER T1	SARM CR+1 ARMER T1	MES.E. (5) CR=0 ARMER T1 DISC	IGN.	IGN.	IGN.	IGN.
2	UA (7) DESARMER T1 INIT. MES.C.	UA (5) DISC CR=0 ARMER T1	IGN.	ARMER T1 CR+1	(4) CR=0 ARMER T1 DISC	IGN.	IGN.	IGN.	IGN.
3	UA	UA (4')	DESARMER T1 (7) INIT MES.C.	SARM ARMER T1 CR+1	MES.E. (5) CR=0 ARMER T1 DISC	IGN.	IGN.	IGN.	IGN.
4	IGN.	UA (5)	ARMER T1 (6) CR=0	DISC ARMER T1 CR+1	MES.E. (0) MES D	IGN.	IGN.	IGN.	IGN.
4'	IGN.	UA	DISC (5) ARMER T1 CR=0	DISC (5) ARMER T1 CR=0	DISC (5) ARMER T1 CR=0	IGN.	IGN.	IGN.	IGN.
5	IGN.	UA	DESARMER T1 (0) MES.D.	DISC ARMER T1 CR+1	MES.E. (0) MES.D.	IGN.	IGN.	IGN.	IGN.
6	IGN.	UA (0) DESARMER T1 MES.D.	IGN.	ARMER T1 CR+1	MES.D. (0)	IGN.	IGN.	IGN.	IGN.
7 TRANSFERT DE DONNEES	VOIR SECONDAIRE	UA (5) DISC CR=0 ARMER T1	VOIR PRIMAIRE	VOIR PRIMAIRE	VOIR PRIMAIRE	IGN.	DISC (4) CR=0 ARMER T1	VOIR PRIMAIRE	VOIR SECONDAIRE

Table séquentielle1: établissement et libération de la liaison

Cette table décrit le comportement de l'automate trame 2 pendant la phase d'établissement ou de libération de la liaison.
Les entrées sont analysées par l'automate qui effectue une action et éventuellement change d'état.
Les actions sont reprises dans les cases de la table.
Lorsqu'il y a lieu de changer d'état, l'état futur est indiqué dans le coin supérieur droit de la case concernée.

Entrées:

SARM, DISC, I : commandes correctes reçues de l'ETCD.

UA,CMDR,RNR,REJ : réponses correctes reçues de l'ETCD.

Commande INV. : trames reçues de l'ETCD portant l'adresse B mais non valide pour une des causes suivantes :
- -champ C non valide pour une commande
- -contient un N(S) incorrect
- -la longueur de la trame est non valable.

Réponse INV. : trames reçues de l'ETCD portant l'adresse A, mais non valide pour une des causes suivantes :
- -champ C non valide pour une réponse
- -contient un N(R) incorrect
- -contient un bit F mis à un alors que le primaire n'est pas en reprise sur temporisateur.
- -la longueur de la trame est non valable.

Time-out : indique que le temporisateur est arrivé en fin de course

CR$\geq$2 : indique que le compteur de retransmission (CR) contient une valeur supérieure ou égale à N2 (nombre maximum de retransmissions) au moment ou le temporisateur arrive en fin de course.

Connexion : le niveau paquet demande au niveau trame d'établir la liaison avec l'ETCD.

Déconnexion : le niveau paquet indique au niveau trame qu'il faut déconnecter la liaison avec l'ETCD.

Actions :

SARM : envoi d'un SARM avec P à 0 vers l'ETCD.

DISC : envoi d'un DISC avec P à 0 vers l'ETCD.

UA : envoi d'un UA vers l'ETCD pour acquitter une commande SARM ou DISC, le bit F est mis à I si le bit P de la commande était à I.

CR = 0 : remise à 0 du compteur de retransmission.

CR + I : le compteur de retransmission est incrémenté d'une unité.

ARMER TI : le temporisateur est armé ou remis au début de la temporisation s'il était déjà armé.

DESARMER TI : le temporisateur est désarmé.

MES. C. : envoi d'un message "indication de connexion" vers le niveau paquet.

MES. D. : envoi d'un message "indication de déconnexion" vers le niveau paquet.

MES. E. : envoi d'un message "erreur fatale" vers le niveau paquet.

IGN : la trame reçue de l'ETCD ou la demande reçue du niveau paquet est ignorée.

ERREUR : indique que l'événement d'entrée ne pouvait pas se produire pour un tel état de l'automate trame2 Ce type d'erreur entraîne une procédure de maintenance.

Voir Primaire : report à la table séquentielle décrivant le comportement du primaire pendant la phase de transfert de données.

Voir Secondaire : idem pour le secondaire.

INIT : initialiser le transfert de données.

```
{ CR := 0 ;
  V(S) := 0 ;
  V(N) := 0 ;
  V(R) := 0 ;
  dernier N(R) transmis := 0 ;
  lancer l'émission des trames d'information.
}
```

Table 2 - Actions du primaire pendant la phase de transfert des données .

ETATS \ ENTREES	CMDR	UA	RR, F=0	RR, F=1	RNR, F=0	RNR, F=1	REJ, F=0	REJ, F=1	REPONSE INV.	TIME-OUT	CR≥N2	N(R) DE INFO
A NORMAL	D REI.	IGN.	ACTION 1	D REI.	B ACTION 3	D REI.	ACTION 5	D REI.	D REI.	C REPETITION	ERREUR	ACTION 11
B BLOQUÉ	D REI.	IGN.	A ACTION 2	D REI.	ACTION 4	D REI.	A ACTION 6	D REI.	D REI.	C REPETITION	ERREUR	IGN.
C REVEIL	D REI.	IGN.	IGN.	A ACTION 7	IGN.	B ACTION 8	IGN.	A ACTION 9	D REI.	REPETITION	D REI.	IGN.
D REINITIALISATION	IGN.	A ACTION 10	IGN.	IGN.	IGN.	IGN.	IGN.	IGN.	IGN.	SARM CR←1 ARMER T1	5 MES.E. DISC CR=0 ARMER T1	IGN.

Table séquentielle 2 : Actions du primaire pendant la phase de transfert de données

Entrées : CMDR, UA, RR, RNR, REJ : trames correctement reçues et dont le champ de commande est valide ainsi que la longueur.

N(R) de INFO : numéro N(R) d'une trame de type I, reçue par le secondaire de la même station. Bien que transmis dans une trame de commande, il s'agit en fait d'un élément de réponse pour le primaire.
Les trames I contenant un N(R) incorrect sont donc traités comme des réponses invalides.

Etats : A. Normal : dès que la liaison est établie le primaire se trouve dans l'état normal et peut transmettre des trames d'information.

B. Bloqué : cet état est atteint lors de la réception d'un RNR indiquant un état occupé dans l'autre station. L'émission des trames d'information est arrêté.

C. Réveil : si le temporisateur TI arrive en fin de course pendant la phase de transfert de l'information, le primaire effectue une reprise sur temporisateur. Il passe dans l'état réveil après avoir réémis la première trame I non encore acquittée.

D. Réinitialisation : lors de la réception d'une réponse invalide ou d'une trame CMDR, le primaire enclenche une procédure de réinitialisation qui le fait passer à l'état D.

Actions : REI: reinitialisation d'un sens de transmission :

```
{ arrêter l'émission des trames I;
  envoyer SARM;
  CR := 0;
  armer TI; }
```

Répétition : reprise sur temporisateur :

{relancer l'émission de la trame I
dont le N(S) est égal à V(N) et avec le
bit mis à I;
armer TI;
CR := CR+I;}

Action I : dans l'état normal, réception d'un RR avec
F = 0;

{Si (N(R) ≠ V(N)
alors { V(N) := N(R);
libérer les paquets acquittés
Si (N(R) = V(S))
alors désarmer TI,
sinon armer TI;
sinon: pas d'action. }

Action 2 : dans l'état bloqué, réception d'un RR avec
F = 0;

{ V(N) := N(R);
libérer les paquets acquittés;
Si (N(R) = V(S)
alors désarmer TI;
sinon armer TI;
relancer l'émission des trames d'information en séquence;}

Action 3 : dans l'état normal, réception d'un RNR avec
F = 0;

{ arrêter l'émission des trames d'information;
V(N) := N(R);
CR := 0;
Si (N(R) := V(S)
alors désarmer TI;
sinon armer TI;}

Action 4 : dans l'état bloqué, réception d'un RNR avec
F = 0
{V(N) := N(R);
Si (N(a) := V(s))
alors désarmer TI;
sinon armer TI;}

Action 5 : dans l'état normal, réception d'un REJ avec
F = 0
{Si (N(R) ≠ V(N))
alors V(N) := N(R);
libérer les paquets acquittés;
CR := 0;
Si (N(R) = V(S))
alors désarmer TI;
sinon relancer l'émission des trames
information à partir de celle
dont le N(S) est = à V(N);

Action 6 : dans l'état bloqué, réception d'un REJ avec
F = 0
V(N) := N(R);
Si (N(R) = V(S))
alors désarmer TI;
relancer l'émission des trames
d'information en séquence;
sinon relancer l'émission des trames
d'information à partir de celle
dont le N(S) est = à V(N);

Action 7 : dans l'état réveil, réception d'un RR avec
F = I
Si (N(R) ≠ V(N))
alors V(N) := N(R);
libérer les paquets acquittés;
désarmer TI;
Si (N(R) ≠ V(S))
alors relancer l'émission des trames
d'information à partir de celle

```
                        dont le N(S) est = à V(N);
                    sinon relancer l'émission des trames
                        d'information en séquence;}
```

Action 8 : dans l'état réveil, réception d'un RNR avec F = I

```
{V(N) := N(R);
 CR := 0;
 Si (N(R) = V(S))
    alors désarmer TI;
    sinon armer TI;}
```

Action 9 : dans l'état réveil, réception d'un REJ avec F = I

```
{ Si (N(R) ≠ V(N))
    alors { V(N) := N(R);
            libérer les paquets acquittés;
            CR := 0;
            désarmer TI;}
  Si (N(R) ≠ V(S))
    alors relancer l'émission des trames
          d'information à partir de celles
          dont le N(S) = V(N);
    sinon relancer l'émission des trames
          d'information en séquence.}
```

Action I0 : réception d'un UA dans l'état réinitialisation

```
{ désarmer TI;
  CR := 0;
  libérer les paquets qui ont été déjà émis
  (même non acquittés);
  N(S) := 0;
  V(N) := 0;
  relancer l'émission des trames d'informa-
  tion en séquence;
  envoyer une indication de réinitialisa-
  tion au niveau paquet;}
```

Action II : réception d'un N(R) d'une trame d'information

```
{ Si (N(R) ≠ V(N))
     alors {V(N) := N(R);
            CR := 0;}
  Si N(R) = V(S)
     alors désarmer TI;
     sinon armer TI; }
```

Table 3 - Actions du secondaire pendant la phase de transfert des données .

ETATS \ ENTREES	SARM	I(N(S)=V(R),P=0)	I(N(S)=V(R),P=1)	I(N(S)≠V(R),P=0)	I(N(S)≠V(R),P=1)	COMMANDE INV.	OCCUPATION	FIN D'OCCUPATION
A NORMAL	RAZ	ACK 0	ACK 1	REJ, F=0 (B)	REJ, F=1 (B)	CMDR (C)	RNR (D)	IGN.
B REJET	RAZ (A)	ACK 0 (A)	ACK 1 (A)	IGN.	REJ, F=1	CMDR (C)	RNR (D)	IGN.
C REJET DE COMMANDE	RAZ (A)	CMDR	CMDR	CMDR	CMDR	CMDR (C)	IGN.	IGN.
D OCCUPÉ	RAZ (A)	IGN.	RNR, F=1	IGN.	RNR, F=1	CMDR (C)	IGN.	REJ, F=0 (B)

Table séquentielle 3 : Actions du secondaire pendant la phase de transfert de données

Entrées : I(N(S) = V(R), P = 0) : trame d'information valide reçue en séquence avec le bit P à 0.

I(N(S) = V(R), P = I) : idem avec P à I.

I(N(S) ≠ V(R), P = 0) : trame d'information valide reçue hors séquence avec le bit P à 0.

I(N(S) ≠ V(R), P = I) : idem avec P à I.

Occupation : détection d'une condition d'occupation entraînant le passage dans l'état occupé, par exemple: plus de place dans un buffer.

Fin d'occupation : fin de la condition d'occupation, par exemple: libération de place dans un buffer saturé.

Etats : Normal: dès que la liaison est établie, le secondaire se trouve dans l'état normal et garde cet état tant que les trames d'information sont reçues correctement et en séquence.

Rejet: le secondaire passe dans l'état rejet après avoir envoyé un REJ pour demander une retransmission suite à la détection d'une erreur de séquence.

Rejet de commande: cet état est atteint après l'envoi d'un CMDR pour indiquer le rejet d'une commande non valable. Dans cet état le secondaire attend une réinitialisation venant du primaire de l'autre station.

Occupé: état atteint après la détection d'une condition d'occupation et l'envoi d'un RNR.

Actions : RAZ: remise à 0 sur demande de réinitialisation :

{ V(R) := 0;
dernier N(R) transmis := 0;
envoyer UA avec F = au P reçu dans la com-

mande SARM;
envoyer une indication de réinitialisation au niveau paquet;}

ACK 0:traitement d'une trame I reçue en séquence avec P=0 :

{ Si (buffer non rempli)
V(R) := V(R)+I;
acquitter avec le N(R) d'une trame I si possible, sinon envoyer RR avec F := 0;
extraire le paquet du champ I pour l'envoyer vers le niveau paquet;
Sinon — condition d'occupation }

ACK I:traitement d'une trame I reçue en séquence avec P=I :

{ Si (buffer non rempli)
V(R) := V(R)+I;
envoyer RR avec F := I;
extraire le paquet du champ I pour l'envoyer vers le niveau paquet;
Sinon — condition d'occupation }

CMDR: envoi d'une trame rejet de commande.

REJ: envoi d'une trame rejet de séquence, le champ information de la trame I reçue hors séquence est ignoré.

RNR: envoi d'une trame "réception non prête" pour indiquer un état occupé.

4.4 Implémentation de l'automate du niveau trame 2

La structure du programme réalisant les fonctions du niveau trame 2 est donnée à la figure 4.6

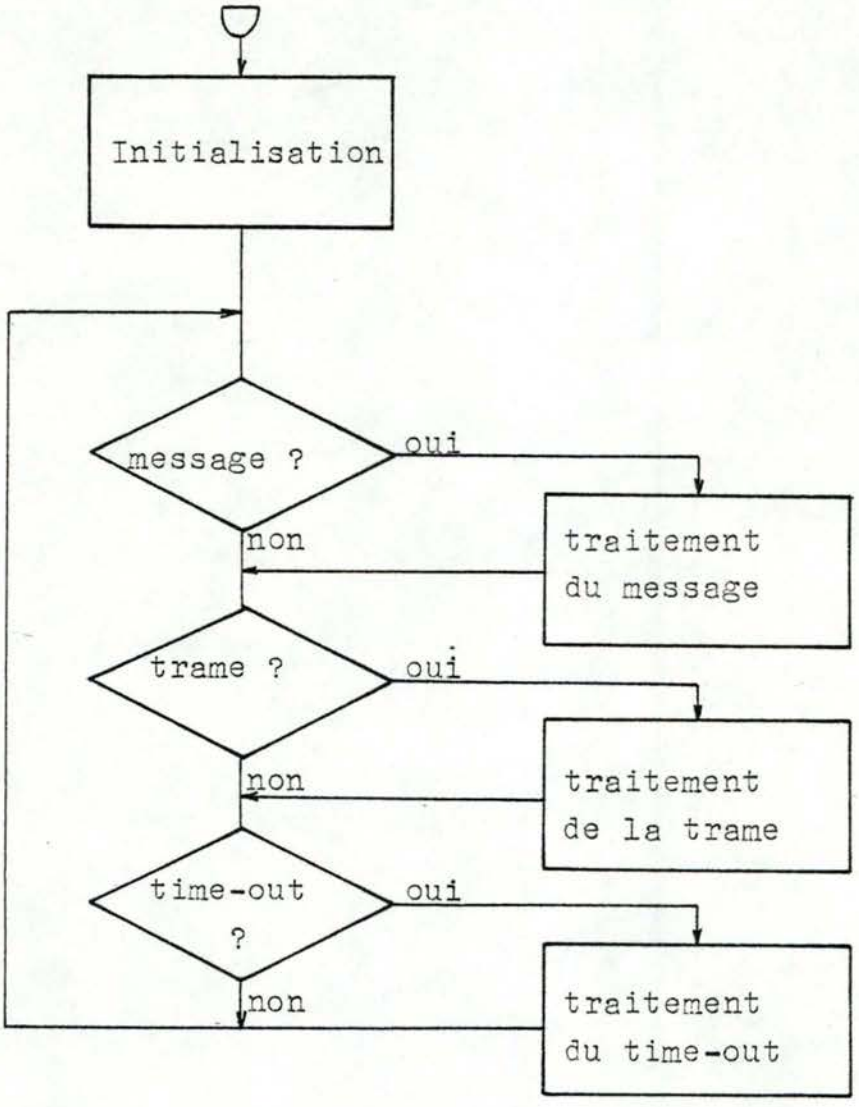


Fig. 4.6 - Structure de traitement de l'automate .

L'automate teste une à une les sources d'entrées possibles qui sont :

I) Les messages : a) -demande de connexion
-demande de déconnexion
venant du niveau paquet

b) -fin de la condition d' occupation; produit par le module qui extrait les paquets de buffin et les passe au niveau paquet.

2) Les trames reçues par le niveau trame I et placées dans le double tampon d'entrée (trbu 0 et trbu I).

3) Le time-out positionné par l'arrivée en fin de course du temporisateur T1.

4.4.I Traitement des messages

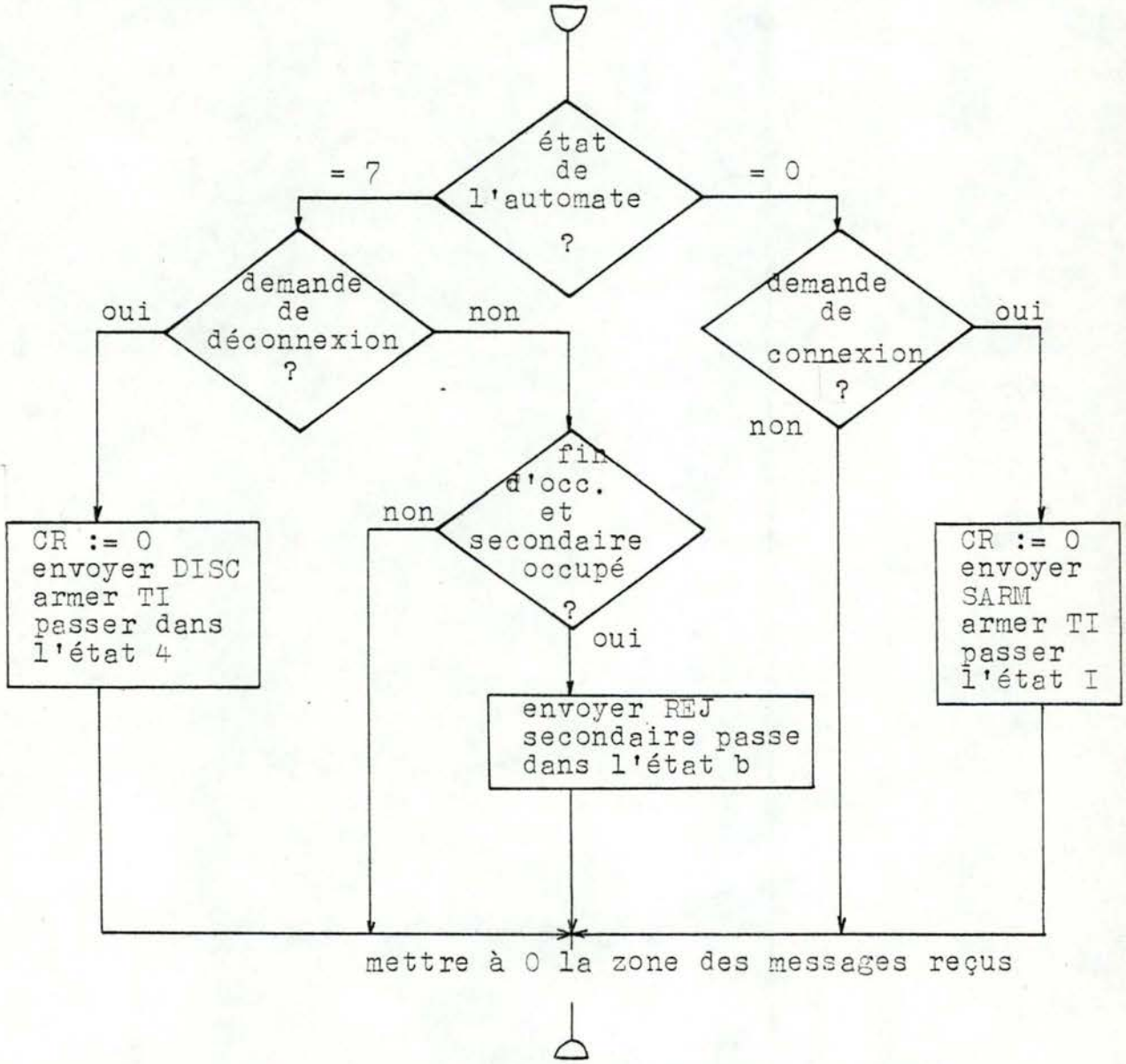


Fig.4.7 - Ce traitement correspondant aux colones "connexion" et "déconnexion" de la table I et "fin d'occupation" de la table 3 .

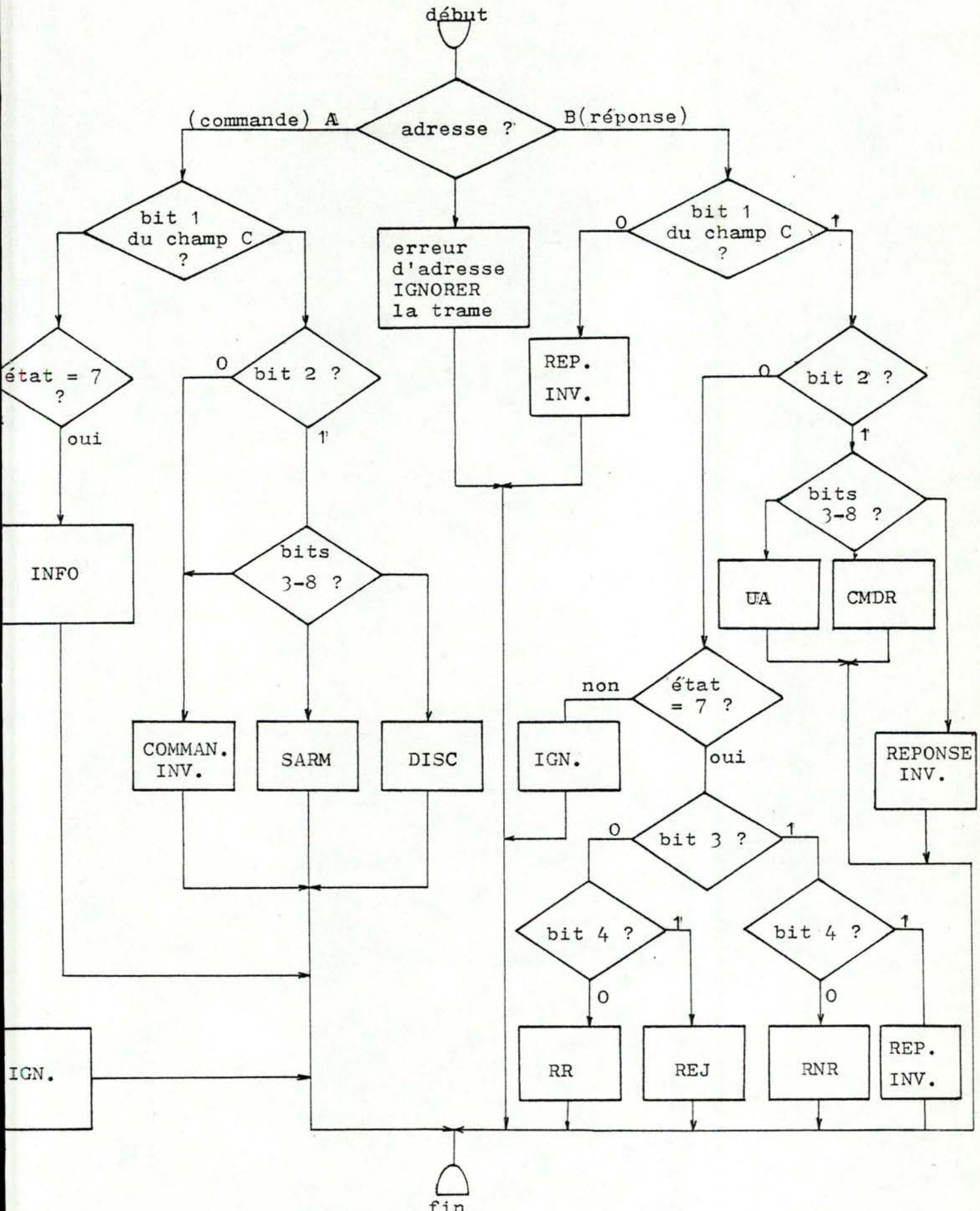


Fig. 4.8 - Analyse du champ ADRESSE et du champ de COMMANDE .

4.4.2 Traitement des trames

L'automate central consulte le tampon d'entrée si une trame y est placée il la traite.
La figure 4.8 montre comment l'analyse du champ d'adresse et du champ de commande est faite afin de déterminer le type de la trame reçue. Cette analyse aboutit sur le choix d'une entrée dans une des 3 tables séquentielles. On peut dès lors décider en fonction de l'état dans lequel se trouve l'automate, quelles actions doit être accomplies et dans quel état l'automate doit passer.
A chaque type de trame correspond une routine qui porte le nom de la trame traitée.

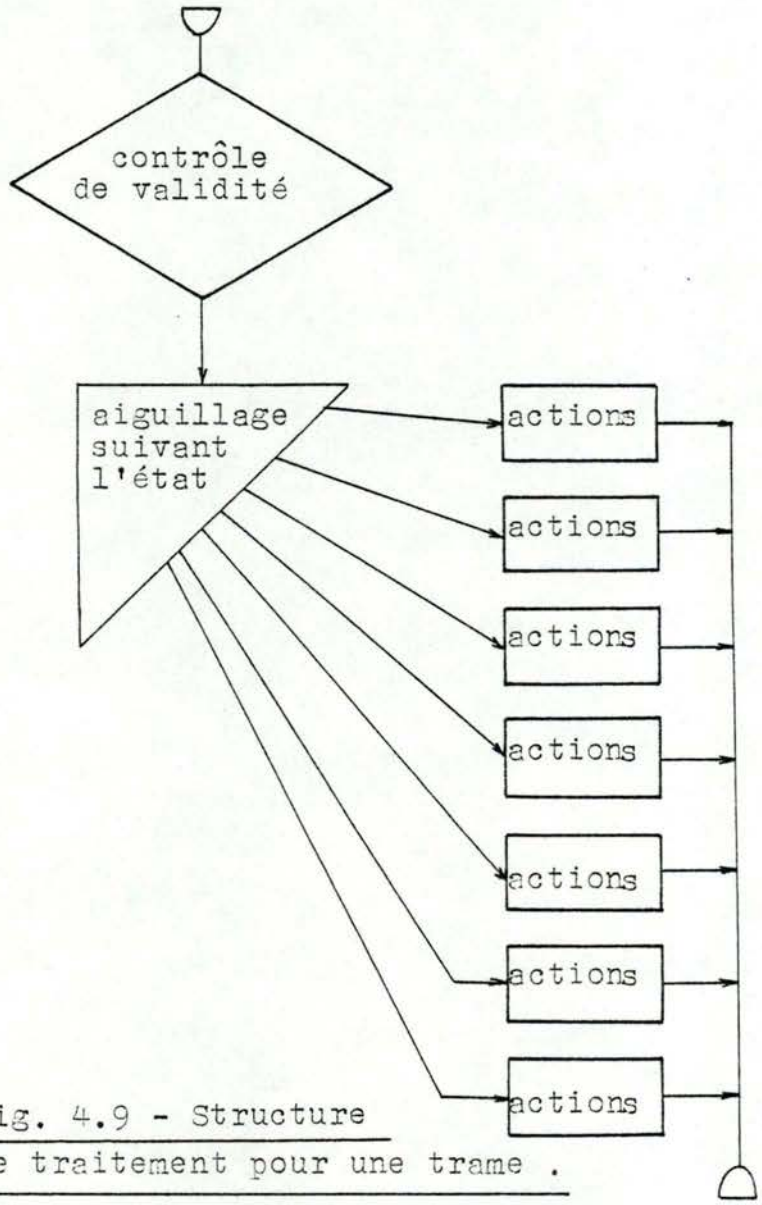


Fig. 4.9 - Structure de traitement pour une trame .

Toutes ces routines ont la même structure (figure 3.9) .
Le traitement commence par un contrôle de validité portant sur la longueur de la trame et sur les valeurs de N(S) et N(R) pour les trames portant ces numéros de séquence.
Les trames non valides sont traitées par des routines appropriées (RP-INV et CO-INV).
Si la trame est correcte, il n'y a plus qu'à déterminer à partir de l'état de l'automate quelle case de la table séquentielle doit être consultée.
Chaque rectangle "Action" de la fig 4.9 correspond à une case. La programmation est très facile car elle peut se faire directement à partir des tables séquentielles.

Exemples : voici quelques exemples de routines qui traitent chacune un type de trame.
Elles sont écrites en langage CMC qui est une version simplifiée du langage C.
(ceci en vue du passage sur le micro-ordinateur).

a) Définitions des variables manipulées par ces programmes

```
char trbu0 [ TRSIZE ] ; /* premier buffer pour la réception des trames */
char trbu1 [ TRSIZE ] ; /* second buffer pour la réception des trames */
char tro_fl ;   /* indique le buffer trame qui doit être examiné */
char tri_fl ;   /* indique le buffer trame qui doit être rempli */
char *tri_pt ;  /* pointeur d'entrée dans les buffers pour trames */
char *tro_pt ;  /* pointeur de sortie des buffers pour trames */
char *trf_pt ;  /* pointeur ou l'on sauve l'adresse de début du buffer
                                                    trame examiné */

char adr_er ;   /* contient le nombre d'erreurs d'adresse détectées
                                                    par l'automate */
char timout ;   /* indicateur de reveil du timer */

char *window [8] ;  /* contient l'adresse de chaque paquet déja émis */
char v_send ;   /* contient le numéro de la prochaine trame à émettre */
char v_rec ;    /* contient le numéro de la prochaine trame à recevoir */
char v_nack ;   /* contient le numéro de la première trame émise mais non
                                                    encore acquitée */
char n_rec ;    /* numéro d'acquittement recu dans la trame traitée */
char n_send ;   /* numéro d'envoi de la trame traitée */
char intNR ;    /* intermédiaire pour l'acquittement en cours de transfert
char lastNR ;   /* dernier acquittement envoyé par l'émetteur */
char rep_ct ;   /* compteur de répétition d'envoi d'une même trame */
char state ;    /* contient l'état général de l'automate */
char stateS ;   /* contient l'état du secondaire */
char stateP ;   /* contient l'état du primaire */
```

```
char mes_rq ;   /* contient les demandes de message à envoyer vers le
                                                    niveau 'paquet' */
char cmdes ;    /* contient les commandes recues du niveau 'paquet' */
char xmi_ab ;   /* flag de demande d'abandon de la trame I en cours à
                                                    l'emetteur */
char xmi_rq ;   /* byte de dépot des demandes d'envoi des trames U et S */
char xmi_fl ;   /* flag d'indication d'activité de l'émetteur */
char xmi_cd ;   /* byte de dépot des commandes concernant l'émission des
                                                    paquets */
char xmi_sv ;   /* contient le type de la trame en cours de transfert */
```

b) Traitement de la trame DISC

contrôle de validité

aiguillage

```
disc () {
        if (*trf_pt == 2 )
                switch (state) {
                        case 0 : send ( UA , (*tro_pt & 020 )) ;
                                 break ;
                        case 1 :
                        case 3 :
                        case 8 : send ( UA , (*tro_pt & 020 )) ;
                                 state = 8 ;
                                 break ;
                        case 2 :
                        case 7 : send ( UA , (*tro_pt & 020 )) ;
                                 send ( DISC , 0 ) ;
                                 rep_ct = 0 ;
                                 startT () ;
                                 state = 05 ;
                                 break ;
                        case 4 :
                        case 5 : send ( UA , (*tro_pt & 020 )) ;
                                 state = 5 ;
                                 break ;
                        case 6 : send ( UA , (*tro_pt & 020 )) ;
                                 stopT () ;
                                 mes_rq =| 02 ;
                                 drssta () ;
                                 state = 0 ;
                                 break ;
                        default : stop (2) ;
                        }
        else co_inv () ;
}
```

(voir colone DISC de la table 1)

c) Traitement de la trame RR dans l'état transfert de données

```
rr_7 () {
        switch (stateP) {
                case 'a' : if (*tro_pt & 020) {
                                xmi_cd = 0 ;
                                rep_ct = 0 ;
                                send ( SARM , 0 ) ;
                                startT () ;
                                stateP = 'd' ;
                                }
                           else
                                if (n_rec != v_nack) {
                                        ack () ;
                                        rep_ct = 0 ;
                                        if (n_rec == v_send) stopT () ;
                                        else startT () ;
                                } ;
                           break ;
                case 'b' : if (*tro_pt & 020) {
                                xmi_cd = 0 ;
                                rep_ct = 0 ;
                                send ( SARM , 0 ) ;
                                startT () ;
                                stateP = 'd' ;
                                }
                           else {
                                ack () ;
                                if ( n_rec == v_send ) stopT () ;
                                else startT () ;
                                xmi_cd = 01 ;
                                stateP = 'a' ;
                                } ;
                           break ;
                case 'c' : if (*tro_pt & 020) {
                                stopT () ;
                                if (n_rec != v_nack) {
                                        ack () ;
                                        rep_ct = 0 ;
                                        } ;
                                if (n_rec != v_send) {
                                        xmi_cd = 02 ;
                                        xmi_ab = 01 ;
                                        }
                                else xmi_cd = 01 ;
                                stateP = 'a' ;
                                }
                           else ignore () ;
                           break ;
                case 'd' : ignore () ;
                           break ;
                default :  stop (3) ;
                } ;
}
```

d) Gestion des demandes d'envoi adressées à l'émetteur

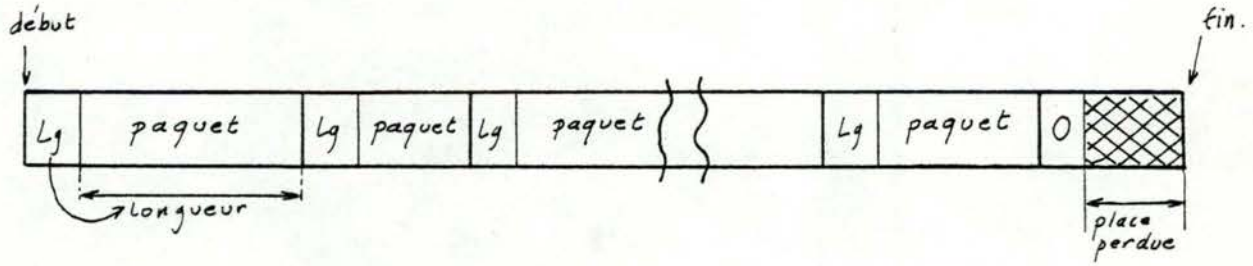
```
/* cette routine gère les demandes d´envoi générées par l´automate.
   si xmi_rq est vide (pas de demande en attente pour l´émetteur),
   la nouvelle demande est placée dans xmi_rq en tenant compte du
   bit P/F .
   si une demande se trouve déja dans xmi_rq il faut attendre que
   l´émetteur la prenne en charge ,ce qu´il indique en remettant
   xmi_rq a zéro .
   dans certains cas ( RR,RNR,REJ ) une nouvelle demande pourra
   se substituer directement a celle déja présente .
   il n´y a donc jamais plus d´une demande en attente à l´emetteur.
*/

send ( code , flag ) char code , flag ; {
        switch ( code ) {
                case RR :
                case RNR :
                case REJ :
                        if (xmi_rq < 05) xmi_rq = (code | flag) ;
                        else {  while (xmi_rq) waitin () ;
                                xmi_rq = (code | flag) ;
                                    } ;
                        break ;
                case SARM :
                case DISC :
                        if (xmi_rq) {
                                xmi_ab = 01 ;
                                while ( xmi_rq ) waitin () ;
                                xmi_rq = (code | flag) ;
                                }
                        else {  xmi_rq = (code | flag) ;
                                xmi_ab = 01 ;
                                } ;
                        break ;
                case UA :
                case CMDR :
                        while (xmi_rq) waitin() ;
                        xmi_rq = (code | flag) ;
                        break ;
                default :
                        stop(9) ;
                } ;
        return ;
}
```

4.5 Gestion des buffers de paquets

Les deux buffers sont gérés de façon identique.
Ce sont en fait des zones continues de mémoire dont la taille est un paramètre de l'implémentation.

I) Mode de rangement des paquets



Les paquets sont rangés l'un à la suite de l'autre.
Chaque paquet est précédé d'un octet contenant sa longueur et est rangé de façon continue dans le buffer.
Si on arrive à la fin du buffer et que le paquet à ranger ne peut être rangé en entier on l'indique en mettant à 0 le premier octet libre (s'il en reste au moins I).
On recommence alors à ranger au début du buffer.

2) Variables de contrôle (Exemples pour buffer d'entrée bu-in (=BUFFIN))

Les variables suivantes sont utilisées pour contrôler les opérations sur le buffer :

char bu_in (BISIZE) ; buffer d'entrée pour les paquets reçus sur la ligne

char bi_ipt ; pointeur sur la première place libre pour l'entrée des caractères dans bu_in

char bi_opt ; pointeur sur le prochain caractère à sortir de bu_in

int bi_fct ; compteur du nombre de places libres dans bu_in

int bi_pct ; contient le nombre de paquets en attente dans bu_in

3) Opérations sur le buffer

Les figures 4.I0 et 4.II donnent les algorithmes pour le rangement et l'extraction d'un paquet.

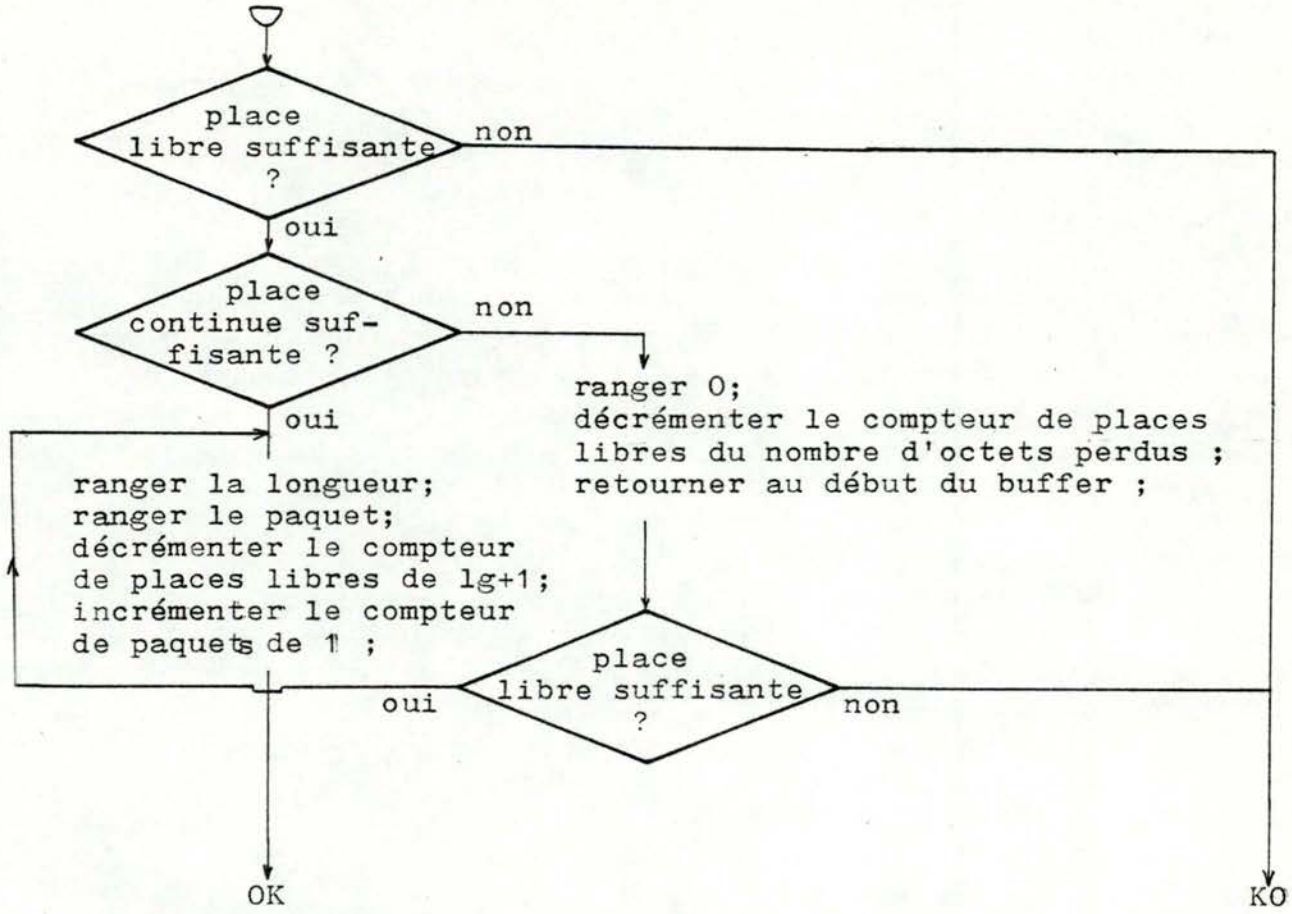


Fig. 4.10 - Rangement d'un paquet dans le buffer .

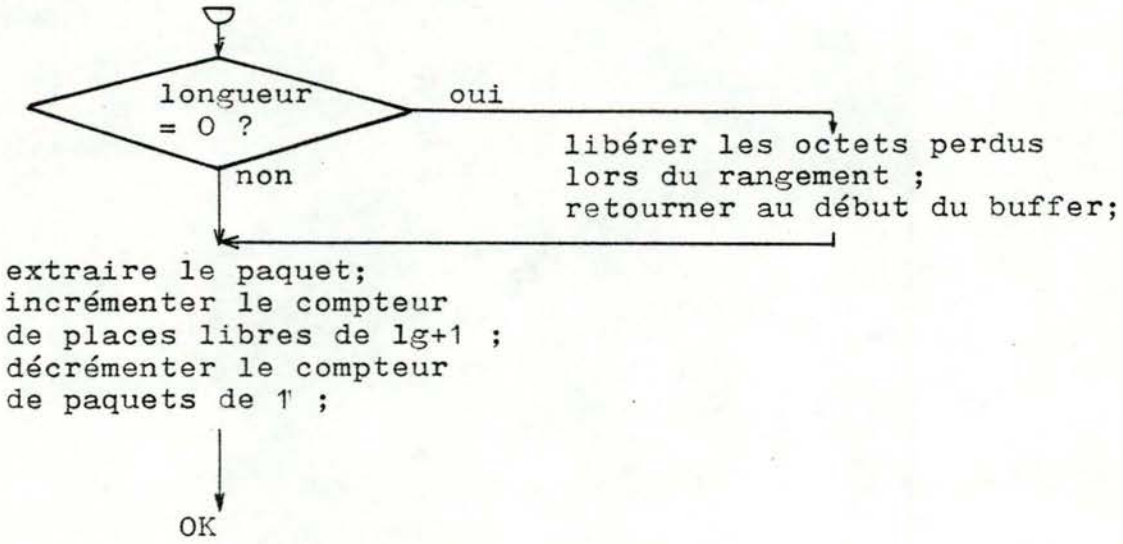


Fig. 4.11 - Extraction d'un paquet hors du buffer .

4.6 Procédure de test de l'implémentation de l'automate du niveau trame 2

La mise au point de l'automate a été faite sur l'ordinateur pdp II/IO qui possédait l'avantage d'être déjà connecté sur le pdp II/45 à l'aide d'un interface DR11-C.

4.6.I Configuration de test : SIMTRA

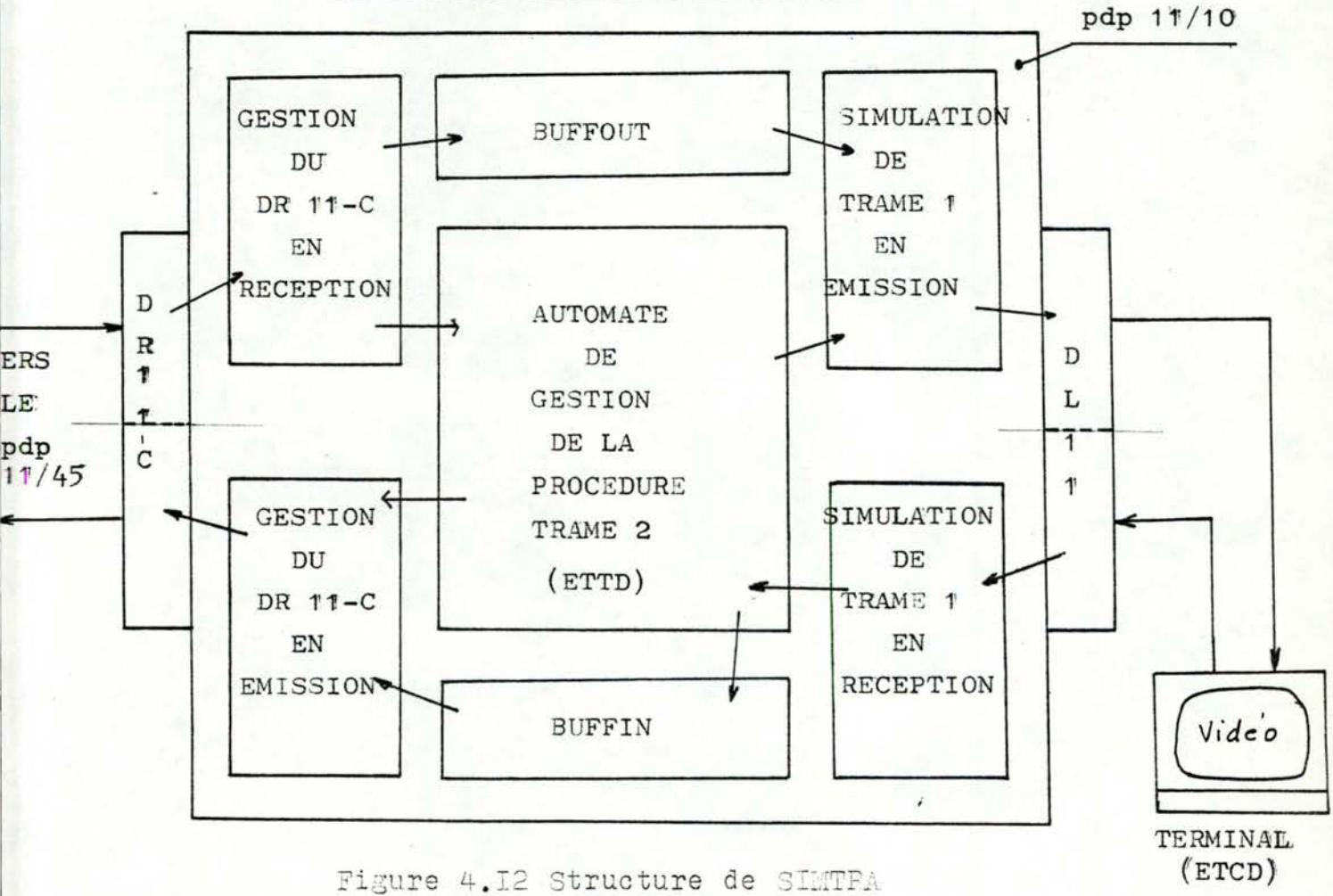


Figure 4.I2 Structure de SIMTRA

La figure 4.I2 montre la configuration utilisée pour tester l'automate. Le rôle de l'ETCD est rempli par un opérateur humain qui converse avec l'ETTD au moyen d'un terminal vidéo.
Ceci est rendu possible grâce aux deux modules qui gère l'interface asynchrone DLII auquel est connecté le terminal. Les trames à envoyer vers l'ETCD sont visualisées de façon

mémonique pour pouvoir être facilement interprétées par l'opérateur.
De la même façon un mécanisme simple permet à l'opérateur d'envoyer les trames à l'ETTD au moyen du clavier du terminal.
Il est également possible de simuler des erreurs sur le champ adresse, sur le champ commande et sur la longueur de la trame envoyée vers l'automate.
Afin de pouvoir suivre l'évolution de l'automate, son état est également visualisé sur l'écran du terminal.

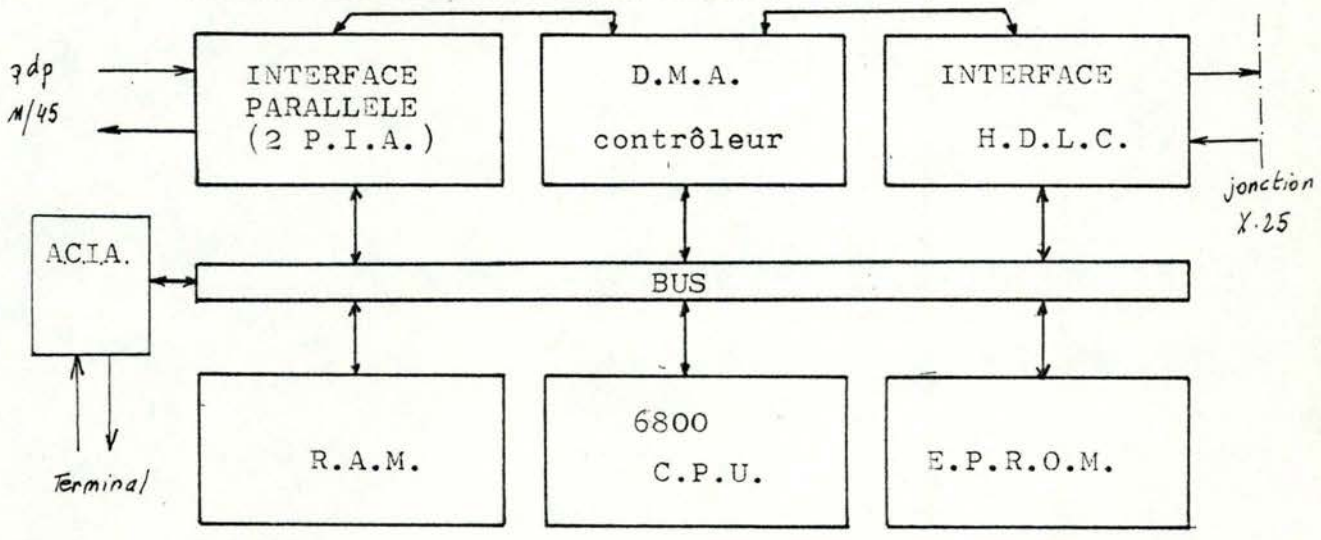
Les fonctions d'interface avec le niveau paquet sont quand à elles remplies par les modules qui gèrent l'interface DR-11C.
De cette façon on peut envoyer des paquets et des messages grâce à un programme tournant sur la pdp II-45.

4.6.2 Procédure de test

La démarche suivie est la suivante :
Pour chaque état que peut prendre l'automate on génèrent tous les événements auxquels il peut être soumis.
Pour chacun de ces événements on vérifie si la transition et les actions de l'automate sont conformes aux tables séquentielles décrites en 4.3 .

4.7 Conversion des programmes pour le micro-ordinateur

4.7.I Structure du micro-ordinateur (Fig. 4.13)



La figure 4.I3 nous montre l'architecture du micro-ordinateur développé par le groupe ELHY.
L'élément central est un micro-processeur Motorola 6800 auquel on a adjoint :
-un interface parallèle pour communiquer avec le pdp (compatible avec le DRII-C) utilisant 2 circuits PIAS.
-un interface HDLC équipé d'un circuit intégré spécialisé réalisant toutes les fonctions du sous- niveau enveloppe de trame (trame 1)
-une mémoire vive (RAM) pour les buffers et les variables.
-une mémoire morte pour ranger les programmes.
-un interface série (ACIA) pour connecter un terminal.
-un contrôleur d'accès direct mémoire (DMAC).
Capable de contrôler l'interface parallèle et l'interface HDLC et dont le rôle est d'accélérer les opérations d'entrées-sorties.

4.7.2 Modifications à apporter aux programmes de SIMTRA

Aucune modification n'a du être apportée à la structure des routines qui ont été testées sur le pdp II/IO.
Seuls quelques points de détails tels que les adresses physiques et le contrôle du temporisateur ont nécessité quelques modifications.
La différence principale entre le pdp II/IO et le micro-ordinateur réside dans la gestion des interruptions. Le micro-ordinateur nécessitant une recherche par programme de la source d'interruption et le branchement à la routine de traitements adéquate.
Ceci nécessite de définir des niveaux de priorités pour la gestion des interruptions en fonction de la nature des événements qui les produisent.
Tous ces problèmes ont été résolus et la version micro-ordinateur de SIMTRA a été testée de la même façon que celle qui avait été mise au point sur le pdp II/IO y compris l'interconnexion avec le pdp II/45.

5. Conclusions

5.I Etat d'avancement du travail

On peut résumer comme suit la liste des travaux effectués :

I) Implémentation de l'automate trame 2 sur le pdp II/IO avec une simulation des entrées- sorties vers la ligne.

Programmes écrits en langage C :

-Ensemble des routines de gestion de la procédure (traitement des messages, des trames et du time-out).

-Routines de gestion du DP11-C, pour l'échange de paquets et de messages sur l'interface trame-paquet.

-Routines de gestion du DLII. Gérant la réception des caractères envoyés par le clavier du terminal (simulation des trames reçues de l'ETCD) et l'émission de caractères vers le terminal pour simuler l'envoi des trames vers l'ETCD.

-Routine d'initialisation.

-En assembleur pdpII : -définitions de vecteurs d'interruption
-routines spécifiques au pdp II/IO.

Ces modules ont été testés puis mis ensembles pour former SIMTRA.

Les tests effectués sur SIMTRA ont été les suivants :

-test complet des fonctions de l'automate trame 2
-test complet des fonctions de l'interface avec le niveau paquet y compris l'intégration avec le driver trapac sur la pdp II/45.
(voir mémoire de P. Lambion)

2) Conversion des programmes de SIMTRA pour le micro-ordinateur.

Les routines de gestion du DRII-C ont été adaptées pour la gestion des PIA'S pour la communication avec le pdp II/45.

Les routines de gestion du DL-II ont été adaptées pour gérer l'ACIA sur lequel est connecté le terminal.

Les routines propres au micro-ordinateur ont été écrites en assembleur motorola 6800. (polling des sources d'interruption, contrôle du temporisateur, etc.).
Les tests comparables à ceux effectués sur le pdp II/IO ont été fait sur le micro-ordinateur.

Version complète du niveau trame

Les routines de gestion de l'interface HDLC et du contrôleur DMA ont été écrites et testées par Y.Beudin et G.Zone.
Leur intégration avec les autres modules du niveau trame est en voie d'achèvement.

Remarque : le champ diagnostic des trames CMDR n'est pas utilisé dans la version implémentée aussi, ce champ est-il toujours à 0.

5.2 Les difficultés de réalisation

La conversion des programmes pour le micro-ordinateur a posé un certain nombre de problèmes. Le cross-compilateur CMC dont nous disposions n'était pas au point, les traductions en assembleur 6800 ont du être corrigées manuellement.
De plus, la mise au point du micro-ordinateur à retarder les tests du logiciel devant tourner sur celui-ci.
Les routines d'émission et de réception des trames ont du être réécrites entièrement en assembleur 6800 pour éviter les problèmes posés par la compilation du langage CMC.

5.3 Evolution possible pour l'avenir

Pour donner suite à notre travail on peut envisager les points suivants :

-améliorer le micro-ordinateur en passant de la version expérimentale à une version plus rationnelle et plus fiable.

-étudier le niveau trame sur le plan des performances en fonction des valeurs données aux paramètres d'implémentation. (taille des buffers, durée du temporisateur, nombre maximum de trames émises en anticipation, etc.).

On peut envisager d'autres applications du niveau trame notamment pour une liaison de point à point entre 2 ordinateurs.

5.4 BIBLIOGRAPHIE

I. Transmission de données sur le réseau téléphonique (avis de la série V.)
Livre orange
Tome 8.I CCITT oct. 76

2. Réseaux publics pour données (avis de la série X.)
Livre orange
Tome 8.2 CCITT oct. 76

3. Version révisée de la section 2 de l'avis X.25.
Commission 7/II juillet 77

4. Manuel utilisateur TRANSPAC.

5. Norme Internationale ISO / DIS 3309.2 (HDLC)
procédure de commande de chainon à haut niveau
Structure de trame.

6. Norme Internationale ISO / DIS 4335
Eléments de procédures

7. Réalisation d'une liaison X.25 à l'aide d'un microprocesseur
Mémoire de : Y. Beudin - G. Zone

8. Conception d'une liaison X.25 : 2e partie
Niveau paquet
Interface avec les utilisateurs
Mémoire de : P. Lambion

9. Computer network protocols
Symposium

I0. Documents for use with the unix time-sharing system
6e édition

II. C Reference Manual
Dennis M. Ritchie
Bell Telephone Laboratories

I2. Microprocesseur
Motorola 6800
Manuel de programmation

I3. Microcomputer
Motorola 6800
System design data

I4. pdp II Processor handbook

I5. pdp II Peripherals handbook

I6. DRII-C General device interface manual

Facultés Notre-Dame de la Paix. - Namur
Institut d'Informatique

Conception d'une liaison X.25

Partie II

Etude et implémentation sur PDP.11/45 du niveau X.25 paquet et des fonctions primitives

Tome I

Patrick LAMBION

Rapport du groupe XYZ

Mémoire présenté en vue de
l'obtention du grade de
Licencié et Maître
en Informatique

- Août 1979 -

Avant-propos.

Ce document est à la fois un mémoire de fin d'études et le résultat d'une partie du travail effectué dans le cadre du groupe XYZ.

Celui-ci rassemble des professeurs, assistants et étudiants des unités informatique et électronique de l'Université Catholique de Louvain et de l'unité architecture de système des Facultés Universitaires Notre-Dame de la Paix à Namur.

Ce groupe a été constitué pour procéder à l'étude des problèmes posés par un raccordement à un réseau d'ordinateurs ainsi qu'à l'utilisation des microprocesseurs dans de telles circonstances.

Deux autres documents ont été élaborés à ce propos: l'un par MM Beudin et Zone; l'autre par Mr J. Art. Ce dernier comprend, en outre, une présentation générale du travail.

Pour une compréhension plus aisée du présent travail et plus particulièrement des chapîtres 3 et 4, il est recommandé d'avoir une bonne connaissance du système d'exploitation UNIX - sa conception et son utilisation -.

La bibliographie se trouvant en fin de ce tome reprend les documents de base utiles à la compréhension du système d'exploitation et à l'élaboration de ce mémoire.

Dans les annexes, on trouvera les sources commentées des nouvelles routines implémentées dans le système ou ailleurs; ainsi que les modifications y apportées.

Que Monsieur le Professeur Brunin voit ici l'expression de ma gratitude pour le choix et l'orientation du travail.

Je tiens à présenter mes remerciements à Monsieur le Professeur Van Bastelaer qui a dirigé le mémoire et organisé la collaboration avec l'UCL dans les meilleures conditions possibles.

Remerciements également à Monsieur le Professeur Milgrom qui non seulement m'a permis d'utiliser son ordinateur et d'en modifier le système d'exploitation, mais qui, grâce à sa disponibilité et ses conseils, a rendu possible la réalisation de ce travail.

Enfin, les autres membres du groupe XYZ, le personnel scientifique et le personnel technique de l'unité informatique de l'UCL trouveront ici le témoignage de ma reconnaissance pour les conditions de travail qu'ils ont créées et leurs conseils judicieux dont j'ai pu profiter.

P. Lambion

TABLE DES MATIERES

TOME I.

TABLE DES MATIERES

TOME I.

TABLE DES MATIERES

TOME II.

INDEX DES SCHEMAS

Chapitre 0: Introduction.

0.1. Description générale.

Le travail consiste en l'implémentation de ce qui est nécessaire pour que des applications appartenant à des utilisateurs puissent effectuer des entrées sorties sur un réseau d'ordinateurs fonctionnant avec le protocole X25. Cela inclut donc un gérant de ce protocole ainsi que les fonctions primitives mises à la disposition des utilisateurs.

0.2. Objectif initial.

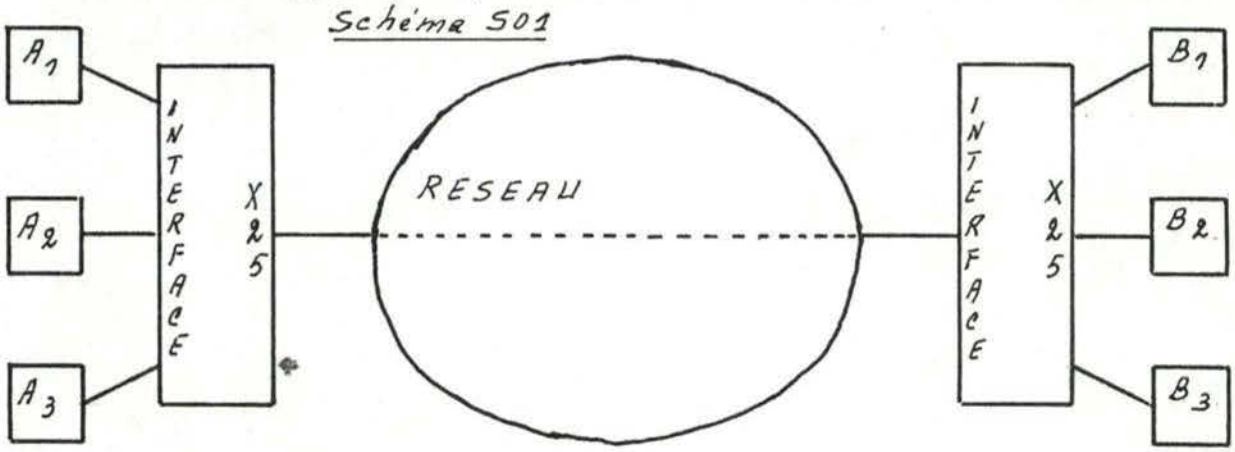
Le but est d'adapter à un ordinateur PDP11/45, avec système d'exploitation UNIX, les éléments nécessaires pour le rendre apte et compatible à une connexion avec un réseau d'ordinateurs fonctionnant suivant le protocole X25 défini par le CCITT. Pour ne pas surcharger le PDP, une partie du travail s'effectuera sur un système à microprocesseur.

Se servant de la bipolarité du protocole X25 logiciel, la gestion de l'un des niveaux (niveau 2) est implémentée sur le microprocesseur, l'autre (niveau 3) sur le PDP. Le niveau 1 est un protocole physique. Sur le PDP sont ajoutées les fonctions (appelé par la suite niveau 4) pour le raccord entre le protocole X25 et les programmes utilisateurs. Voir schéma S01.

La première partie du mémoire qui a été écrite par J. ART et traite du niveau 2 contient également une description générale et plus fouillée du travail. Il est utile d'en prendre connaissance pour se faire une idée de l'ensemble implémenté.

Les transferts peuvent être de type quelconque et couvrir par exemple l'utilisation interactive d'un système à partir de l'autre, l'échange de données entre applications, le transfert de fichiers. Voir schéma S01.

Schéma S01



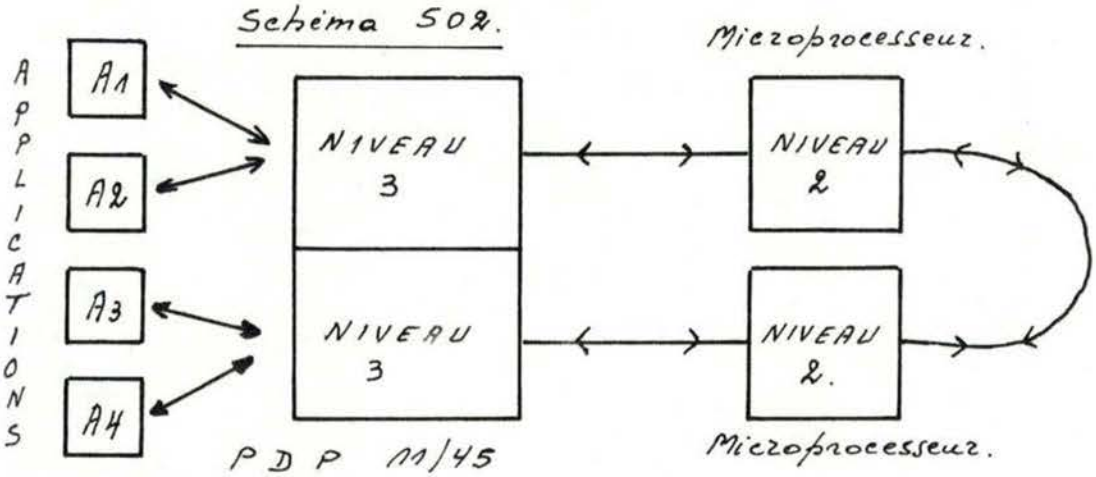
A1 · A2 · A3 ≡ applications sur ordin. A B1 · B2 · B3 ≡ applications sur ordin. B.

0.3. Contraintes et objectif réel.

Comme il n'est pas possible de disposer d'un réseau ni même de deux ordinateurs identiques, le réseau est court-circuité et le PDP 11/45 remplit le rôle de deux ordinateurs. Cela est obtenu de la façon suivante: ce qui sortira par un interface vers un microprocesseur, rentrera par un autre interface en venant directement d'un deuxième microprocesseur. Ces deux

microprocesseurs sont directement reliés. Ce qui donne le schéma S02.

Schéma S02.



Cela nécessite évidemment quelques modifications au niveau 2 et au niveau 3 mais rien qui soit fondamental. Certes, la solution ne permet pas de vérifier la compatibilité avec un réseau mais au moins permet de se rendre compte si l'implémentation est cohérente: ce qui sort doit pouvoir entrer et réciproquement.

Remarque:

Par la suite, lorsqu'il sera question du moniteur réseau, il faudra entendre l'ensemble composé

- des fonctions de gestion du protocole X25 paquet.
- des fonctions de liaison avec les applications des utilisateurs.
- des fonctions de contrôle de ces applications.

Chapitre 1: Le protocole X25, niveau paquet.

1.1. Notions et définitions principales.

1.1.1. Eléments principaux.

Le protocole logiciel X25 a été défini par le CCITT pour régir le raccordement d'une machine à un réseau travaillant par commutation des paquets. On dit qu'il y a commutation des paquets car l'unité de transfert est le paquet, séquence de bits de longueur variable mais limitée à un maximum défini.

Chaque paquet possède un champ de commande de contenu bien défini exprimant le travail demandé au réseau et d'un champ de données à remplir au gré de l'utilisateur du réseau (gérant du protocole sur la machine et par extension l'utilisateur au sens large).

La machine à relier au réseau est couramment appelée ETTD (Elément Terminal de Traitement de Données) et le point de raccordement (noeud du réseau) est l'ETCD (Elément Terminal de Commutation de Données).

Le Protocole permet la communication entre plusieurs utilisateurs d'un ETTD et des utilisateurs d'un nombre indéterminé d'ETTDs, et ce, par une seule liaison physique au réseau. Voir schéma S11. Cela est dû au fait que le protocole dispose de deux niveaux :

- le niveau trame (niveau 2) qui s'occupe de la gestion de la connexion physique entre l'ETTD et l'ETCD.
- le niveau paquet (niveau 3) qui s'occupe de la gestion des voies logiques.

Une voie logique est la liaison logique, non nécessairement attachée à une liaison physique, entre une application d'un ETTD et une autre application d'un autre ETTD. La ligne physique ETTD - ETCD est multiplexée entre plusieurs voies logiques. Voir schéma S11. Chaque voie logique est définie par un numéro valable pour une liaison ETTD - ETCD; le réseau fera la correspondance entre les numéros d'une même voie logique au départ et à l'arrivée.

Le niveau trame s'occupe de la gestion de la voie physique en faisant passer des paquets des voies logiques entre l'ETTD et l'ETCD. Le niveau paquet gère la voie logique.
L'action du niveau trame est strictement limitée à la liaison ETTD - ETCD tandis que l'action du niveau paquet a des répercussions d'un ETTD à l'autre.

1.1.2. Etablissement et rupture d'une voie logique.

La voie logique a une durée de vie limitée à la nécessité d'une liaison entre deux ETTDs(cela peut représenter le temps d'exécution d'une application , le temps de travail d'une personne à un terminal) alors qu'une voie physique (ETTD - ETCD) peut avoir une durée beaucoup plus grande (session d'ordinateur ou plus). Une voie logique doit

donc être établie et rompue à la demande. Il faut donc qu'il existe des commandes de l'ETTD vers le réseau et du réseau vers l'ETTD pour effèctué ces deux actions; ce sont :

- paquet d'appel.
- paquet de confirmation d'appel.
- paquet de libération.
- paquet de confirmation de libération.

Deux paramètres sont nécessaires dans un paquet d'appel issu d'un ETTD:

- un identificateur de la voie logique choisi par l'ETTDa car chaque liaison doit être caractérisée dès le début.
- une adresse de l'interlocuteur ETTDb (destinataire de l'appel).

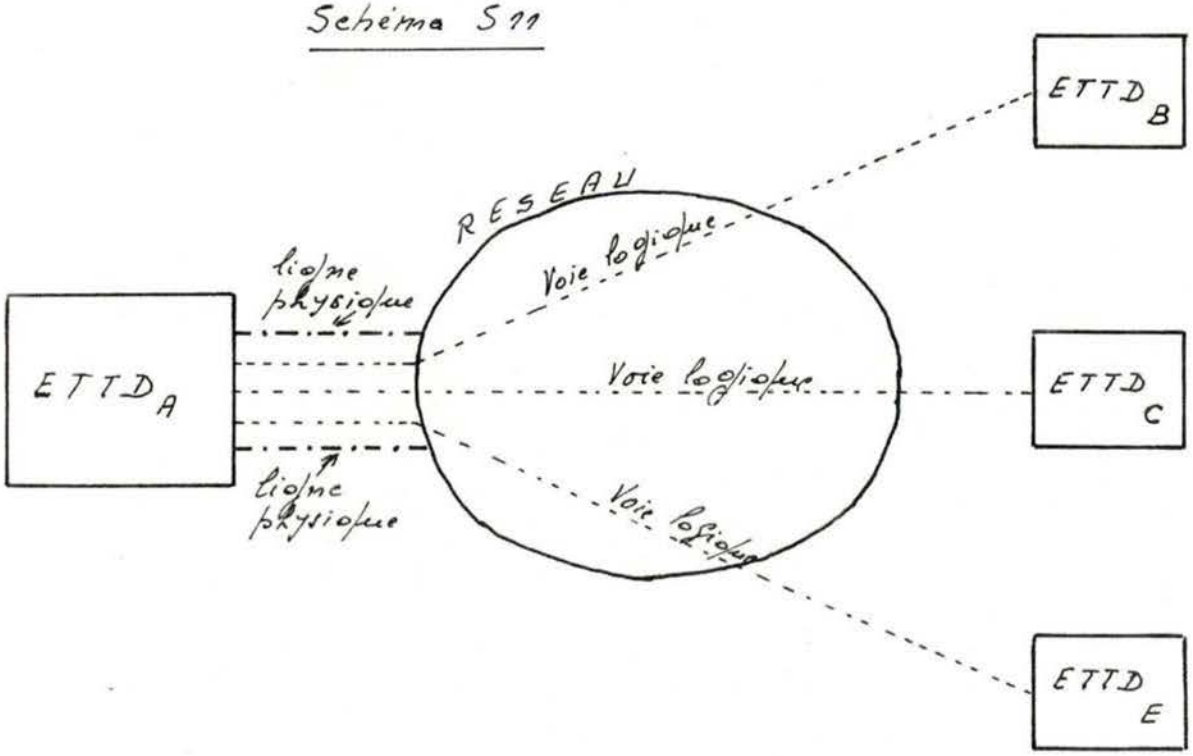
Le paquet d'appel est acheminé par le réseau jusqu'à cet interlocuteur ETTDb. Les paramètres du paquet d'appel sont alors:

- un identificateur de la voie logique choisi par l'ETCD, éventuellement différent de celui selectionné par ETTDa (le réseau fera la correspondance).
- une adresse de l'interlocuteur ETTDa (expéditeur de l'appel).

Ayant pris connaissance de cet appel, l'ETTDb pourra approuver la communication (paquet de confirmation d'appel) ou la refuser (paquet de demande de libération). Ces paquets ont pour argument l'identificateur de voie logique défini entre le réseau et ETTDb

L'ETTDa recevra l'un de ces paquets avec l'identificateur de voie logique défini entre l'ETTDa et le réseau. Voir schémas S12 et S13.

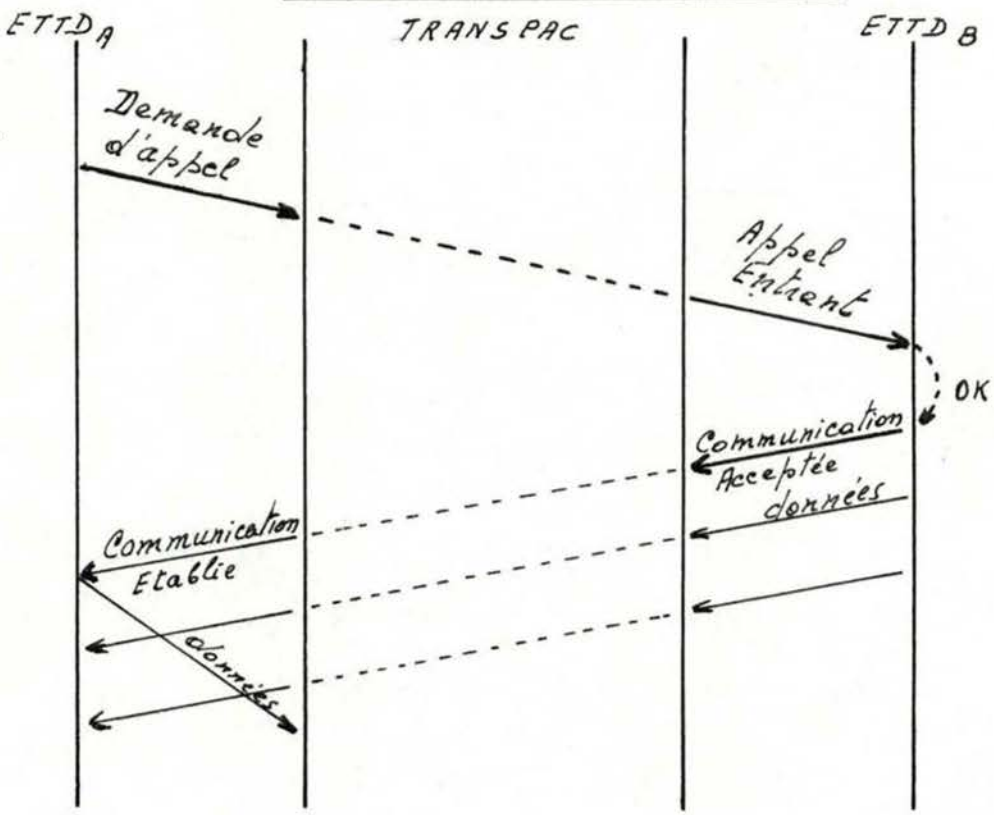
Schéma S11



Schémas:

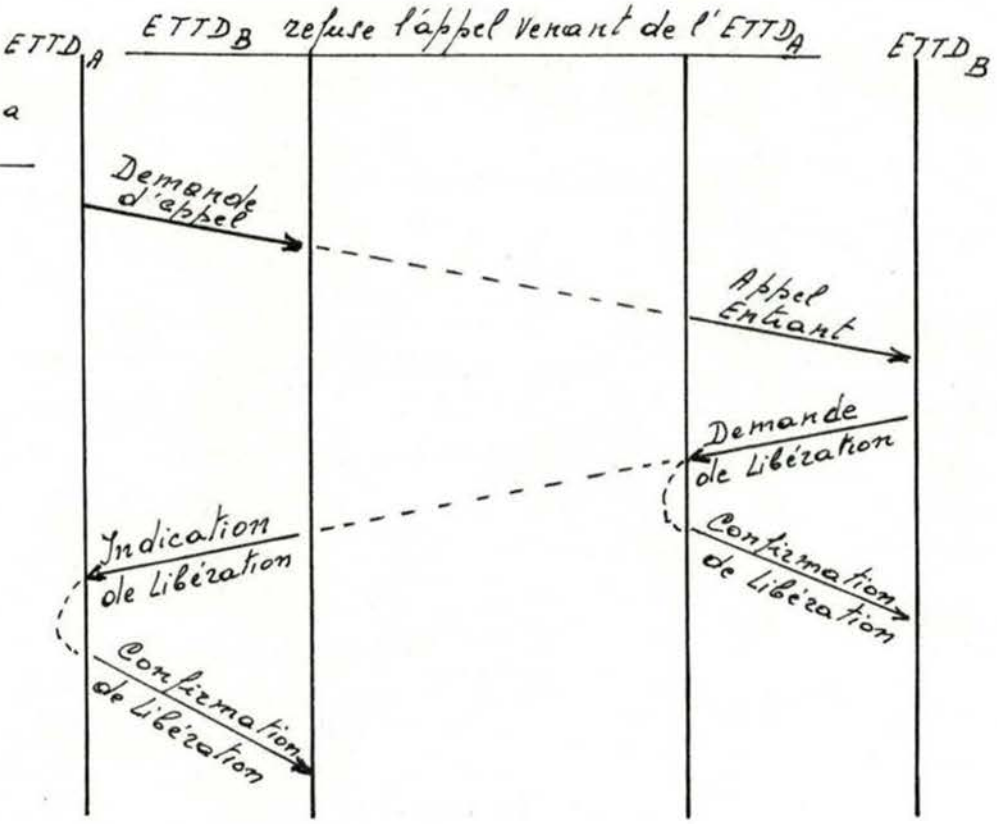
$ETTD_A$ appelle $ETTD_B$

Schéma S12



$ETTD_B$ refuse l'appel venant de l'$ETTD_A$

Schéma S13



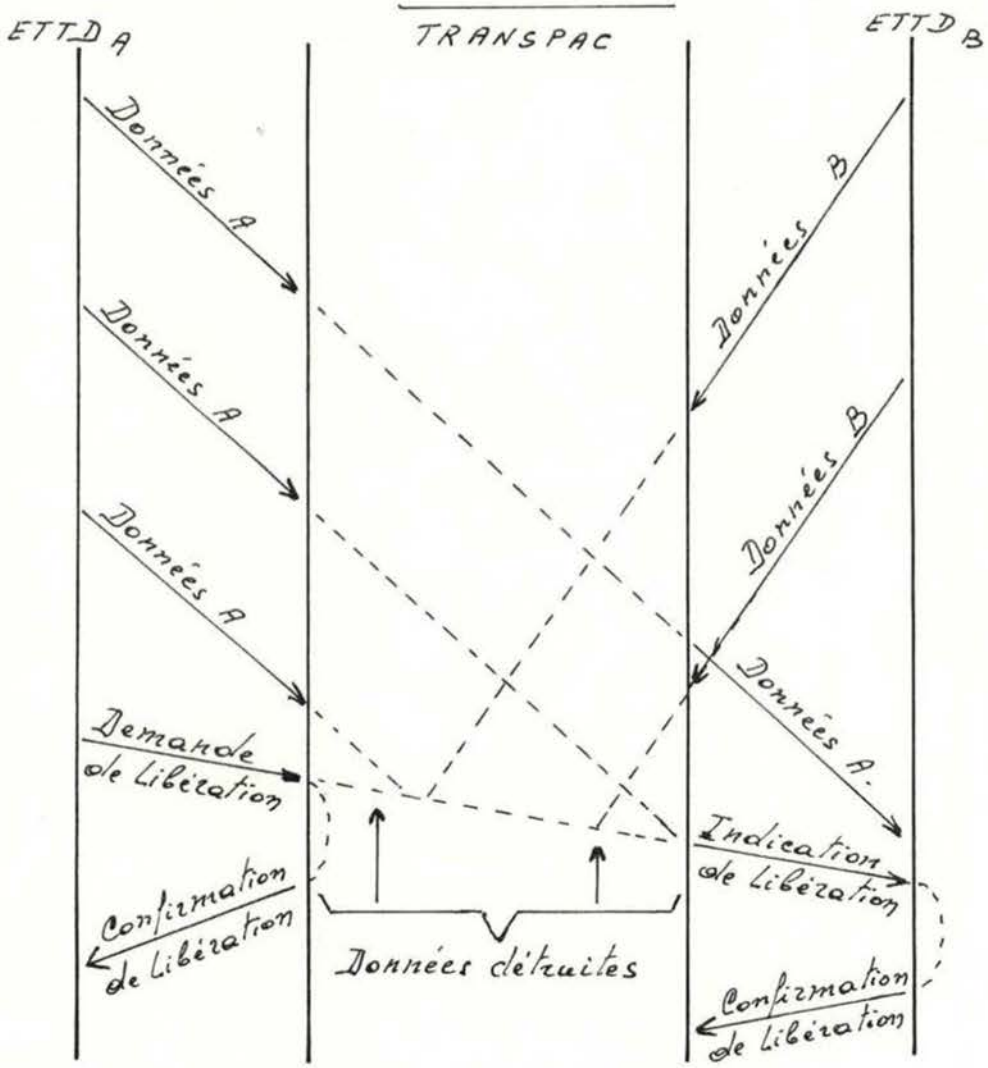
Le paquet de liberation est envoyé :

- par un ETTD s'il refuse la liaison lors de l'appel ou parce qu'il veut arrêter cette liaison.
- par le réseau pour une raison technique interne.

La réponse à une demande de libération est un paquet de confirmation de libération et dès cet instant, la voie logique est considérée libre. Voir schéma S14

Remarques: Alors que la réponse à un paquet d'appel vient de l'ETTD interlocuteur, la réponse a une demande de libération émise par l'ETTD vient de l'ETCD auquel il est connecté; la procédure d'appel est directe d'un ETTDa vers un ETTDb, la procédure de libération se fait en cascade ETTDa vers ETCD puis à l'intérieur du réseau et enfin de l'ETCD vers ETTDb. Dans le cas de saturation de l'ETTD appelé, le réseau prend la décision de libérer la voie logique de l'appelant. Voir schémas S15 et S16.

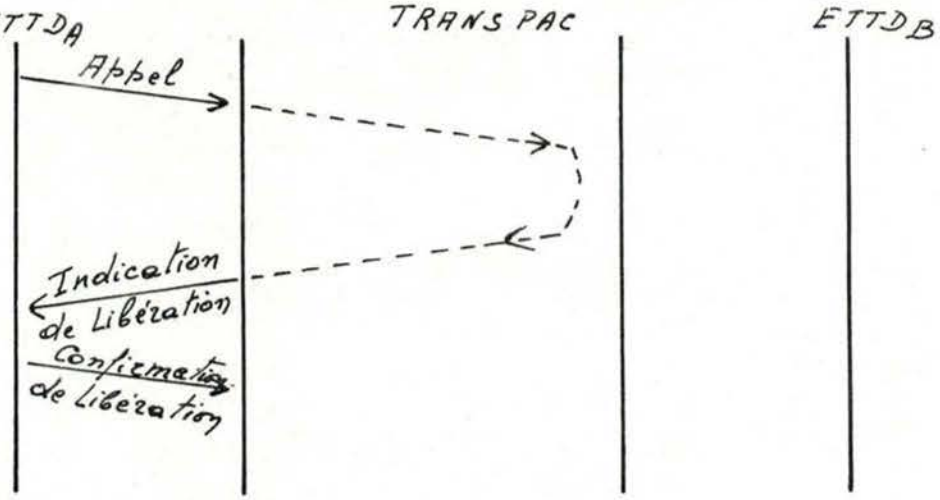
Schéma S14



Schémas

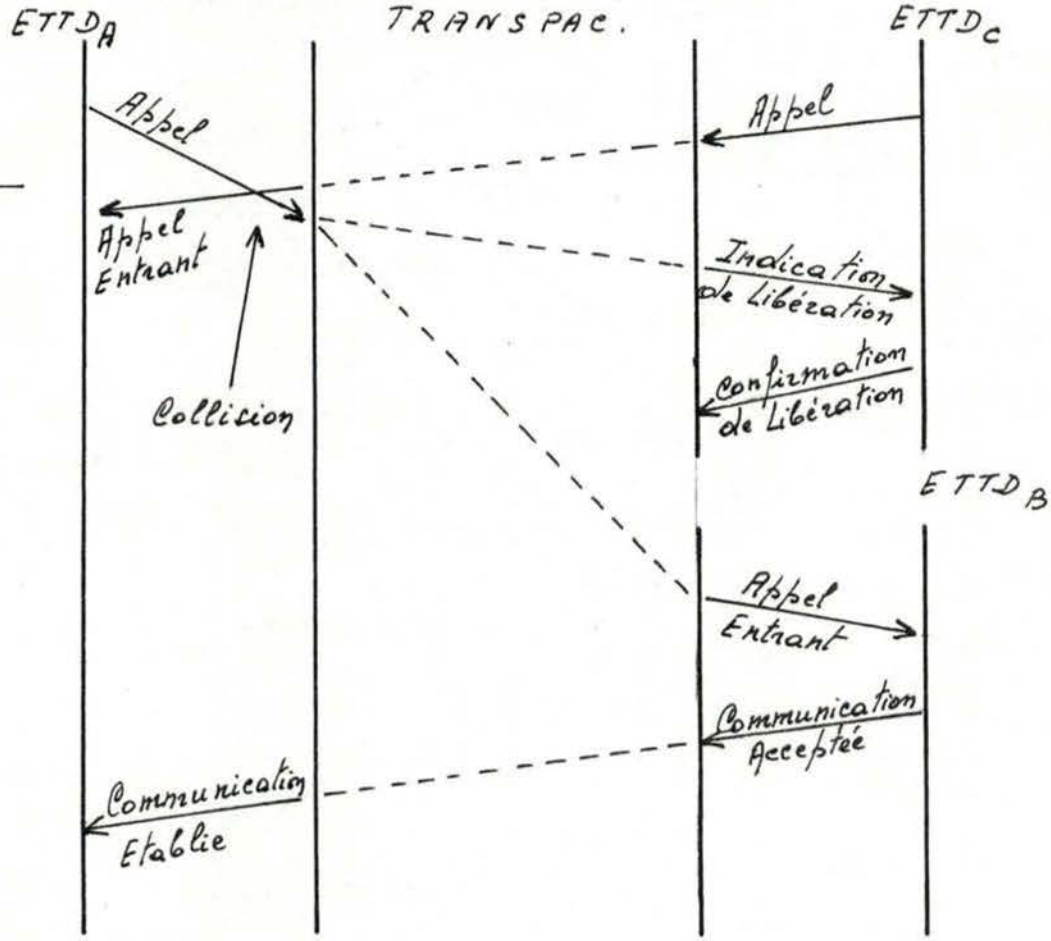
Transpac ne peut établir le circuit virtuel entre ETTD$_A$ et ETTD$_B$

Schéma S15



ETTD$_A$ appelle ETTD$_B$ mais ETTD$_C$ appelle en même temps ETTD$_A$

Schéma S16



1.1.3. Transfert de données.

La voie logique étant établie, commence le transfert des paquets de données. Ceux-ci ont pour arguments:

- le numéro de voie logique.
- un système de contrôle de flux.

Chaque paquet est numéroté modulo 8 au départ de 0. Le principe de base du contrôle de flux est la notion de crédit ou notion de fenêtre. Le destinataire et l'expéditeur sont d'accord sur un crédit(nombre) de paquets qui peuvent être envoyés sans avoir recu un acquittement de ceux précédemment envoyés. Ce nombre s'appelle aussi fenêtre.

Exemple:
Soit un crédit de 3
On peut envoyer les paquets numérotés:
0, 1, 2 sans avoir reçu un acquittement.
Supposons que l'acquittement du paquet 0 est reçu, alors on peut envoyer le numéro 3; et ainsi de suite.

Pour acquitter un paquet, on envoie le numéro du paquet que l'on s'attend à recevoir ou que l'on va prendre en considération. (dans l'exemple pour acquitter 0 on envoie 1; pour acquitter 1 on envoie 2;...).

Ce "numéro" d'acquittement peut être envoyé par un paquet particulier (paquet RR) ou dans un champ spécial du paquet de données (dans le cas d'un échange dans les deux sens).

Il faut bien préciser que ce système de contrôle de flux est valable uniquement entre l'ETTD et l'ETCD, dédicacé à chaque voie logique. Il ne s'agit pas d'un contrôle de bout en bout de voie logique.

Comme le protocole X25 n'oblige pas l'acquittement d'un paquet dès son arrivée à destination, le récepteur peut contrôler le trafic sur la voie logique. Dans le cas où le récepteur est l'ETTD, tout ralentissement ou arrêt peut provoquer à l'autre bout de la voie logique un ralentissement ou un arrêt car la capacité de mémoire du réseau est limitée.

Tout n'est pas parfait, dès lors une faute dans le contrôle de flux (mauvais séquencement, dépassement du crédit, acquittement non valable) provoque l'activation d'une procédure de réinitialisation dont les conséquences dans le réseau sont: Voir schéma S18

- remise à zéro de la numérotation des paquets des deux côtés de la voie logique.
- élimination des paquets de données de la voie logique se trouvant dans l'ensemble du réseau.

Les deux ETTDs sont avertis par un paquet (demande de réinitialisation) et tant qu'ils n'auront pas marqué leur accord (confirmation de réinitialisation), aucun paquet de données ne sera pris en considération. Une fois que la confirmation aura eu lieu, les transferts reprendront sur cette voie logique.

L'ETTD, tout comme le réseau, peut prendre l'initiative d'une réinitialisation (mauvais contrôle de flux ou cause interne). Il ne pourra envoyer de paquet de données tant que l'ETCD n'aura pas confirmé. Tout comme la libération, la réinitialisation se fait en cascade jusqu'à l'autre ETTD.

1.1.4. Initialisation du niveau paquet.

Tout ce qui vient d'être décrit est relatif à une voie logique et touche chacune séparément; il est cependant nécessaire d'initialiser l'ensemble des voies logiques c'est à dire le niveau paquet. (par exemple au début de la session d'ordinateur ou lors de l'activation du niveau physique). Pour ce faire, il existe le paquet de reprise dont l'effet est identique au paquet de libération sauf que son action s'étend simultanément à toutes les voies logiques d'un même ETTD.

La reprise aura lieu lors de la connexion d'un ETTD au réseau (après le SARM du niveau trame), lors de la déconnexion (avant le DISC du niveau trame), ou encore lors de problèmes touchant la ligne elle-même. Normalement la reprise viendra de l'ETCD mais rien n'empêche l'ETTD d'en prendre l'initiative.

Après une demande de reprise, les voies logiques retrouvent l'état normal lorsqu'une confirmation de reprise y a répondu. Lors d'une reprise, et pour les voies logiques qui seraient actives, le réseau générera une demande de libération pour les interlocuteurs de l'ETTD victime d'une demande de reprise. Voir schéma S17.

1.2. Quelques détails et options du protocole X25 paquet.

IL est utile de préciser quelques points plus pratiques et quelques options proposées par le protocole X25 et/ou par TRANSPAC. Les citer toutes est impossible mais quelques unes ont un interêt pour la suite (implémentation ou protocole d'un niveau encore supérieur).

1.2.1. Système d'adresse; diagnostics et causes.

Le système d'adresse est basé sur la localisation physique et non sur une localisation logique. C'est à dire que si une application "facture" peut être implémentée sur plusieurs ETTDs, un utilisateur désirant se servir de cette application devra adresser l'ETTD où elle est située. Le réseau ne possède pas de tables de correspondances: nom d'une application (logique) <-> lieu actuel d'implémentation (physique). L'adresse elle-même est habituellement divisée en sous-adresses:

- un indicatif de la région géographique.
- un indicatif de l'ETCD. En effet plusieurs ETCDs peuvent être reliés à un même ETTD; sans compter les possibilités de connexions multiples.

- un indicatif de l'ETTD.
- un indicatif d'une application de l'ETTD ou du protocole "bout en bout" valable pour cet ETTD.

Les adresses (appelant et appelé) sont codées en séquence de 1/2 octet, l'appelant immédiatement suivi de l'appelé. (packed décimal de IBM).

Pour ce qui est des paquets de demande (libération, réinitialisation, reprise), ils disposent d'un champ d'explication composé de deux octets:

- un octet (cause) rempli par le réseau indique la source (demande par l'ETTD, par le niveau technique du réseau,...).
- un octet (diagnostic) étant à la disposition de l'utilisateur (ETTD) donne des renseignements supplémentaires. Ce diagnostic doit être défini par un protocole de bout en bout.

1.2.2. Fonctions spéciales.

En plus des fonctions déjà décrites, le protocole X25 paquet offre encore quelques options ou facilités pour étoffer le travail avec le réseau:

- ajoutes au contrôle de flux.

  Le paquet RNR est une version du paquet RR disposant en plus d'une fonction de "non prêt à recevoir". Seul un ETTDa peut prendre l'initiative de l'émission d'un RNR. Ce paquet est transmis à l'autre ETTDb et ainsi lui "demande" d'arrêter l'émission. La transmission sera débloquée par un envoi RR. Il n'y a aucune obligation dans le paquet RNR et entr'autre il n'empêche pas la réception de paquets par l'ETTDa, à concurrence du crédit. Si l'ETTDa ne tient pas compte de ces paquets, il peut se servir de la mémoire du réseau qui garde copie des paquets recus après le passage du RNR et recevoir à nouveau ces paquets par une demande spéciale (paquet REJ).

- modification du débit de la ligne.

  Cette modification est possible en changeant la dimension de la fenêtre et la longueur maximum des paquets. Ces deux variables sont liées au nombre maximum de voies logiques sur cet ETTD et au débit physique de la ligne par une règle donnée dans les documents TRANSPAC. La dimension de la fenêtre et la longueur maximum des paquets peuvent être déterminées lors du raccordement ou être modifiées dynamiquement lors de l'établissement de chaque voie logique
  (elles figurent alors dans le paquet d'appel).

- bits spéciaux du paquet de données.

  Dans le paquet de données, il existe:
  - d'une part un qualificateur de données dont l'usage est déterminé par accord entre les deux ETTDs.

- d'autre part un indicateur de données à suivre signalant que ce paquet est suivi d'un autre dont les informations constituent un prolongement de celle-ci.

- données d'interruption.

Il s'agit d'un paquet possédant un champ (1 octet) qui contient des données de l'utilisateur. Ce paquet possède les propiétés suivantes:
- il n'est pas soumis au contrôle de flux.
- il est prioritaire en cas de file d'attente.
- il est détruit par un réinitialisation.
- il possède son propre acquittement (paquet confirmation d'interruption).

- voies logiques permanentes.

C'est une liaison entre deux ETTDs qui est établie pour une très longue durée; il n'existe plus de paquets d'appel, de libération ou de reprise pour ces voies logiques car ils sont remplacés par des paquets de réinitialisation.

- voies logiques spécialisées en entrée ou en sortie.

Certaines voies logiques sont particularisées lors de l'abonnement, elles interdisent l'émission ou la réception d'appel.

1.2.3. Options externes.

Pour éviter des blocages intempestifs, TRANSPAC conseille l'utilisation

- d'un temporisateur armé lors de l'envoi de demandes (appel, libération, réinitialisation, reprise) et désarmé lors de la réception des confirmations. Cela évite de rester bloqué si la ligne physique est rompue ou en cas d'incident de l'ETCD et/ou du réseau.

- d'un système évitant l'interblocage lors de l'échange des données. Deux applications créant des informations à envoyer avant de consommer les données recues peuvent saturer la communication dans les deux sens en bloquant les mémoires des ETTDs et du réseau.

- d'un algorithme pour choisir les numéros de voies logiques. Cela pour éviter les collisions d'appels. Les numéros des voies logiques établies au départ de l'ETTD commencent à 0; tandis que pour celles établies au départ de l'ETCD, les numéros partent du nombre maximum vers 0.

- d'un système de sauvetage sur disques qui peut être utile mais doit être envisagé avec un protocole supérieur prévu entre les deux ETTDs.

1.3. Protocole X25 paquet implémenté.

1.3.1. Conditions de travail.

Dès que l'on pense à l'implémentation, un problème se pose : celui des tests. Comment vérifier la validité du programme gérant le protocole X25; surtout lorsqu'on ne dispose ni d'un réseau ni d'un système simulant correctement le réseau?

L'idée est alors
- d'abord de construire un ETTD existant en deux versions ETTDa et ETTDb dans une même machine;
- ensuite de construire un ETCD (noeud de réseau) simulant le comportement du réseau.
- et finalement on relie le tout. voir schéma S19.

Ce genre de solution nécessite deux machines pour le niveau paquet et quatre pour le niveau trame. Cela permet de vérifier au moins la cohérence des programmes.

Cette solution est cependant fort compliquée dans le chef de l'ETCD paquet dont on ignore le comportement sous l'influence du réseau. En mettant certaines contraintes on peut supprimer l'ETCD paquet.

1.3.2. Connexion directe ETTD-ETTD.

Quel est le travail de l'ETCD?
Vu ce qui précède, on constate que le protocole X25 paquet possède plusieurs degrés de complexité. Un ETTD peut comprendre le strict minimum pour être compatible ou bien posséder toutes les options et facilités. L'ETCD doit alors, soit suppléer à la "pauvreté" de l'ETTD, soit contrôler l'usage des options par l'ETTD. Cependant, parmi ces options et facilités, beaucoup sont des tests ou systèmes qui peuvent servir à vérifier le travail d'un autre ETTD plus "pauvre". Aussi, on essaie de construire un ETTD en ajoutant au strict minimum des fonctions pour pouvoir

- émettre, recevoir et contrôler des appels.
- travailler et contrôler un échange de données avec une fenêtre supérieure à 1.
- prendre l'initiative de réinitialisation, libération et reprise.
- contrôler la validité des paquets et de leurs paramètres.
- éviter le blocage sur commandes et/ou sur données. Limiter les collisions d'appels.

Ainsi l'ETTD peut effectuer les tâches qu'un ETCD devrait faire pour le contrôler. Par la suite, cette combinaison, ETTD plus fonctions de contrôle de l'ETCD, sera nommée ETTD tout simplement.

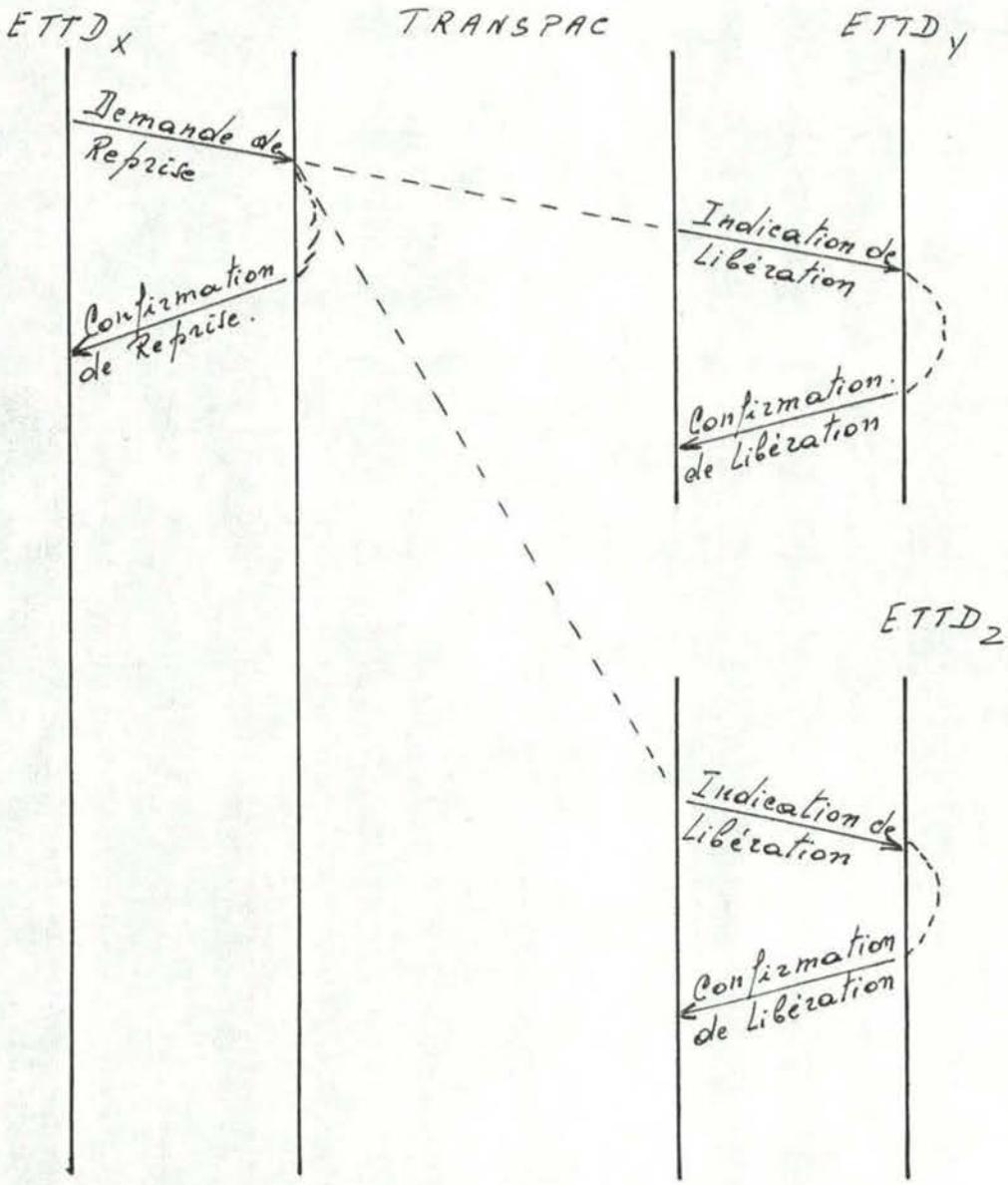
La connexion ETTD-ETTD devient possible en résolvant le dernier problème: la numérotation des voies logiques; ce qui est possible en inversant l'algorithme de la fonction conçue pour limiter les collisions d'appels.

Le niveau trame peut aussi être aménagé, et la solution devient

comme sur le schéma S110.

Schéma S17

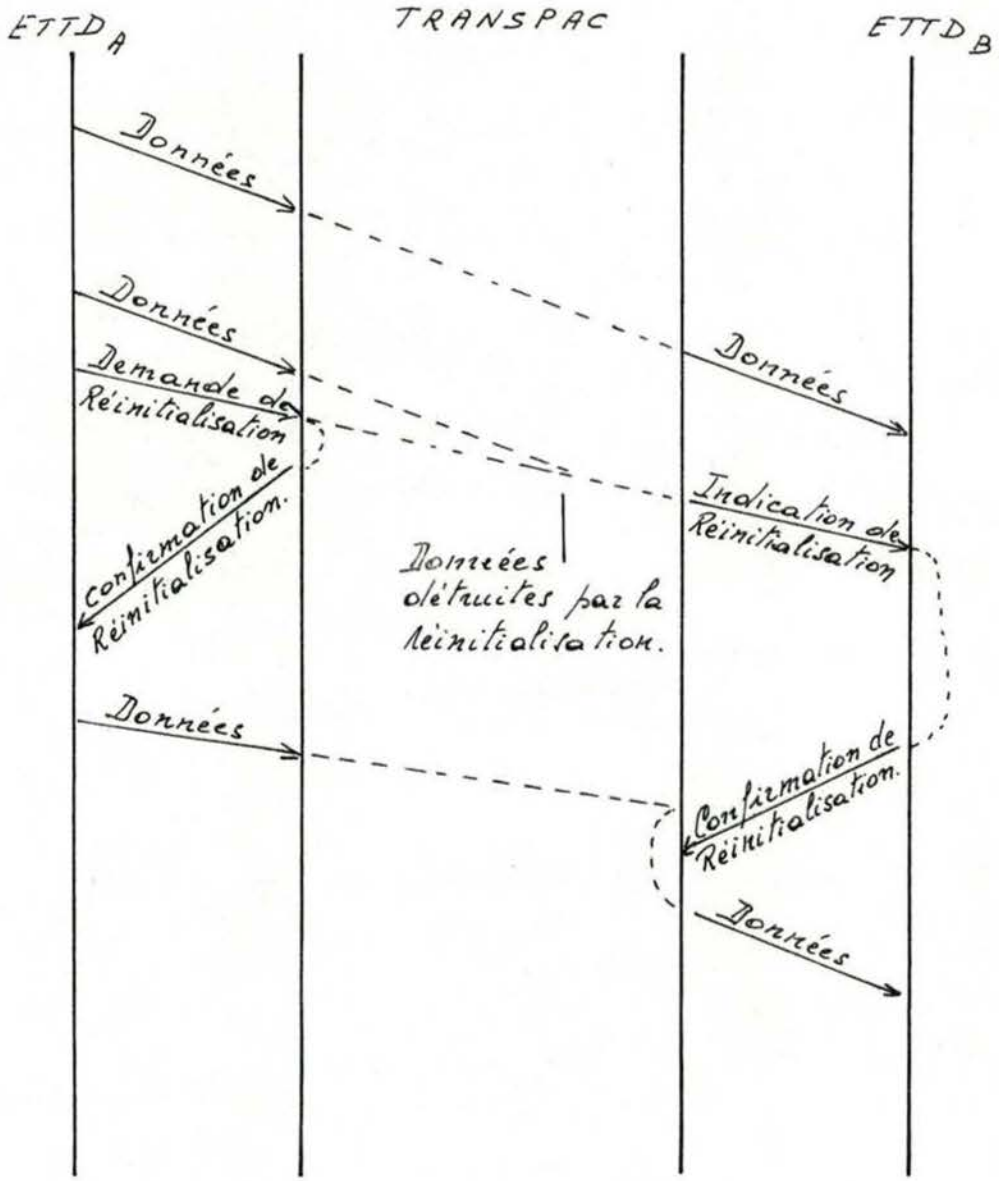
Reprise demandée par l'ETTD$_X$



Schémas:

Schéma S18

Réinitialisation par l'ETTD A.

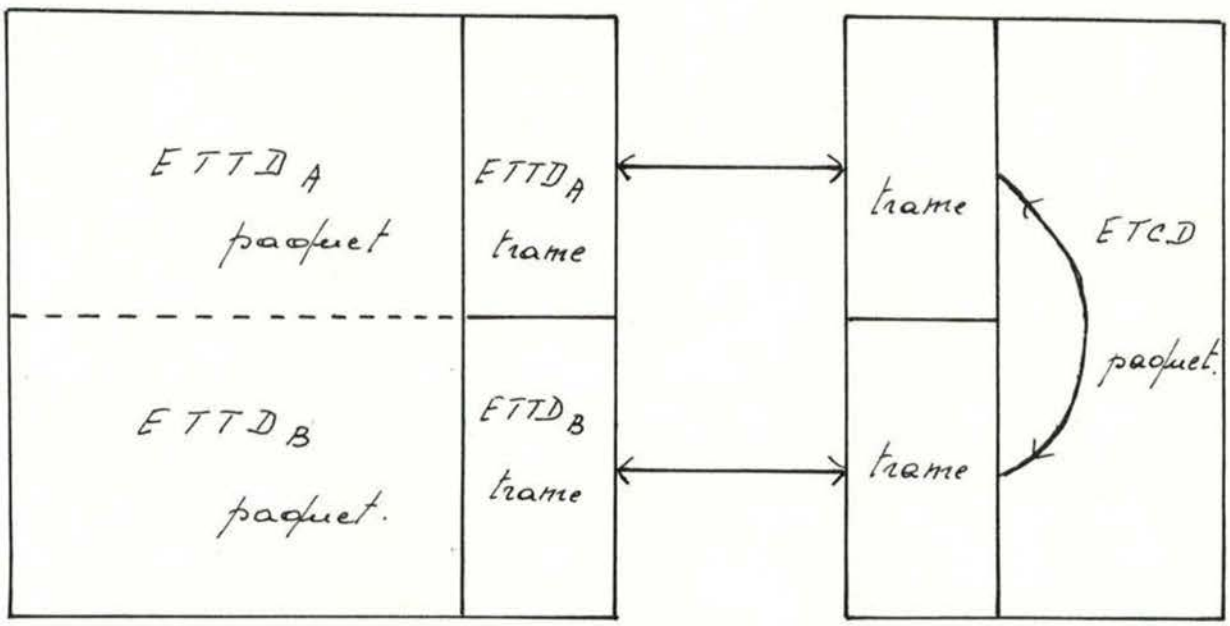


Ce qui nécessite une machine pour les deux niveaux "paquet" et deux machines pour les niveaux "trame".

Le niveau paquet peut encore évoluer en décidant que les n premières voies logiques correspondent au niveau trame ETTDa et les autres au niveau trame ETTDb. Ce qui donne le schéma S111.

Il est bien évident que cette solution ne permet pas de vérifier la compatibilité de l'ETTD avec un réseau ni de "bénéficier" de ce qui est généré ou accepté par le réseau uniquement. Mais au moins on pourra vérifier la cohérence des protocoles implémentés.

Schéma S 19.



Schémas:

Schéma S 110

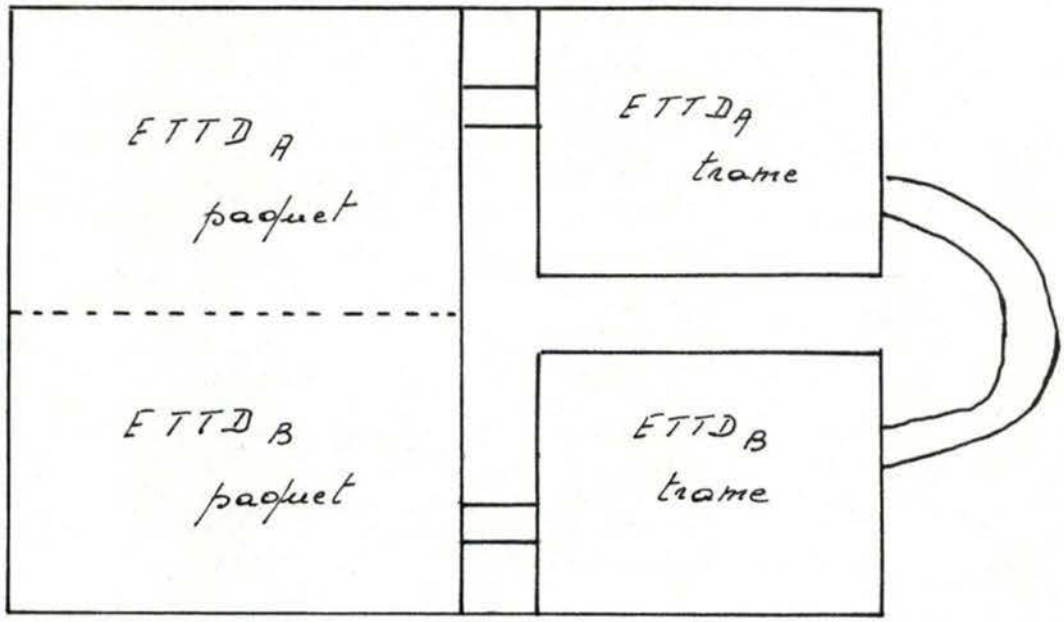
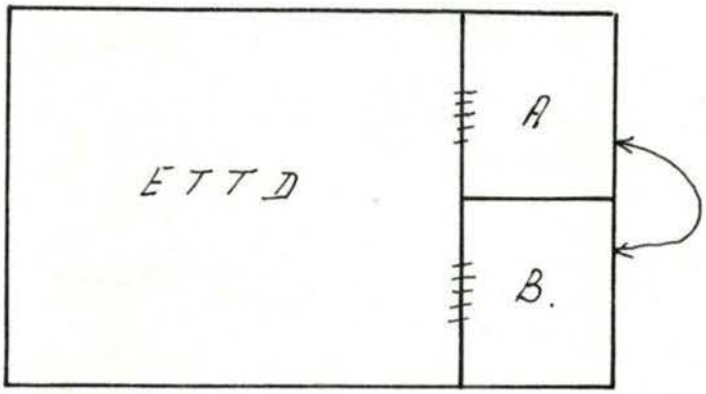


Schéma S 111



1.3.3. Options et facilités.

Parmi les options et facilités décrites plus haut, certaines ont été implémentées :

- temporisateur, détection de blocage, algorithme de numérotation de voies logiques car elles sont nécessaires pour éliminer l'ETCD.
- possibilités d'émettre, de recevoir et de traiter tous les paquets, y compris RNR et interruptions sauf le paquet REJ. Cela a été implémenté dès à présent car les ajouter pouvait provoquer des perturbations dans l'automate représentant le protocole (contrôle de flux).
- dimension de la fenêtre supérieure à 1, bit M et bit Q pour leur interêt.

D'autres options peuvent être implémentées sans difficultés, elles ont été prévues:

- modification dynamique de la dimension de la fenêtre et de la longueur des paquets.
- introduction des voies logiques spécialisées.
- mémorisation des paquets sur disques.

Seul le paquet REJ ne peut et ne pourra être implémenté car il est prévu qu'un ETTD ne reçoit jamais ce type de paquet or on effectue une connexion directe ETTD-ETTD. De plus il perturbe gravement le contrôle de flux.

1.3.4. Table séquentielle.

Pour analyser l'ensemble du protocole, sa représentation sous forme d'automate et de table séquentielle facilite le travail.

Table 1 X25, niveau paquet 1.16

Etat / Evèn.	P1	P2	P3	P4	P5	P6	P7	P81	P82
A	(i)	(i)	4	(i)	(i)	(i)	(i)	(i)	(i)
B	2	(i)	5	(i)	(i)	(i)	(i)	(i)	(i)
C	(i)	(i)	(i)	(i)	(i)	(i)	1	(i)	(i)
D	(i)	6.	6.	6.	6.	6.	1	(i)	(i)
E	(i)	81	81	81	81	81	81	81	1
F	3	5	(e)	(e)	(e)	(i)	(e)	(i)	(e)
G	(e)	4	(e)	(e)	4	(i)	(e)	(i)	(e)
H	(e)	7	7	7	7	1	7	(i)	(e)
I	(e)	(e)	(e)	(e)	(e)	1	(e)	(i)	(e)
J	(e)	82	82	82	82	82	82	1	82
K	(e)	(e)	(e)	(e)	(e)	(i)	(e)	(i)	(e)

A	envoi comm. acceptée	P1	état initial
B	envoi appel	P2	état après envoi d'appel
C	envoi conf. libération	P3	état lors réception d'appel
D	envoi demande libération	P4	état de transferts de données
E	envoi reprise	P5	état lors de collision d'appels
F	réception appel entrant	P6	état après demande libération
G	réception comm. établie	P7	état lors réception ind. libér.
H	réception ind. libération	P81	état après demande reprise
I	réception. conf. libération	P82	état lors réception ind. reprise
J	réception dem. reprise	(i)	événement ignoré
K	réception conf. reprise	(e)	événement déclenchant procédure erreur

Table 2. Transferts de données X25, niveau paquet Détail de l'état P4

Etat / Evèn.	P400	P410	P420	P430	P440	P401	P411	P421	P431	P441	P450
A	400	(i)	420	?	?	401	(i)	421	?	?	(i)
B	400	(i)	400	430	430	401	(i)	401	431	431	(i)
C	420	(i)	420	440	440	421	(i)	421	441	441	(i)
D	410	(i)	410	410	410	411	(i)	411	411	411	400
E	400	(i)	420	430	440	401	(i)	421	431	441	(e)
F	400	(i)	420	400	420	401	(i)	421	401	421	(e)
G	430	(i)	440	430	440	431	(i)	441	431	441	(e)
H	410	(i)	410	410	410	411	(i)	411	411	411	(e)
I	450	400	450	450	450	450	400	450	450	450	450
J	(e)	400	(e)	(e)	(e)	(e)	400	(e)	(e)	(e)	(e)
K	411	(i)	421	431	441	(i)	(i)	(i)	(i)	(i)	(i)
L	400	(i)	420	430	440	401	(i)	421	431	441	(e)
M	(e)	(i)	(e)	(e)	(e)	400	(i)	420	430	440	(e)
N	(i)	(i)	(i)	(i)	(i)	(i)	(i)	(i)	(i)	(i)	400

A	envoi données possibles	P400	état courant.
B	envoi RR possibles	P410	état attente conf. réinit.
C	envoi RNR	P420	état courant + RNR récept.
D	envoi réinitialisation.	P430	état courant + RNR émis.
E	réception données valables	P440	état RNR récept. + RNR émis.
F	réception RR valables.	P401	P400 et attente conf. inter.
G	réception RNR	P411	P410 et attente conf. inter.
H	réception non valables	P421	P420 et attente conf. inter.
I	réception réinit.	P431	P430 et attente conf. inter.
J	réception conf. réinit.	P441	P440 et attente conf. inter.
K	envoi interruption.	P450	état récept. dem. réinit.
L	récept. interr ⇒ envoi conf.	(i)	évènement ignoré
M	récept. conf. interruption.	(e)	évènement déclenchant procédure erreur.
N	envoi conf. réinit.		

Chapitre 2: Protocole supérieur.

2.1. Influence du protocole X25 paquet.

Le protocole X25 paquet est utilisé entre un ETTD et un réseau; même si son influence porte jusqu'à l'autre ETTD, il n'est pas un protocole de bout en bout. Or le problème est de relier et de contrôler deux processus en activité sur des machines différentes. Il manque certaines choses pour atteindre cet objectif.

2.2. Réseau d'ordinateurs différents.

La première difficulté vient de ce que X25 ne s'occupe pas de la nature ni des conditions d'emploi des ETTDs au niveau de l'utilisateur. La puissance, au point de vue X25, du moniteur réseau (voir 0.2) est rééquilibrée et contrôlée par celui-ci; c'est à dire qu'un moniteur disposant de toutes les options et facilités du protocole X25 peut communiquer avec un moniteur qui n'effectue que les fonctions de base. Le réseau aura équilibré leur puissance.

Des problèmes se posent donc à un niveau supérieur:

- Comment signaler qu'une machine travaille en interactif ou en traitement par lots? Les interprêteurs de commandes ne sont pas nécessairement les mêmes; le format d'entrée et de sortie des données peut être aussi différent. Si de plus, la machine fonctionne alternativement suivant les deux systèmes, le problème est d'autant plus compliqué. Certes, on pourrait supposer que l'utilisateur est au courant des conditions d'emploi mais cela n'est pas souhaitable et peu probable.

- Comment savoir où et comment sera connecté le réseau? S'agit-il d'une entrée type terminal ou lecteur de cartes et d'une sortie imprimante ou encore une entrée/sortie spéciale? Tout cela peut avoir une influence sur les formats.

  Exemple:

  pour une entrée lecteur de carte, format:
  8 caractères de contrôle et 72 caractères effectifs.
  pour une sortie imprimante, format:
  1 caractère de contrôle et 131 caractères effectifs.
  pour une entrée/sortie type disque, format:
  bloc physique.

- Comment savoir le code caractère en vigueur sur l'autre machine? La liaison dans ce cas pose des problèmes de conversion d'un code à un autre (éventuellement octet <-> sextet <-> nonet).

Il faut donc construire un protocole supérieur qui définirait par exemple:

- un format commun de données.
- quelques commandes spéciales lisibles par n'importe quel interprèteur dans les conditions d'emploi les plus diverses.
- des fonctions de conversion de formats.

La liste d'exemples n'est pas exhaustive.

2.3. Absence de contrôle de bout en bout.

La deuxième difficulté est le transfert de données à travers le réseau. Celui-ci certifie que les paquets de données seront acheminés dans l'ordre et avec une probabilité d'exactitude élevée mais non égal à 1; et il ne fournit aucun moyen de contrôle. Cela oblige l'utilisateur à faire "confiance" au réseau d'autant plus qu'il n'a même pas accès aux vérifications de parité (FCS au niveau trame) puisqu'elles sont faites à ce niveau et même dans un module hardware. Donc difficultés

- pour récupérer une erreur détectée,
- pour insérer un nouveau contrôle de parité qui utiliserait les résultats de l'ancien,
- pour repérer l'origine de l'erreur c'est à dire voir si l'erreur vient de l'intérieur du réseau ou seulement de la liaison ETTD<->ETCD.

A remarquer également que tout nouveau protocole de transfert de données entre ETTDs (utilisateurs) est tributaire de ce qui est défini en 2.1 et ne peut entrer en contradiction avec le contrôle de flux du niveau paquet, principalement pour ce qui est du flux de paquets et surtout des procédures de réémission qui pourraient être définies.

2.4. Echange de données indépendantes de l'utilisateur.

Troisième difficulté et la plus proche de l'implémentation: c'est l'échange, entre les deux ETTDs, d'informations indépendantes de celles de la communication entre les applications. Ces informations se rapportent:

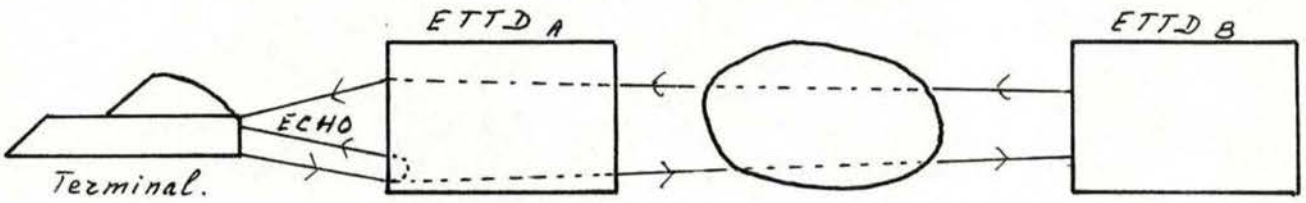
- à l'utilisateur: type de l'utilisateur, sa priorité au niveau de l'un des ETTDs. Un super utilisateur sur un ETTD doit-il posséder un même statut sur un autre ETTD? N'est-il pas intéressant de définir des utilisateurs particuliers au niveau du réseau ou d'une communication ETTD-ETTD.
- aux applications: nombre, type, état, numéro de processus, caractère à envoyer pour les tuer.
- au contrôle des terminaux, ou lecteur de cartes, ou... : caractères à envoyer pour lire une carte, stopper une imprimante, changer le mode du terminal. Exemple:

> Lors du login: l'entrée des caractères doit se faire sans écho pour le mot de passe; or ce n'est pas la ligne qui se mettre "non écho" mais l'ETTD où est relié le terminal. Voir schéma S21. Il faut arrêter l'écho en A même si l'ordre vient de B.

Toutes ces informations sont à échanger entre les deux ETTDs mais leur origine et leur destination ne sont pas les applications mais les

systèmes d'exploitation ou les protocoles supérieurs

Schéma S 21



ou autres fonctions spéciales. Il faut donc les qualifier pour l'émission, les repérer en réception et les traiter de façon bien précise.

2.5. Conclusions.

Tous ces exemples prouvent l'utilité de protocole supérieur au niveau X25, défini pour plusieurs types de machines de fonctions différentes.

Le protocole X25 fournit quelques armes entr'autres:

- groupe fermé d'abonnés.
- voies logiques spécialisées.
- qualificateur dans les paquets de données.

Les moniteurs réseaux des ETTDs doivent être renforcés pour supporter des protocoles supérieurs (sauvetage sur disques,...).

Il existe certainement d'autres problèmes; le but était d'en montrer quelques uns, les principaux, et non d'en faire une liste complète et encore moins d'y apporter une solution.

Le moniteur réseau implémenté ne s'occupe d'aucun de ces problèmes; pour l'instant, il se contente de gérer le protocole X25, d'effectuer la liaison avec les applications et éventuellement de les controler; les seules vérifications ont pour but de lui éviter tout problème et corruption: arrêt immédiat de l'exécution de l'application fautive et libération de la voie logique sont les solutions à la moindre difficulté.

Chapitre 3: Le réseau et ses utilisateurs.

3.1. Utilisateurs internes et externes.

Lorsqu'on regarde les utilisateurs d'un ordinateur qui se servent d'un réseau, on peut les répartir en deux catégories:

- ceux qui, à l'origine, étaient déjà actifs sur l'ordinateur et qui ont demandé à utiliser le réseau. Ce sont les utilisateurs internes.
- ceux qui sont devenus actifs sur cette machine-ci en venant du réseau. On les nommera par la suite les utilisateurs externes.

Par rapport au réseau, un utilisateur est toujours interne sur l'un des ETTDs et externe sur l'autre.

Au niveau du moniteur réseau, on essaye de tendre vers une banalisation des utilisateurs; ce qui veut donc dire que le moniteur ne fera rien de plus pour contrôler un utilisateur interne qu'un utilisateur externe.

Pour les premiers, il faut donc construire un système de contrôle capable de prendre des initiatives lorsque la liaison par le réseau est coupée.

Pour les seconds, il faut un système qui traduira en clair les indications venant du moniteur réseau ou codera et vérifiera les demandes de l'utilisateur.

3.2. SHELL - interprêteur de commandes.

Comme à tout système d'exploitation, un interprêteur de commandes est relié à UNIX, son nom: le SHELL. Pour chaque terminal, un processus est créé et y est rattaché; cela est fait par INIT, autre processus qui a été lancé par UNIX lui-même.

Chaque fois qu'un processus meurt, INIT est réactivé et en créera un nouveau pour ce terminal. Chaque processus exécute un SHELL, c'est à dire

- commence l'exécution directement après le 'login' de l'utilisateur.
- interprète les commandes et programmes de l'utilisateur en créant un processus fils pour chacun d'eux.
- meurt lorsque l'utilisateur fait un 'break'.

3.3. Etapes de la transmission.

Tout ce qui est relatif à la transmission (établissement et rupture de la communication, transfert de données) se fait en deux étapes.

- étape entre l'utilisateur et le moniteur réseau.
- étape entre le moniteur réseau et le réseau proprement dit. Voir schéma S31.

Pour établir une communication avec un autre ETTD au départ d'une application:

- on établit d'abord une liaison avec le moniteur réseau.
- on prolonge ensuite cette liaison vers l'autre ETTD.

Pour ce faire, on dispose des fonctions primitives "rqline" et "netopen".

A ce moment, on a accès au fonctions du moniteur réseau qui permettent:

- le transfert de ou vers l'autre ETTD.
- le nettoyage ou vidange des buffers relatif à la communication.
- la possibilité d'envoyer des données rapides c'est à dire de provoquer l'envoi d'un paquet d'interruption.
- la possibilité de remettre à zéro toutes les voies logiques; ce qui correspond à une reprise.
- la modification des priorités et variables internes du moniteur réseau.

Les deux dernières fonctions ne sont accessibles qu'à un utilisateur privilégié.

A cela s'ajoutent deux fonctions:

- rupture de la liaison entre le moniteur réseau et l'autre ETTD.
- rupture de la liaison entre l'application de l'utilisateur et le moniteur.

Cette séparation en deux étapes vient du fait que:

- les vitesses de transmission sur le réseau et de consommation (ou production) des données sont sensiblement différentes.
- une rupture de la communication dans le réseau ne doit pas rendre inaccessible les informations bufferisées dans les tampons du moniteur réseau.

Ainsi on peut arriver à la situation suivante:

il existe une coupure au niveau de la liaison à travers le réseau mais, par contre, l'application de l'utilisateur continue à tourner jusqu'à l'épuisement des informations lui fournies par sa liaison avec le moniteur réseau.

D'autre part pour prévenir l'utilisateur qu'il y a des nouvelles informations de contrôle pour lui, le moniteur réseau provoque une interruption du processus. Si l'utilisateur n'a pas prévu de traiter cette interruption, le processus sera tué. D'où nécessité de construire une routine qui prendra connaissance de ces informations et en fera le traitement.

3.4. Commande NETWORK pour les utilisateurs internes.

En voyant le paragraphe précédent, on se rend compte qu'il y a beaucoup à faire lorsqu'on veut travailler en faisant des entrées/sorties sur le réseau; c'est beaucoup trop pour les programmes quelconques déjà écrits (qu'il faudrait modifier) ou à écrire. Le programmeur devrait en effet connaitre toutes les subtilités permises et demandées par le moniteur réseau. Aussi a-t-on partagé le travail:

- d'une part une commande, NETWORK, qui, tout comme le SHELL est responsable des processus utilisateurs vis à vis du système d'exploitation, sera, elle, responsable des processus vis à vis du moniteur réseau.
- d'autre part les programmes utilisateurs ne pourront que lire et écrire sur le réseau puisque celui-ci sera considéré comme un simple fichier spécial (/dev/net) au même titre qu'un lecteur de cartes ou une imprimante.

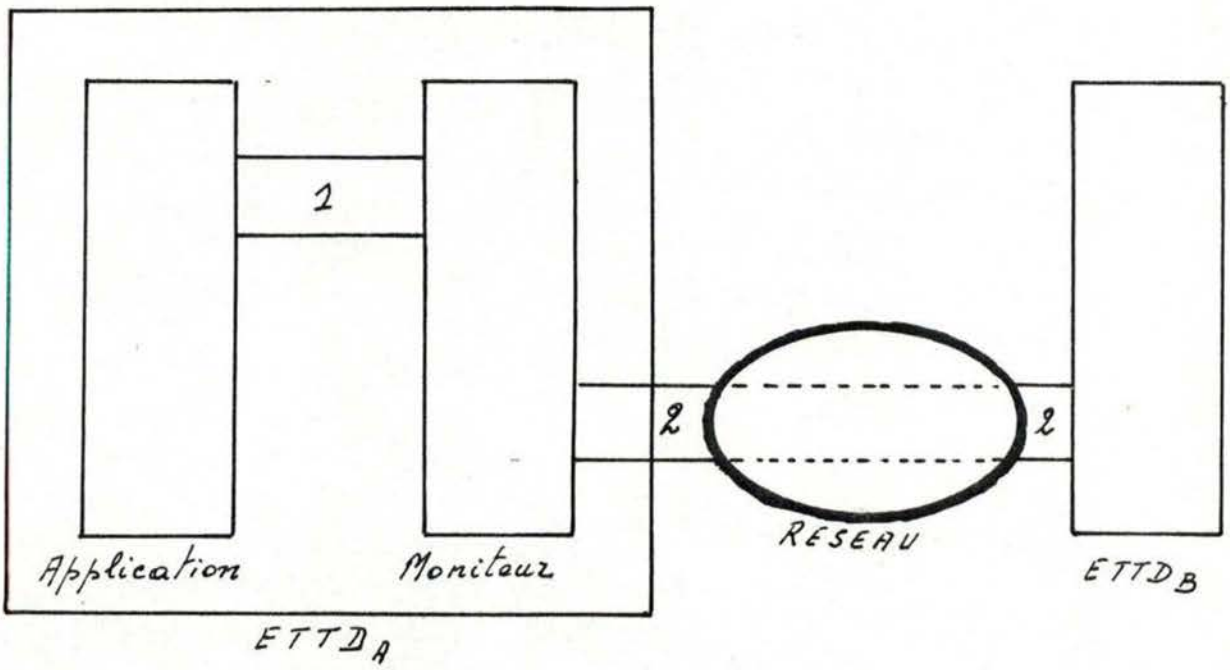
Quel sera le travail de NETWORK? En réalité, il s'agit d'une boucle dont les actions sont: Voir schéma S32.

- après lancement par le SHELL, capter les caractères que l'utilisateur tape sur le clavier du terminal.
- analyser cette chaîne de caractères et
  - soit exécuter le programme ou la commande s'il appartient à l'ordinateur (création d'un processus fils).
  - soit transmettre la chaîne au moniteur réseau pour émission vers l'autre ETTD.
  - soit traduire en commande et l'envoyer au moniteur pour exécution.
- Lire les caractères envoyés par le moniteur réseau et les afficher à l'écran.
- traiter les interruptions pour réceptionner les commandes et données de contrôle, et les traiter en les affichant en clair au terminal.

Remarque:
C'est l'utilisateur qui reste maître pour toutes les décisions à prendre.

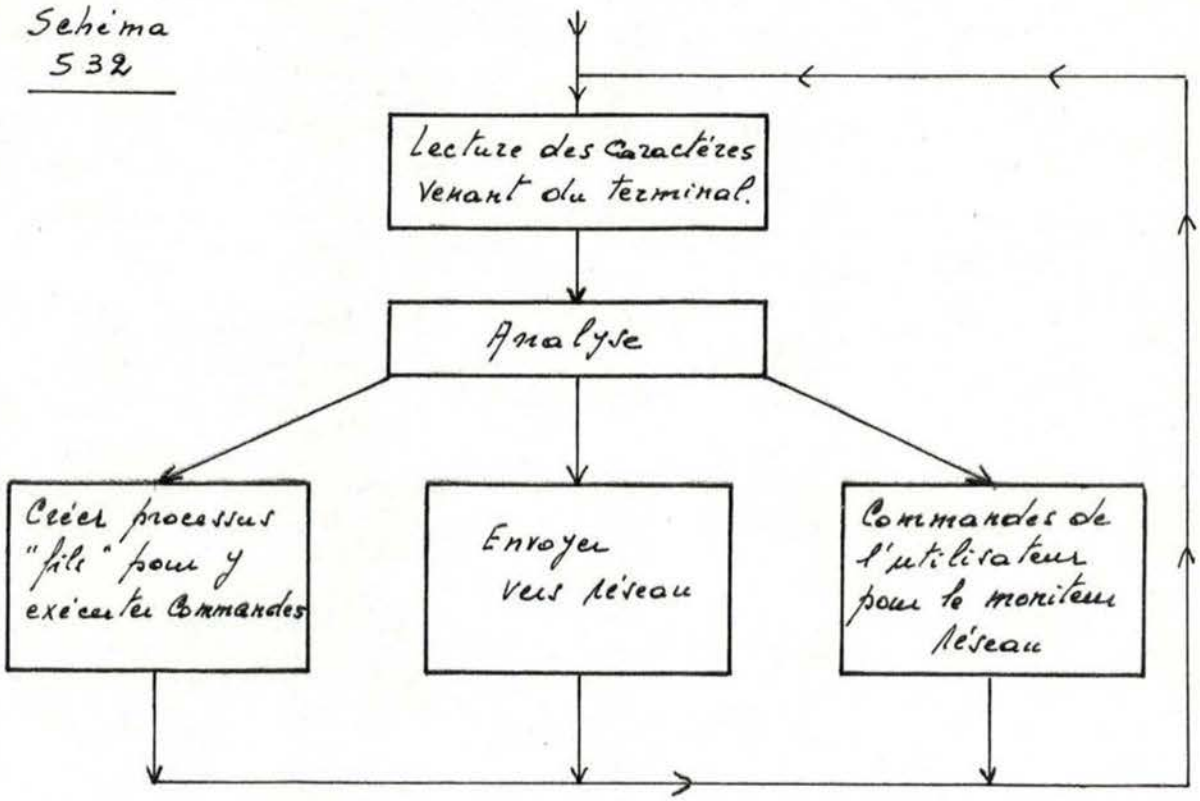
Schémas:

Schéma S32



1 = Liaison application et moniteur 2 = Liaison moniteur réseau autre ordinateur

Schéma S32



Dans le cas où des processus fils sont créés pour exécuter des programmes de l'ordinateur, ceux-ci peuvent effectuer des E/S sur le réseau (uniquement lecture et écriture, pas de commandes). Tout ce qu'ils écriront sur le fichier /dev/net sera transmis à l'interlocuteur, de même tout ce qui aura été reçu sera accessible en lisant /dev/net.

NETWORK contrôlera (comme le SHELL) ces processus fils, intervenant s'il y a des problèmes avec le réseau.

3.5. Commande NETIN pour les utilisateurs externes.

NETIN est une version du SHELL où sont ajoutées:

- une routine modifiant les entrées-sorties standards.
- une routine de traitement d'interruption pour réception des commandes et des données de contrôle. Le traitement est fait immédiatement; il en résulte des décisions qui ont pour but la sécurité de l'ensemble du système. (mort des processus concernés et rupture de communication).

Pour l'instant, NETIN ne vérifie rien quant aux commandes et programmes exécutés. Comme on l'a vu au 2, des effets de bord peuvent contrarier l'exécution de ces programmes.

3.6. Initialisation des NETINs, processus NETINIT.

Le nombre de processus externes est limité lors de l'activation du moniteur. Ils sont créés et initialisés par un processus "NETINIT" qui supervisera leur travail. Voir schémas S33 et S34.

Le rôle du moniteur se limite alors à mettre en liaison une voie logique et un processus "NETIN" jusque là en attente. C'est la seule chose qui différencie, pour le moniteur réseau, un utilisateur interne d'un utilisateur externe.

La fonction du processus "NETINIT" n'est pas sans rappeler celle du processus N'1 de UNIX: "INIT" qui accroche un processus à chaque terminal.

Schémas:

Schéma S 33 — Après Initialisation - Avant Réception d'appel.

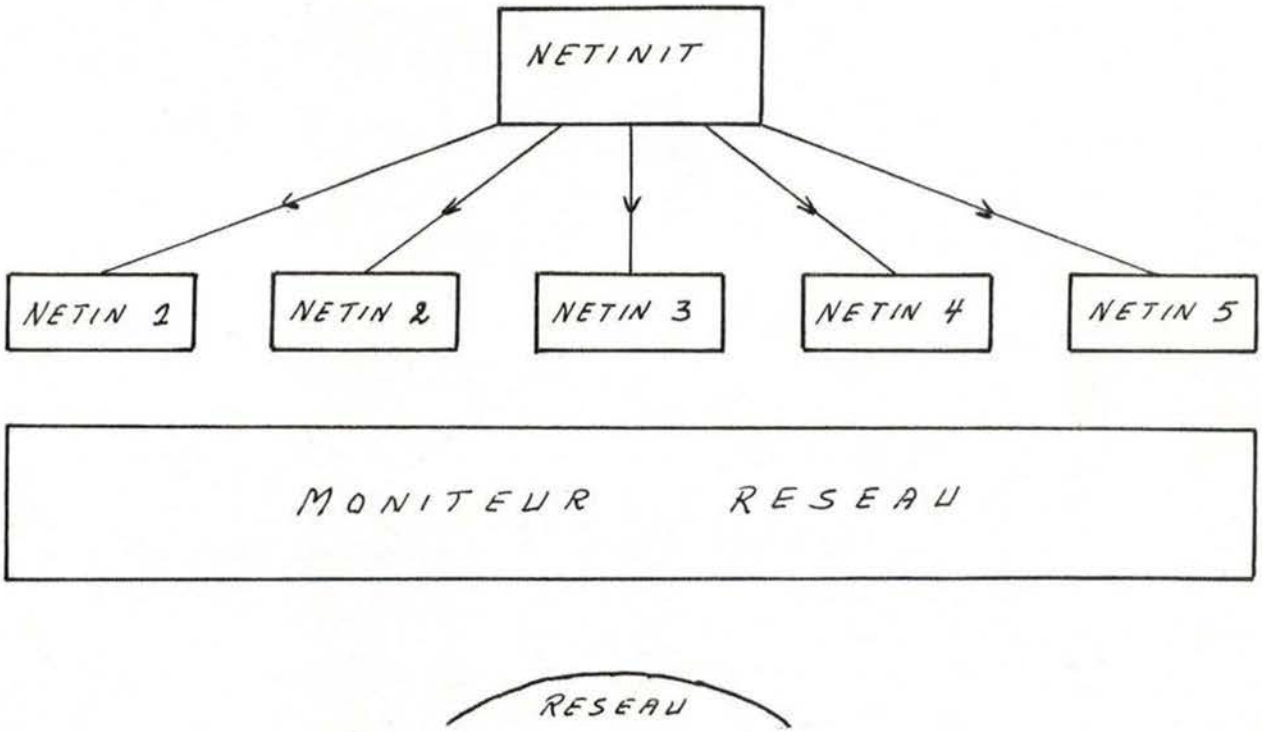
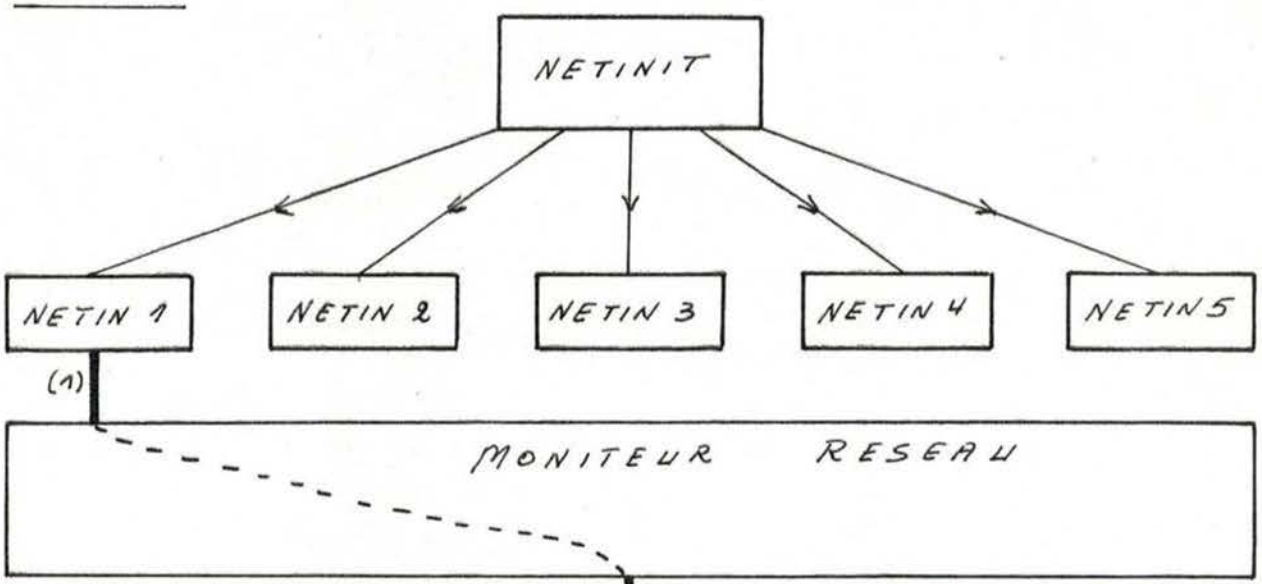


Schéma S 34 — Lors de la réception d'appel



(1) Liaison utilisateur - moniteur
(2) Voie logique - appel entrant.

Chapitre 4: Le moniteur réseau RTTX25.

4.1. Composition du moniteur réseau.

Le moniteur réseau comprend:

- Le moniteur gérant du protocole X25 paquet (fichier PCKX25).
- Un pseudo protocole de bout en bout et responsable de la liaison avec l'utilisateur (fichier ENDEND).
- Des fonctions accessoires (fichiers UCLX25, DRX25, GERANT).

4.2. Place du moniteur réseau dans le système.

4.2.1. Définition.

Par "système", on entend l'ensemble composé du système d'exploitation UNIX et des processus utilisateurs, travaillant ou non avec le réseau, internes ou externes.

4.2.2. Solutions possibles.

Trivialement, on constate qu'un moniteur a deux places possibles:

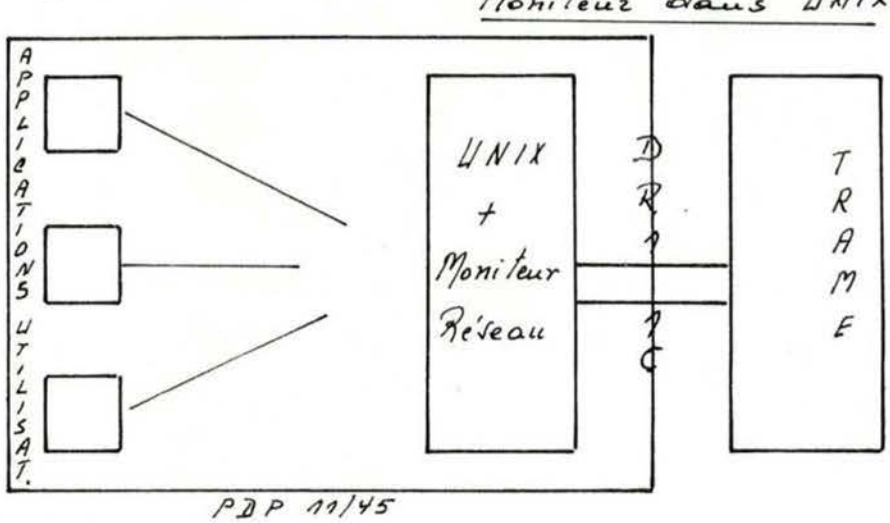
- l'inclure dans le système d'exploitation. Voir schéma S41.
- le laisser à l'extérieur en lui donnant un statut utilisateur. Voir schéma S42.

Cette dernière solution possède plusieurs variantes:

- utilisateur à statut normal.
- utilisateur à statut spécial (super utilisateur).
- définition d'un nouveau type d'utilisateur (résident par exemple).

Schéma S41

Moniteur dans UNIX



Schémas:

Schéma 542 — Moniteur hors UNIX

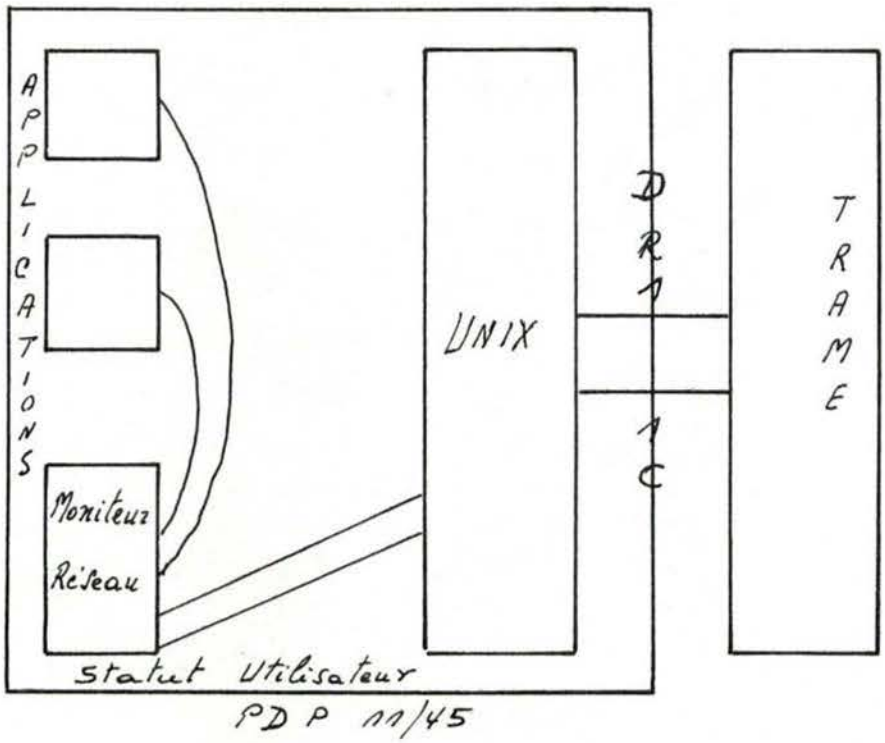
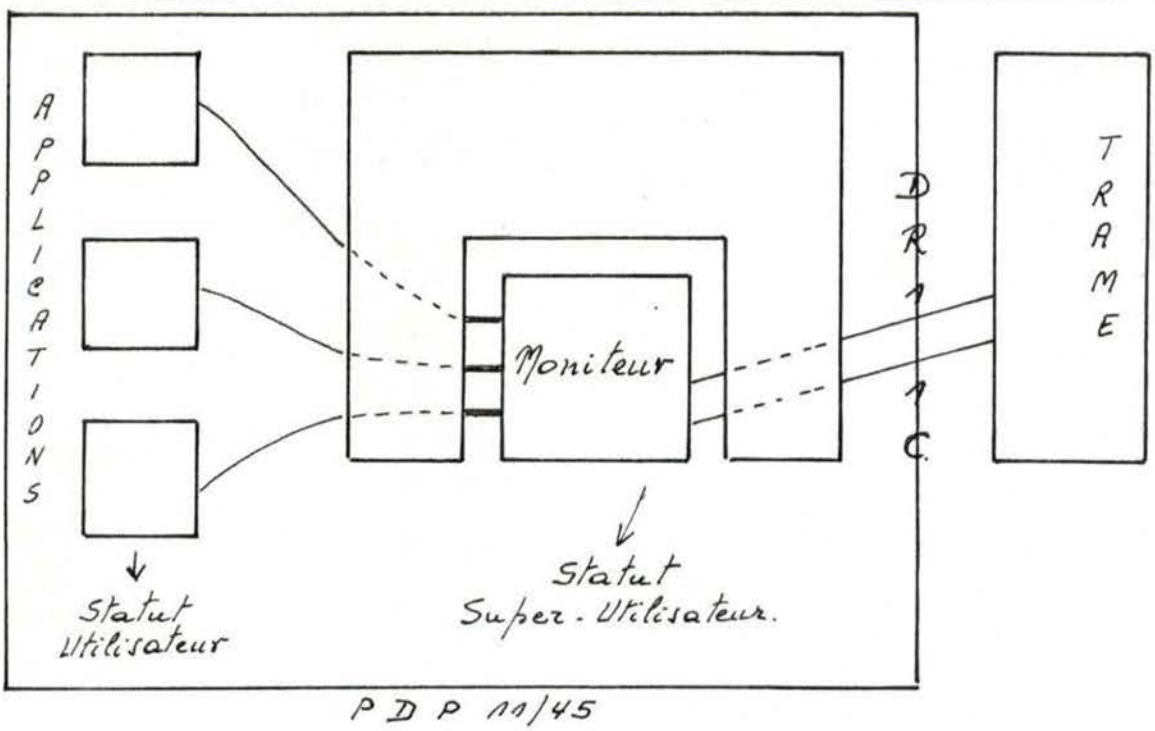


Schéma 543 — Moniteur hors UNIX avec aménagements



Inclure le moniteur dans UNIX semble, du point de vue réseau la meilleure solution:

- Rapidité puisqu'il serait prioritaire sur l'ensemble des utilisateurs, peut être sur des fonctions du système d'exploitation. De plus il serait indépendant de la gestion de la mémoire (swap in swap out) UNIX étant résident.

- Facilité de liaison avec les applications utilisant le réseau grâce aux appels systèmes déjà existant ou à rajouter.

- Transfert avec le niveau trame piloté par les interruptions de l'interface physique DR11-C.

Par contre, si on considère l'ensemble du système, une baisse sensible des performances risque de se produire.

Même en implémentant une version minimum du protocole X25 paquet, le travail est important d'où consommation du processeur et de place mémoire pour le code objet et surtout pour les données; il est également à noter que peu de fonctions de UNIX serviront dans le moniteur réseau.

L'espace disponible pour les utilisateurs sera diminué d'où:

- augmentation des aller retour avec le disque de mémoire secondaire.
- temps de réponse augmenté pour tous les utilisateurs.

Laisser le moniteur à l'extérieur en lui donnant un statut utilisateur n'offre évidemment pas les mêmes avantages pour le réseau. Le fait d'être considéré comme un processus utilisateur le rend tributaire des algorithmes d'allocation de mémoire et du processeur d'où:

- baisse sensible de la vitesse d'exécution.
- ralentissement dans la construction et le traitement des paquets.

Pour toutes les applications n'utilisant pas le réseau, pour les entrées-sorties, il n'y a pas d'augmentation du temps de réponse puisque le moniteur ne risque pas de monopoliser processeur et mémoire.

Quant aux autres applications, elles subiront le contre coup du ralentissement du moniteur et leurs entrées-sorties seront bien moins rapides.

Cette solution possède un autre désavantage se situant dans les problèmes de liaison entre le moniteur réseau et les processus utilisateurs d'une part, et le niveau trame d'autre part. (microprocesseur via interface DR11-C); Des solutions ont été trouvées en ra-

joutant des modules spéciaux dans le système d'exloitation.

Finalement, le point de vue système général, ses performances et le temps de réponse de tous les utilisateurs est plus important que le moniteur réseau. Le PDP est un mini ordinateur, il ne peut supporter une telle surcharge à son système d'exploitation.

C'est pourquoi, il a été décidé de faire exécuter le moniteur en statut utilisateur.

Un autre avantage a également fait pencher la balance: c'est la facilité de mise au point surtout lorsque la machine ne sert pas seulement à construire un moniteur réseau.

Comme précisé, plus haut, plusieurs variantes existent dans cette solution:

- utilisateur à statut normal.
- utilisateur à statut spécial (super utilisateur).
- définition d'un nouveau type d'utilisateur (résident par exemple).

Trois aménagements ont donc été apportés au moniteur réseau et à l'environnement: Voir schéma S43.

- Ajoute dans UNIX d'un module facilitant la communication entre processus répertoriés sous des utilisateurs différents.
- Ajoute dans UNIX d'un module, moniteur d'interface physique DR11-C disposant de tampons et fonctionnant à l'aide d'interruptions hardware.
- Modification du statut du moniteur réseau qui sera activé en mode super utilisateur, lui donnant ainsi une priorité supérieure et un pouvoir de contrôle sur les autres processus.

4.2.3. Moniteur de communication inter processus.

4.2.3.1. Solutions et justifications.

Le statut dans le système étant décidé, il reste à trouver une solution pour communiquer avec les applications sous statut utilisateur. Quel est le problème?

D'un côté, des processus regroupés en applications faisant des entrées-sorties sur le réseau c'est à dire envoyant et recevant des caractères, demandant des actions spéciales (commandes), recevant des données de contrôle. Ces processus sont répertoriés par UNIX sous des utilisateurs différents.

D'un autre côté, un processus, le moniteur réseau, traitant et générant tous ces caractères et commandes. Il faut mettre en

communication ces processus.

Une solution est de définir des zones de stockage temporaire sur disques accessibles alternativement par l'un puis l'autre. Voir schéma S44.

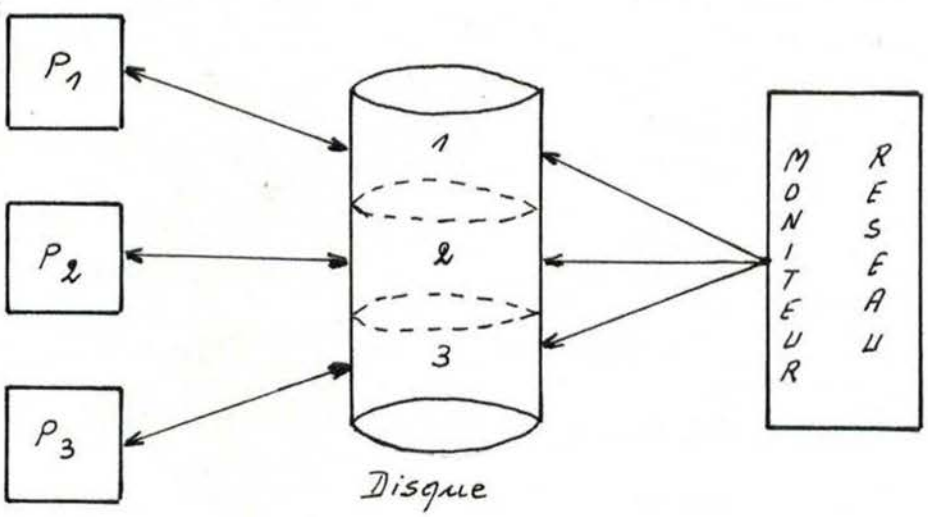
Cette solution relève du bricolage, de plus, elle est d'une grande lenteur. Aussi, a-t-on pensé faire communiquer le processus directement avec les autres.

Il existe déjà un système plus ou moins identique dans UNIX: un processus voudrait entrer en liaison avec le fils qu'il va créer; aussi définit-il un ou plusieurs "pipes" puis il crée son fils. Chaque "pipe" défini est identifiable par deux numéros "descripteur de fichier" (identique à celui attribué lors de l'ouverture d'un fichier quelconque). Ces numéros sont passés comme argument au fils qui les utilisera pour lire et écrire. Le père fera de même et sera ainsi en liaison avec son fils. Voir schéma S45.

Ce système a un défaut: la liaison doit être définie avant la création du fils

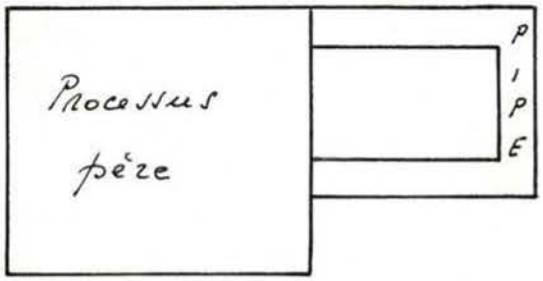
Or, dans le cas présent, le processus moniteur existe depuis longtemps lorsque subitement le processus utilisateur nouvellement créé veut communiquer.

Schéma S44



Schémas:

Schéma 545 Création du "pipe" par le processus "père"



Création du "fils"

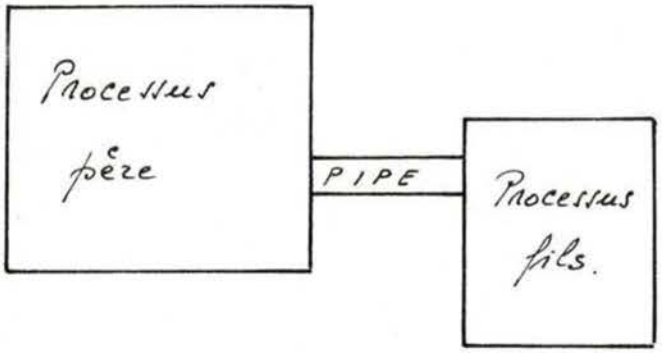
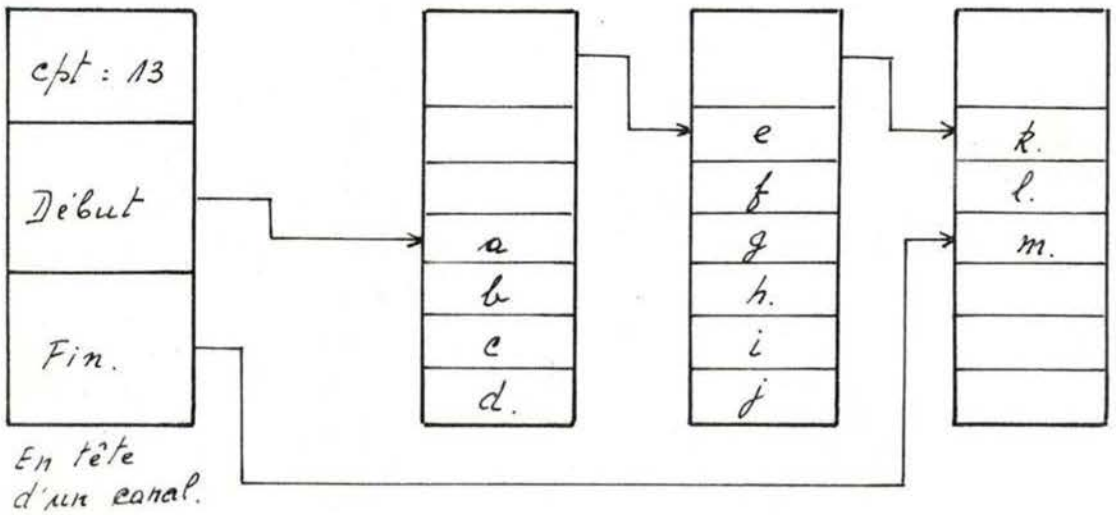


Schéma 548 Bufférisation.



4.2.3.2. Description de la fonction du moniteur réseau.

Pour résoudre ce problème, on a introduit la notion de faux device c'est à dire des éléments possédant un nom au même titre qu'une imprimante par exemple, éléments sur lesquels les opérations d'entrées-sorties habituelles ont une influence (open, read, write,...).

A ces éléments correspondent dans UNIX non pas un appareil physique mais deux files d'attente. Ainsi, deux processus d'utilisateurs différents s'étant mis d'accord sur le nom de ce faux device (il en existe 16) l'ouvrent chacun séparément en mode écriture et en mode lecture. Chaque fois que l'un deux écrit, il remplit une file d'attente et, chaque fois qu'il lit, il vide l'autre. Voir schéma S46. Réciproquement pour l'autre processus.

Une dernière amélioration a encore été faite en ajoutant un 17ème élément dont le comportement est un peu spécial: l'ouvrir (appel système open) équivaut à ouvrir les 16 premiers éléments dans un sens. Les ordres d'écriture (write) et de lecture (read) possèdent un nouveau paramètre qui est un indice pour un des 16 éléments.
Ce 17ème élément permet donc de multiplexer ou de démultiplexer les entrées-sorties vers les 16 premiers ou en provenance de ceux-ci. Voir schéma S47.

Un dernier détail. Outre les quatres appels systèmes habituels (open, read, write et close), les 17 faux devices réagissent également aux appels systèmes "stty" et "gtty" permettant ainsi un passage d'information de longueur limitée (3mots). (stty = write; gtty =read).

On pourra donc échanger des commandes et données de contrôle (stty et gtty) et des informations d'entrées-sorties sur le réseau (read et write). De plus, on reste compatible avec la philosophie de UNIX puisque les entrées sorties avec le réseau se font à l'aide des mêmes ordres que pour un fichier (fichier normal sur disque ou sur fichier spécial = device = imprimante ou terminal).

4.2.3.3. Quelques détails d'implémentation.

Les canaux de communication.

Chaque moniteur PCK possède deux canaux, un pour chaque sens. Chacun est repéré par une entité composée d'un compteur d'utilisation, d'un pointeur vers le premier caractère à lire et d'un autre vers le dernier caractère. Ces caractères sont rangés dans des buffers de 6 octets chaînés entr'eux (le chaînage se fait à l'aide d'un pointeur). L'ensemble pointeur de chaînage plus buffer fait donc 8 octets et se trouve à une adresse multiple de 8. (3 derniers bytes à 0). Voir schéma S48.

Shémas :

Schéma S4b — Utilisation normale du Moniteur.

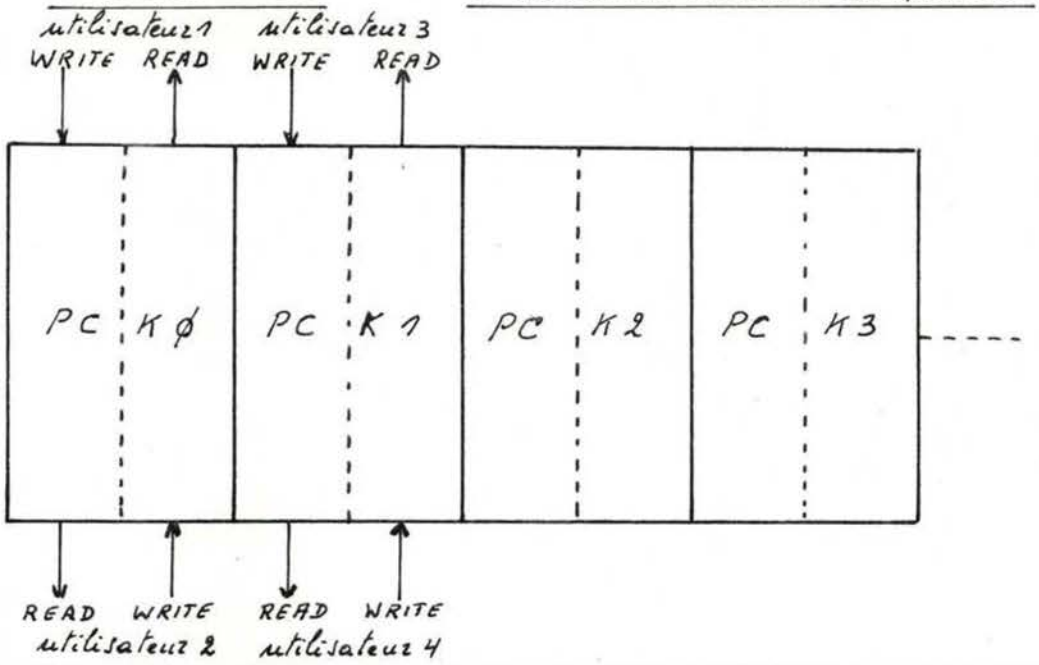
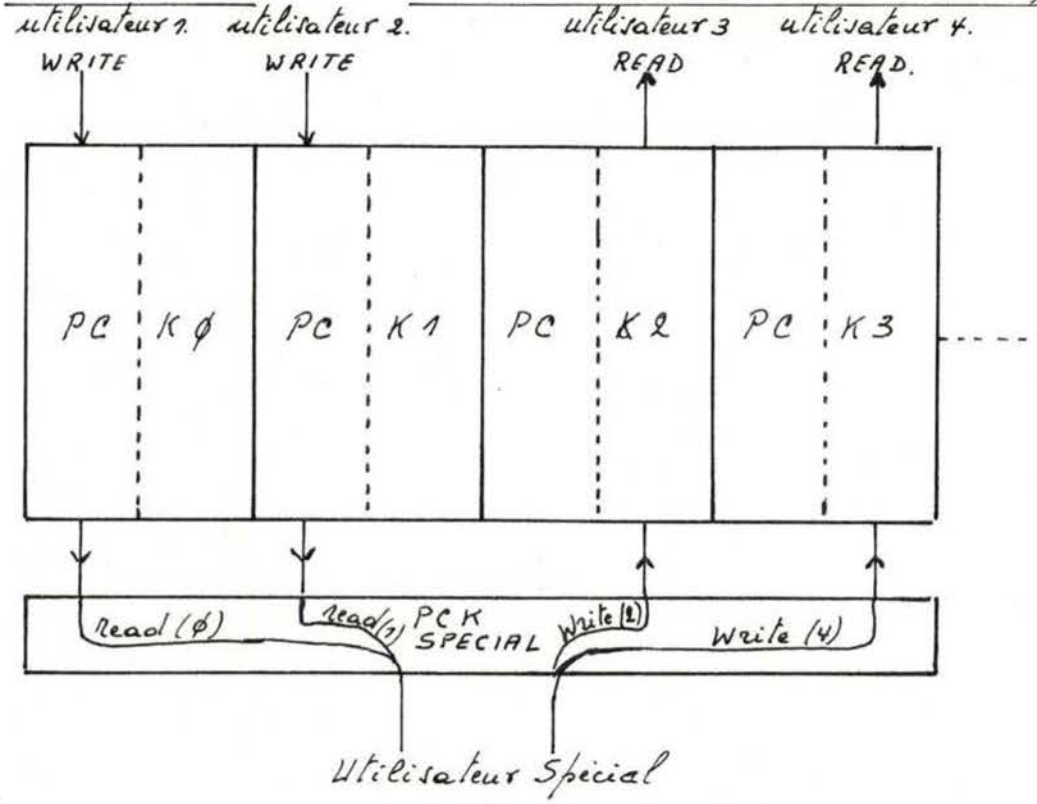


Schéma S47 — Utilisation Spéciale du Moniteur (Multiplexeur)



Des routines spéciales en assembleur ajoutent ou retirent les caractères et gèrent la file de blocs. Ce système de files est identique à celui pour la réception des caractères au terminal.

Canal saturé, canal vide.

Lorsqu'un processus veut écrire alors que le canal est saturé, ou lire dans un canal vide, le moniteur réagit de façon différente suivant l'élément. Pour les entrées-sorties faites directement sur l'un des 16 premiers, le processus est endormi.

Pour celles effectuées sur le 17ème, le moniteur n'endort pas le processus mais lui retourne le nombre de caractères transférés. Il faut bien faire cela puisque le moniteur réseau qui utilise cet élément ne peut être endormi.

Protection.

C'est un même processus qui doit ouvrir un élément dans un sens, et séparément, en mode lecture et mode écriture. Par la suite, cette ouverture pourra être dédoublée autant de fois que l'on veut et tous les fils de ce processus hériteront des descripteurs de fichiers.

Attention: Deux ouvertures en même mode et dans un même sens ne peuvent être faites explicitement.

4.2.4. Communication avec le niveau trame.

4.2.4.1. Conditions de travail.

Le moniteur réseau étant au niveau utilisateur implique que l'accès au fichier (fichier normal ou spécial) se fait par les appels systèmes habituels. Pour piloter l'interface physique DR11-C qui relie le PDP 11/45 au microprocesseur, il doit exister un module logiciel dans le système d'exploitation.

A l'origine, un module (moniteur DR) était déjà présent mais guère performant dans les conditions actuelles: En effet, lors d'un appel système (écriture ou lecture d'une séquence de caractères), il transférait entre l'espace utilisateur et le DR11-C les caractères à la vitesse à laquelle le DR11-C les acceptait ou les émettait; et comme cela était fonction du microprocesseur, le transfert dépendait de la vitesse de celui-ci.

Le moniteur DR ne rendait la main au processus utilisateur qu'une fois le transfert entièrement terminé; puisque la vitesse d'exécution du PDP est plus rapide que celle du microprocesseur,

le processus était souvent endormi.

Aussi, un nouveau moniteur pour le DR11-C a-t-il été implémenté dans UNIX: TRAPAC.

4.2.4.2. Protocole trame-paquet.

Puisqu'on travaille sur deux machines différentes, sachant d'autre part qu'outre les paquets, des commandes et des données de contrôle doivent être échangées entre les deux machines et entre les deux niveaux du protocole X25, il faut définir ce que l'on va passer et comment on va le passer.

Niveau physique.

On utilisera l'interface DR11-C permettant la transmission bidirectionnelle de mots de 16 octets. Les fils de contrôle sont reliés comme suit:

CSRO----->REQUEST-A

CSR1----->REQUEST-B

REQUEST-A<-----CSRO

REQUEST-B<-----CSR1

DATA-IN(0, 15)<-----DATA-OUT(0, 15)

DATA-OUT(0, 15)<-----DATA-IN(0, 15)

Niveau logique.

- La synchronisation des machines au démarage est réalisée comme suit:
  - l'activité d'une machine est signalée par un REQUEST-B.
  - la machine lancée la première attend l'autre. (avec message éventuel à l'opérateur).

- Eléments transmis:
  - Les messages:

NIVEAU PAQUET NIVEAU TRAME

- demande de connexion -->
<-- indication de connexion

- demande de déconnexion -->
<-- indication de déconnexion

- demande de statut -->
<-- indication de statut
<-- demande de statut
- indication de statut -->

<-- indication d'erreur "GRAVE"
- entête de paquet -->
<-- entête de paquet

- Les paquets: Les échanges sont gérés par les bits de contrôle REQUEST-A mis à 1 pour signaler ou'un nouveau mot de 16 octets se trouve dans le DATA-OUT de l'émetteur.

Le REQUEST-B est mis à 1 pour signaler que le mot se trouvant dans le DATA-IN du récepteur est pris en compte et qu'un nouveau mot peut être envoyé.

Les paquets ont une longueur maximum de 131 octets (voir protocole) et sont transmis mot par mot.

Ils sont précédés d'une entête spécifiant leur longueur.

Les messages sont transmis entre les paquets et ne peuvent donc être échangés pendant la transmission d'un paquet. La transparence du contenu des paquets est ainsi assurée.

- Format des messages.

a . type commande
---

bit 15 --> mis a 0 pour une commande
bit 14 - 9 --> indique l'etat de buffout (nombre de place libres / 8)
bit 8 --> mis a 1 si buffin est rempli
bit 7 - 0 --> donne le code de la commande

codification des commandes

bit 7 --> erreur grave entraînant une procédure de maintenance
bit 6 - 4 --> type de l'erreur grave
bit 3 --> indication de statut
bit 2 --> demande de statut
bit 1 --> deconnexion (demande ou indication)
bit 0 --> connexion (demande ou indication)

b. type entête ----------

bit 15 --> mis a 1 pour une entête
bit 14 - 9 --> indique l'état de buffout (nombre de place libres / 8)
bit 8 --> mis a 1 si buffin est rempli
bit 7 - 0 --> donne la longueur réelle (en bytes) du paquet
qui va suivre

Tout ajoute entre le niveau paquet et le niveau trame devra être transparent pour ce contrôle.

4.2.4.3. Fonctionnement du moniteur TRAPAC.

Dans chaque sens de circulation, le moniteur TRAPAC dispose d'une file d'attente (buffer circulaire).

Dans le sens trame vers paquet, ce buffer se remplit à chaque interruption grâce aux routines de traitement de celles-ci. Il se videra lors de chaque appel système pour la lecture effectuée par le processus utilisateur (moniteur réseau). Voir schéma S49.

Le sens paquet vers trame possède la même philosophie de travail.

Compte tenu du fait que circulent non seulement des paquets mais aussi des commandes prioritaires, il ne peut être question de bloquer l'échange pour éviter un dépassement de capacité des buffers; d'où le fait que les transferts entre le niveau trame et le moniteur TRAPAC ne se déclenchent que pour des blocs entiers (bloc = entête + un paquet X25).

Pour être sûr qu'on peut débuter un tel échange, TRAPAC possède un compteur estimant la place libre restant dans les buffers du microprocesseur. Ce compteur sera donc rafraichi comme prévu plus haut.

L'avantage de TRAPAC sur DR est que le moniteur réseau n'est plus influencé par la "lenteur" du microprocesseur. En effet, il transfère rapidement des buffers utilisateurs vers ceux de TRAPAC qui seront vidés par interruption à la vitesse du microprocesseur entretemps: le moniteur réseau aura récupéré le processeur et continuera son travail.

Pour éviter des problèmes entre TRAPAC et l'utilisateur, le transfert se fera aussi par blocs entiers. A chaque demande de transfert d'un bloc, TRAPAC le prendra s'il a de la place ou retournera 0 pour le nombre de caractères lus.

En aucun cas le moniteur réseau ne sera endormi.

4.3. Le moniteur réseau.

4.3.1. Avertissement.

Au moment où ces lignes sont écrites, il existe deux versions opérationnelles du moniteur réseau:

- La version 0 qui comprend seulement un système de gestion du protocole X25 paquet. Son existence avait pout but de vérifier la cohérence de la table séquentielle.
- La version 1 qui est décrite par la suite et est cours d'implémentation.

Une troisième version (version 2) est déjà définie au point de vue fonctionnel et devrait être implémentée une fois que la version 1 aura été terminée et testée avec un niveau trame complet (matériel et logiciel).

Cette version 2 diffère de la version 1 par une gestion unique de tous les buffers et files d'attente et par l'existence de sauvetages continus sur disques.

4.3.2. Description générale.

Le moniteur réseau comporte deux phases:

- une phase d'initialisation dont la fonction est:
  - d'affecter une série de valeurs de base à des variables et à des paramètres.
  - de sauver tout cela sur un fichier temporaire.

  Par la suite, cette phase prendra le nom de INITX25.

- Une phase d'exécution dont les fonctions sont:
  - récupérer les variables et paramètres du fichier temporaire. Cette fonction est remplie par le "main program" connu sous le nom MO
  - gérer le protocole X25 paquet. Fonction connue sous le nom PCKX25
  - gérer un pseudo-protocole de bout en bout, effectuer la liaison avec les applications des utilisateurs et contrôler celles-ci. Fonction ENDEND.
  - effectuer la liaison (entrées-sorties) avec les moniteurs de communication inter-processus (moniteur /dev/pck). Fonction UCLX25.
  - effectuer la liaison (entrées-sorties) avec les moniteurs de communication pour interface physique DR11-C (interface avec microprocesseur) (moniteur /dev/trapac). Fonction DRX25.

- gérer certaines ressources: buffers, temporisateur, files d'attente. Fonction GERANT.

Voir schéma S410.

Schéma S49

Communication TRAME - TRAPAC - PAQUET

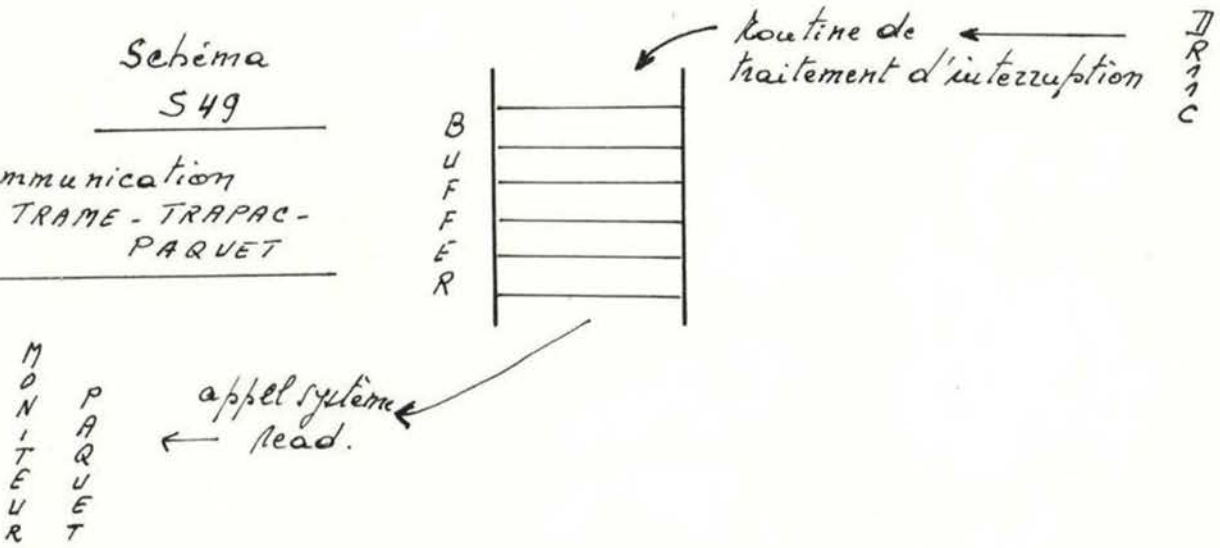
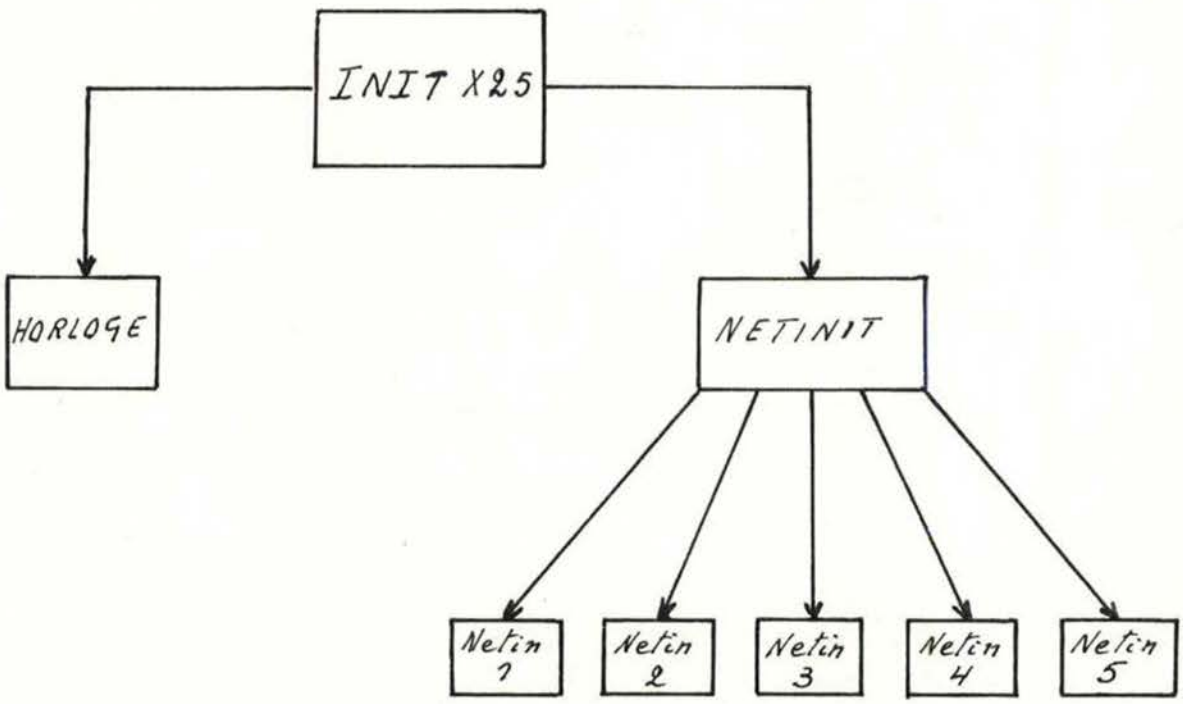


Schéma S 411.

Création des Processus



Schémas:

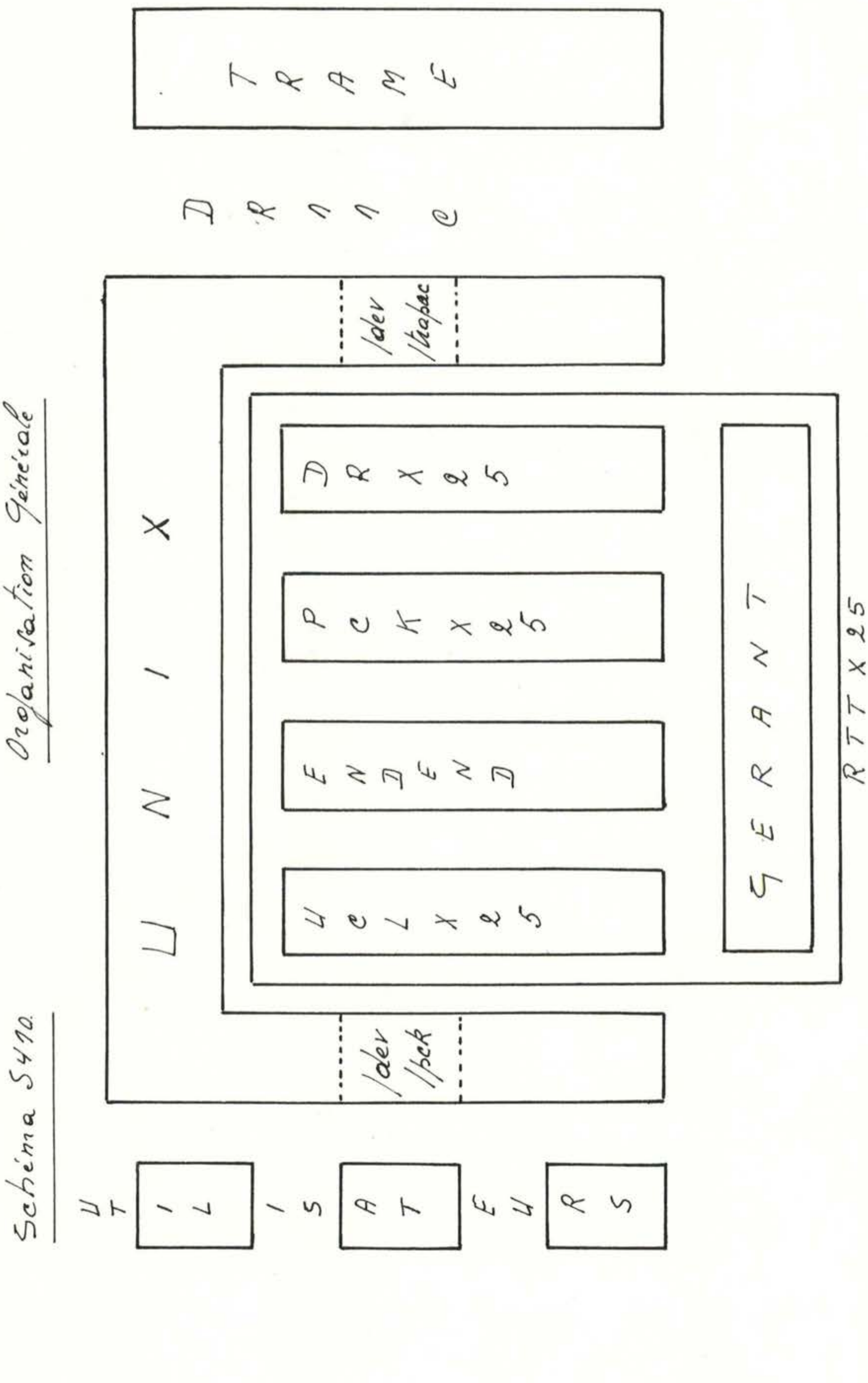


Schéma S410
Organisation Générale
TRAME
UNIX
DRX25
PCKX25
ENDEND
GERANT
RTTX25
UTILISATEURS

4.3.3. Phase d'initialisation.

Le moniteur réseau que l'on appelle RTTX25 ne peut s'exécuter immédiatement, il n'est que la deuxième passe après un programme d'initialisation nommé INITX25.

4.3.4. Description fonctionnelle de INITX25

Comme son nom l'indique, le travail d'INITX25 est d'affecter des valeurs à une série de variables.

- Initialiser des paramètres variables du réseau; ainsi sans recompiler le moniteur réseau, il est possible de modifier la longueur maximum des paquets et la dimension de la fenêtre; de particulariser des voies logiques en les spécialisant pour l'une ou l'autre condition; ou encore de créer des conditions spéciales pour les tests.

- Créer la table séquentielle: en partant d'un fichier (protocol) écrit en français, il construit une table avec les adresses des routines correspondant à chaque transition de la table en utilisant la table des symboles de RTTX25. On peut donc modifier facilement la table séquentielle. Il est nécessaire de garder la table des symboles, résultat de la compilation de RTTX25.

- Construire les différents chaînages nécessaires (buffers).

- Ouvrir les fichiers normaux: "admvl" et "init.h" et les fichiers spéciaux: "/dev/pckz" de communication avec les utilisateurs et "/dev/trapac" de communication avec le niveau trame.

- Créer un processus fils "horloge" qui toutes les trente secondes enverra un signal software au processus RTTX25. Il constituera un des éléments du temporisateur.

- Créer un processus fils pour initialiser les interprêteurs de commandes "SHELL spéciaux" (netin) pour les utilisateurs externes. Voir schéma S411.

Une fois tout cela exécuté, les informations sont sauvées sur un fichier temporaire "init.h". Le programme RTTX25 est lancé; il hérite de tous les fichiers ouverts par INITX25, il commence par lire le fichier "init.h" pour récupérer les informations.

L'avantage de cette solution est de:

- décharger RTTX25 d'un travail long dont le code objet est superflu pour le reste de l'exécution.

- pouvoir changer dynamiquement les paramètres et l'initialisation des variables.

Il faut bien noter que INITX25 a besoin de la table des symboles du fichier exécutable RTTX25 générée lors des compilations et éditions des liens.

4.3.5. Table des symboles.

Actuellement, il est nécessaire de disposer de deux copies du code objet de RTTX25:

- une contenant le code objet seul; c'est elle qui sera chargée pour être exécutée.
- une autre composée du code objet et de la table des symboles sur laquelle travaillera INITX25.

Dans une phase ultérieure, une solution devra être trouvée pour disposer de la table des symboles sans être encombré par le code objet.

4.3.6. Description de RTTX25.

La découpe en ces modules (voir schéma S410) peut se justifier en constatant que chacun a un objectif bien précis:

- PCKX25 qui contient les fonctions pour la gestion du protocole X25 paquet.

- UCLX25 qui contient les fonctions pour la liaison avec les applications des utilisateurs et ce via les moniteurs de communications inter-processus (voir chap 4.2).

- DRX25 qui contient les fonctions pour la liaison avec les moniteurs des interfaces physiques DR11-C, dernière étape avant les microprocesseurs du niveau trame.

- ENDEND qui contient les fonctions de liaison logique avec les applications des utilisateurs. ENDEND contient également les fonctions pour l'intervention "en catastrophe" dans les processus utilisateurs et pour comptabiliser l'utilisation du réseau.

- GERANT qui contient les fonctions utilitaires: gestion des buffers, une partie du temporisateur, gestion de certaines files d'attente.

A cela il faut ajouter une routine principale (main) qui contient la lecture du fichier "init.h" et puis appelle une des routines de GERANT pour commencer réellement le travail.

De cette façon, si un des protocoles (X25 paquet ou bout en bout) ou un des interfaces (/dev/pck ou /dev/trapac) venaient à changer, un seul module serait à recommencer complètement.

4.3.6.1. Répartition du travail.

Parmi les fonctions regroupées dans GERANT, il y a celle de répartir le travail le plus uniformément possible.
Quel est ce travail?
On peut considérer que le moniteur réseau a plusieurs travaux à faire:

- saisir les commandes envoyés par l'utilisateur.
- saisir les états du niveau trame.
- réceptionner les données de l'utilisateur.
- transformer ces données et ces commandes en paquet.
- envoyer ces paquets.
- etc.

On se rend compte que certains travaux ne pourront pas être menés à terme en une seule fois.

Exemple:

l'envoi de données vers l'utilisateur est tributaire de la place libre restant dans les canaux du moniteur de communication inter-processus (/dev/pck). Si on veut y écrire 128 octets, il n'y a peut être place que pour 50. Donc le travail devra être suspendu et reprendre au bout d'un certain temps; temps qui est indéterminé car fonction de plusieurs paramètres (vitesse d'acquisition de l'utilisateur, nombre de processus utilisant le moniteur de communication,...).

D'où nécessité d'arrêter un travail pour le reprendre quelque temps après.

Il faut aussi tenir compte que certains travaux doivent se reproduire régulièrement et indéfiniment (ex: la saisie de l'état du niveau trame car on ne sait jamais quand arrive un nouvel état).

L'idée est donc de construire une file d'attente où sont rangés les travaux:

- ceux qui se reproduisent régulièrement.
- ceux qui n'ont pu être menés à terme.

Continuellement, le travail de tête est sélectionné et poursuit son exécution:

- jusqu'à son terme et est remis en file d'attente s'il doit de reproduire.
- jusqu'à son terme puis abandonné s'il est fini.
- jusqu'à son arrêt pour cause externe (ex: saturation) et remis en file d'attente.

Ce système pourrait, dans une certaine mesure, être comparé à un "scheduler" ou système d'allocation du processeur.

On peut déduire que l'exécution de RTTX25 revient à une boucle sans fin dont le minimum est:

- saisie de l'état du niveau trame.
- saisie des commandes des utilisateurs.

Voir schéma S412.

Ces deux travaux doivent se reproduire éternellement puisque:

- en analysant l'état du niveau trame, on détecte l'arrivée d'un paquet.
- en analysant l'état du moniteur de communication interprocessus, on obtient les commandes et la quantité de données venant de l'utilisateur.

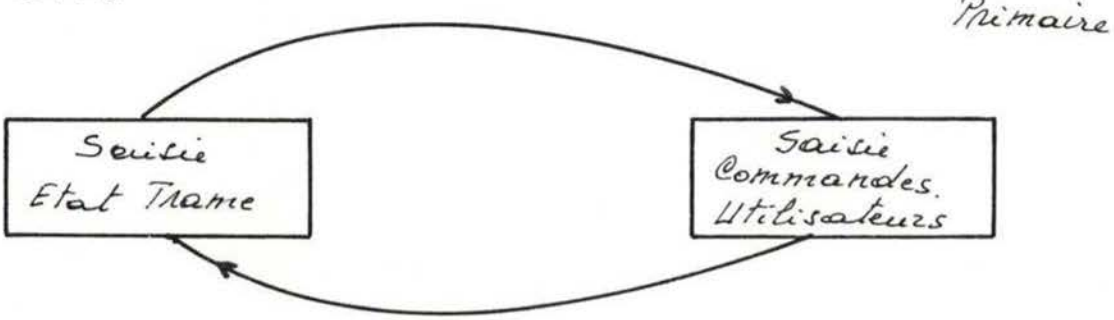
La boucle pouvant être agrandie par l'un ou l'autre nouveau travail; elle évoluera d'ailleurs continuellement.

4.3.6.2. Allocation des buffers.

A la fin de la zone donnée, est située une zone pour les buffers. Chacun possède 134 octets pour les informations, deux pointeurs et un compteur. Non utilisés, ils sont chaînés entr'eux. Pour en obtenir un, il suffit d'appeler la routine "bufp()" qui retournera l'adresse de ce buffer qui est alors alloué et donc retiré de la chaîne des libres. Pour l'y remettre, la routine "bufv(adresse du buffer)" le remettra dans la chaine.

La zone buffer peut être agrandie en demandant à UNIX d'augmenter la mémoire accordée au processus RTTX25. Un garde fou est cependant prévu pour limiter cette incrémentation.

Schéma. Séquence des Travaux
S412



Schémas:

Schéma 5412

Séquence des Travaux

Evolution

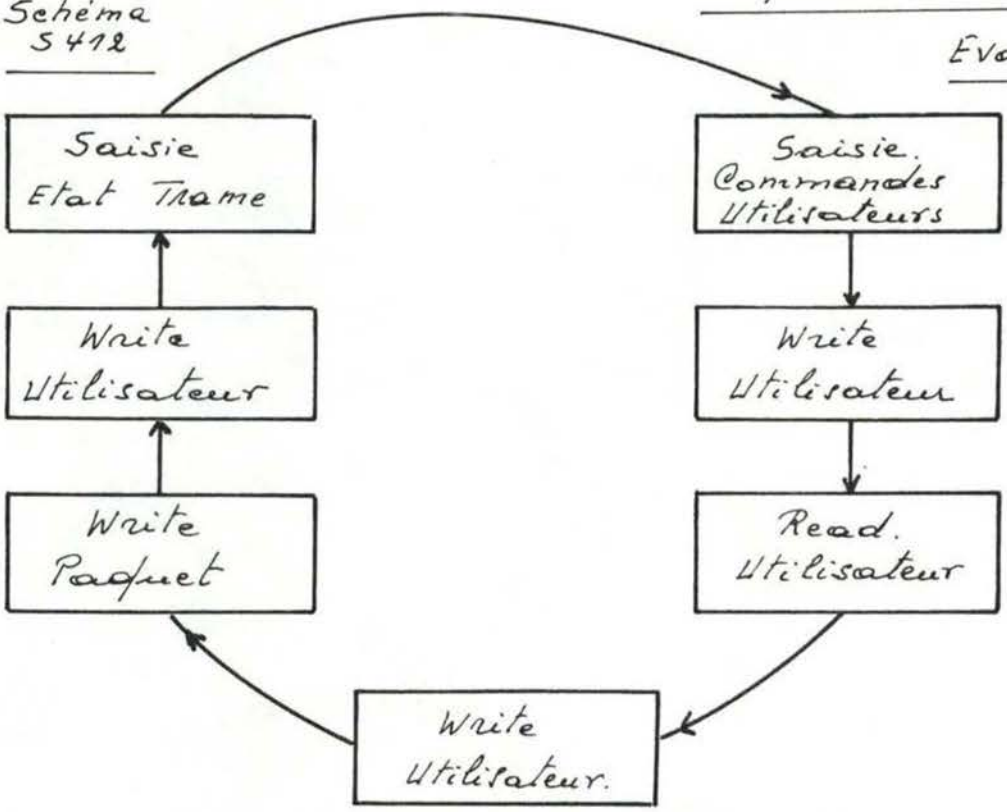
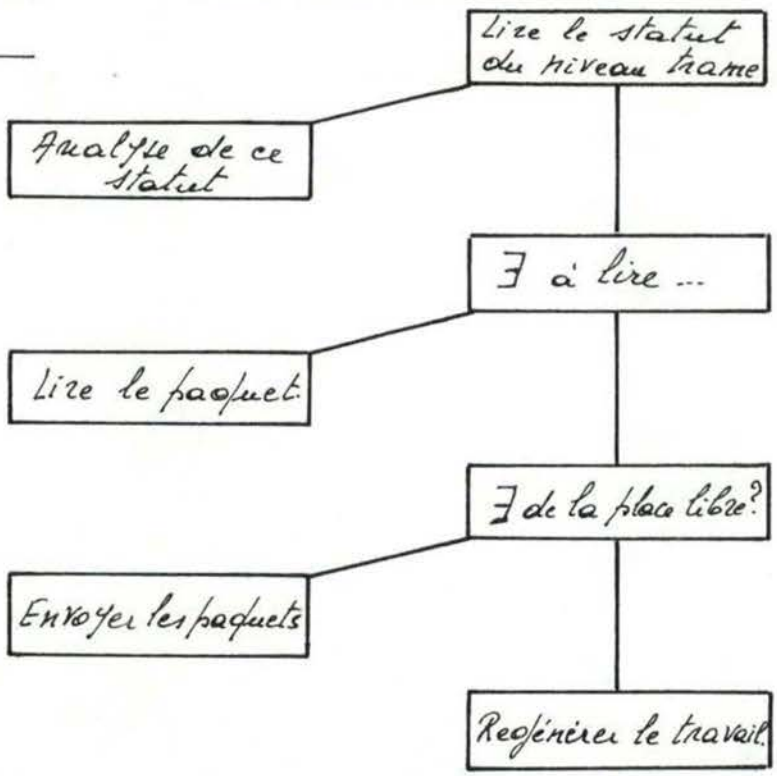


Schéma 5413

Travaux de IRX25



4.3.6.3. Description de DRX25.

Comme on vient de le voir, régulièrement le moniteur réseau doit se renseigner sur l'état du niveau trame; c'est donc une des fonctions de DRX25 que de l'obtenir du moniteur des interfaces physiques (Voir schéma S413.) DR11-C, celui-ci étant directement relié au niveau trame.

DRX25 possède encore deux autres fonctions:

- l'une permet de fournir des paquets au moniteur et par là au niveau trame. Ces paquets ont été préalablement stockés dans des files d'attente:
  - file d'attente pour les paquets de données.
  - file d'attente pour les paquets de commandes.
- l'autre permet de demander au moniteur un paquet en provenance du niveau trame.

Remarque:
Les gestions des files d'attente précitées sont assez différentes l'une de l'autre:

- pour les paquets de données, on se contente de chaîner entr'eux les buffers les contenant.
- pour les paquets de commandes, on essaye d'abord de transférer chaque nouveau paquet dans le dernier buffer déjà chaîné. Si c'est possible, le buffer actuel est libéré; sinon il est alors chaîné aux autres.

Voir schémas S414 et S415.

Un dernier détail : précisons que les paquets de données doivent encore être retravaillés avant l'envoi pour le contrôle de flux. (Voir protocole X25).

Schémas:

Files D'Attente

Schéma S414.

Données

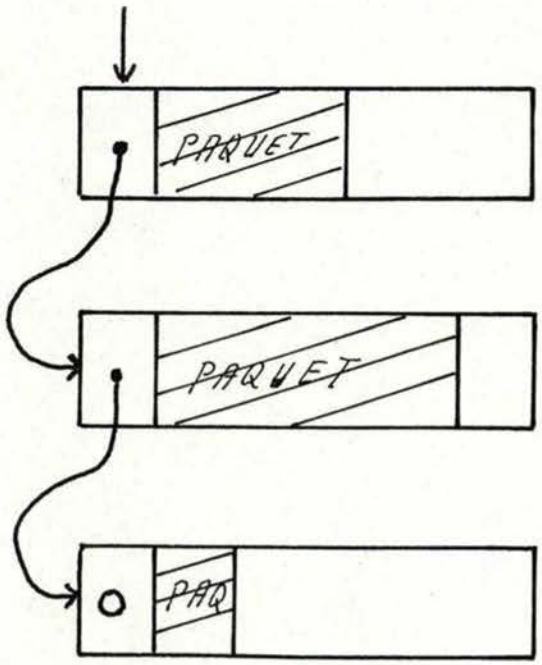
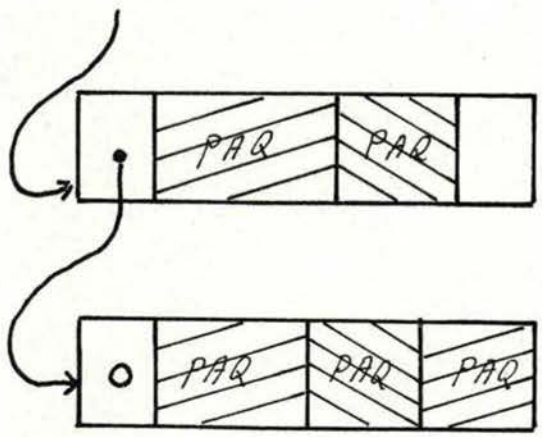


Schéma S415

Files D'Attente.

Commandes.



4.3.6.4. Description de UCLX25.

On l'a vu précédemment, il existe une deuxième fonction permanente: c'est la réception de l'état du moniteur de communication inter-processus et donc la saisie des commandes des utilisateurs.

Rappel:
Comment fonctionne la liaison avec l'utilisateur?
Deux chemins sont prévus à travers le moniteur de communication:

- un pour s'échanger un nombre très limité d'octets que l'on appelle commandes ou données de contrôle.
- un autre pour s'échanger des flots de caractères appelés données ou informations.

Voir schéma S416.

Comme l'envoi d'une commande par un utilisateur est aléatoire, régulièrement on va demander au moniteur de communication si un des utilisateurs en a envoyé une. Si oui, celle-ci doit être traitée immédiatement.

L'exécution du moniteur réseau peut ammener celui-ci à utiliser une des trois autres fonctions de UCLX25:

- l'écriture des données provenant du réseau et à destination des utilisateurs.
- la lecture des données que l'utilisateur désire voir envoyé vers le réseau.
- l'envoi d'une commande ou d'une donnée de contrôle vers l'utilisateur et le signal qui le préviendra qu'il y a quelque chose pour lui. (Ce signal provoque la mort du processus s'il ne le traite pas).

4.3.6.5. Description de ENDEND.

ENDEND regroupe des fonctions faisant partie d'un pseudo protocole supérieur (bout en bout) et d'une liaison logique avec l'utilisateur.

Pour cela, il existe deux niveaux de liaison:

- liaison entre l'utilisateur et le moniteur réseau.
- liaison entre l'utilisateur et son interlocuteur de l'autre ETTD.

La deuxième ne pouvant être établie que si la première existe et elle seule a une influence sur le niveau paquet.

ENDEND possède deux routines pour interpréter:

- l'une les commandes venant de l'utilisateur et en fonction de celles-ci active le niveau paquet.
- l'autre les informations venant du niveau paquet et provoque l'envoi de commandes à l'utilisateur.

Ces commandes ont déjà été décrites au chapître 3. La première routine se chargera en plus de vérifier la validité des commandes et de contrôler le travail du processus pour récuperer les erreurs qui auraient échappé aux tests de NETWORK ou de NETIN.

Ces deux routines s'occuperont également du sauvetage sur disques des informations relatives aux utilisateurs (voir fichier "admvl").

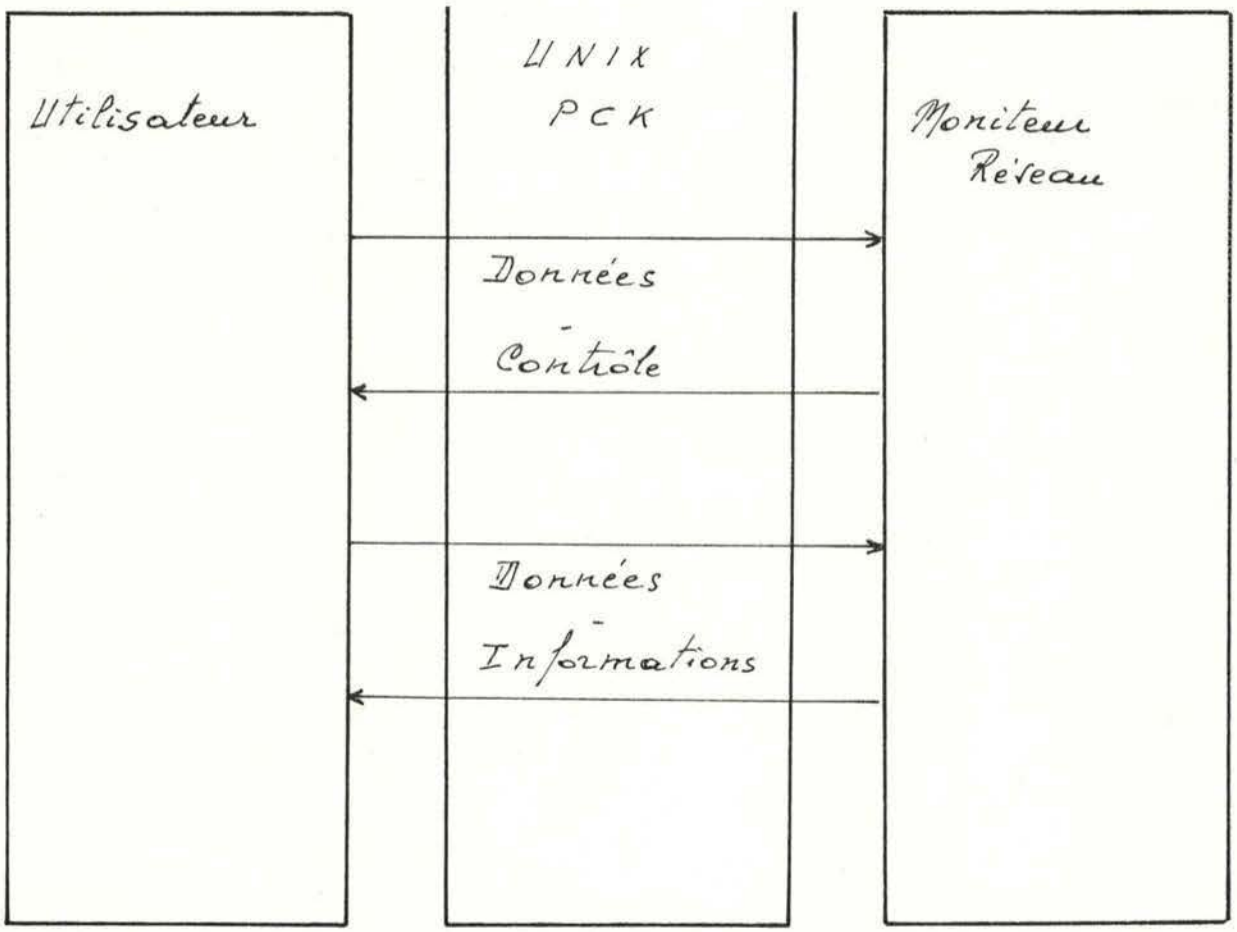
ENDEND possède aussi seize files d'attente destinées à recevoir les paquets de données venant du réseau à destination des utilisateurs. Ce sont les routines de gestion de ces files d'attente qui décident de freiner ou d'accélérer le transfert des paquets sur la voie logique correspondante.

La dernière fonction de ENDEND est de mettre en relation une voie logique et un processus libre lorsque l'appel (au niveau paquet) vient de l'extérieur.

Schémas :

Schéma S416

Communication. Paquet - Utilisateur



Chapitre 5: Conclusions.

C'est d'une façon fort générale que le moniteur réseau a été décrit dans le présent fascicule. Une visualisation plus détaillée requiert la consultation des programmes se trouvant dans le tome II.

Cette première implémentation n'est qu'une ébauche, à caractère expérimental et didactique, de ce que devrait être un moniteur réellement opérationnel.

Une des imperfections les plus importantes est le peu de fiabilité pour l'exécution de certains programmes lorsque cette dernière est demandée de l'extérieur (via le réseau). En effet, UNIX ne possède pas encore d'éléments prévus pour les utilisateurs venant d'un réseau. Il faut bien admettre que la réalisation de tels aménagements constituerait un travail de mémoire tant sont diversifiés les problèmes sous-jacents au statut des utilisateurs.

Les deux principales améliorations à apporter se définissent de la façon suivante:

- la liaison entre le moniteur réseau et les applications.
- la liaison paquet trame ou plutôt PDP11-45 microprocesseur. Cette liaison doit être considérée dans l'optique de rendre accessible tous les paquets non encore traités par le niveau trame et qui doivent être soit modifiés soit détruits.

Il faut reconnaître qu'il n'y a jamais eu de difficultés majeures pour la réalisation de ce mémoire.

Les obstacles rencontrés ne furent pas d'ordre informatique mais leur origine est due soit à une approche inexperte du problème posé soit au bricolage, nécessité par le temps imparti, de certaines solutions (liaison moniteur-utilisateur).

Je formule l'espoir que ce travail que j'ai eu plaisir à réaliser soit non seulement source d'intérêt mais augure de réalisations.

BIBLIOGRAPHIE

I. - MEMOIRES ET RAPPORTS ANNEXES.

- Réalisation d'une liaison X.25 lère partie.
  Mémoire de fin d'études J. ART - F.N.D.P. I979.
- Réalisation d'une liaison X.25 à l'aide d'un micro-processeur.
  Mémoire de fin d'études BEUDIN et ZONE - U.C.L. I979.

2. - A PROPOS DES RESEAUX.

- Manuel "TRANSPAC" par l'équipe technique de RENNES.
- CCITT Livre orange tome VIII. 2.
  Réseaux publics pour données.
  6ème assemblée plénière.
- COMMUNICATION NETWORKS FOR COMPUTERS.
  DAVIES et BARBER.

3. - A PROPOS DE UNIX.

- BELL SYSTEM TECHNICAL JOURNAL.
  Juillet-Août I978 Vol. 57 n° 6 part. 2 (chap. 1 à 6).
- Manuel utilisateur UNIX (chap. 1 à 8).
- PDP 11 PROCESSOR HANDBOOK.
- PDP 11 PERIPHERALS HANDBOOK.
  (DR 11-C Interface).

Errata de la 1ère partie.

<numéro de page>L<nombre de ligne> : compter les lignes en remontant du bas de la page.
<numéro de page>l<nombre de ligne> : compter les lignes en commencant au sommet de la page.

1.15L0 : ajouter:
- La première table représente les séquences d'établissement, rupture d'une voie logique et de reprise du niveau paquet.
- La seconde est le détail d'un des états de la première (P4) qui est la situation active d'une voie logique: transfert de données.

2.1L3 : ajouter:

Ou se fera l'écho des caractères si on utilise un ordinateur de façon interactive au départ d'un autre machine sur laquelle est connecté un terminal?.

4.3L11 :

Pour toutes les applications n'utilisant pas le réseau comme périphérique d'entrées-sorties, il n'y a pas d'augmentation...

Facultés Notre-Dame de la Paix. - Namur
Institut d'Informatique

Conception d'une liaison X.25

Partie II

Etude et implémentation sur PDP. 11/45 du niveau X. 25 paquet et des fonctions primitives

Tome II

Patrick LAMBION

Rapport du groupe XYZ

Mémoire présenté en vue de
l'obtention du grade de
Licencié et Maître
en Informatique

- Août 1979 -

FMB16/1979/7/2/2

Facultés Notre-Dame de la Paix. - Namur
Institut d'Informatique

Conception d'une liaison X.25

Partie II

Etude et implémentation sur PDP. 11/45 du niveau X. 25 paquet et des fonctions primitives

Tome II

Patrick LAMBION

Rapport du groupe XYZ

Mémoire présenté en vue de
l'obtention du grade de
Licencié et Maître
en Informatique

- Août 1979 -

Avertissement.

Ce présent volume contient la plupart des routines du moniteur réseau et des moniteurs de communication (avec le niveau trame et entre les processus). Certaines (fichier ENDEND) sont à prendre avec certaines réserves car tous les tests n'ont pas encore été faits.

On remarquera l'absence de quelques programmes dont:

- les routines de DRX25 qui, à l'heure actuelle, n'ont pas été testées puisque le niveau trame n'est pas encore opérationnel sur le microprocesseur.
- les routines de NETWORK et de NETIN car elles sont beaucoup trop grandes (500 et 1500 lignes en langage 'C' et sans aucun commentaire); de plus elles doivent encore être modifiées dans un autre mémoire.

Une version cohérente et complète sera arrêtée dans le courant du mois de septembre; le lecteur qui désirerait l'analyser peut toujours me contacter pour disposer de cette version ainsi que des programmes qui ont permis de tester chacun des modules.

P. Lambion

TABLE DES MATIERES

TOME II.

Annexe A: Nouvelles routines primitives

A.1. Introduction.

Dans cette annexe sont reprises les routines élémentaires qui apparaissent dans les programmes utilisateurs pour y effectuer des actions de base: entrées sorties en général, retour en fin de routine.

Ces routines, habituellement écrites en assembleur, contiennent le plus souvent un appel système ou au moins une modificationde la pile.

Celles qui vont suivre se divisent en deux groupes:

- les routines existant déjà, elles sont dans la librairie "libc", et ont été modifiées en fonction de leurs nouvelles utilisations dans le moniteur réseau ou dans les interprèteurs de commandes NETWORK et NETIN.
- les routines créées pour les besoins de l'implémentation. Parmi celles-ci, quelques unes sont écrites en langage 'C'; ce sont des routines qui font appel à d'autres d'un niveau inférieur; elles peuvent être considérées comme
  fonctions élémentaires pour la liaison avec le réseau.

A.2. Arrêt de l'exécution d'un processus.

Quatre routines peuvent provoquer une suspension dans l'exécution d'un processus, ce sont:

- "wait.s" : qui attend la mort de l'un des processus fils créés.
- "sleep.s" : qui suspend le processus pendant un temps déterminé.
- "read.s" et "write.s" : qui peuvent suspendre l'exécution d'un processus si le transfert des caractères n'est pas terminé.

Or il se fait que lorsqu'il y a interruption software de ce processus endormi (routine "signal.s" et "kill.s"), ce processus est réveillé, exécute la routine de traitement de l'interruption, puis reprend le cours de l'exécution normale après l'appel système qui a provoqué l'arrêt; et ce, même si l'événement ne s'est pas produit (mort de fils, fin de transfert).

On peut repérer ces redémarages anormaux en testant le contenu du registre R0 après l'appel système et prendre alors les mesures qui s'imposent:

- relancer en repartant à zéro pour ce qui est du "wait" et du "sleep".
- recommencer ou poursuivre le transfert dans le cas d'un "read" ou d'un "write". En ce qui concerne les entrées sorties avec le moniteur de communication inter-processus /dev/pck?, ceux-ci ont été équipés pour renvoyer dans le registre R1 le nombre de bytes qui doivent encore être transférés. Grâce à cela, on peut poursuivre le transfert; mais dans les autres cas, il faut recommencer au risque de perdre des caractères.

A.2.1. Nouvelle version de "wait.s".

```
/ special library

/ wait system call for program using network monitor RTTX25


/ pid = wait();
/   or,
/ pid = wait(&status);
/
/ pid == -1 if error
/ status indicates fate of process, if given


_wait:
        mov     r5,-(sp)
        mov     sp,r5
        jsr     pc,_nargs
        mov     r0,-(sp)
  3:
        sys     wait
        bec     1f
        cmp     $4, r0          / in case of abnormal termination by 'signal'
        beq     3b              / repeat the 'wait' system call
        tst     (sp)+
        jmp     cerror
1:
        tst     (sp)+
        beq     2f
        mov     r1,*4(r5)       / status return
2:
        mov     (sp)+,r5
        rts     pc
```

A.2.2. Nouvelle version de "sleep.s".

```
/ special library

/ sleep system call for program using network monitor RTTX25

/ detects detects and forgets software system call 'signal'

/ result = sleep(time)

sleep = 35

_sleep :
        mov     r5, -(sp)
```

```
        mov     sp, r5
        mov     4(r5), r0
2:
        sys     sleep
        bec     1f
        cmp     $4, r0
        bne     3f
        mov     4(r5), r0
        ash     $1, r0
        br      2b
1:
        mov     (sp)+, r5
        rts     pc

3:
        jmp     cerror
```

A.2.3. Nouvelle version de "read.s".

```
/result = read(file descriptor, buffer address, trans. size)

/result == -1 means error

/if signal interupt, system call continue

_read :
        mov     r5, -(sp)
        mov     sp, r5
        clr     r1
 2:
        mov     4(r5), r0
        mov     6(r5), 0f
        mov     8(r5), 0f+2
        sys     0; 9f
        bec     1f
        cmp     r0, $4
        bne     3f
        tst     r1
        beq     3f
        mov     8(r5), r0
        sub     r1, r0
        add     r0, 6(r5)
        mov     r1, 8(r5)
        br      2b
 1:
        mov     (sp)+, r5
        rts     pc
 3:
        jmp     cerror
```

```
.data
9:
        sys     read
0:
        ..;..
```

A.2.4. Nouvelle version de "write.s".

```
/result = write(file descriptor, buffer address, trans. size)

/result == -1 means error

/if signal interupt, system call continue

_write :
        mov     r5, -(sp)
        mov     sp, r5
        clr     r1
  2:
        mov     4(r5), r0
        mov     6(r5), 0f
        mov     8(r5), 0f+2
        sys     0; 9f
        bec     1f
        cmp     r0, $4
        bne     3f
        tst     r1
        beq     3f
        mov     8(r5), r0
        sub     r1, r0
        add     r0, 6(r5)
        mov     r1, 8(r5)
        br      2b
  1:
        mov     (sp)+, r5
        rts     pc
  3:
        jmp     cerror

.data
9:
        sys     write
0:
        ..;..
```

A.3. Echange d'états par le moniteur /dev/pck?

Comme il a été défini précédemment, il est possible d'échanger des données de contrôle de dimension limitée par les moniteurs de communication inter-processus. Pour envoyer ces données de contrôle on a construit une version particulière de la routine "stty.s" qui fonctionne pour les terminaux. Cette nouvelle routine s'appelle "putcmd.s". De même pour obtenir des données de contrôle, il existe une variante de "gtty.s" qui est nommée "getcmd.s".

A.3.1. Routine "putcmd.s".

```
/result = putcmd(fildes, point)
/point is a pointer to a cmd structure

/result == -1 means error


_putcmd:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     stty
0:
        ..;
```

A.3.2. Routine "getcmd.s".

```
/result = getcmd(fildes, point)
/point is a pointer to a cmd structure

/result == -1 means error


_getcmd:
        mov     r5,-(sp)
```

```
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     gtty
0:
        ..;
```

A.4. Entrées sorties sur le moniteur /dev/pckz.

Le moniteur /dev/pckz se différencie des autres moniteurs de communication inter-processus par le fait qu'il multiplexe et démultiplexe les informations en provenance ou vers ceux-là.

Un paramètre supplémentaire est donc nécessaire pour les entrées sorties sur /dev/pckz: le numéro de l'un des autres moniteurs. Ce paramètre doit être rangé dans le registre R1.

Il faut noter que /dev/pckz ne suspend jamais l'exécution du processus et que les modifications précédentes sont inutiles pour lui.

Tout comme "putcmd.s" et "getcmd.s", "ptcpk.s" et "gtpck.s" sont des variantes de "stty.s" et de "gtty.s" et effectuent le même genre de travail.

A.4.1. Routine "rdpck.s".

```
/result = rdpck(file descriptor, buffer address, trans. size, pck number)

/result == -1  means error


_rdpck:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        mov     10(r5),0f+2
        mov     12(r5),r1
```

```
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     read
0:
        ..;..
```

A.4.2. Routine "wrpck.s".

```
/result = wrpck(file descriptor, buffer address, trans. size, pck number)

/result == -1  means error


_wrpck:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        mov     10(r5),0f+2
        mov     12(r5),r1
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     write
0:
        ..;..
```

A.4.3. Routine "ptpck.s".

```
/result = ptpck(fildes, point, pcknumber)
/point is a pointer to a cmd structure

/result == -1 means error


_ptpck:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        mov     10(r5),r1
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     stty;
0:
        ..;
```

A.4.4. Routine "gtpck.s".

```
/result = gtpck(fildes, point, pcknumber)
/point is a pointer to a cmd structure

/result == -1 means error


_gtpck:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        mov     10(r5),r1
        sys     0;9f
        bec     1f
        jmp     cerror
1:
        mov     (sp)+,r5
        rts     pc

.data
9:
        sys     gtty;
0:
```

..;

A.5. Ouverture du fichier réseau.

Les programmes utilisant le réseau pour faire des entrées sorties sont exécutés comme processus fils des commandes NETWORK ou NETIN qui se chargent d'ouvrir ce réseau à l'avance. Les processus héritent donc des descripteurs de fichiers 3 et 4. Lorsque les programmes font un "open" du réseau, ils ne doivent pas provoquer un appel système mais recevoir les numéros 3 ou 4.

Par convention, le réseau est nommé "/dev/net"; la nouvelle routine "open" teste si c'est ce nom qui est passé comme argument:
- si oui, elle retourne 3 ou 4 suivant le mode d'ouverture.
- si non, elle provoque un appel système "open".

A.5.1. Nouvelle routine "open".

```
/ special library

/ open system call for program using network monitor RTTX25

/ detects special file name "/dev/net"

/ file = open(string, mode)

/ file == -1 means errror


_open :
        mov     r5, -(sp)
        mov     sp, r5
        mov     r2, -(sp)
        mov     4(r5), r2
        mov     $L1, r1
  3:
        movb    (r2), r0
        inc     r2
        cmpb    (r1), r0
        jne     8f
        tstb    (r1)
        jeq     1f
        inc     r1
        jbr     3b
  1:
        mov     $3, r0
        tstb    6(r5)
```

```
        jeq     2f
        mov     $4, r0
2:
        mov     (sp)+, r2
        mov     (sp)+, r5
        rts     pc
8:
        mov     (sp)+, r2
        mov     4(r5), 0f
        mov     6(r5), 0f + 2
        sys     0; 9f
        bec     3f
        jmp     cerror
3:
        mov     (sp)+, r5
        rts     pc

.data
9:
        sys     open; 0: ..; ..
L1:
        .byte 57, 144, 145, 166, 57, 156, 145, 164, 0, 0
```

A.6. Instruction logique.

La routine "exor.s" a été écrite pour effectuer un "ou exclusif" entre deux arguments d'appel en les considérant comme des variables booléennes et non en travaillant au niveau du bit.

La routine "exor.s".

```
_exor :
        mov     r5,-(sp)
        mov     sp,r5
        clr     r0
        tst     2(r5)
        beq     1f
        inc     r0
1 :
        clr     r1
        tst     4(r5)
        beq     2f
        inc     r1
2 :
        xor     r1,r0
        mov     (sp)+,r5
        rts     pc
```

A.7. Echange de données de contrôle avec le moniteur /dev/trapac.

Pour échanger des commandes et des données de contrôle avec le moniteur /dev/trapac des interfaces physiques DR11-C, et par là avec le niveau trame su r microprocesseur, on utilise aussi des variantes des routines "stty.s" et "gtty.s" nommées "pstat.s" et "gstat.s".

A.7.1. La routine "pstat.s".

```
/ C library -- pstat

/ error = pstat(filep, pnt);

/ filep is descriptor of open trapac


_pstat:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        sys     0; 9f
        bec     1f
        jmp     cerror
1:
        clr     r0
        mov     (sp)+,r5
        rts     pc
  .data
9:
        sys     stty; 0:..
```

A.7.2. La routine "gstat.s".

```
/ C library -- gstat

/ error = gstat(filep, pnt);

/ filep is descriptor of open trapac


_gstat:
        mov     r5,-(sp)
        mov     sp,r5
        mov     4(r5),r0
        mov     6(r5),0f
        sys     0; 9f
        bec     1f
        jmp     cerror
```

```
1:
        clr     r0
        mov     (sp)+,r5
        rts     pc
  .data
9:
        sys     gtty; 0:..
```

A.8. Primitives d'entrées sorties sur le réseau.

Des routines ont été écrites pour effectuer les actions de base avec le moniteur réseau. (entrées sorties et contrôle de la communication) Elles contiennent des appels à des routines d'un niveau inférieur, du même type que les précedentes. Elles ont été écrites en langage 'C' et certaines utilisent des structures définies dans le fichier "open.h" dont on trouvera copie dans les fichiers du moniteur réseau.

A.8.1. Recherche d'un moniteur de communication.

Pour établir la première partie d'une communication, il faut d'abord entrer en liaison avec le moniteur réseau et pour ce faire, il faut disposer d'un moniteur de communication inter-processus libre. La routine "srchlin" recherche un moniteur de libre et le réserve. Avant, elle aura vérifié que le moniteur réseau est actif en testant l'existence du fichier "admvl". Le paramètre d'appel est l'adresse d'un tableau de deux entiers qui contiendront les descripteurs de fichiers. Le paramètre de retour sera:

- l'identificateur du moniteur /dev/pck? .
- -1 si le moniteur réseau n'est pas actif.
- -2 si aucun moniteur /dev/pck? n'est libre.

La routine "srchlin".

```
srchlin(fildes)
int     *fildes;
{
        register  char  *name;
        register  int   result;
        register  int   i;
        int     imax;

        name = "admvl";
        i = open(name, 0);
        if (i < 0) return(-1);
        close(i);


        name = "/dev/pck0";
        i = 0;
```

```
        imax = 10;

loop:
        for (; i < imax; i++){
                result = open(name, 0);
                if (result < 0) {
                        name[8] =+ 1;
                } else {
                        fildes[0] = result;
                        result = open(name, 1);
                        if (result < 0) {
                                close(fildes[0]);
                                name[8] =+ 1;
                        } else {
                                fildes[1] = result;
                                return(i);
                        }
                }
        }
        if (imax == 16) return(-2);
        name[8] = 'a';
        i = 10;
        imax = 16;
        goto loop;
}
```

A.8.2. Etablissement de la liaison avec le moniteur réseau.

Deux routines sont utilisées pour établir cette liaison:

- "pcklin" qui assemble les différents paramètres à envoyer au moniteur réseau: identificateur de l'utilisateur, du terminal, du processus.
- "rqline" qui expédie une commande de demande de liaison par le chemin des commandes (putcmd) puis les paramètres par le chemin des données-informations (write).

Les paramètres d'appel sont:

- l'identificateur du moniteur /dev/pck?.
- l'adresse des descripteurs de fichiers.
- le pointeur vers la structure des paramètres ("rqline").

La routine "pcklin".

```
#include "open.h"

#define INT             0       /* flag utilisateur interne */
#define EXT             1       /* flag utilisateur externe */

pcklin(nucl, fildes, status)
int     nucl, status;
```

```
int     *fildes;
{
        register  int  result;
        register  struct  rq  *rqpt;

        struct {
                int     longueur;
                struct  rq  resrq;
        } *pnt;

        rqpt = &(pnt->resrq);

        rqpt->rq_fgs =| (status ? EXT : INT);
        rqpt->rq_pid = getpid();
        rqpt->rq_usid = (getuid() & 017);
        rqpt->rq_ttid = ttyn();
        rqpt->rq_uc = nucl;


        pnt->longueur = RQLNG;

        result = rqline(nucl, fildes, pnt);
        return(result);
}
```

La rutine "rqline".

```
#define BEGIN   14      /* code operatoire d'une demande de ligne */

rqline(nucl, fildes, param)
int     nucl;
int     *fildes, *param;
{

        register  int  result;
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = *param++;
        pnt->cop = (nucl << 4);
        pnt->cop =| BEGIN;

        if (putcmd(fildes[1], pnt) < 0) return(-1);
        if (write(fildes[1], param, pnt->argu) < 0) return(-1);
        return(0);
}
```

A.8.3. Etablissement d'une liaison avec un autre ETTD.

Deux routines sont également nécessaires pour prolonger un liaison jusqu'à un autre ETTD.

- "liaison" qui construit la liste des paramètres nécessaires: adresse de l'interlocuteur, options et facilités si elles existent.
- "netopen" qui envoie une commande de demande de liaison par le chemin des données de contrôle puis les paramètres par le chemin des données-informations.

Les paramètres d'appel sont:

- l'identificateur du moniteur /dev/pck?.
- l'adresse des descripteurs de fichiers.
- l'adresse d'une chaîne de caractères (nom de l'interlocuteur) (liaison).
- le pointeur vers la liste des paramètres (netopen).

La routine "liaison".

```
#include "open.h"

liaison(nucl, fildes, nom)
int     nucl;
int     *fildes;
char    *nom;
{
        register  struct  op  *opnt;
        register  char  *pntchar;
        register  int  i;
        int     result;

        struct {
                int     longueur;
                struct  op  oprq;
        } *pnt;

        opnt = &(pnt->oprq);

        opnt->op_uc = nucl;
        opnt->op_ailn = 4;
        opnt->op_iad[0] = "UNIX";
        pntchar = opnt->op_oad;
        while(*nom != '0') {
                *pntchar++ = *nom++;
                i++;
        }
        opnt->op_aoln = i;
        pnt->longueur = OPLNG;
        result = netopen(nucl, fildes, pnt);
        return(result);
}
```

La routine "netopen".

```
#define CALL    1       /* code operatoire d'une demande de communication */

netopen(nucl, fildes, param)
int     nucl;
int     *fildes, *param;
{

        register  int  result;
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = *param++;
        pnt->cop = (nucl << 4);
        pnt->cop =| CALL;

        if (putcmd(fildes[1], pnt) < 0) return(-1);
        if (write(fildes[1], param, pnt->argu) < 0) return(-1);
        return(0);
}
```

A.8.4. Saisie des données de contrôle.

Les données de contrôle venant du moniteur réseau via les moniteurs /dev/pck? sont difficilement lisibles car chaque bit a un sens particulier. La routine "control" lit les 3 mots venant du moniteur et réarrange les bits pour rendre ces données de contrôle plus lisible. Les paramètres d'appel sont:

- l'identificateur du moniteur /dev/pck?.
- l'adresse des descripteurs de fichiers.
- le pointeur vers une structure où seront rangées ces données de contrôle.

La routine "control".

```
#define NOTRANS         0100000         /*flag de premiere lecture */

struct commande {
        int     cmd_1param;
        int     cmd_2param;
        int     cmd_cop;
        int     cmd_fgs;
        int     cmd_rdcpt;
        int     cmd_wrcpt;
```

```
};

control(nucl, fildes, pointer)
int     nucl;
int     *fildes;
struct  commande  *pointer;
{
        int     param[3];
        register  char  *pnt;

        pnt = pointer;
        if(getcmd(fildes[1], param) < 0) return(-1);

        pnt->cmd_1param = param[0];
        pnt->cmd_2param = (param[1] & 0360);
        pnt->cmd_2param =>> 4;
        pnt->cmd_cop = (param[1] & 017);
        pnt->cmd_fgs = (param[1] & 0177400);
        pnt->cmd_fgs =>> 8;
        pnt->cmd_rdcpt = (param[2] & 0177400);
        pnt->cmd_rdcpt =>> 8;
        pnt->cmd_wrcpt = (param[2] & 0377);

        if(pnt->cmd_fgs & NOTRANS) return(0);
        return(1);
}
```

A.8.5. Commandes disponibles.

Une fois la liaison établie, des commandes peuvent influencer la communication:

- "frline" qui coupe la liaison avec le moniteur réseau et implicitement libère la voie logique.
- "netclose" qui libère la voie logique.
- "netvv" qui vidange tous les buffers dans les deux sens de la communication et implicitement provoque une réinitialisation.
- "netint" qui provoque l'envoi de la donnée urgente (1 octet) passée comme paramètre.
- "netrun" qui réactive une communication lors d'un problème sur la voie logique du réseau.
- "netrep" qui provoque une reprise du niveau paquet du protocole X25; accessible uniquement à un utilisateur privilégié.
- "netpri" qui modifie les priorités dans le moniteur réseau. (voir chap 4.2.3); accessible à un utilisateur privilégié.

Les paramètres d'appel sont:

- l'identificateur du moniteur /dev/pck?.
- l'adresse des descripteurs de fichiers.
- une cause (frline, netclose, netvv, netrep).
- une donnée rapide (netint).

La routine "frline".

```
#define BREAK   15      /*code operatoire de fin de travail */


frline(nucl, fildes, param)
int     nucl, param;
int     *fildes;
{
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = param;
        pnt->cop = (nucl << 4);
        pnt->cop =| BREAK;
        if (putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netclose".

```
#define LIB     2       /*code operatoire d'une fin de communication */


netclose(nucl, fildes, param)
int     nucl, param;
int     *fildes;
{
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = param;
        pnt->cop = (nucl << 4);
        pnt->cop =| LIB;
        if (putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netvv".

```
#define VIDVIT  3       /* code operatoire d'un nettoyage de la ligne */


netvv(nucl, fildes, param)
int     nucl, param;
int     *fildes;
{
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = param;
        pnt->cop = (nucl << 4);
        pnt->cop =| VIDVIT;
        if (putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netint".

```
#define INT     4       /*code operatoire d'une interruption pour donnee urgent


netint(nucl, fildes, param)
int     nucl, param;
int     *fildes;
{
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = param;
        pnt->cop = (nucl << 4);
        pnt->cop =| INT;
        if (putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netrun".

```
#define RUN     16      /* code operatopn de RUN */
netrun(nucl, fildes)
int     nucl;
```

```
int     *fildes;
{
        struct {
                int     param;
                char    cop;
        } *pnt;

        pnt->cop = (nucl << 4);
        pnt->cop =| RUN;
        if(putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netrep".

```
#define REP     7       /* code operatoire d'une reprise */

netrep(nucl, fildes, param)
int     nucl, param;
int     *fildes;
{
        struct {
                int     argu;
                char    cop;
        } *pnt;

        pnt->argu = param;
        pnt->cop = (nucl << 4);
        pnt->cop =| REP;
        if (putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

La routine "netpri".

```
#define SETPRI          13      /* code operatoire pour une modification de la

netpri(nucl, fildes, param1, param2)
int     nucl, param1, param2;
int     *fildes;
{
        struct {
                char    argu1;
                char    argu2;
                char    cop;
        } *pnt;
```

```
        pnt->argu1 = param1;
        pnt->argu2 = param2;
        pnt->cop = (nucl << 4);
        pnt->cop =| SETPRI;
        if(putcmd(fildes[1], pnt) < 0) return(-1);
        return(0);
}
```

Annexe B: Le mode d'emploi du moniteur réseau.

B.1. Chargement et exécution du moniteur réseau.

B.1.1. Rappel.

Comme décrit dans le chapître 4, le moniteur réseau travaille en deux passes:

- une phase d'initialisation.
- une phase d'exécution proprement dite.

B.1.2. Chargement et exécution de la phase d'initialisation.

Le code objet se trouve sur le fichier INITX25. Le programme doit être lancé comme une commande SHELL par un utilisateur privilégié (root ou su).

```
% initx25
```

En s'exécutant le programme va afficher:

- les numéros des descripteurs des fichiers et des fichiers spéciaux.
- les valeurs par défaut des variables et des paramètres des voies logiques. Il demande alors si l'utilisateur veut modifier certaines valeurs.
  - si oui, il est demandé de préciser les voies logiques qui doivent être ainsi particularisées et de quelle façon. (énumération des variables pouvant être modifiées).
  - si non, l'exécution du programme continue.
- le numéro du processus qui va exécuter le programme RTTX25 moniteur réseau.

Dès cet instant, l'utilisateur récupère l'interprèteur de commandes; le processus continue en "stand alone" et va encore afficher:

- un signal lors de la création d'un processus qui va exécuter le programme HORLOGE, élément du temporisateur.
- un signal lors de la création d'un processus qui va exécuter le programme NETINIT responsable des processus des utilisateurs externes.
- un résumé des actions entreprises et l'organisation des informations sur le fichier "init.h".
- un message signalant qu'on va passer à la deuxième phase.

Le programme INITX25 demande lui même au système d'exploitation le chargement et l'exécution du programme RTTX25.

En cas d'erreurs, un message est affiché précisant en clair le type et l'origine de l'erreur. Le programme est alors stoppé et le processus meurt.

B.1.3. Exécution des programmes HORLOGE et NETINIT.

Ces deux programmes sont exécutés par des processus créés par INITX25. Dès le début, HORLOGE affiche le numéro du processus père vers lequel il va envoyer des interruptions software. Ce numéro doit être le même que celui signalé précédemment.

Le programme NETINIT affiche:

- un message signalant le nombre de processus qu'il va créer et qui seront mis à la disposition des utilisateurs externes.
- les numéros de ces processus.

Dans la suite, NETINIT affichera chaque numéro de processus qu'il aura dû créer pour remplacer un décédé.

Si les processus exécutant HORLOGE ou NETINIT venaient à mourir, il est possible à l'utilisateur de les remplacer en donnant la commande:

- % horloge <numéro du processus RTTX25> &
- % netinit &

Le numéro du processus exécutant RTTX25 a été affiché dès le début. Il est cependant conseillé de réinitialiser au préalable le niveau paquet c'est à dire d'effectuer une reprise (Voir commande plus loin).

B.1.4. Contrôle de RTTX25.

Dès l'instant où l'interprèteur des commandes SHELL est redevenu actif, le moniteur réseau continue tout seul; il n'effectue plus aucune saisie de caractères au terminal. Pour le contrôler, il faut se relier au moniteur réseau de la même manière qu'un utilisateur mais à l'aide du moniteur de communication inter-processus /dev/pckB. Celui-ci transforme son utilisateur en utilisateur privilégié; il permet de demander au moniteur réseau d'effectuer des actions spéciales.

B.2. Commande utilisateur.

B.2.1. Lancement de la commande.

Pour se servir du réseau, il faut d'abord lancer l'exécution du programme NETWORK en tapant:

% network

Cela provoque le remplacement de l'interprèteur des commandes SHELL par un autre interprèteur. Dès cet instant, tout ce qui est entré au terminal est analysé suivant des critères différents.

- Toute chaîne de caractères doit être précédée d'un caractère spécial indiquant sa destination:

  - "$" : ce qui suit est une commande pour le moniteur réseau.$
  - "&" : ce qui suit doit être envoyé sur la ligne vers l'autre machine.
  - "%" : ce qui suit doit être interprèté comme une commande par l'ancien interprèteur SHELL.
  - "c" : signale à l'interprèteur qu'il doit poursuivre l'affichage des caractères venant du réseau, affichage qui était interrompu.

- Les commandes qui sont destinées au moniteur réseau sont:

  - "$ line" : qui demande une liaison avec le moniteur réseau. Dès cet instant, les descripteurs de fichiers 3 et 4 sont réservés aux entrées sorties avec le moniteur réseau.
  - "$ netopen <nom de l'interlocuteur> " : qui demande une liaison entre le moniteur réseau et un autre ETTD spécifié par <nom de l'interlocuteur>.
  - "$ netclose <cause> " : demande de rompre la liaison avec l'autre ETTD; une cause de rupture peut être précisée par un entier inférieur à 255.
  - "$ vidvit <cause> " : demande le nettoyage des buffers; une cause peut également être spécifiée.
  - "$ interrupt <argument> " : demande l'envoi en urgence de l'argument qui est un entier inférieur à 255.
  - "$ break <cause> " : demande la rupture de la liaison avec le moniteur réseau et par voie de conséquence avec l'autre ETTD; une cause peut être spécifiée.
  - "$ run " : demande la poursuite de la communication lorsqu un problème est signalé.

  Valables uniquement pour un super utilisateur:

  - "$ reprise <cause> " : demande une reprise du niveau paquet. Une cause peut être spécifiée.
  - "$ setpri <élément> <nouvelle priorité> " : demande de modifier la priorité (0 < priorité < 29) de l'élément.
  - "$ chuclid " : demande de changer d'utilisateur courant, c'est à dire qu'on travaille à la place d'un autre utilisateur. Ce qui permet de l'obliger à libérer la ligne par exemple.

- L'interprèteur réceptionne aussi les données envoyées par l'autre ETTD et que le moniteur réseau lui transmet. Il les affiche à l'écran. De même il traite les interruptions provoquées par le moniteur réseau lorsque celui-ci veut signaler l'arrivée de données de contrôle; l'interprèteur lit ces données de contrôle, les affiche en clair à l'écran et bloque la liaison avec le moniteur réseau. Celle-ci pourra être débloquée par la commande "$ run ".

- Plusieurs commandes doivent encore être écrites pour faciliter le travail de contrôle:
  - affichage en clair du fichier "admvl" qui contient les renseignements sur les utilisateurs.
  - déclenchement simultané d'une reprise du niveau paquet et d'un initialisation du niveau trame.
  - nettoyage complet lors de l'arrêt du moniteur réseau.
  - etc...

B.2.2. Exécution de programmes ou de commandes.

Toutes les commandes qui effectuent des entrées sorties sur le réseau doivent être compilées ou assemblées avec les routines de la librairie "lamb" décrite en annexe A.

Le réseau doit être référencé par "/dev/net" à l'exclusion de tout autre nom. Il est strictement interdit d'utiliser les appels systèmes "stty" et "gtty" en direction du réseau.

B.3. Exemples concrets.

Soit à effectuer le transfert vers une autre machine d'un fichier pour l'y faire compiler. On suppose que l'autre machine est identique en tout point à celle-ci.

```
network
$ line
$ netopen WQR
                /* commentaire: lancer sur l'autre ETTD une commande
                   pour réceptionner ce qui vient du réseau.
                 */
& cp /dev/net test.c
                /* commentaire: lancer l'expédition du fichier essai.c
                   sur le réseau.
                 */
% cp essai.c /dev/net
                /* commentaire: stopper le programme de réception. */
$ interrupt 4
                /* commentaire: demander la compilation. */
& cc test.c
                /* commentaire: arrêter l'interprèteur de commande
                   de l'autre ETTD.
                 */
$ interrupt 0
$ netclose 10
$ break 10
                /* commentaire: revenir à l'interprèteur SHELL */
effectuer ctrl d
```

Annexe C: Le moniteur pour interface DR11-C.

```
1 #
2 /*
3    This driver shares the DR11-C interface with driver mxpar.c and dr.c:
4    only one driver at a time may be opened!
5    All opening and closing must go through routines clopen and clclose.
6  */

7 /* This driver for use with up to two DR11-C interfaces connecting UNIX to
8    another processor; the communication is bi-directional; it consists of blocks
9    of varying length (max. 128 words of 16 bytes). The main purpose of this
10   driver is to be used in a X25 connection, whereby UNIX handles the "packet"
11   level and the other processor handles the "frame" level. The present driver
12   procides the interface between the two levels.
13 */

14 /*
15 La fonction de TRAPAC est d'assurer le transfert de blocs d'information
16 (des paquets) entre les niveaux trame et paquet.
17 Il est conçu pour :
18 - respecter le protocole de transmission défini par J. ART dans le rapport TP
19 - accélérer la transmission en n'obligeant pas les deux niveaux à se
20   synchroniser.
21 - ne pas imposer des arrêts intempestifs("sleep") aux programmes du niveau
22   paquet.
23 - éviter que des paquets ne soient à cheval sur les trois étapes
24   niveau trame--trapac--niveau paquet.
25 - pour repérer la fin des paquets afin d'y insérer ou d'en retirer
26   les mots d'état de la transmission.
27 */



28 #include "/binar/usr/sys/param.h"
29 #include "/binar/usr/sys/user.h"
30 #include "/binar/usr/sys/conf.h"
31 #include "/binar/usr/sys/drmxp.h"

32 /* flags de controle */

33 #define ASKSTAT 010             /* code pour demande d'etat par niveau trame */
34 #define INOPEN  0100            /* semaphore pour ouverture en mode lecture */
35 #define OUTOPEN 040             /* semaphore pour ouverture en mode ecriture */
36 #define STOPOU  0200            /* flag indiquant l'arret de l'expedition */
37 #define STOPIN  0400            /* flag indiquant l'arret de la reception */
38 #define EMPTY   01000           /* buffer de sortie vide */
39 #define NOPLACE 02000           /* pas assez de places libres dans le niveau trame
40 #define FULL    040000          /* buffer d'entree surcharge */

41 #define NTRPC   2               /* nombre d'interfaces prevues */
42 #define BUFMAX  128             /* taille des buffers en mots */
43 #define LNGMAX  BUFMAX * 2      /* taille des buffers en bytes */

44 #define TRAPA1  0767760         /* adresse des registres du premier interface */
```

```
45 #define TRAPA2  0767750         /* adresse des registres du second interface */

46 #define BITFAI  0377            /* validation des bits de poids faible */
47 #define CFREE   077000          /* validation des bits du nouveau compteur */
48 #define BITPAQ  0100000         /* indicatif d'un début de paquet */
49 #define FACTEUR 6               /* facteur multiplicatif du compteur */

50 int     tpcia(), tpcib();      /* routine de traitement des interruptions */

51 struct inter inter[NDR11];
52 struct trpc {
53     int   tp_stflag;     /* flags de contrôle voir ci-dessus */
54     int   tp_cmdo;       /* état de la ligne sens paquet->trame */
55     int   tp_cmdi;       /* état de la ligne sens trame->paquet */
56     int   tp_cin1;       /* compteur de remplissage du buffer d'entrée */
57     int   tp_cin2;       /* compteur pour repérage de fin de paquet (voir tpcib()) */
58     int   tp_cou1;       /* compteur de remplissage du buffer de sortie */
59     int   tp_cou2;       /* compteur pour repérage de fin de paquet (voir tpcia()) */
60     int   tp_cfree;      /* compteur indiquant la place libre dans le buffer du microp
61     int   tp_bfou[BUFMAX];   /* buffer de sortie circulaire */
62     int   tp_bfin[BUFMAX];   /* buffer d'entrée circulaire */
63     int   tp_iou;        /* indice première place libre dans buffer de sortie */
64     int   tp_jou;        /* indice dernière place occupée dans le même buffer */
65     int   tp_iin;        /* indice première place libre dans buffer d'entrée */
66     int   tp_jin;        /* indice dernière place occupée dans ce même buffer */
67     int   tp_usid;       /* identificateur de l'utilisateur */
68     int   *tp_drxad;
69 } trpc[NDR11];

70 tpcopen(dev, flag)
71 int     dev, flag;
72 {

73         register  struct  trpc  *pnttr;
74         register  int  *trapa;

75                 /* selection d'un des interfaces physiques DR11-C */

76         if(dev.d_minor >= NDR11) goto erreur;
77         pnttr = &trpc[dev.d_minor];

78                 /* un seul et meme utilisateur peut ouvrir plusieurs fois */

79         if (pnttr->tp_usid) {
80                 if (pnttr->tp_usid != u.u_uid) goto erreur;
81         } else {
82                 pnttr->tp_usid = u.u_uid;
83         }


84                 /* demande de l'interface physique; 'clopen' le réservera */
85                 /* désormais à ce moniteur. */

86         if(clopen(&pnttr->tp_drxad, &tpcia, &tpcib, dev.d_minor, flag) < 0) {
87                 pnttr->tp_usid = 0;
88                 goto erreur;
89         }
```

```
90                  /* en fonction du mode d'ouverture, placer les flags */
91                  /* nécessaires. OUTOPEN et INOPEN sont les flags principaux */
92                  /* dépendant du mode d'ouverture. */

93          if(flag){
94                  if (pnttr->tp_stflag & OUTOPEN) {
95                          return;
96                  }
97                  pnttr->tp_stflag =| (EMPTY | NOPLACE | STOPOU | OUTOPEN);
98          } else {
99                  if (pnttr->tp_stflag & INOPEN) {
100                         return;
101                 }
102                 pnttr->tp_stflag =| INOPEN;
103         }
104
105                 /* Mise a zéro du mot de commande à envoyer. */

106         pnttr->tp_cmdo = 0;
107         return;

108 erreur:
109         u.u_error = ENXIO;
110         spl0();
111         return;
112 }


113 tpcclose(dev, flag)
114 int     dev, flag;
115 {

116         register  struct  trpc  *pnttr;
117         register  int  *trapa;

118         if (dev.d_minor >= NDR11) {
119                 u.u_error = ENXIO;
120                 return;
121         }

122         pnttr = &trpc[dev.d_minor];

123                 /* Liberer l'interface physique. */

124         clclose(&pnttr->tp_drxad, dev.d_minor);

125                 /* Remettre tout dans l'état initial : compteurs, indices */
126                 /* et mot de commande. */

127         pnttr->tp_stflag = 0;
128         pnttr->tp_cfree = 0;
129         pnttr->tp_usid = 0;
130         pnttr->tp_cmdo = pnttr->tp_cmdi = 0;
131         pnttr->tp_cin1 = pnttr->tp_cin2 = 0;
132         pnttr->tp_iin = pnttr->tp_jin = 0;
133         pnttr->tp_cou1 = pnttr->tp_cou2 = 0;
134         pnttr->tp_iou = pnttr->tp_jou = 0;
```

```
135         return;
136 }


137 /*
138 tpcread() est conçu pour :
139 - ne renvoyer vers le niveau paquet (c.a.d. le programme qui a fait un tpcread)
140   qu'un paquet qui est déjà entièrement arrive dans TRAPAC.
141 - ne libérer la place dans 'bfin' qu'une fois le paquet complètement envoyer.
142 - ne renvoyer qu'un seul paquet par appel système "read" effectue dans le niveau
143   paquet.
144 - réactiver la transmission depuis le niveau trame si celle-ci a été bloquée
145   suite à une surcharge des buffers.
146 - prévenir le niveau trame qu'une plus grande place est disponible dans 'bfin';
147   et au besoin, réactiver la routine de transmission pour envoyer un nouveau
148   mot d'etat.
149 */


150 tpcread(dev)
151 int     dev;
152 {
153         register   struct  trpc  *pnttr;
154         register  int  *pnt;
155         register  int  cptrd1;
156         int     cptrd2;
157         int     *trapa;
158         int     savind;
159
160         if (dev.d_minor >= NDR11){
161                 u.u_error = ENXIO;
162                 return;
163         }

164         pnttr = &trpc[dev.d_minor];
165         trapa = pnttr->tp_drxad;

166         if (pnttr->tp_cin1 <= 0) return;     /* TRAPAC n'a lu aucun caractere */

167                 /* Pointer le premier élément à donner au processus utilisateur. */

168         pnt = &pnttr->tp_bfin[pnttr->tp_jin];

169                 /* Sauver l'indice de cet élément pour le cas ou il y */
170                 /* aurait des problèmes. */

171         savind = pnttr->tp_jin;

172                 /* initialiser le compteur de transmission. */
173                 /* arrondir au nombre pair de bytes immédiatement supérieur. */

174         cptrd1 = *pnt & 0377;
175         ++cptrd1;
176         cptrd1 =>> 1;
177         if(cptrd1 > pnttr->tp_cin1) return;
178         cptrd2 = cptrd1;
```

```
179                 /* transfert */

180         while (givec(pnttr) == 0 && cptrd1-- > 0);

181                 /* tout doit être envoyé sinon il y a erreur. */

182         if(cptrd1 > 0) {
183                 u.u_error = EFBIG;
184                 pnttr->tp_jin = savind;
185                 return;
186         }

187                 /* Monter la priorité car on travaille maintenant sur */
188                 /* les compteurs et sur les interruptions.  */

189         spl5();
190         pnttr->tp_cin1 =- ++cptrd2;
191         if (pnttr->tp_stflag & STOPIN) {
192                 pnttr->tp_stflag =&  STOPIN;
193                 trapa->drcsr =| SIGB;   /* signaler a l'emetteur qu'il peut emettre *
194         }
195         if (pnttr->tp_stflag & STOPOU){
196                 pnttr->tp_stflag =| ASKSTAT;
197                 tpcib(dev.d_minor);
198         }
199         spl0();
200         return;
201 }

202 /*
203 tpcwrite est conçu pour :
204 - n'accepter un paquet du programme activant tpcwrite que s'il a assez de place
205   dans 'bfou' pour le recevoir entièrement.
206 - refuser le paquet s'il ne lui est pas parvenu complètement.
207 - accepter plusieurs paquets pour un même appel système "write".
208 - réactiver la routine de transmission si celle-ci a été suspendue suite à un
209   manque de données à envoyer.
210 */



211 tpcwrite(dev)
212 int     dev;
213 {
214         register  struct  trpc  *pnttr;
215         register  int  *pnt;
216         register  int  cptw1;
217         int  result, cptw2;
218         int  trapa;
219         int     savind;

220         if (dev.d_minor >= NDR11) {
221                 u.u_error = EIO;
222                 return;
223         }

224         pnttr = &trpc[dev.d_minor];
```

```
225         trapa = pnttr->tp_drxad;

226                 /* Le nombre de bytes à envoyer doit être pair car */
227                 /* il sera transformé en nombre de mots. */

228         if(u.u_count & 01){
229                 u.u_error = ENXIO;
230                 return;
231         }
232                 /* Recevoir de l'utilisateur tous les paquets possibles */
233                 /* jusqu'à saturation des buffers. */

234 loop:

235                 /* sauver l'état actuel pour le restituer en cas de problèmes */

236         savind = pnttr->tp_iou;
237         pnt = &pnttr->tp_bfou[savind];
238         if (takec(pnttr) < 0) goto retour;
239         cptw1 = (*pnt & 0377) ;
240         cptw1++;
241         cptw1 =>> 1;
242         cptw2 = cptw1;
243         if(++cptw2 > (BUFMAX - pnttr->tp_cou1)){

244                 /* pas assez de place, remettre en situation initiale */

245                 pnttr->tp_iou = savind;

246                 /* Les deux derniers caractères sont considérés comme */
247                 /* non lus. */

248                 u.u_count =+ 2;
249                 goto retour;
250         }
251         while( cptw1-- > 0 && (result = takec(pnttr)) >= 0 );

252                 /* Si le compteur n'est pas négatif l'arrêt est */
253                 /* provoqué par "takec" . */

254         if (cptw1 >= 0) {
255                 u.u_error = ENXIO;
256                 pnttr->tp_iou = savind;
257                 return;
258         }

259                 /* Sous une haute priorité, on modifie le compteur. */

260         spl5();
261         pnttr->tp_cou1 =+ cptw2;
262         spl0();

263                 /* Si le resultat du "takec" est négatif, il n'y a */
264                 /* plus rien à lire. */

265         if(result < 0) goto retour;
266         goto loop;
```

```
267 retour:
268         spl5();

269                 /* Le buffer n'est plus vide maintenant ! */

270         pnttr->tp_stflag =&  EMPTY;
271         if(pnttr->tp_stflag & STOPOU){
272                 pnttr->tp_stflag =&  (STOPOU | EMPTY);
273                 tpcib(dev.d_minor);
274         }
275         spl0();
276 }



277 /*
278 tpcib() est conçu pour :
279 -traiter les interruptions dans l'émetteur survenant pour signaler un "prêt a
280   recevoir" de la part du récepteur (niveau trame).
281 - repérer la fin d'un paquet.
282 - envoyer le mot d'état si c'est nécessaire.
283 - se suspendre si le buffer de sortie est vide.
284 - estimer la place restante dans le niveau trame et au besoin s'arrêter
285 - construire les deux premiers octets d'un paquet conformément au protocole TP.
286 */


287 tpcib(dev)
288 int     dev;
289 {
290                 /* Pointeur vers les registres du DR11-C. */
291         register  int  *trapa;
292                 /* pointeur vers la structure de "trapac". */
293         register  struct  trpc  *pnttr;
294         register  int  *pnt;

295         pnttr = &trpc[dev];
296         trapa = pnttr->tp_drxad;

297         trapa->drcsr =&  SIGA;          /* le mot précédant a été reçu */

298                 /* C'est le début d'un nouveau paquet. */

299    if(pnttr->tp_cou2 <= 0){

300       /* Que reste-t-il comme place dans le buffer ? */

301       tpcfree(pnttr);

302       /* Demande-t-on l'envoi du status? */

303       if(pnttr->tp_stflag & ASKSTAT){
304
305       /* Construire la nouvelle commande. */

306          trapa->drobuf = pnttr->tp_cmdo;
307          pnttr->tp_stflag =&  ASKSTAT;
```

```
308       } else {

309       /* Il n'y a plus rien à envoyer ou plus de place pour */
310       /* recevoir. */

311          if(pnttr->tp_stflag & (EMPTY + NOPLACE)){
312             pnttr->tp_stflag =| STOPOU;
313             return;
314          }
315          if(pnttr->tp_cou1 <= 0){
316             trapa->drobuf = pnttr->tp_cmdo;
317             pnttr->tp_stflag =| EMPTY;
318          } else {

319       /* Construire l'entête du paquet. */

320             pnt = &pnttr->tp_bfou[pnttr->tp_jou];
321             pnttr->tp_cou2 = *pnt & 0377;
322             if((pnttr->tp_cou2 + 2) > pnttr->tp_cfree){
323                trapa->drobuf = pnttr->tp_cmdo;
324                pnttr->tp_cou2 = 0;
325                pnttr->tp_stflag =| NOPLACE;
326             } else {
327                pnttr->tp_cfree =- (pnttr->tp_cou2 + 2);
328                trapa->drobuf = pnttr->tp_cou2++ + ((pnttr->tp_cmdo) & CFREE) + BITPA
329                pnttr->tp_cou2 =>> 1;
330                pnttr->tp_cou1--;
331                pnttr->tp_jou++;
332                pnttr->tp_jou =& (BUFMAX - 1);
333             }
334          }
335       }
336    } else {

337       /* On a envoyé un mot, décrémenter les compteurs et */
338       /* modifier les indices. */

339       pnt = &pnttr->tp_bfou[pnttr->tp_jou];
340       trapa->drobuf = *pnt;
341       pnttr->tp_cou2--;
342       pnttr->tp_cou1--;
343       pnttr->tp_jou++;
344       pnttr->tp_jou =& (BUFMAX - 1);
345    }

346       trapa->drcsr =| SIGA;   /* signaler au récepteur qu'un nouveau mot est prêt

347 }
348
349 /*
350 tpcia() est conçu pour :
351 - traiter des interruptions dans le récepteur causées par l'arrivée d'un mot
352   dans le buffer d'entrée.
353 - repérer la fin d'un paquet.
354 - différencier un mot d'état d'une en-tête de paquet.
355 - mettre a jour l'estimateur de places libres dans le niveau trame.
356 - répondre à une demande de mise à jour du mot d'état de la transmission et
```

```
357   au besoin réactiver la routine de transmission.
358 - bloquer l'arrivée du paquet si le buffer déborde.
359 */



360 tpcia(dev)
361 int     dev;
362 {

363         register  int  *trapa;
364         register  struct  trpc  *pnttr;
365         register  int  *pnt;

366         pnttr = &trpc[dev];
367         trapa = pnttr->tp_drxad;

368         trapa->drcsr =&  SIGB;
369         pnt = &pnttr->tp_bfin[pnttr->tp_iin];
370
371                 /* Début d'un nouveau paquet. */

372         if(pnttr->tp_cin2 <= 0){
373                 pnttr->tp_cin2 = trapa->dribuf;

374                 /* Mettre a jour le compteur estimant la place libre */
375                 /* sur le microprocesseur. */

376                 pnttr->tp_cfree = pnttr->tp_cin2 & CFREE;
377                 pnttr->tp_cfree =>> FACTEUR;
378                 pnttr->tp_stflag =&  NOPLACE;
379                 if(pnttr->tp_cin2 < 0){
380                         pnttr->tp_cin2 =& BITFAI;
381                         *pnt = pnttr->tp_cin2++;
382                         pnttr->tp_cin2 =>> 1;
383                 } else {

384                 /* Une commande -> traitement special. */

385                         pnttr->tp_cin2 = 0;
386                         pnttr->tp_cmdi = trapa->dribuf;
387                         if(pnttr->tp_cmdi & ASKSTAT) pnttr->tp_stflag =| ASKSTAT;
388                         goto activ;
389                 }
390         } else {
391                 pnttr->tp_cin2--;
392                 *pnt = trapa->dribuf;
393         }

394                 /* mise a jour des compteurs et indices. */

395         pnttr->tp_iin++;
396         pnttr->tp_iin =& (BUFMAX - 1);
397         pnttr->tp_cin1++;
398         if(pnttr->tp_cin1 >= BUFMAX){
399                 pnttr->tp_stflag =| STOPIN;
400                 return;
```

```
401         }

402                 /* accuser réception. */

403         trapa->drcsr =| SIGB;
404         return;


405                 /* Réactiver l'émission des paquets. Les conditions ont */
406                 /* évolués */

407   activ :
408         if (pnttr->tp_stflag & STOPOU){
409                 pnttr->tp_stflag =&  STOPOU;
410                 tpcib(dev);
411         }
412         trapa->drcsr =| SIGB;
413         return;
414 }
415 /*
416 Les routines de passage (passc() et cpass()) fonctionnent au niveau des
417 caractères, or TRAPAC fonctionne au niveau du mot. takec() et givec() ont
418 pour mission d'effectuer la transition entre ces deux modes de travail.
419 */


420 takec(pointer)
421 int     *pointer;
422 {
423         register  struct  trpc  *pnttr;
424         register  int  c;
425         register  int  dc;

426         pnttr = pointer;

427         if((c = cpass()) == -1) return(-1);
428         dc = (c & 0377);
429         if((c = cpass()) == -1) return(-1);
430         c =<< 8;
431         dc =| (c & 0177400);
432         pnttr->tp_bfou[pnttr->tp_iou++] = dc;
433         pnttr->tp_iou =& (BUFMAX - 1);
434         return(0);
435 }


436 givec(pointer)
437 int     *pointer;
438 {
439         register  struct  trpc  *pnttr;
440         register  int  c;
441         register  int  dc;

442         pnttr = pointer;

443         dc = pnttr->tp_bfin[pnttr->tp_jin++];
444         pnttr->tp_jin =& (BUFMAX - 1);
```

```
445         c = (dc & 0377);
446         if(passc(c) < 0) return(-1);
447         dc =>> 8;
448         passc(dc);
449         return(0);
450 }


451 /*
452 tpcfree() sert a donner le nombre de places libres suivant le code mis au point
453 dans le protocole TP.
454 */


455 tpcfree(pointer)
456 int     *pointer;
457 {
458         register  struct  trpc  *pnttr;
459         register  int  reste;

460         pnttr = pointer;

461         pnttr->tp_cmdo =& BITFAI;

462         reste = LNGMAX - pnttr->tp_cin1;
463         reste =<< FACTEUR;
464         pnttr->tp_cmdo =| (reste & CFREE);
465 }



466 /*
467 tpcstat() permet l'échange des mots d'état entre TRAPAC et le niveau paquet.
468 Il est activé par le niveau paquet pour fournir un nouveau mot d'état par
469 l'appel système stty et pour obtenir le mot d'état par gtty.
470 */


471 tpcstat(dev,pointer)
472 int     dev;
473 int     *pointer;
474 {
475         register  struct  trpc  *pnttr;
476         register  int  *pnt;
477
478         pnt = pointer;
479         pnttr = &trpc[dev.d_minor];

480         if(pnt == 0){
481                 pnttr->tp_cmdo = u.u_arg[0] & 077777;
482                 pnttr->tp_stflag =| ASKSTAT;
483                 spl5();
484                 if(pnttr->tp_stflag & (STOPOU)){
485                         pnttr->tp_stflag =&  (EMPTY | NOPLACE | STOPOU);
486                         tpcib(dev.d_minor);
487                 }
488                 spl0();
```

```
489         } else {
490                 *pnt++ = pnttr->tp_cmdi;
491                 *pnt++ = pnttr->tp_cin1;
492                 *pnt = pnttr->tp_cou1;
493                 pnttr->tp_cmdi = 0;
494         }
495 }
```

Annexe D: Le moniteur de communication inter-processus

```
1 #
2 /*
3    Driver for inter-process communication via named channels
4 */

5 /*
6   This driver allows processes to communicate via /dev/pck? pseudo-devices.
7   Synchronization between readers and writers occurs by means of sleep/
8   wakeup mechanism. However, for one of the /dev/pck?s (with minor number
9   equal to NOSLEEP), the writer or reader never sleeps:  a writer terminates
10  when no more characters may be written (due to buffer congestion), and the
11  write operation returns the number of characters effectively written;
12  similarly, a read operation returns when there are no more characters to
13  be read at that moment, the return value is the number of characters
14  effectively read.

15  Also, a reader/writer on this peculiar channel is able to communicate
16  with writers and readers which operate on any other channel:  all the
17  output produced by the writers is read by this special reader, and
18  contains information about its origin;
19  the output produced by this special writer contains routing information which
20  allows to send that output to designated readers.
21  The first byte on a read operation on this special channel is the minor number
22  of the source device; the first byte to be written on the special channel must
23  be the minor number of the destination channel.

24  It should therefore be obvious that one should have:
25        - either pairs of readers and writers operating on common channels,
26        - or readers and writers communicating exclusively with a single
27          reader/writer which operates on the special channel.

28  Note: even though the channels are bi-directional, both streams are
29        assumed, if they are used, to flow between a single pair of
30        processes.

31 */

32 /* This driver uses routines pckgetc and pckputc
33    which are defined in file pcks.s
34 */


35 #include "/binar/usr/sys/param.h"
36 #include "/binar/usr/sys/conf.h"
37 #include "/binar/usr/sys/user.h"
38 #include "/binar/usr/sys/proc.h"
39 #include "/binar/usr/sys/seg.h"
40 #include "/binar/usr/sys/reg.h"
41 #include "/binar/usr/sys/file.h"

42 #define NPCK     16      /* Nombre de moniteurs normaux */
43 #define NOSLEEP NPCK     /* Numéro du moniteur spécial */
44 #define PCKPRI  8        /* priorité de réveil */
```

```
45 #define NPCKBLK 50      /* Nombre de blocs chaines */
46 #define PCKHI   130     /* Borne supérieure pour stopper input */
47 #define PCKLO   34      /* Borne inférieure pour réactiver input */

48 #define CHAR    0377    /* Validation des 8 bits faibles */
49 #define STAT    077400  /* Validation des 8 bits forts */

50 #define ENDFIL  0       /* Caractère fin de fichier */

51 /* flags */

52 #define ORAB    0400    /* open to read on AB */
53 #define OWAB    01000   /* open to write on AB */
54 #define ORBA    02000   /* open to read on BA */
55 #define OWBA    04000   /* open to write on BA */

56 #define WRITINGAB       010000  /* writingon AB */
57 #define WRITINGBA       020000  /* writing on BA */


58 #define SRAB    01      /* reader AB asleep on sleep channel &pck_prAB */
59 #define SWAB    02      /* writer AB asleep on sleep channel &pck_qAB  */
60 #define SRBA    04      /* reader BA asleep on sleep channel &pck_prBA */
61 #define SWBA    010     /* writer BA asleep on sleep channel &pck_qBA  */

62 #define NOTRANS         0100000         /* flag de no transmission de status */

63 /* pckflag values */
64 #define OPEN    01      /* normal open */
65 #define OPENSPEC  03777         /* special pck open */

66 struct cblock {
67         struct cblock *c_next;  /* Pointeur vers bloc suivant */
68         char info[6];           /* buffer information */
69 };

70 struct cblock pckfr[NPCKBLK];
71 struct cblock *pckfrlist;

72 struct clist {
73         int     c_cc;           /* character count */
74         int     c_cf;           /* pointer to first block */
75         int     c_cl;           /* pointer to last block */
76 };

77 /* There are two independent channels for every minor "device",
78    hence two queues:     AB and BA
79 */

80 struct pck {
81         int pck_fgs;    /* Flags, voir plus haut */
82         int pck_prAB;   /* pid of reader on AB and writer on BA */
83         int pck_prBA;   /* pid of reader on BA and writer on AB */
84         int pck_frAB;   /* pointer to file structure read on AB */
85         int pck_fwAB;   /* pointer to file structure write on AB */
86         int pck_frBA;   /* pointer to file structure read on BA */
87         int pck_fwBA;   /* pointer to file structure write on BA */
```

```
88          int pck_stAB[2];        /* status de A -> B */
89          int pck_stBA[2];        /* status de B -> A */
90          struct clist pck_qAB;
91          struct clist pck_qBA;
92  } pck[NPCK];

93  int pckflag 0;  /* zero until first call to pckopen */

94  /* A "device" may be opened either for input or for output,
95     but not for both!
96  */

97  pckopen(dev,flag) {

98          int vl, iof;
99          register  int  pid;
100         register  int  ptf;
101         struct proc *pp;register  struct pck *tp;

102         if(!pckflag) { pckflag++; pckinit(); }
103                 /* Reperer le moniteur */
104         vl = dev.d_minor;
105                 /* Tester si ce moniteur existe et si le mode d'ouverture
106                    n'est pas lecture-ecriture. */
107         if(vl < 0 || vl > NPCK || u.u_arg[1] == 3) {
108                 u.u_error = ENXIO;
109                 return;
110         }

111                 /* Reperer le numero de descripteur de fichier */
112         iof = u.u_ar0[R0];
113                 /* Reperer le pointeur dans la table des fichiers ouverts */
114         ptf = u.u_ofile[iof];

115                 /* Reperer le pointeur vers le descripteur de processus */
116         pp = u.u_procp;
117                 /* Reperer le numero de l'utilisateur */
118         pid = pp->p_pid;

119         /* handle all cases for actual opening */
120         if(vl != NOSLEEP){

121                 /* Affecter l'adresse de la structure de ce moniteur */
122                 tp = &pck[vl];

123                 switch(tp->pck_fgs & 07400){            /* what is open ? */
124         case 0  :        tp->pck_prAB = pid;            /* nothing */
125                         if(flag) {
126                                 tp->pck_fwBA = ptf;
127                                 tp->pck_fgs =| OWBA;
128                         }else{
129                                 tp->pck_frAB = ptf;
130                                 tp->pck_fgs =| ORAB;
131                         }
132                         return;
133         case 0400:       if (tp->pck_prAB == pid){      /* already reader on AB */
134                 /* same pid then writer is allowed */
```

```
135                             if (flag) {
136                                     tp->pck_fwBA = ptf;
137                                     tp->pck_fgs =| OWBA;
138                             } else break;           /* reader then error */
139                     } else {
140             /* new pid make what it likes */
141                             tp->pck_prBA = pid;
142                             if (flag) {
143                                     tp->pck_fwAB = ptf;
144                                     tp->pck_fgs =| OWAB;
145                             }else{
146                                     tp->pck_frBA = ptf;
147                                     tp->pck_fgs =| ORBA;
148                             }
149                     }
150                     return;
151     case 01400 :     if (flag) goto cas7;   /* one channel is occupied */
152                     else goto cas13;
153                     break;
154     case 02400 :     if(!flag) break;       /* two readers then no third */
155                     if (tp->pck_prAB == pid){       /* either this reader */
156                             tp->pck_fwBA = ptf;
157                             tp->pck_fgs =| OWBA;
158                             return;
159                     }
160                     if (tp->pck_prBA == pid){               /* or this */
161                             tp->pck_fwAB = ptf;
162                             tp->pck_fgs =| OWAB;
163                             return;
164                     }
165                     break;                  /* another then error */
166     case 03400 :     cas7 : if((tp->pck_prAB == pid) && flag){
167             /* one channel is occupied */
168                             tp->pck_fwBA = ptf;
169                             tp->pck_fgs =| OWBA;
170                             return;
171                     }
172                     break;
173     case 04000 :     if (tp->pck_prAB == pid) {     /* already writer on BA */
174                             if (!flag){     /* same pid then only reader is allo
175                                     tp->pck_frAB = ptf;
176                                     tp->pck_fgs =| ORAB;
177                             } else break;           /* writer then error */
178                     } else {
179                             tp->pck_prBA = pid;   /* new pid then what it likes
180                             if ( flag ){
181                                     tp->pck_fwAB = ptf;
182                                     tp->pck_fgs =| OWAB;
183                             } else {
184                                     tp->pck_frBA = ptf;
185                                     tp->pck_fgs =| ORBA;
186                             }
187                     }
188                     return;
189     case 04400 :     if (tp->pck_prAB == pid) break;
190             /* one process writes and reads */
191                     tp->pck_prBA = pid;     /* a new makes what it likes */
```

```
192                     if ( flag) {
193                             tp->pck_fwAB = ptf;
194                             tp->pck_fgs =| OWAB;
195                     } else {
196                             tp->pck_frBA = ptf;
197                             tp->pck_fgs =| ORBA;
198                     }
199                     return;
200     case 05000 :     if (flag) break;   /* already two readers, no third */
201                     if (tp->pck_prAB == pid){       /* either this process */
202                             tp->pck_frAB = ptf;
203                             tp->pck_fgs =| ORAB;
204                             return;
205                     }
206                     if (tp->pck_prBA == pid){               /* or this */
207                             tp->pck_frBA = ptf;
208                             tp->pck_fgs =| ORBA;
209                             return;
210                     }
211                     break;
212     case 05400 :     cas13 : if((tp->pck_prBA == pid) && (!flag)){
213                             tp->pck_frBA = ptf;
214                             tp->pck_fgs =| ORBA;
215                             return;
216                     }
217                     break;
218     case 06000 :     if (flag) goto cas15;  /* one channel is occupied,  */
219                     /* a new process makes what it likes */
220                     else goto cas16;
221                     break;
222     case 06400 :     cas15 : if((tp->pck_prBA == pid) && flag){
223                             tp->pck_fwAB = ptf;
224                             tp->pck_fgs =| OWAB;
225                             return;
226                     }
227                     break;
228     case 07000 :     cas16 : if((tp->pck_prAB == pid) && (!flag)){
229                             tp->pck_frAB = ptf;
230                             tp->pck_fgs =| ORAB;
231                             return;
232                     }
233                     break;
234             }

235             u.u_error = EACCES;     /* third process or bad mode */
236             return;
237     }else{
238             tp = pck;
239             if (pckflag != OPENSPEC) {
240                     for(vl = 0;vl < NPCK;vl++){
241                             if(tp->pck_fgs & (OWAB | ORAB | OWBA | ORBA)){
242                                     u.u_error = ENXIO;
243                                     return;
244                             }
245                             tp++;
246                     }
247                     pckflag = OPENSPEC;
```

```
248                 }
249                 /* all normal channels are closed */
250                 tp = pck;
251                 if(flag){       /* open  normal channels in input mode */
252                         for(vl = 0;vl < NPCK;vl++){
253                                 tp->pck_prAB = pid;
254                                 tp->pck_fwBA = ptf;
255                                 tp->pck_fgs =| OWBA;
256                                 tp++;
257                         }
258                 }else{          /* open normal channels in output mode */
259                         for(vl = 0;vl < NPCK;vl++){
260                                 tp->pck_prAB = pid;
261                                 tp->pck_frAB = ptf;
262                                 tp->pck_fgs =| ORAB;
263                                 tp++;
264                         }
265                 }
266         }
267 }

268 pckclose(dev,flag) {

269         register  int iof;
270         register  int  vl;
271         register  struct pck *tp;

272         vl = dev.d_minor;

273                 /* Monter la priorité; cela évite crash; aucun raison
274                    valable */
275         spl7();

276         /* cleanup queue, if necessary */

277         if(vl != NOSLEEP){
278                 tp = &pck[vl];
279                 pckqfl(&tp->pck_qAB);
280                 pckqfl(&tp->pck_qBA);
281                 tp->pck_stAB[1] = 0;
282                 tp->pck_stBA[1] = 0;
283                 if (pckflag == OPENSPEC) {
284                         tp->pck_fgs = (OWBA | ORAB | WRITINGBA);
285                 } else {
286                         tp->pck_fgs = 0;
287                         tp->pck_prAB = 0;
288                 }
289                 tp->pck_prBA = 0;
290         }else{                                  /* special channel thus general purge */
291                 tp = pck;
292                 /* Couper chaque communication */
293                 for(vl = 0;vl < NPCK;vl++){
294                         tp->pck_fgs =& "(OWBA | ORAB | WRITINGBA);
295                         tp->pck_stBA[1] = 0;
296                         tp->pck_stAB[1] = 0;
297                         tp->pck_prAB = 0;
298                         tp++;
```

```
299                 };
300                 pckflag = OPEN;
301         }
302         spl0();
303         return;
304 }

305 /*
306 "pckput" appelle la routine assembleur pour ajouter un caractère et
307 teste le compteur pour éviter qu'un canal ne prenne toute la place.
308 Il faut aussi réveiller le processus qui lit sur ce canal s'il est endormi.
309 */

310 pckpt(c,atp,ptf)
311 char c;
312 struct pck *atp;
313 int ptf;
314 {

315         register struct pck *tp;
316         register int ppp;

317                 /* Repérer le canal dans lequel il faut ajouter ce caractère */
318         tp = atp; ppp = ptf;

319         if(ppp == tp->pck_fwBA) {       /* writer on  BA */
320                 /* On est au maximum, refuser d'ajouter */
321                 if(tp->pck_qBA.c_cc >= PCKHI) return(-1);
322                 /* On ne peut ajouter, plus de places libres */
323                 if(pckputc(c,&tp->pck_qBA) < 0) return(-1);
324                 /* Tester si un processus est endormi */
325                 if(tp->pck_fgs & SRBA) {
326                         tp->pck_fgs =& "SRBA;
327                         wakeup(&tp->pck_prBA);
328                 }
329                 return(1);
330         } else {                        /* writer on AB */
331                 if(tp->pck_qAB.c_cc >= PCKHI) return(-1);
332                 if(pckputc(c,&tp->pck_qAB) < 0) return(-1);
333                 if(tp->pck_fgs & SRAB) {
334                         tp->pck_fgs =& "SRAB;
335                         wakeup(&tp->pck_prAB);
336                 }
337                 return(1);
338         }
339 }

340 pckgt(atp,ptf) struct pck *atp; int ptf; {

341         register struct pck *tp;
342         register int ppp;
343         register  int  c;

344                 /* Reperer le canal */
345         tp = atp; ppp = ptf;

346         if(ppp == tp->pck_frAB) {       /* reader on AB */
```

```
347                 /* Il n'y a rien dans ce canal */
348                 if((c = pckgetc(&tp->pck_qAB)) < 0) return(-1);
349                 /* Si on passe en dessous de la borne inférieure et
350                    qu'un processus est endormi sur l'écriture */
351                 if((tp->pck_qAB.c_cc < PCKLO) && (tp->pck_fgs & SWAB)) {
352                         tp->pck_fgs =& "SWAB;
353                         wakeup(&tp->pck_qAB);
354                 }
355                 return(c);
356         } else {                        /* reader on BA */
357                 if((c = pckgetc(&tp->pck_qBA)) < 0) return(-1);
358                 if((tp->pck_qBA.c_cc < PCKLO) && (tp->pck_fgs & SWBA)) {
359                         tp->pck_fgs =& "SWBA;
360                         wakeup(&tp->pck_qBA);
361                 }
362                 return(c);
363         }
364 }

365 /* Sert à effectuer une vidange d'un canal */

366 pckqfl(atp)
367 struct clist *atp; {

368         while(pckgetc(atp) >= 0);
369 }

370 pckread(dev) {
371         register  char c;
372         int nvl;
373         int vl, iof;
374         register  int  ptf;
375         register  struct pck *tp;
376         int *wchan;

377                 /* Reperer le canal */
378         vl = dev.d_minor;
379         iof = u.u_ar0[R0];
380         ptf = u.u_ofile[iof];

381         if(vl == NOSLEEP){
382                 /* Repérer l'interlocuteur, 4eme paramètre de l'appel
383                    systeme ranger dans le registre R1 */
384                 nvl = u.u_ar0[R1];
385                 nvl =& 017;
386                 tp = &pck[nvl];
387         }else{
388                 tp = &pck[vl];
389         };

390   retry:
391         spl7();
392         while((c = pckgt(tp,ptf)) != -1)
393         {
394                 if((passc(c) == -1) || (c == ENDFIL)) {
395                         spl0();
396                         return;
```

```
397                 }
398         }

399         /* more to be read, but not enough in buffers */
400         if(vl != NOSLEEP) {
401                 if(ptf == tp->pck_frAB) {       /* reader on AB */
402                         tp->pck_fgs =| SRAB;
403                         wchan = &tp->pck_prAB;
404                 } else {                        /* reader on BA */
405                         tp->pck_fgs =| SRBA;
406                         wchan = &tp->pck_prBA;
407                 }
408                 u.u_ar0[R1] = u.u_count;
409                 sleep(wchan, PCKPRI);
410                 spl0();
411                 goto retry;
412         }
413
414                 /* Le processus qui utilise le moniteur special n'est
415                    jamais endormi */

416         spl0();
417         return;
418 }

419 pckwrite(dev) {
420         register  int  c;
421         int vl, iof;
422         register  int  ptf;
423         register  struct pck *tp;
424         int *wchan;

425                 /* Reperer le moniteur */
426         vl = dev.d_minor;
427         iof = u.u_ar0[R0];
428         ptf = u.u_ofile[iof];

429         if(vl == NOSLEEP){
430                 /* Repérer l'interlocuteur dont le numéro est le 4eme
431                    paramètre de cet appel système rangé dans le registre R1 */
432                 c = u.u_ar0[R1];
433                 tp = &pck[c&017];
434         }else{
435                 tp = &pck[vl];
436         };

437         if(ptf == tp->pck_fwBA) tp->pck_fgs =| WRITINGBA;   /* when it is set,*/
438                         /* user has still something to write */
439         else tp->pck_fgs =| WRITINGAB;
440         spl7();

441         while((c = cpass()) != -1) {
442         retry :
443                 if(pckpt(c,tp,ptf) < 0) {
444                         if(vl != NOSLEEP) {
445                                 if(ptf == tp->pck_fwBA) {   /* writer on BA */
446                                         tp->pck_fgs =| SWBA;
```

```
447                                     wchan = &tp->pck_qBA;
448                                 } else {        /* writer on AB */
449                                     tp->pck_fgs =| SWAB;
450                                     wchan = &tp->pck_qAB;
451                                 }
452                                 u.u_ar0[R1] = u.u_count;
453                                 sleep(wchan, PCKPRI);
454                                 spl7();
455                                 goto retry;
456                             }
457                             spl0();
458                             u.u_count++;
459                             return;
460                         }
461                 }
462                 spl0();
463                 if(ptf == tp->pck_fwBA) tp->pck_fgs =& "WRITINGBA;   /* user finished */
464                 else tp->pck_fgs =& "WRITINGAB;
465 }
466                 /* Créer le premier chainage */
467 pckinit() {
468                 register int ccp;
469                 register struct cblock *cp;

470                 ccp = pckfr;
471                         /* Commencer la division a un multiple de 8 */
472                 for(cp = (ccp+07) & "07; cp <= &pckfr[NPCKBLK-1]; cp++) {
473                         cp->c_next = pckfrlist;
474                         pckfrlist = cp;
475                 }
476 }


477 pckstat(dev,p)
478 int     dev;
479 int     *p;
480 {

481                 register  struct  pck  *tp;
482                 register  int  vl;
483                 register  int  *pnt;
484                 int  ptf;
485                 int  iof;

486                 pnt = p;
487                 vl = dev.d_minor;
488                 iof = u.u_ar0[R0];
489                 ptf = u.u_ofile[iof];

490                 if(vl == NOSLEEP) {
491                         vl = u.u_ar0[R1];
492                         vl =& 017;
493                 }
494                 tp = &pck[vl];

495                 if (pnt == 0) {                 /* stty */
496                         if(ptf == tp->pck_fwAB) {
```

```
497                     if (tp->pck_stAB[1] & NOTRANS) {
498                             u.u_error = EINVAL;
499                     } else {
500                             tp->pck_stAB[0] = u.u_arg[0];
501                             tp->pck_stAB[1] = u.u_arg[1] & CHAR;
502                             tp->pck_stAB[1] =| NOTRANS;
503                     }
504             };
505             if(ptf == tp->pck_fwBA) {
506                     if (tp->pck_stBA[1] & NOTRANS) {
507                             u.u_error = EINVAL;
508                     } else {
509                             tp->pck_stBA[0] = u.u_arg[0];
510                             tp->pck_stBA[1] = u.u_arg[1] & CHAR;
511                             tp->pck_stBA[1] =| NOTRANS;
512                     }
513             }
514     } else {
515             if(ptf == tp->pck_fwAB) {
516                     *pnt++ = tp->pck_stBA[0];
517                     *pnt = tp->pck_stBA[1];
518                     *pnt++ =| tp->pck_fgs & STAT;
519                     tp->pck_stBA[1] =& "NOTRANS;
520                     *pnt = (tp->pck_qBA).c_cc;
521                     *pnt =<< 8;
522                     *pnt =| (tp->pck_qAB).c_cc & CHAR;
523             } ;
524             if(ptf == tp->pck_fwBA) {
525                     *pnt++ = tp->pck_stAB[0];
526                     *pnt = tp->pck_stAB[1];
527                     *pnt++ =| tp->pck_fgs & STAT;
528                     tp->pck_stAB[1] =& "NOTRANS;
529                     *pnt = (tp->pck_qAB).c_cc;
530                     *pnt =<< 8;
531                     *pnt =| (tp->pck_qBA).c_cc & CHAR;
532             }
533     }
534 }
```

Annexe E: Variables, structures et paramètres.

Les variables et structures déclarées globalement sont répétoriées dans des fichiers particuliers caractérisés par leur nom en '.h' . Il en est de même pour les paramètres importants.

Tous ces fichiers sont utilisés lors de la compilation du moniteur réseau tant pour la phase 1 (INITX25) que pour la phase 2 (RTTX25). De plus, quelques uns servent pour la compilation des routines de la librairie ou pour d'autres programmes.

Fichier pckx25.h (variables relatives à pckx25.c).

```
#
/* identification des paquets */
#define PCKA            013     /* ident. paquet d'appel */
#define PCKL            023     /* ident. paquet de demande de libération */
#define PCKD            0       /* ident. paquet de données */
#define PCKR            033     /* ident. paquet de demande de reinitialisation */
#define PCKREP          0373    /* ident. paquet de demande de reprise */
#define PCKDAT          0       /* ident. paquet de données */
#define PCKPRR          01      /* ident. paquet RR */
#define PCKPRNR         05      /* ident. paquet RNR */
#define PCKCA           017     /* ident. paquet confirmation acceptee */
#define PCKCL           027     /* ident. paquet confirmation de libération */
#define PCKCR           037     /* ident. paquet confirmation de reinitialisation */
#define PCKCREP         0377    /* ident. paquet confirmation de reprise */
#define PCKIT           043     /* ident. paquet d'interruption */
#define PCKCIT          047     /* ident. paquet de confirmation d'interruption */

/* flags des facilites */
#define FACILI  0100000         /* existence de facilites */


/* flags de voie logique */
#define BITM            01
#define MBIT            02
#define BITQ            04
#define QBIT            010
#define STOPIN          0100
#define LOWIN           0200
#define STOPOU          0400
#define INTOUT          01000
#define RNROU           02000


/* nombres importants */
#define HEADPCK     020         /* entête de paquet */
#define NVL     16              /* nombre de voies logiques disponibles */
#define LPCK    128             /* longueur des paquets par défaut */
```

```
#define LNGMAX  131
#define DFEN    4               /* dimension de la fenêtre par défaut */

#define LINEIN          5       /* nombre d´utilisateur externe prevu */

struct vl {
        int     v_usflag;       /* flags de l´utilisateur */
        int     v_facili;       /* flags des facilites demandées */
        int     v_mode;         /* mode de table sequentielle */
        int     v_lnpaq;        /* long paquet (facilité) */
        int     v_dimfen;       /* dimension de la fenêtre (facilité) */
        char    *v_bufout;      /* adresse de la structure des E/S avec le niveau t
                        /* compteurs du controle de flux */
        char    v_recu;
        char    v_rack;
        char    v_send;
        char    v_sack;
        int     v_uc;           /* numéro de liaison par pck */
        char    *v_tim;         /* indicatif pour temporisateur */
} vl[NVL], *pntvl;

#define VLLNG           20
#define VLLNGTOT        320


                /* structure des paquets */
struct pk1 {
        int     pk1_bf[3];
        int     pk1_lng;
        char    pk1_head;
        char    pk1_vlid;
        char    pk1_act;
        char    pk1_ar1;
        char    pk1_ar2;
};


int     (*func[2][020][012])();

#define FUNCLNG         640
```

Fichier open.h (structure pour les fichiers "admvl" et "init.h").

```
                /* Structure de l´entete du fichier ´init.h´ */
struct head {
        int     dfile;  /* lng. descripteurs de fichiers */
        int     dstrvl; /* lng. structure des voies logiques */
        int     dtable; /* lng. table sequentielle */
        int     dchain; /* lng. chaines des travaux */
} dimen, *pntdim;
```

```
                /*
                 * structure des paramètres nécessaires pour l'
                 * établissement d'une liaison entre un utilisateur
                 * et le moniteur réseau. Cette structure se retrouve
                 * dans le fichier "admvl".
                 */
struct rq{
        int     rq_fgs;
        int     rq_pid;         /* Numéro du processus utilisateur */
        int     rq_ttid;        /* Numéro du terminal utilisateur. */
        int     rq_usid;        /* Numéro de l'utilisateur */
        int     rq_uc;          /* Numéro de moniteur pck? */
        int     rq_vl;          /* Numéro de la voie logique */
};

#define RQLNG   12      /* longueur de la structure 'rq' */

                /*
                 * Structure pour l'établissement d'une liaison entre
                 * le moniteur réseau et un autre ETTD. Cette structure
                 * se retrouve dans le fichier "admvl".
                 */

struct op {
        int     op_fgs;
        int     op_fac;         /* Facilités si elles existent */
        int     op_uc;          /* Numéro du moniteur pck? */
        int     op_vl;          /* Numéro de la voie logique */
        int     op_ailn;        /* Longueur de l'adresse suivante */
        char    op_iad[20];     /* Adresse interne */
        int     op_aoln;        /* Longueur de l'adresse suivante */
        char    op_oad[20];     /* Adresse externe */
        int     op_wln;
        int     op_pln;
};

#define OPLNG   46      /* longueur de la structure 'op' */


struct  deb {
        struct  rq  deb_rq;
        struct  op  deb_op;
};

#define DEBLNG  58      /* longueur de la structure 'deb' */
```

Fichier drx25.h (voir drx25.c).

```
#

/* paramètres pour routine d'envoi vers le niveau trame */
```

```
#define NDR11           2       /* nombre de niveaux trame */
#define TRSTILL         077000  /* flag de place libre dans le niveau trame */
#define CRITIC          0217    /* problème grave au niveau trame */
#define BFDRO           6       /* seuil de saturation de la chaine à envoyer */
#define TRFREE          060000  /* seuil de place libre au niveau trame */
#define PLUSVITE        0400    /* flag d'impatience du niveau trame */
#define RMAX            133     /* longueur max de la lecture */
#define NDR1            8       /* voies logiques pour le premier interface */


/* structure des données des routines d'envoi vers le niveau trame */
struct trame {
        int     *tr_ptdat;      /* pointeur vers la chaine des paquets de données */
        int     *tr_ptcmd;      /* pointeur vers la chaine des paquets de commande */
        int     tr_cpt;         /* nombre de buffers a écrire */
        int     tr_cmdo;        /* status prêt a partir */
        int     tr_cmdi[3];     /* dernier status reçu */
        char    tr_fdr;         /* descripteur de fichier en lecture */
        char    tr_fdw;         /* descripteur de fichier en ecriture */
} trame[NDR11];
```

Fichier task.h (voir gerant.c, routines select(), taskin() et detimo()).

```
#

/* parametres des taches */

#define NJOB    16              /* nombre max de taches */
#define SEMA    1               /* sémaphore de blocage de taskin */


/* structure d'une tache */
struct task {
        struct  task  *tk_pnt;
        int     tk_prio;        /* priorité de la tache */
        int     (*tk_job)();    /* adresse de la routine */
        int     tk_arg[3];      /* argument de l'appel */
}travail[NJOB];

#define TKLNG           192

/* structure du début de chainage pour le temporisateur */
struct tim{
        struct  task  *tim_pnt; /* pointeur vers la premiere tache */
        int     tim_curt;       /* heure courante */
}tim;

#define TIMLNG          4

/* structure du debut chainage des taches */
```

```
struct sched {
        struct  task  *sc_pnt;  /* pointeur vers la prochaine tache a effectuer */
        int     sc_flag;        /* flag de contrôle */
        struct  task  *sc_free; /* pointeur vers premier bloc libre */
} sch;

#define SCHLNG          6
#define TKLNGTOT        202

char    priority[10] {0,1,2,3,4,5,6,7,8,9};
```

Fichier buffer.h (voir gerant.c, routines bufp() et bufv()).

```
#
/* parametres des buffers */

#define LBUF            134     /* longueur utile des buffers */
#define NBFMAX          50      /* nombre max de buffers admis, 50 * 140bytes = 7KB */
#define PAGE            704     /* dimension d'une page de buffers */


/*structure d'un buffer */
struct bf{
        struct  bf  *bf_pnt;            /* pointeur vers le buffer suivant */
        char    *bf_to;                 /* pointeur libre */
        int     bf_cpt;                 /* compteur d'utilisation */
        char    bf_buf[LBUF];           /* longueur utile du buffer */
};


/* structure de la reserve de buffers */
struct bfres {
        struct  bf  *bf_first;          /* pointeur vers le premier buffer libre */
        int     bf_free;                /* compteur des buffers libres  */
        int     bf_res;                 /* nombre de buffers existant */
} bfres;
```

Fichier uclx25.h (voir uclx25.c).

```
#define WMAX            30      /* longueur max possible à ecrire sur chaque pck */
#define TROP            3       /* première borne pour la file d'attente vers l'utilisa
#define SATURATION      5       /* garde fou pour la file d'attente vers l'utilisateur

struct ucl {
        int     uc_flag;        /* flags de l'utilisateur */
```

```
        int     uc_ppid;
        char    *uc_bufin;
        char    *uc_bufout;
        char    uc_nvl;
        int     *uc_vl;
        char    *uc_point;
        char    *uc_tim;
        int     uc_rdcpt;
        int     uc_wrcpt;
        int     uc_fgs;
}ucl[NVL], *pntucl;

#define UCLNG           22      /* longueur de la structure ucl */

int     uc_frd;         /* descripteur de fichier pour l'ouverture en mode lecture */
int     uc_fwr;         /* descripteur de fichier pour l'ouverture en mode ecriture */

int     fstat;          /* fichier de renseignement */

/* valeur des flags */
#define OCCUP           01      /* ucl occupée */
#define LINING          02      /* on est en train d'établir la liaison */
#define LINED           04      /* liaison etablie par pck */
#define CALLING         010     /* on est en train d'appeler le réseau */
#define ONLINE          020     /* correspondant en ligne */
#define STOPREAD        0100    /* suspendre les lectures */
#define DELREAD         0200    /* détruire ce qui est lu */
#define STOPWRITE       0400    /* suspendre les écritures vers l'utilisateur */
#define ARREAD          01000   /* lectures arrêtées */

/* codes operatoires */
#define CALL            1       /* netopen : demande de liaison avec le réseau */
#define INT             4       /* donnée d'interruption */
#define LIB             2       /* netclose : fin de communication */
#define VIDVIT          3       /* nettoyer le circuit */
#define RR              5       /* demande contrôle de flux */
#define REP             7       /* reprise */
#define BREAK           15      /* frline : tout fini */
#define BEGIN           14      /* rqline : demande une ligne */
#define RUN             16      /* en route */
#define OKCALL          9       /* appel accepté */

/*valeur des flags de uc_fgs */
#define OPENED          03000   /* pck ouvert par l'utilisateur */
#define NOTRANS         0100000 /* la commande est lue pour la première fois */
#define STILLWR         030000  /* en train d'écrire */

#define USER            0177400 /* demandeur est utilisateur */
```

Fichier routine.h (index des routines existante).

```
/*
```

```
                * Routines de gestion du protocole X25 paquet.
                * Voir fichier pckx25.c
                */
int     pcallin();
int     pckx25();
int     plibin();
int     plibo();
int     plibed();
int     preinin();
int     preinou();
int     preined();
int     pintin();
int     pintout();
int     pinted();
int     prrout();
int     prnrout();
int     pdatin();
int     pdatout();
int     prepin();
int     prepout();

               /*
                * Routine de liaison avec les utilisateurs et de
                * contrôle de leurs processus.
                * Voir fichier endend.c
                */
int     ewrite();
int     eread();
int     ecmdin();
int     ecmdo();
int     ertrait();
int     elining();
int     ecalling();

/* routines d'entree/sortie sur les moniteurs PCK */
int     ugtcmd();       /* U4   routine d'acquisition de l'état de l'utilisateur */
int     uread();
int     uwrite();
int     upcmd();
int     dwrite();
int     dread();

/* routine de gestion speciale */
int     bufp();         /* G1   attribution de buffer */
int     bufv();         /* G2   récupération de buffer */
int     bufover();      /* G3   incrémentation de la zone des buffers */
int     select();       /* G4   sélection du travail suivant */
int     taskin();       /* G5   tache a rentrer */
int     detimo();       /* G61  tache a retirer */
int     timout();       /* G7   changement de file d'attente */
int     sendat();       /* G8   mise en file d'attente des paquets de données */
int     bipbip();       /* G9   tache a temporisateur */
int     sendcmd();      /* G10  mise en file d'attente des paquets de cmd */
int     bfclean();      /* G11  nettoyage de la file d'attente des données */
int     newpck();       /* G12  arrivée nouveau paquet */
int     trastat();      /* G14  arrivée stutus du niveau trame */
int     tempor();       /* G15  réveil par temporisateur */
```

Fichier mnemo.h (cause et diagnostics des libérations,...).

```
#
/* Ce fichier comprend les codes et diagnostics pour les reprises,
 * les libérations et les réinitialisations                          */

/* code des causes de reprise */
#define CREPTD          0       /* reprise par l'ETTD (voir diagnostic) */
#define CPROCLOC        01      /* erreur de procédure locale (voir diagnostic) */
#define CFINDER         05      /* fin de dérangement */
#define CFINPRO         07      /* fin d'incident */

/* code des diagnostics pour cause d'erreur de procedure locale */
#define DPCKINC         01      /* type de paquet inconnu */
#define DERREP          02      /* erreur lors d'une reprise */
#define DVLERR          03      /* nature de VL incorrecte */
#define DVLFAULT        010     /* No VL non a zero dans un paquet de reprise */
#define DDEBUT          020     /* debut d'une connection */

/* code des causes de liberation */
#define CLIBTD          0       /* libération par l'ETTD */
#define COCCUPE         01      /* numéro occupé ou collision d'appel */
#define CDERANG         011     /* dérangement */
#define CPROCDIS        021     /* erreur de procédure distante */
#define CNONTAX         031     /* abonné distant refusant la taxe au demande */
#define CERRAPP         03      /* appel non valide */
#define CACCES          013     /* accès interdit */
#define CPRLOCAL        023     /* erreur procédure locale */
#define CINCID          05      /* incidents sur le réseau */
#define CINCONNU        015     /* numero inconnu */
#define CFINDER         041     /* fin de dérangement */

/* code des diagnostics lors d'une erreur de procédure distante */
#define DPCKINC         01      /* type de paquet inconnu */
#define DERREP          02      /* erreur lors d'une reprise */
#define DVLERR          03      /* nature de VL incorrecte */
#define DERRAPP         04      /* erreur dans la phase d'établissement ou de rupture de
#define DTOREINI        05      /* non acquittement lors d'une reinitialisation */
#define DVLREP          010     /* Vl non zero dans un paquet de reprise */

/* code des diagnostics lors d'un appel nen valide */
#define DLNGPAQ         01      /* taille du paquet incorrecte */
#define DFACILI         02      /* demande d'une facilité non autorisée a l'abonnement
#define DBADFAC         03      /* erreur d'emploi d'une facilité */

/* code des diagnostics lors d'une erreur de procédure locale */
#define DERRAPP         04      /* erreur dans la phase d'établissement ou de rupture de
#define DTOREINI        05      /* non acquittement lors d'une reinitialisation */

/* code des causes de reinitialisation */
#define CRETD           0       /* demande faite par l'ETTD */
#define CRDER           01      /* dérangement */
#define CREPRODIS       03      /* erreur de procédure distante */
#define CREPROLO        05      /* erreur de procédure locale */
#define CREINC          07      /* incidents dans le réseau */
```

```
#define CREFINDER       011     /* fin de dérangement */
#define CREFININC       017     /* fin d´incident */

/* code des diagmostics pour une erreur de procédure distante */
#define DPCKINC         01      /* type de paquet inconnu */
#define DERREP          02      /* erreur lors d´une reprise */
#define DVLERR          03      /* nature de VL incorrecte */
#define DLNGDAT         04      /* taille du champ de données incorrectes*/
#define DFLUX           05      /* erreur de contrôle de flux P(S) ou P(R) */
#define DERRPROC        06      /* erreur de procédure dans une reinitialisation */
#define DNOREJ          07      /* l´abonné n´a pas l´option REJ */
#define DVLREP          010     /* Vl non zero dans un paquet de reprise */

/* diagnostic pour erreur de procedure locale */
#define DLNGDAT         04      /* taille du champ de données incorrecte */
#define DFLUX           05      /* erreur de controle de flux P(S) ou P(R) */
#define DERRPROC        06      /* erreur de procédure dans une reinitialisation */
#define DNOREJ          07      /* l´abonné n´a pas l´option REJ */
/* diagnostic pour demande par ETTD */
#define DSATUR          020     /* saturation de l´ETTD */
#define DPROTO          022     /* erreur protocole */
#define DTIMOUT         024     /* dépassement du temporisateur */
```

Fichier protocol (table séquentielle).

```
int     (*func[2][020][012])(){
/* réception d'un paquet de données */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pdatin,        &pnotrait,
/* réception d'un appel */
&pcallin,       &pnotrait,      &pnotrait,      &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
/* réception d'une demande de libération */
&pckerr,        &pnotrait,      &plibin,        &plibin,        &plibin,
&plibin,        &plibed,        &pnotrait,      &plibin,        &plibin,
/* réception demande de réinitialisation */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &preinin,       &preined,
/* réception d'une interruption */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pintin,        &pnotrait,
/* réception d'un contrôle de flux */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pfcin,         &pnotrait,
/* rien à faire */
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
/* réception demande de reprise */
&prepin,        &preped,        &prepin,        &prepin,        &prepin,
&prepin,        &prepin,        &prepin,        &prepin,        &prepin,
/* rien à faire */
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
/* réception confirmation d'appel */
&pckerr,        &pnotrait,      &pokcall,       &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
/* réception d'une confirmation de libération */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &plibed,        &pnotrait,      &pckerr,        &pckerr,
/* réception d'une confirmation de réinitialisation */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &preined,
/* réception confirmation d'une interruption */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pinted,        &pnotrait,
/* réception d'une demande d'arrêt d'émission */
&pckerr,        &pnotrait,      &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pnotrait,      &pckerr,        &pfcin,         &pnotrait,
/* rien à faire */
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
/* rien à faire */
&pckerr,        &preped,        &pckerr,        &pckerr,        &pckerr,
&pckerr,        &pckerr,        &pckerr,        &pckerr,        &pckerr,
/* émission d'un paquet de données */
&pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,      &pnotrait,
&pnotrait,      &pnotrait,      &pnotrait,      &pdatout,       &pnotrait,
/* émission d'un paquet d'appel */
```

```
&pcallout,      &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* émission d'une demande de libération */
&plibo,         &pnojob,       &plibo,        &plibo,        &plibo,
&plibo,         &pnojob,       &plibo,        &plibo,        &plibo,
/* émission d'une demande de réinitialisation */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &preinou,      &pnojob,
/* émission d'une interruption */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pintout,      &pnojob,
/* émission d'un contrôle de flux */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &prrout,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* émission d'une demande de reprise */
&prepout,       &prepout,      &prepout,      &prepout,      &prepout,
&prepout,       &prepout,      &prepout,      &prepout,      &prepout,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* émission d'un paquet pour arrêter l'entrée */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &prnrout,      &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
/* rien à faire */
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
&pnojob,        &pnojob,       &pnojob,       &pnojob,       &pnojob,
};
```

Annexe F: Le programme d'initialisation.

```
 1 #
 3 #include "mnemo.h"
 4 #include "pckx25.h"
 5 #include "net25.h"
 6 #include "open.h"
 7 #include "task.h"
 8 #include "uclx25.h"
 9 #include "drx25.h"
10 #include "buffer.h"


11 int     finit;          /* descripteur du fichier d'initialisation */
12         /* doit avoir le numero '3' sous peine d'erreur */

13                 /* Fichier contenant la deuxieme passe et la table des symboles */
14 char    *origin         "rttx25";

15                 /* Structure pour l'accs  la table des symboles. */
16 struct  n {
17         char    nl_name[8];
18         int     nl_type;
19         int     nl_value;
20 } ;

21                 /* Structure de l'entete du fichier 'init.h' */
22 struct head {
23         int     dfile;  /* lng. descripteurs de fichiers */
24         int     dstrvl; /* lng. structure des voies logiques */
25         int     dtable; /* lng. table sequentielle */
26         int     dchain; /* lng. chaines des travaux */
27 } dimen, *pntdim;

28 /* Le main programme est chargé de l'affichage preliminaire au terminal
29  * et d'orchestrer les appels aux routines.
30  */

31 main() {

32                 /* routine de création et d'ouverture des fichiers */
33         ifile();
34                 /* routine pour l'initialisation de la structure des voies logiques *
35         ivl();
36                 /* routine de création de la table séquentielle */
37         itable();
38                 /* routine de création de la chaine des travaux */
39         ichain();

40         printf("0");
41         printf(" Organisation on file 'init.h' : 0);
42         printf(" - files descriptors : 0);
43         printf("    TRAPAC1 in read mode, and in write mode; 0);
```

```
44          printf("    TRAPAC2 in read mode, and in write mode; 0);
45          printf("    PCKZ in read mode, and in write mode; 0);
46          printf("    file 'admvl' in read/write mode. 0);
47          printf(" - structure 'vl'0);
48          printf(" - automat table, from file 'protocol'0);
49          printf(" - task chain0);
50          printf("*************************************************0);
51          printf("0*  LET'S GO  **0);
52          execl(origin, origin, 0);
53  }

54  /*
55   * "ifile" s'occupe de tout ce qui concerne les fichiers:
56   *      - créer le fichier temporaire.
57   *      - ouvrir le fichier de communication inter-processus.
58   *      - ouvrir les fichiers de communication avec le niveau trame.
59   *      - créer un fichier d'administration et de renseignements.
60   */

61  ifile() {
62                  /* code d'erreur pour les entrées sorties */
63          extern  int  errno;
64          register  struct  trame  *pttr;
65          register  int  result;

66                  /* création du fichier temporaire 'init.h' contenant les paramètres
67                   * de l'initialisation. */
68          finit = creat("init.h", 0600);
69          close(finit);
70          finit = open("init.h", 2);
71          if(finit != 3) {
72                  printf("0Can't creat 'init.h' file. 0);
73                  exit();
74          }
75
76                  /* mise en place de l'entête du fichier 'init.h' */
77          pntdim->dfile = 14;
78          pntdim->dstrvl = sizeof vl;
79          pntdim->dtable = sizeof func;
80          pntdim->dchain = (sizeof travail) + (sizeof tim) + (sizeof sch);

81          result = write(finit, pntdim, sizeof dimen);

82          if (result != sizeof dimen) {
83                  printf("0ANIC! can't creat 'init.h' 0);
84                  exit();
85          }


86                  /* ouverture du moniteur de communication inter-processus */
87          uc_frd = open("/dev/pckz",0);
88          if (uc_frd < 0) {
89                  printf("0PANIC can't open pck. errno = %d 0, errno);
90                  exit();
91          } else {
92                  printf("pck read mode : file descriptor : %d 0,uc_frd);
93          }
```

```
94          uc_fwr = open("/dev/pckz",1);
95          if (uc_fwr < 0) {
96                  printf("OPANIC can't open pck. errno = %d 0, errno);
97                  exit();
98          } else {
99                  printf("pck write mode : file descriptor : %d 0, uc_fwr);
100         }

101                 /* ouverture des fichiers de communication avec les
102                  * microprocesseurs. */
103         pttr = &trame[0];
104         pttr->tr_fdw = open("/dev/trapac1",1);
105         pttr->tr_fdr = open("/dev/trapac1",0);
106         if(pttr->tr_fdr < 0 || pttr->tr_fdw < 0) {
107                 printf("OPANIC can't open trapac1. errno : %d .0, errno);
108                 exit();
109         }
110         pttr++;
111         if (NDR11 > 1) {
112                 pttr->tr_fdr = open("/dev/trapac2",0);
113                 pttr->tr_fdw = open("/dev/trapac2",1);
114                 if(pttr->tr_fdr < 0 || pttr->tr_fdw < 0 ){
115                         printf("OPANIC can't open trapac2. errno : %d .0, errno);
116                         exit();
117                 }
118         }

119                 /* création du fichier administration. */
120         fstat = creat("admvl", 0644);
121         close(fstat);
122         fstat = open("admvl", 2);
123         if(fstat < 0) {
124                 printf("OPANIC can't creat 'admvl' file. 0);
125                 exit();
126         }

127         result =+ write(finit, &(trame[0].tr_fdr), 2);
128         result =+ write(finit, &(trame[0].tr_fdw), 2);
129         result =+ write(finit, &(trame[1].tr_fdr), 2);
130         result =+ write(finit, &(trame[1].tr_fdw), 2);
131         result =+ write(finit,&(uc_frd), 2);
132         result =+ write(finit, &(uc_fwr), 2);
133         result =+ write(finit, &fstat, 2);
134         if (result != pntdim->dfile) {
135                 printf("OPANIC error in save on 'init.h' 0);
136                 exit();
137         }
138 }

139 /*
140  * "ivl" s'occupe d'initialiser la voie logique en fournissant des valeurs
141  * de l'utilisateur ou des valeurs par défaut.
142  * Chaque voie logique est associée avec un microprocesseur; on calcule
143  * les adresses des structures relatives aux moniteurs pour DR11-C (trame)
144  */

145 ivl() {
```

```
146         register  int  i;
147         register  struct  vl *vlpt;
148         register  int   result;
149         char    c;
150         struct  trame  *pttr0, *pttr1;
151         struct  n  nl[2];
152         char    *pntch;

153         printf(" Values of 'vl' structure : 0);
154         printf("v_usflag = 0        ");
155         printf("v_facili = 0        ");
156         printf("v_mode = 0   0);
157         printf("v_lnpaq = %d        ", LPCK);
158         printf("v_dimfen = %d       ", DFEN);
159         printf("v_recu = 0 0);
160         printf("v_rack = 0  ");
161         printf("v_send = 0  ");
162         printf("v_sack = 0 0);
163         printf("v_tim = 0   ");

164                 /* recherche de l'adresse de la structure 'trame' */
165         pntch = &nl[0].nl_name[0];
166         *pntch++ = '_';
167         *pntch++ = 't';
168         *pntch++ = 'r';
169         *pntch++ = 'a';
170         *pntch++ = 'm';
171         *pntch++ = 'e';

172         nlist(origin, nl);
173         pttr0 = nl[0].nl_value;
174         pttr1 = pttr0 + 1;


175         vlpt = vl;
176         for (i = 1 ; i <= 16 ; i++) {
177                 vlpt->v_usflag = 0;
178                 vlpt->v_facili = 0;
179                 vlpt->v_mode = 0;
180                 vlpt->v_lnpaq = LPCK;
181                 vlpt->v_dimfen = DFEN;
182                 /* chaque voie logique est en relation avec un microprocesseur */
183                 vlpt->v_bufout = (i < NDR1 ? pttr0 : pttr1);
184                 vlpt->v_recu = 0;
185                 vlpt->v_rack = 0;
186                 vlpt->v_send = 0;
187                 vlpt->v_sack = 0;
188                 vlpt++;
189         }
190
191 loop :
192         printf("0o you change these values ? 0);

193         c = getchar(); getchar();
194         if (c == 'y') {
195                 printf("0ATTENTION ! Your changes must be in agreement");
196                 printf("0with network rules. 0);
```

```
197                 printf("  Wich structure will you change ? 0);
198                 i = getint();
199                 vlpt = &vl[i];

200                 printf("v_usflag ?  ");
201                 result = getint();
202                 vlpt->v_usflag =  result;

203                 printf("v_facili ?  ");
204                 result = getint();
205                 vlpt->v_facili = result;

206                 printf("v_lnpaq ?   ");
207                 result = getint();
208                 vlpt->v_lnpaq = result;
209
210                 printf("v_dimfen ?  ");
211                 result = getint();
212                 vlpt->v_dimfen = result;
213


214                 goto loop;
215         }

216         result = write(finit, vl, sizeof vl);
217         if (result != sizeof vl) {
218                 printf("OPANIC error in vl structures initialization. 0);
219                 exit();
220         }
221         return;
222 }

223 /*
224  * "itable" transforme le fichier 'protocol' écrit en francais et donc
225  * modifiable en un fichier d'adresses des routines de traitement pour
226  * chaque cas correspondant à une transition de la table.
227  */


228 itable(){
229         struct {
230                 char    nl_name[8];
231                 int     nl_type;
232                 int     nl_value;
233         }nl[321],*pnl;

234         register  int  result, i;
235         char    c;
236         int     frd, fwr;

237         frd = open("protocol",0);
238         pnl = nl;

239   loop:
240         result = read(frd, &c, 1);
241         if (result < 0) goto err;
```

```
242                 /* construction de la liste des symboles nécessaires */
243                 /* en partant du fichier 'protocol' */
244         if (c == '&') {
245                 pnl->nl_name[0] = '_';
246                 i = 1;
247                 result = read(frd, &c, 1);
248                 if (result < 0) goto err;
249                 while ( c != ',') {
250                         pnl->nl_name[i++] = c;
251                         result = read(frd, &c, 1);
252                         if (result < 0) goto err;
253                 }
254                 pnl++;
255         }
256         if (c == '}') goto fin;
257         goto loop;

258   fin :
259         nlist(origin,nl);
260         pnl = nl;
261
262         for (i = 0; i < 320;i++) {
263                 result = write(finit, &(pnl->nl_value), 2);
264                 if (result < 0) goto err;
265                 pnl++;
266         }
267         close(frd);
268         return;
269   err :
270         printf("OPANIC. error in table initialization. 0);
271         exit();
272 }

273 /*
274  * Construction de la chaine des buffers qui contiendront les renseignements
275  * pour chacun des travaux à effectuer.
276  */

277 ichain() {
278         struct  n  nl[2];
279         char    *pntch;
280         register  struct  task  *pnt, *npnt;
281         register  int   i;
282         int     result;
283         extern  int  errno;


284         pntch = &nl[0].nl_name[0];

285         *pntch++ = '_';
286         *pntch++ = 't';
287         *pntch++ = 'r';
288         *pntch++ = 'a';
289         *pntch++ = 'v';
290         *pntch++ = 'a';
291         *pntch++ = 'i';
292         *pntch++ = 'l';
```

```
293         nlist(origin, nl);

294         pnt = &travail[1];
295         npnt = nl[0].nl_value;
296         npnt++;
297         sch.sc_free = npnt++;
298
299         for(i = 1; i < NJOB - 1; i++){
300                 (pnt++)->tk_pnt = npnt++;
301         }
302         pnt->tk_pnt = 0;
303         result = write(finit, travail, (sizeof travail) + (sizeof tim) + (sizeof sch
304         if (result < 0) {
305                 printf("0PANIC. error in save of tasks chain 0);
306                 exit();
307         }
308         return;
309 }

310 /*
311  * "iline" crèe un processus qui va executer le programme "netinit" de
312  * contrôle des processus pour utilisateurs externes.
313  */

314 iline() {
315         register  int  result;
316         register  int  i;

317         result = fork();
318         if (result == 0) result = execl("netinit", "netinit",0);
319         if (result < 0) printf("Pas d'initialisation dees taches internes0);
320         else printf("controle des taches internes par le processus %d 0, result);
321         printf("%d processus sont initialiss pour les utilisateurs externes0,LINEIN)
322 }

323 /*
324  * "itask" crèe le processus horloge qui toutes les trentes secondes
325  * va envoyer un signal software vers ce processus-ci.
326  */

327 itask(){
328         char  pid[3];
329         register  int  result;
330         register  int  *apid;

331         apid = pid;
332         result = getpid();
333         pid[2] = '0';
334         *apid = result;
335         result = fork();
336         if(result == 0) result = execl("horloge","horloge",pid,0);
337         if(result < 0) goto err;
338         tim.tim_curt = 0;
339         printf("0clock process running !. 0);
340         return;
341   err:
342         printf("0PANIC, no clock. 0);
```

```
343         exit();
344 }
```

Annexe G: Le programme pour temporisateur

```
1 #

2 /*
3  * Ce programme est activé par "initx25" et toutes les trentes secondes
4  * il interrompt le processus moniteur réseau par un signal 'kill 1'.
5  * Si le processus réseau n'existe pas ou est mort, l'horloge s'arrête.
6  * Pour bien prouver qu'il a été activé, ce programme affiche le
7  * numero du processus moniteur reseau; une comparaison avec la table
8  * des processus permet de vérifier la validite de fonctionnement.
9  */

10 /*
11  * L'argument d'appel est le numéro du processus père.
12  */
13 /*
14  * Affichage de:
15  *      PANIQUE, horloge en panne.
16  * si il y a une erreur.
17  */

18 main(argc,argv)
19 int     argc;
20 char    **argv;
21 {
22         register  int  *pnt;
23         register  int  result;
24         register  int  pid;
25
26         pnt = argv[1];
27         pid = *pnt;
28         sleep(20);
29         printf("processus père : RTTX25 a le numero: %d . 0, pid);
30         for(;;){
31                 sleep(30);
32                 result = kill(pid, 1);
33                 if(result < 0){
34                         printf("panique, horloge en panne. 0);
35                         exit();
36                 }
37         }
38 }
```

Annexe I: Les fonctions utilitaires.

```
 1 #
 2 #include "param.h"
 3 #include "mnemo.h"
 4 #include "pckx25.h"
 5 #include "net25.h"
 6 #include "open.h"
 7 #include "task.h"
 8 #include "uclx25.h"
 9 #include "drx25.h"
10 #include "routine.h"
11 #include "buffer.h"


12 /*
13  * Routine bufp:
14  *      - pour chaque appel, elle attribue un buffer c.a.d. une zone
15  *        de 140 bytes dont deux pointeurs, un compteur et 134 bytes
16  *        libres pour les informations.
17  *      - elle met à jour le chainage des buffers libres.
18  *      - si aucun buffer n'est libre, il y a appel de la routine
19  *        bufover() qui incrémentera si possible la zone des buffers.
20  *        Si c'est impossible, bufover() retourne -1 qui sera la valeur
21  *        retournee a la routine appelant bufp().
22  */


23 bufp(){
24         register  struct  bfres  *pntbf;
25         register  int  *presult;

26         pntbf = &bfres;
27         if(pntbf->bf_free == 0) {
28                 if(bufover() < 0) return(-1);
29         }
30         pntbf->bf_free--;
31         presult = pntbf->bf_first;
32         pntbf->bf_first = *presult;
33         *presult = 0;
34         return(presult);
35 }

36 /*
37  * Routine bufv(pt):
38  *      - replace un buffer libre pointé par 'pt' dans la file des
39  *        buffers libres.
40  *      - met a jour les compteurs.
41  */

42 bufv(pt)
43 int     *pt;
44 {
45         register  struct  bf  *pntbf;
```

```
46          register  struct  bfres  *pbfres;
47          register  struct  bf  *point;

48          pbfres = &bfres;
49          pntbf = pt;
50          pntbf->bf_pnt = pbfres->bf_first;
51          pbfres->bf_first = pntbf;
52          pbfres->bf_free++;
53          return;
54  }


55  /*
56   * Routine bufover():
57   *      - demande une incrémentation de la mémoire disponible pour
58   *        le processus a concurrence de NBFMAX * 140.
59   *      - l´incrémentation est appelee PAGE.
60   *      - divise l´incrémentation en buffers, les chaine entr´eux et
61   *        les ajoute à la chaine des buffers libres.
62   *      - retourne -1 si il y a erreur lors de la demande d´incrémenta
63   *        tion ou si on atteint le garde fou.
64   */


65  bufover(){
66          register  struct  bfres  *ptbfres;
67          register  int  *ptbf1;
68          register  struct  bf  *ptbf2;
69          char   *pnt;
70          int    i;
71          extern  int  errno;

72          ptbfres = &bfres;

73          if(ptbfres->bf_res >= NBFMAX) return(-1);
74          pnt = sbrk(PAGE);
75          if(pnt == -1){
76                  taskin(&ertrait, 29, ´EG´, 11, errno);
77                  return(-1);
78          }

79          ptbf2 = pnt;
80          ptbfres->bf_first = ptbf2;
81          for(i = 4; i > 0; i--){
82                  ptbf1 = ptbf2++;
83                  *ptbf1 = ptbf2;
84          }
85          ptbfres->bf_free =+ 5;
86          ptbfres->bf_res =+ 5;
87          return;
88  }

89  /*
90   * Routine select():
91   *      - sélectionne le premier élément de la file d´attente des travaux
92   *        qui sont prêts a continuer leur exécution.
93   *        (voir définition des travaux dans le chapitre 4).
```

```
 94 *      - incrémente toutes les priorités des travaux qui restent en
 95 *        en attente.
 96 *      - lance l'exécution du travail choisi.
 97 *      - boucle sur lui-meme.
 98 *
 99 * Cette routine est appelée par le programme principal et sa boucle
100 * s'exécutera jusqu'à la mort du processus.
101 */

102 select(){
103         register  int  *pointer;
104         register  int  *fonct;
105         register  struct  task  *pntk;
106
107  loop:
108         sch.sc_flag =| SEMA;
109         pointer = fonct = sch.sc_pnt;
110         sch.sc_pnt = pointer->tk_pnt;
111         pntk = sch.sc_pnt;
112         sch.sc_flag =& "SEMA;
113         while(pntk != 0) {
114                 pntk->tk_prio++;
115                 pntk = pntk->tk_pnt;
116         }
117         (*fonct->tk_job)(pointer->tk_arg[0], pointer->tk_arg[1], pointer->tk_arg[2])
118         if (fonct == travail) goto loop;
119         fonct->tk_pnt = sch.sc_free;
120         sch.sc_free = fonct;
121         goto loop;
122 }

123 /*
124  * Routine taskin():
125  *      - ajoute un travail dans la file d'attente pour sélection
126  *        par select (priorite < 30) ou pour sélection par le
127  *        temporisateur (priorite >= 30 et qui donne le temps pour
128  *        d'attente pour le temporisateur).
129  *      - ajoute suivant la priorité des autres.
130  *      - ranger les paramètres d'appel dans les zones prévues.
131  *      - signaler qu'il est impossible de rajouter un nouveau
132  *        travail et retourner alors -1.
133  */

134 taskin(job, prior, A1, A2, A3)
135 int     job, prior, A1, A2, A3;
136 {
137         register  int  *pntk;
138         register  int  *npnt;
139         register  int  *pointer;

140         npnt = sch.sc_free;
141         if(!npnt) goto err;
142         sch.sc_free = npnt->tk_pnt;

143         sch.sc_flag =| SEMA;

144         if(prior >= 30){
```

```
145                 prior =+ tim.tim_curt + 60;
146                 pntk = &tim.tim_pnt;
147         } else {
148                 pntk = &sch.sc_pnt;
149         }
150         pointer = *pntk;
151   loop:
152         if(!pointer){
153                 *pntk = npnt;
154                 *npnt = pointer;
155         } else {
156                 if(pointer->tk_prio >= prior){
157                         pntk = pointer;
158                         pointer = *pntk;
159                         sch.sc_flag =& "SEMA;
160                         goto loop;
161                 } else {
162                         *pntk = npnt;
163                         *npnt = pointer;
164                 }
165         }
166         sch.sc_flag =& "SEMA;
167         pointer = &npnt->tk_prio;

168         *pointer++ = prior;
169         *pointer++ = job;
170         *pointer++ = A1;
171         *pointer++ = A2;
172         *pointer = A3;
173         return(npnt);
174   err:
175         printf("saturation des taches. 0);
176         return;
177 }

178 /*
179  * Routine detimo():
180  *      - retirer ce travail de la file d'attente relative au
181  *        temporisateur.
182  *      - rajouter la zone libre dans la liste.
183  */

184
185 detimo(pointer)
186 int     *pointer;
187 {
188         register  int  *pntk;
189         register  int  *npnt;
190         register  int  *pntest;

191         pntk = &tim.tim_pnt;

192   loop :
193         if (!pntk) return(-1);
194         if (pntk == pntest) {
195                 sch.sc_flag =| SEMA;
196                 *npnt = *pntk;
```

```
197                 sch.sc_free = pntk;
198                 sch.sc_flag =& "SEMA;
199                 return(0);
200         }
201         npnt = pntk;
202         pntk = *npnt;
203         goto loop;
204 }

205 /*
206  * Routine timout():
207  *      - faire passer un travail de la file du temporisateur dans
208  *        la file des travaux choisis par select.
209  *      - les mettre en tête de cette file.
210  */

211 timout(){
212         register  struct  task  *pointer;
213         register  struct  task  *pnt;
214         register  int  temps;
215         struct  task  *savpnt;

216         pnt = tim.tim_pnt;
217         pointer = &tim;
218         temps = tim.tim_curt;

219   loop:
220         if(!(pnt)) return;
221         if(pnt->tk_prio > temps){
222                 pointer = pnt;
223                 pnt = pointer->tk_pnt;
224                 goto loop;
225         }
226         sch.sc_flag =| SEMA;
227         pointer->tk_pnt = 0;
228         savpnt = sch.sc_pnt;
229         sch.sc_pnt = pnt;
230         pointer = pnt;
231         while(pnt != 0) {
232                 pointer = pnt;
233                 pnt = pointer->tk_pnt;
234         }
235         pointer->tk_pnt = savpnt;
236         sch.sc_flag =& ~SEMA;
237         return;
238 }

239 /*
240  * Routine bipbip():
241  *      - routine de traitement de l'interruption provoquée par
242  *        le processus horloge.
243  *      - placer l'adresse de la routine timout() en première position
244  *        de la file de select().
245  *      - si le sémaphore de protection de cette file d'attente est mis,
246  *        rien n'est effectué.
247  */
```

```
248 bipbip(){
249         register  int  *pntk;

250         pntk = &sch;

251         if(pntk->sc_flag & SEMA) return;
252         travail[0].tk_pnt = pntk->sc_pnt;
253         pntk->sc_pnt = travail;
254         travail[0].tk_prio = 20;
255         tim.tim_curt =+ 30;
256         return;
257 }

258 /*
259  * Routine bfclean():
260  *       - retirer d'une des files d'attente pour expédition vers le
261  *         niveau trame, les buffers relatifs à une voie logique.
262  *       - chaque buffer possède dans un de ses pointeurs l'adresse
263  *         de la structure de sa voie logique.
264  *       - l'adresse de la structure de la voie logique et l'adresse de
265  *         la structure des entrées sorties vers le microprocesseur.
266  *       - les buffers libérés seront remis dans la file des buffers
267  *         libres.
268  */


269 bfclean(pointvl, pointr)
270 char    *pointvl, *pointr;
271 {
272         register  char  *vlpt;
273         register  int  *pnt1;
274         register  int  *pnt2;
275         struct  trame  *pttr;
276         int  i;

277         pttr = pointr;
278         pnt2 = pointr;
279         vlpt = pointvl;
280         i = 0;

281         pnt1 = &pnt2->tr_ptdat;
282         while(*pnt1){
283                 pnt2 = *pnt1;
284                 if(pnt2->bf_to == vlpt){
285                         i++;
286                         *pnt1 = *pnt2;
287                         bufv(pnt2);
288                 } else {
289                         pnt1 = pnt2;
290                 }
291         }
292         pttr->tr_cpt =- i;
293         return;
294 }


295 /*
```

```
296  *Routine ertrait():
297  *       - traitement des erreurs.
298  *       - actuellement se limite à afficher le type d'erreur à l'écran.
299  */

300 ertrait(par1, par2, par3)
301 int     par1, par2, par3;
302 {
303         register  char  *pnt;

304         pnt = &par1;
305         printf("err in %c_%c, type %d, arg: %d 0,pnt[1], pnt[0], par2, par3);
306         return;
307 }

308 /*
309  * Routine trastat()
310  *       - traitement du statut du niveau trame.
311  *       - actuellement, se contente de répondre et d'afficher à l'ecran.
312  */

313 trastat(pointer)
314 struct  trame  *pointer;
315 {
316         register  int  status;
317         register  int  *ptcmd;

318         status = pointer->tr_cmdi[0];
319         ptcmd = pointer->tr_cmdo;

320         pckx25(0, USER + REP, DDEBUT);
321
322         if (status & 02) *ptcmd = 0100002;
323         if (status & 01) *ptcmd = 0100001;
324         return;
325 }

326 /*
327  * Routine tempor():
328  *       - appelée lors de l'expiration du temporisateur.
329  *       - en fonction de la confirmation attendue, déclenche une
330  *         libération, une reprise.
331  */

332 tempor(nvl, idact)
333 int     nvl, idact;
334 {
335         if (idact == PCKA || idact == PCKR) {
336                 pckx25(nvl, USER + LIB, DTIMOUT);
337                 ecmdo(vl[nvl].v_uc, LIB, DTIMOUT);
338         } else {
339                 pckx25(0, USER + REP, DTIMOUT);
340         }
341         return;
342 }

343 /*
```

```
344  * Routine drclean():
345  *      - vide les files d'attente de l'expédition vers le niveau
346  *        trame.
347  *      - files des paquets de données et des paquets de commandes.
348  */

349 drclean()
350 {
351         register  struct  trame  *pttr;
352         register  int  *pnt;
353         register  int  *savpnt;
354         int  i;

355         pttr = trame;

356         for(i = 0; i < 2; i++){
357                 pnt = pttr->tr_ptcmd;
358                 pttr->tr_ptcmd = 0;
359                 while(pnt) {
360                         savpnt = pnt;
361                         bufv(savpnt);
362                         pnt = *pnt;
363                 }
364                 pnt = pttr->tr_ptdat;
365                 (pttr++)->tr_ptdat = 0;
366                 while(pnt){
367                         savpnt = pnt;
368                         bufv(savpnt);
369                         pnt = *pnt;
370                 }
371         }
372 }
```

Plusieurs routines qui n'ont pu encore être testées dans des conditions d'utilisation normales, ne sont pas reprises pour le moment dans ce fichier. Il s'agit des routines "newpck()" qui effectue les premiers tests sur les paquets en provenance du niveau trame, "sendcmd()" et "sendat()" qui ajoutent les paquets dans la file d'attente pour l'expédition vers le niveau trame. (Voir chapître 4 pour la description de ces files).

Annexe J: Les routines du protocole X25 paquet.

```
#

/* Ce fichier contient toutes les routines relatives au protocole X25
 * niveau paquet. A plusieurs reprises, elles font appel à des routines
 * d'autres fichiers : "endend.c" et "gerant.c".
 * Le protocole est représenté par une table sequentielle dont une version
 * française existe dans le fichier "protocol". Lors de l'initialisation
 * du moniteur reseau, ce fichier servira de base à la création d'une table
 * d'aiguillages : "fonc[2][020][012]". Toute modification dans ce fichier
 * aura une répercussion dans la table "fonc" et par le fait même dans la
 * séquence d'appel des routines de "pckx25.c".
 *
 * Les variables globales ainsi que les paramètres sont définis dans les
 * fichiers cités ci-dessous.
 */

                /* Fichier contenant les paramètres principaux et les codes
                 * pour la construction des paquets.
                 */
#include "param.h"
                /* Fichier contenant les valeurs des causes et des diagnostics
                 * pour les libérations, reprises et réinitialisations.
                 */
#include "mnemo.h"
                /* Fichier contenant les variables globales sur lesquelles
                 * travaille "pckx25.c". Entre autres la structure "vl".
                 */
#include "pckx25.h"
                /* Fichier contenant les variables globales servant à
                 * l'initialisation des voies logiques. Principalement
                 * la structure "net".
                 */
#include "net25.h"
                /* Fichier contenant la structure des données sur le fichier
                 * "admvl".
                 */
#include "open.h"
                /* Fichier contenant les structures pour la gestion des travaux.
                 */
#include "task.h"
                /* Fichier contenant les structures relatives à la communi-
                 * cation avec les utilisateurs. Dont la structure "ucl".
                 */
#include "uclx25.h"
                /* Fichier contenant les variables pour les entrées et sorties
                 * sur les moniteurs "trapac". Dont la structure "trame".
                 */
#include "drx25.h"
                /* Fichier contenant la liste complète de toutes les routines
                 * du moniteur réseau.
                 */
#include "routine.h"
                /* Fichier contenant la structure de gestion des buffers.
```

```
52                  */
53 #include "buffer.h"



54 /*
55  * Routine pckx25.c :
56         fonction : aiguiller la suite du travail vers la bonne routine selon :
57                 - l'origine : utilisateur ou réseau.
58                 - l'état dans lequel on se trouve. (voir table sequentielle).
59                 - ce qui est demandé par l'utilisateur ou le type de paquet
60                   à traiter.
61         Ces éléments sont représentés respectivement par les variables
62         locales 'i', 'k' et 'j'.
63
64         Cette routine peut être appelée soit par "newpck" ( paquet venant du
65         réseau), soit par "ecmdin" (demande d'action faite par l'utilisateur).

66         Le résultat retourné par la routine appelée sera lui-même renvoyé vers
67         la routine appelante.
68  *
69  */


70 pckx25 (nvl, action, arg1)
71 int     nvl, action, arg1;
72 {
73         register  int  i;
74         register  int  j;
75         register  int  k;
76         int     result;

77         if(action & USER) i = 1;
78         else i = 0;
79         j = action & 017;
80         k = vl[nvl].v_mode;

81         result = (*func[i][j][k])(nvl, arg1);
82         return(result);
83 }

84 /*
85  * Routine "pcallin" :
86         fonction :
87                 - Analyser un paquet d'appel entrant dans l'ETTD.
88                 - Demander au niveau supérieur de mettre en relation un
89                   processus utilisateur déjà créé avec cette voie logique.
90                 - Sauver l'adresse de l'ETTD appelant.
91                 - Demander l'envoi d'une confirmation d'appel si tout va bien
92                   ou une demande de libération de la voie logique si il y a
93                   un problème ou si l'ETTD est saturé.
94                 - Des ajoutes sont prévues pour le traitement des facilités
95                   ou des données incluses dans le paquet d'appel.
96         Variables locales :
97                 - ptvl : pointeur vers la structure de la voie logique
98                   prise en considération.
```

```
 99                - ptbuf : pointeur se déplacant a l'intérieur du paquet d'appel
100                - pnt : pointeur se déplacant dans l'image de la zone de sauvetage
101                  sur disque. Lors de la mise en relation d'un processus
102                  utilisateur et de la voie logique, la zone de sauvetage est
103                  amenée en memoire. Elle sera renvoyée lorsque tout sera fini.
104        Cette routine ne peut être appelée que par "pckx25".
105        Suivant les cas, elle fera appel à "plibo", "ecmdo" ou " pconfou".

106 *
107 */


108 pcallin(nvl, pointer)
109 int     nvl;
110                 /* Pointer est un pointeur vers le buffer contenant le paquet
111                  * d'appel. Ce buffer sera libéré à la fin du traitement.
112                  */
113 char    *pointer;
114 {
115         register  struct  vl  *ptvl;
116         register  char  *ptbuf;
117         register  char  *pnt;
118         int     result;
119
120         ptvl = &vl[nvl];
121         ptvl->v_mode = 03;
122                 /* On commence au début de la partie intéressante. */
123         ptbuf = &(pointer->pk1_ar1);

124                 /* Demande de mise en relation avec un processus libre. */

125         result = ecmdo(-nvl, CALL, 0);
126         if ( result < 0) {
127                         /* Il n'y en pas, saturation! */
128                 plibo(nvl, DSATUR);
129         } else {
130                 ptvl->v_uc = result ;
131                 pnt = ucl[result].uc_point;
132                 pnt = &pnt->deb_op;
133                         /* début du traitement des adresses. */
134                 pnt->op_aoln = (*ptbuf >> 4);
135                 pnt->op_ailn = (*ptbuf++ & 017);
136                 pointer->bf_to = ptbuf;
137                         /* "pnwad" s'occupera du transfert. */
138                 ptbuf = pointer->bf_to + pnwad(&pnt->op_ailn, pointer, pnwad(&pnt->op_
139                         /* Place prévue pour le traitement des facilités et
140                          * données spéciales.                                    */
141                 if (*ptbuf++) {
142                         if (!(net.n_facili)) {
143                                 result = DFACILI;
144                                 goto err;
145                         }
146                 }
147                 if (*ptbuf)  {
148                         result = DERRAPP;
149                         goto err;
150                 }
```

```
151                     /* Tout est valable, on accepte la communication. */
152             pconfou(nvl, PCKCA);
153             ecmdo(ptvl->v_uc, OKCALL, 0);
154             ptvl->v_mode = 010;
155             bufv(pointer);
156             return;
157         }
158             /* Il y a un problème, la communication est refusée; tout ce qui
159              * a été établi doit être libéré.                              */
160   err :
161         bufv(pointer);
162         ecmdo(ptvl->v_uc, LIB, result);
163         plibo(nvl, result);
164         return;
165 }

166 /*
167  * Routine "pnwad" :
168         fonction :
169                 - Transférer les adresses du paquet d'appel dans une zone
170                   prévue.
171                 - Tenir compte de la position du premier demi-octet.
172                 - Tenir compte de la position du dernier demi-octet et la
173                   signaler en retour.
174         Variables locales :
175                 - pnt1 : pointeur vers la zone réceptrice.
176                 - pnt2 : pointeur vers  la buffer contenant l'adresse.
177                   Le pointeur local de ce buffer (bf_to) pointe vers la zone
178                   adresse.
179                 - cpt1 : décomptera les demi-octets transmis. Il est initialisé
180                   par pnt1 car la longueur de l'adresse a déjà été rangée
181                   dans la zone réceptrice.

182         La routine renverra :
183                 0 si la place libre suivante commence au début d'un octet.
184                 1 si il y a encore un demi-octet non utilise.
185  *
186  */


187 pnwad( pointer1, pointer2, half)
188 int     *pointer1;
189 struct  bf  *pointer2;
190             /* "half" prévient si on commence au début d'un octet ou si
191              * le premier demi-octet est déjà occupé.                     */
192 int     half;
193 {
194         register  char  *pnt1;
195         register  char  *pnt2;
196         register        cpt1;

197         cpt1 = *pointer1++;
198         pnt1 = pointer1;
199             /* Faire pointer pnt2 vers le début de la zone adresse. */
200         pnt2 = pointer2->bf_to;

201   loop :
```

```
202         if (cpt1 > 1) {
203                 if (half) {
204                         *pnt1 = (*pnt2++ & 017);
205                         *pnt1++ =| (*pnt2 & 0360);
206                 } else {
207                         *pnt1++ = *pnt2++;
208                 }
209                 cpt1 =- 2;
210                 goto loop;
211         }

212         if (cpt1 == 1)  {
213                 if (half) *pnt1 = (*pnt2++ & 017);
214                 else *pnt1 = (*pnt2 >> 4) & 017;
215         }

216         pointer2->bf_to = pnt2;
217         return(cpt1 /\ half);
218 }


219 /*
220  * Routine "pcallout" :
221         fonction :
222                 - Construire un paquet d'appel.
223                 - Demander un buffer libre pour y construire ce paquet.
224                 - Y installer l'entête.
225                 - Transférer la longueur des adresses, puis faire appel à une
226                   routine qui les transfèrera.
227                 - Armer le temporisateur et donner le paquet a "sencmd" pour
228                   expédition.
229         variables locales :
230                 - ptcal : pointeur dans l'image des données d'envoi sauvées
231                   sur disques.
232                 - crpt : pointeur dans le buffer pour la construction du paquet.
233                 - vlpt : pointeur vers la voie logique concernee.
234                 - ptbuf : pointeur vers le buffer.
235         cette routine retournera :
236                 - 0 si tout va bien;
237                 - -1 si on ne peut envoyer de paquet d'appel.
238  *
239  */


240 pcallout(nvl, pointer)
241 int     nvl;
242 char    *pointer;
243 {
244         register  char  *ptcal;
245         register  char  *crpt;
246         register  struct  vl  *vlpt;
247         int     result;
248         char    *ptbuf;

249         ptcal = pointer + 6 + RQLNG;
250         vlpt = &vl[nvl];
251
```

```
252                 /* Demande d'un buffer. */
253         crpt = bufp();
254         if (crpt == -1) return(-1);


255         ptbuf = crpt;
256         crpt = crpt->bf_buf;
257         crpt =+ 2;
258
259         *crpt++ = HEADPCK;
260         *crpt++ = nvl;
261         *crpt++ = PCKA;

262         *crpt = ptcal->op_aoln;
263         *crpt =<< 4;
264         *crpt++ =| (ptcal->op_ailn & 017);
265         ptbuf->bf_to = crpt;
266         crpt = ptbuf->bf_to + paddr(&ptcal->op_ailn, ptbuf, paddr(&ptcal->op_aoln, pt

267         *crpt++ = 0;
268         *crpt++ = 0;

269         ptbuf->bf_cpt = crpt - ptbuf->bf_buf;
270         ptbuf->pk1_lng = ptbuf->bf_cpt++ - 2;
271         ptbuf->bf_cpt =& 0377776;
272         ptbuf->bf_to = ptbuf->bf_buf;

273         vlpt->v_tim = taskin(&tempor, 180, nvl, PCKA);
274         vlpt->v_mode = 02;
275         vlpt->v_uc = ptcal->op_uc;

276         sendcmd(vlpt->v_bufout, ptbuf);
277         return(0);
278 }


279 /*
280  * Routine "paddr" :
281         fonction :
282                 - Transférer les adresses d'une zone donnée vers le paquet
283                   en construction.
284                 - Les longueurs ont déjà été transférées.
285                 - Tenir compte de l'endroit ou commence la place libre.
286                 - Retourner soit 0 si la place libre suivante commence au debut
287                   d'un octet, soit 1 si elle commence au milieu d'un octet.
288         Variables locales :
289                 - pnt1 : pointeur dans la zone donnée.
290                 - pnt2 : pointeur dans le paquet en construction, zone réceptrice.
291                 - cpt : contenant la longueur en demi-octets.
292  *
293  */


294 paddr(point1, point2, half)
295 char    *point1, *point2;
296                 /* "half" prévient si on commence au début d'un octet ou au milieu.
297 int     half;
```

```
298 {
299         register  char  *pnt1;
300         register  char  *pnt2;
301         register  int   cpt;

302         pnt1 = point1;
303         cpt = *pnt1;
304         pnt1 =+ 2;

305         cpt =& 017;
306         pnt2 = point2->bf_to;

307   loop:
308         if (cpt > 1) {
309                 if (half) {
310                         *pnt2++ =| (*pnt1 & 017);
311                         *pnt2 = (*pnt1++ & 0360);
312                 } else {
313                         *pnt2++ = *pnt1++;
314                 }
315                 cpt =- 2;
316                 goto loop;
317         }
318         if (cpt == 1) {
319                 if (half) *pnt2++ =| (*pnt1 & 017);
320                 else *pnt2 = (*pnt1 << 4);
321         }
322         point2->bf_to = pnt2;
323         return(cpt ^ half);
324 }

325 /*
326  * Routine "pokcall" :
327         fonction :
328                 - Réceptionner une confirmation d'appel.
329                 - Faire évoluer l'état dans la table sequentielle.
330                 - Prévenir le niveau supérieur et le processus utilisateur.
331                 - Désarmer le temporisateur.
332         Cette routine fait appel a :
333                 - "ecmdo" pour prévenir le niveau superieur.
334                 - "detimo" pour désarmer le temporisateur.
335                 - "bufv" pour libérer le buffer contenant le paquet.
336  *
337  */


338 pokcall(nvl, pointer)
339 int     nvl;
340 char    *pointer;
341 {
342         detimo(vl[nvl].v_tim);
343         vl[nvl].v_tim = 0;
344         vl[nvl].v_mode = 010;
345         bufv(pointer);
346         ecmdo(vl[nvl].v_uc, OKCALL);
347         return;
348 }
```

```
349 /*
350  * Routine "plibin" :
351         fonction :
352                 - Réceptionner une demande de libération.
353                 - En extraire la cause et le diagnostic pour les transmettre
354                   au niveau superieur.
355                 - Prévoir le nettoyage de la voie logique correspondante en
356                   appelant la routine "pclean".
357                 - Prévenir le niveau supérieur par "ecmdo".
358                 - Acquitter la demande de libération en envoyant une confirmation
359                   par la routine "pconfou".
360                 - Remettre l'état de la table sequentielle en position initiale.
361         Variables locales :
362                 - vlpt : pointeur vers la structure de la voie logique courante.
363                 - cause : entier qui recevra la cause et le diagnostic de la
364                   libération.
365  *
366  */


367 plibin(nvl, pointer)
368 int     nvl;
369 char    *pointer;
370 {
371         register  struct  vl  *vlpt;
372         register  int  cause;

373         vlpt = &vl[nvl];
374         cause = (pointer->pk1_ar1 << 8) + (pointer->pk1_ar2 & 0377);

375         vlpt->v_mode = 07;

376         bfclean(vlpt, vlpt->v_bufout);
377         bufv(pointer);
378         pconfou(nvl, PCKCL);
379         ecmdo(vlpt->v_uc, LIB, cause);
380         pclean(vlpt);
381
382         vlpt->v_mode = 0;
383 }
384 /*
385  * Routine "plibo" :
386         fonction :
387                 - Préparer l'emission d'une demande de libération.
388                 - Prévoir le nettoyage de la structure de la voie logique
389                   concernee.
390                 - Armer un temporisateur.
391                 - faire evoluer l'etat de la table sequentielle.
392         Variables locales :
393                 - vlpt : pointeur vers la structure de voie logique courante.
394  *
395  */



396 plibo(nvl, cause)
397                 /* cause contient la cause de la libération. Sur un demi-octet. */
```

```
398 int     nvl, cause;
399 {
400         register  struct  vl  *vlpt;
401
402         vlpt = &vl[nvl];
403
404         bfclean(vlpt, vlpt->v_bufout);
405         pdemout(nvl, PCKL, cause & 0377);
406         vlpt->v_tim = taskin(&tempor, 30, nvl, PCKL);
407         pclean(vlpt);
408         vlpt->v_mode = 06;
409 }

410 /*
411  * Routine "plibed"
412         fonction :
413                 - Réceptionner une confirmation de liberation.
414                 - Faire évoluer l'état de la table sequentielle en le remettant
415                   dans la position initiale.
416                 - Libérer le buffer contenant le paquet.
417         Variable locale :
418                 - vlpt : pointeur vers la voie logique courante.
419  *
420  */


421 plibed(nvl, pointer)
422 int     nvl;
423 char    *pointer;
424 {
425         register  struct  vl  *vlpt;

426         vlpt = &vl[nvl];
427         bufv(pointer);
428         detimo(vl[nvl].v_tim);
429         vlpt->v_tim = 0;
430         vl[nvl].v_mode = 0;
431 }


432 /*
433  * Routine "prepin"
434         fonction :
435                 - Réceptionner une demande de reprise.
436                 - Appeler la routine "prepris" pour le signaler à tous les
437                   utilisateurs.
438                 - Remettre l'état de la table sequentielle dans la position
439                   initiale.
440                 - Envoyer une confirmation de reprise.
441                 - Extraire la cause de cette reprise pour la transmettre au
442                   niveau superieur.
443  *
444  */

445 prepin(nvl, pointer)
446 int     nvl;
447 char    *pointer;
```

```
448 {
449         register  int  cause;
450         register  struct  vl  *vlpt;
451         register  int  i;

452         cause = (pointer->pk1_ar1 << 8) + (pointer->pk1_ar2);
453         vlpt = vl;
454         for(i = 0; i <= 15; i++)
455                 (vlpt++)->v_mode = 0;
456         prepris(cause);
457         bufv(pointer);
458         pconfou(0, PCKCREP);
459         return;
460 }

461 /*
462  * Routine "prepout"
463         fonction :
464                 - Préparer l'emission d'une demande de reprise.
465                 - Armer le temporisateur.
466                 - Appeler la routine "prepris" pour le signaler aux autres
467                   utilisateurs.
468                 - Faire évoluer l'état de la table séquentielle.
469  *
470  */



471 prepout(nvl, cause)
472 int     nvl, cause;
473 {
474         register  struct  vl  *vlpt;
475         register  int  i;

476         prepris(cause);
477         vlpt = vl;
478         vlpt->v_tim = taskin(&tempor, 30, nvl, PCKREP);
479         for(i = 0; i <= 15; i++)
480                 (vlpt++)->v_mode = 01;
481         pdemout(0, PCKREP, cause & 0377);
482 }


483 /*
484  * Routine "preped"
485         fonction :
486                 - Réceptionner une confirmation de reprise.
487                 - Faire évoluer l'état de la table sequentielle en le remettant
488                   en position initiale.
489                 - Libérer le buffer contenant le paquet.
490                 - Désarmer le temporisateur.
491  *
492  */


493 preped(nvl, pointer)
494 int     nvl;
```

```
495 char    *pointer;
496 {
497         register  struct  vl  *vlpt;
498         register  int  i;

499         vlpt = vl;
500         detimo(vlpt->v_tim);
501         for(i = 0; i <= 15; i++)
502                 (vlpt++)->v_mode = 0;
503         bufv(pointer);
504         return;
505 }


506 /*
507  * Routine "prepris"
508         fonction :
509                 - Prévenir chaque utilisateur q'une reprise est en cours.
510                   Appel de la routine "ecmdo".
511                 - Nettoyer tout ce qui est relatif a chaque voie logique
512                   occupée.
513                 - Afficher au terminal principal qu'il y a une reprise.
514         Variables locales :
515                 - vlpt : pointeur vers les structures des voies logiques.
516                 - cause : cause et diagnostic de la reprise.
517  *
518  */


519 prepris(cause)
520 int     cause;
521 {
522         register  struct  vl  *vlpt;
523         register  int  i;
524
525         vlpt = vl;
526         drclean();
527         for(i = 0; i <= 15; i++){
528                 ecmdo(vlpt->v_uc, LIB, cause);
529                 bfclean(vlpt, vlpt->v_bufout);
530                 pclean(vlpt++);
531         }
532         printf(" OREPRISE 0 ");
533         return;
534 }
535 /*
536  * Routine "pdatout"
537         fonction :
538                 - Vérifier qu'on peut lancer un paquet de donnees. Si non le
539                   retourner 1.
540                 - Vérifier la compatibilite BITM et BITQ, si nécessaire lancer
541                   un paquet d'appel vide pour obtenir cette compatibilité.
542                 - Tester si on demande l'option BITM et vérifier si elle peut
543                   être appliquée en regardant la longueur de la zone donnée.
544                 - Tester si on demande l'option BITQ.
545                 - Construire le contrôle de flux actuel.
546                 - Voir si c'est le dernier paquet qu'on peut envoyer.
```

```
547                     Positionner les bits en fonction.
548                 - Donner le paquet a "sendat" pour expédition.
549         Variables locales :
550                 - vlpt : pointeur vers la structure de la voie logique
551                   correspondante.
552                 - pnt : pointeur vers le buffer contenant le paquet de données.
553                 - crpt : pointeur vers le buffer pour l'expédition d'un paquet
554                   vide.
555         Cette routine retournera les valeurs suivantes :
556                 - 0 si tout va bien.
557                 - 1 si le paquet n'a pas été envoyé .
558 *
559 */


560 pdatout (nvl, pointer)
561 int     nvl;
562 char    *pointer;
563 {
564         register  struct  vl  *vlpt;
565         register  char  *pnt;
566         register  int  result;
567         char    *crpt;

568         vlpt = &vl[nvl];
569         pnt = pointer;
570         if (vlpt->v_usflag & (STOPOU | RNROU)) return(1);
571         if (vlpt->v_usflag & BITM) {
572                 if (exor((vlpt->v_usflag & BITQ), (pnt->pk1_lng & 0177400))){
573                         crpt = bufp();
574                         if (crpt == -1) {
575                                 return(1);
576                         }
577                         crpt->bf_cpt = 0;
578                         crpt->pk1_lng = 0;
579                         pdatout (nvl, crpt);
580                 }
581         }
582         if ((pnt->bf_cpt == LNGMAX) && (pnt->pk1_lng & 0377)){
583                 vlpt->v_usflag =| BITM;
584                 pnt->pk1_act = 020;
585         } else {
586                 vlpt->v_usflag =& ~BITM;
587                 pnt->pk1_act = 0;
588         }

589         if (pnt->pk1_lng & 0177400) {
590                 vlpt->v_usflag =| BITQ;
591                 pnt->pk1_head = 0200;
592         } else {
593                 vlpt->v_usflag =& ~BITQ;
594                 pnt->pk1_head = 0;
595         }

596         result = vlpt->v_rack & 07;
597         result =<< 4;
598         result =| vlpt->v_send & 07;
```

```
599         result =<< 1;
600         vlpt->v_send ++;
601         vlpt->v_send =& 07;
602
603         pnt->pk1_lng = pnt->bf_cpt + 3;
604         pnt->bf_cpt =+ 5;
605         pnt->pk1_head =| HEADPCK;
606         pnt->pk1_act =| result;
607         pnt->pk1_vlid = nvl;

608         sendat(vlpt->v_bufout, pnt);
609         result = vlpt->v_sack + vlpt->v_dimfen;
610         if (result <= vlpt->v_send) vlpt->v_usflag =| STOPOU;
611         return(0);
612 }


613 /*
614  * Routine "pdatin"
615         fonction :
616                 - Réceptionner un paquet de données.
617                 - Vérifier si il est compatible pour ce qui est des BITQ et BITM.
618                 - Tester si l´une ou l´autre option est demandée; en tenir
619                   compte pour prévenir l´utilisateur.
620                 - Vérifier le contrôle de flux pour ce qui est du numéro d´entrée
621                   de ce paquet.
622                 - Faire appel à la routine "packdin" pour vérifier si le numéro
623                   d´acquittement est bon.
624                 - En fonction de ces tests, passer ce paquet au niveau supérieur
625                   ou déclencher une procédure de réinitialisation.
626                 - Suivant le résultat de la mise en file d´attente ("eadata"),
627                   bloquer le contrôle de flux(fenêtre), la faire évoluer lente-
628                   ment, ou normalement.
629         Variables locales :
630                 - vlpt : pointeur vers la structure de voie logique courante.
631                 - pnt : pointeur vers le buffer contenant le paquet.
632                 - num : variable auxiliaire pour le contrôle de flux.
633         Cette routine fera appel a :
634                 - "preinou" pour déclencher une procédure de réinitialisation.
635                 - "ecmdo" pour le signaler au niveau supérieur.
636                 - "eadata" pour ajouter ce paquet à la file d´attente en direction
637                   de l´utilisateur.
638                 - "packdou" pour envoyer un acquit ou une demande de blocage.
639                   (paquet RR ou RNR).
640                 - "exor" pour effectuer un ´ou exclusif" entre les deux
641                   arguments.
642  *
643  */

644 pdatin(nvl, pointer)
645 int     nvl;
646 char    *pointer;
647 {
648         register  struct  vl  *vlpt;
649         register  char *pnt;
650         register  int  num;
651         char    *crpt;
```

```
652         int     result;

653         vlpt = &vl[nvl];
654         pnt = pointer;

655         result = pnt->pk1_head & 0200;

656         if (vlpt->v_usflag & MBIT) {
657                 if (exor((vlpt->v_usflag & QBIT), result))
658                         goto bad;
659         }
660         vlpt->v_usflag =& ~(MBIT | QBIT);
661         if (pnt->pk1_act & 020) {
662                 if (pnt->pk1_lng != LNGMAX) goto bad;
663                 pnt->pk1_lng = 0377;
664         } else {
665                 pnt->pk1_lng = 0;
666         }
667         if (result) {
668                 vlpt->v_usflag =| QBIT;
669                 pnt->pk1_lng =+ 0177400;
670         } else {
671                 vlpt->v_usflag =& ~QBIT;
672         }

673         if ((packdin(vlpt, pnt->pk1_act)) < 0) goto bad;

674         num = pnt->pk1_act & 016;
675         num =>> 1;
676         if (num != vlpt->v_recu++) goto bad;
677         vlpt->v_recu =& 07;
678         result = vlpt->v_rack + vlpt->v_dimfen;
679         if(num < vlpt->v_rack) result =& 07;
680         if( num >= result) goto bad;

681         pnt->bf_cpt =- 3;
682         pnt->bf_to = &pnt->pk1_ar1;
683         result = eaddata(vlpt->v_uc, pnt);
684         if (result < 0) {
685                 vlpt->v_usflag =| STOPIN;
686                 packdou(nvl, PCKPRNR);
687                 return;
688         } else {
689                 vlpt->v_usflag =& ~(STOPIN | LOWIN);
690         }
691         if (!result) vlpt->v_usflag =| LOWIN;
692         packdou(nvl, PCKPRR);
693         return(0);
694   bad:
695         bufv(pnt);
696         preinou(nvl, DFLUX);
697         ecmdin(vlpt->v_uc, VIDVIT, DFLUX);
698         return(0);
699 }

700 /*
701  * Routines "prrout" et "prnrout"
```

```
702         fonction :
703                 - Préparer l'appel de "packdou" pour l'envoi d'un paquet
704                   RR ou RNR.
705  *
706  */


707 prrout(nvl)
708 int     nvl;
709 {
710         packdou(nvl, PCKPRR);
711 }
712 prnrout(nvl)
713 int     nvl;
714 {
715         packdou(nvl, PCKPRNR);
716 }

717 /*
718  * Routine "pfcin"
719         fonction :
720                 - Réceptionner un paquet de contrôle de flux RNR ou RR.
721                 - Vérifier le contrôle de flux pour l'acquit des paquets envoyés
722                   en appelant la routine "packdin".
723                 - Déclencher une reinitialisation si nécessaire. Prévenir l'
724                   utilisateur.
725                 - Tester si on demande un blocage (RNR) de la transmission ou
726                   de reprendre(ou continuer) celle-ci. (RR).
727                 - Agir en fonction, positionner les bits necessaires.
728                 - Libérer le buffer contenant le paquet.
729  *
730  */

731 pfcin(nvl,pointer)
732 int     nvl;
733 char    *pointer;
734 {
735         register  struct  vl  *vlpt;
736         register  char  *pnt;
737
738         pnt = pointer;
739         vlpt = &vl[nvl];
740         if (packdin(vlpt, pnt->pk1_act) < 0) {
741                 preinou(nvl, DFLUX);
742                 ecmdin(vlpt->v_uc, VIDVIT, DFLUX);
743         } else {
744                 if ((pnt->pk1_act & 017) == 05) vlpt->v_usflag =| RNROU;
745                 else vlpt->v_usflag =& ~RNROU;
746         }
747         bufv(pnt);
748         return;
749 }


750 /*
751  * Routine "packdin"
752         fonction :
```

```
753                 - Vérifier la validite du contrôle de flux en ce qui concerne
754                   l'acquit des données déjà envoyées et uniquement ce contrôle.
755                 - La vérification est faite en testant si le numéro n'est
756                   pas dans une zone interdite.
757         Cette routine renverra :
758                 - 0 si tout va bien.
759                 - -1 si le contrôle est faux.
760  *
761  */


762 packdin(pointvl, num)
763 char    *pointvl;
764 char    num;
765 {
766         register  struct  vl  *vlpt;
767         register  int  nsen;

768         vlpt = pointvl;
769         nsen = num & 0340;
770         nsen =>> 5;

771         if(vlpt->v_sack < vlpt->v_send){
772                 if(nsen < vlpt->v_sack || nsen > vlpt->v_send) return(-1);
773         } else {
774                 if(nsen < vlpt->v_sack && nsen > vlpt->v_send) return(-1);
775         }

776         if (vlpt->v_sack < nsen) {
777                 vlpt->v_sack = nsen;
778                 vlpt->v_usflag =& ~STOPOU;
779         }

780         return(0);
781 }


782 /*
783  * Routine "packdou"
784         fonction :
785                 - Créer les paquets de contrôle de flux RR ou RNR.
786                 - Calculer les nouvelles valeurs de ce contrôle.
787                 - Modifier le contrôle de flux de paquets de données précédents
788                   et non encore envoyés.
789                 - Donner le paquet "sendcmd" pour expédition.
790  *
791  */



792 packdou(nvl, id)
793 int     nvl, id;
794 {
795         register  struct  vl  *vlpt;
796         register  char  nrec;
797         register  char  *pnt;
```

```
798         vlpt = &vl[nvl];
799         if (!(vlpt->v_usflag & STOPIN)) {
800                 if (vlpt->v_usflag & LOWIN) vlpt->v_rack++;
801                 else {
802                         vlpt->v_rack = vlpt->v_recu;
803                 }
804         }

805         nrec = vlpt->v_rack;
806         nrec =<< 5;

807         pnt = (vlpt->v_bufout)->tr_ptdat;
808         if(pnt){
809                 do {
810                         if (pnt->bf_to == vlpt) {
811                                 pnt->pk1_act =& 037;
812                                 pnt->pk1_act =| nrec;
813                         }
814                         pnt = pnt->bf_pnt;
815                 } while (pnt);
816                 if (id == PCKPRR) return;
817         }
818         pnt = bufp();
819         if (pnt == -1) {
820                 taskin(&packdou, 30, nvl, id);
821                 return;
822         }
823
824         pnt->bf_cpt = 6;
825         pnt->pk1_lng = 3;
826         pnt->pk1_head = HEADPCK;
827         pnt->pk1_vlid = nvl;
828         pnt->pk1_act = id & 017;
829         pnt->pk1_act =| (nrec & 0160);

830         sendcmd(vlpt->v_bufout, pnt);
831         return(0);
832 }

833 /*
834  * Routine preinou:
835         fonction :
836                 - envoyer, sur demande de l'utilisateur, une demande de
837                 réinitialisation.
838                 - remettre les compteurs du contrôle de flux a 0.
839                 - nettoyer les files d'attente des buffers dependant de
840                 cette voie logique.
841                 - demander l'envoi d'une confirmation.
842                 - faire évoluer la table sequentielle.
843  *
844  */


845 preinou(nvl, cause)
846 int     nvl, cause;
847 {
848         register  struct  vl  *vlpt;
```

```
849         vlpt = &vl[nvl];
850         bfclean(vlpt, vlpt->v_bufout);
851         pdemout(nvl, PCKR, cause & 0377);

852         vlpt->v_send = 0;
853         vlpt->v_sack = 0;
854         vlpt->v_recu = 0;
855         vlpt->v_rack = 0;
856         vlpt->v_usflag =& ~(STOPIN | STOPOU | LOWIN | INTOUT | BITM | MBIT | BITQ | Q

857         vlpt->v_mode = 011;
858         vlpt->v_tim = taskin(&tempor, 30, nvl, PCKR);
859 }


860 /*
861  * Routine preined:
862         fonction:
863                 - faire évoluer la table sequentielle lors de la
864                 réception d'une confirmation de réinitialisation.

865  *
866  */


867 preined(nvl, pointer)
868 int     nvl;
869 char    *pointer;
870 {
871         register  struct  vl  *vlpt;

872         vlpt = &vl[nvl];

873         detimo(vlpt->v_tim);
874         vlpt->v_tim = 0;
875         vlpt->v_mode = 010;
876         bufv(pointer);
877         return;
878 }


879 /*
880  * Routine preinin:
881         fonction: réceptionner une demande de réinitialisation venant
882                 du réseau.
883
884                 - remettre a 0 les compteurs du contrôle de flux.
885                 - enlever des files d'attente les buffers de cette voie
886                 logique.
887                 - prévenir le niveau superieur.
888                 - demander l'envoi d'une confirmation.
889  *
890  */


891 preinin(nvl, pointer)
```

```
892 int     nvl;
893 char    *pointer;
894 {
895         register  struct  vl  *vlpt;

896         register  int  cause;

897         vlpt = &vl[nvl];
898         bfclean(vlpt, vlpt->v_bufout);

899         vlpt->v_send = 0;
900         vlpt->v_sack = 0;
901         vlpt->v_recu = 0;
902         vlpt->v_rack = 0;
903         vlpt->v_usflag =& ~(STOPIN | STOPOU | LOWIN | INTOUT | MBIT | BITM | QBIT | E

904         cause = (pointer->pk1_ar1 << 8) + (pointer->pk1_ar2 & 0377);
905         bufv(pointer);
906         pconfou(nvl, PCKCR);
907         ecmdo(vlpt->v_uc, VIDVIT, cause);
908         return;
909 }


910 /*
911  * Routine pintout
912         fonction:
913                 - vérifier si on peut envoyer.
914                   sinon mettre en file d'attente.
915                 - réserver un buffer pour la construction.
916                 - construire un paquet d'interruption avec la donnée
917                 fournie par le niveau superieur.
918                 - mettre le flag en position pour éviter tout nouvel
919                 envoi.
920                 - mettre ce buffer dans la file.
921  *
922  */


923 pintout(nvl, arg)
924 int     nvl, arg;
925 {
926         register  struct  vl  *vlpt;
927         register  char  *crpt;

928         vlpt = &vl[nvl];
929         if(vlpt->v_usflag & INTOUT) goto bad;
930         crpt = bufp();
931         if(crpt == -1) goto bad;
932         crpt->bf_cpt = 6;
933         crpt->pk1_lng = 4;
934         crpt->pk1_head = HEADPCK;
935         crpt->pk1_vlid = nvl;
936         crpt->pk1_act = PCKIT;
937         crpt->pk1_ar1 = arg;
938
939         sendcmd(vlpt->v_bufout, crpt);
```

```
940
941         vlpt->v_usflag =| INTOUT;
942         return(0);
943    bad:
944         taskin(&pintout, 30, nvl, arg);
945         return(0);
946 }


947 /*
948  * Routine pinted:
949         fonction:
950                 - réceptionner une confirmation d'interruption.
951                 - enlever le flag interdisant l'envoi de donnees
952                 d'interruption.
953  *
954  */


955 pinted(nvl, pointer)
956 int     nvl;
957 char    *pointer;
958 {
959         register  struct  vl  *vlpt;

960         vlpt = &vl[nvl];
961
962         vlpt->v_usflag =& ~INTOUT;
963         bufv(pointer);
964         return;
965 }


966 /*
967  * Routine pintin:
968         fonction:
969                 - réceptionner un paquet d'interruption.
970                 - demander envoi confirmation.
971                 - transmettre la donnée au niveau supérieur.
972
973  *
974  */


975 pintin(nvl, pointer)
976 int     nvl;
977 char    *pointer;
978 {
979         register  int  argu;
980         register  struct  vl  *vlpt;

981         vlpt = &vl[nvl];

982         argu = pointer->pk1_ar1;
983         bufv(pointer);
984         pconfou(nvl, PCKCIT);
985         ecmdo(vlpt->v_uc, INT, argu);
```

```
986         return;
987 }


988 /*
989  * Routine pdemout
990         fonction:
991                 - construire un paquet de demandes avec l'argument fourni.
992                 - mettre en file d'attente s'il n'y a pas de buffer libre
993                 pour la construction.

994  *
995  */

996 pdemout(nvl, id, ar1)
997 char    nvl, id;
998 int     ar1;
999 {
1000         register  char  *crpt;

1001         crpt = bufp();
1002         if (crpt == -1){
1003                 taskin(&pdemout, 30, nvl, id, ar1);
1004                 return;
1005         }
1006
1007         crpt->bf_cpt = 8;
1008         crpt->pk2_lng = 5;
1009         crpt->pk2_head = HEADPCK;
1010         crpt->pk2_vlid = nvl;
1011         crpt->pk2_act = id;
1012         crpt->pk1_ar1 = 0;
1013         crpt->pk1_ar2 = ar1;
1014         sendcmd(vl[nvl].v_bufout, crpt);
1015         return;
1016 }


1017 /*
1018  * Routine pconfou
1019         fonction:
1020                 - construire un paquet de confirmation suivant
1021                 le paramètre d'appel.
1022  *
1023  */



1024 pconfou(nvl, id)
1025 char    nvl, id;
1026 {
1027         register  char  *crpt;

1028         crpt = bufp();
1029
1030         crpt->bf_cpt = 6;
1031         crpt->pk1_lng = 3;
```

```
1032            crpt->pk1_head = HEADPCK;
1033            crpt->pk1_vlid = nvl;
1034            crpt->pk1_act = id;
1035            sendcmd(vl[nvl].v_bufout, crpt);
1036            return;
1037    }


1038    /*
1039     * Routine pclean
1040            fonction:
1041                    - remettre les variables de la voie logique dans
1042                    l'état initial.
1043     *
1044     */

1045    pclean(pointvl)
1046    char    *pointvl;
1047    {
1048            register  int  *vlpt;

1049            vlpt = pointvl;
1050
1051            vlpt->v_send = 0;
1052            vlpt->v_sack = 0;
1053            vlpt->v_recu = 0;
1054            vlpt->v_rack = 0;
1055            vlpt->v_tim = 0;
1056            vlpt->v_usflag = 0;
1057            vlpt->v_facili = 0;
1058            vlpt->v_uc = -1;
1059    }


1060    /*
1061     * Routine notrait :
1062            fonction:
1063                    - réceptionner un paquet venant du réseau et ne rien en
1064                    faire. (cas attente confirmation libération,...).
1065     *
1066     */


1067    pnotrait (nvl, pointer)
1068    int     nvl;
1069    char    *pointer;
1070    {
1071            bufv(pointer);

1072    }


1073    /*
1074     * Routine pnojob
1075            fonction:
1076                    - refuser d'exécuter une action demandée par le niveau
1077                    supérieur.
```

```
1078  *
1079  */


1080 pnojob()
1081 {
1082         return(-1);
1083 }


1084 /*
1085  * Routine pckerr
1086         fonction:
1087                 -réceptionner un paquet qui est faux pour le protocole
1088                 X25 paquet.
1089                 - déclencher une erreur, une libération.
1090                 - prévenir le niveau supérieur.
1091  *
1092  */



1093 pckerr(nvl, pointer)
1094 int     nvl;
1095 char    *pointer;
1096 {
1097         bufv(pointer);
1098         plibo(nvl, DPROTO);
1099         if (vl[nvl].v_uc < 0) return(-1);
1100         ecmdin(vl[nvl].v_uc, LIB, DPROTO);
1101         return(-1);
1102 }
```

Annexe K: Les fonctions de liaison avec l'utilisateur.

```
#
#include "param.h"
#include "mnemo.h"
#include "pckx25.h"
#include "net25.h"
#include "open.h"
#include "task.h"
#include "uclx25.h"
#include "drx25.h"
#include "routine.h"
#include "buffer.h"
/*
 * Routine ewrite():
 *      - sortir un buffer de la file d'attente pour expédition vers
 *        l'utilisateur.
 *      - passer ce buffer a la routine ewrite qui se chargera de
 *        l'introduire dans le moniteur de communication inter processus.
 *      - éventuellement prévenir le niveau paquet qu'il y a de la place.
 */
ewrite(id)
int     id;
{
        register  struct  ucl  *ptucl;
        register  int  *pntbf;

        ptucl = &ucl[id];
        pntbf = ptucl->uc_bufin;
        if (pntbf->bf_pnt) {
                ptucl->uc_bufin = pntbf->bf_pnt;
                bufv(pntbf);
                pntbf = ptucl->uc_bufin;
                if (*pntbf == 0) {
                        taskin(&pckx25, priority[3], ptucl->uc_nvl, USER + RR);
                }
                uwrite(id, pntbf);
        }
        return;
}
/*
 * Routine eaddata():
 *      - recevoir un paquet du niveau paquet.
 *      - le ranger dans la file d'attente.
 *      - compter le nombre de paquets.
 *      - bloquer, ralentir ou laisser aller l'arrivée des paquets.
 */
eaddata(nucl, pointer)
int     nucl;
int     *pointer;
{
        register  struct  ucl  *ptucl;
        register  int  i;
        register  *pnt;
```

```
52          i = 0;
53          ptucl = &ucl[nucl];
54          pnt = &ptucl->uc_bufout;
55          while (*pnt) {
56                  pnt = *pnt;
57                  i++;
58          }
59          *pnt = pointer;
60          *pointer = 0;
61          if (i) {
62                  if (i > TROP) return(0);
63                  if (i > SATURATION) return(-1);
64          } else {
65                  uwrite(nucl, pointer);
66          }
67          return(1);
68  }
69  /*
70   * Routine eread():
71   *      - appelée pour traiter les données-informations reçues de
72   *        l'utilisateur.
73   *      - éliminer le paquet si on ne peut plus envoyer.
74   *      - proposer au niveau paquet (PCKX25) d'envoyer ce paquet.
75   *      - s'il refuse se mettre en file d'attente.
76   *      - s'il accepte relancer le travail de lecture sur le moniteur
77   *        pck?.
78   *      - si on veut arrêter, relancer la lecture pour purger pck?.
79   *      - traiter les bit Q et M si nécessaire.
80   */
81  eread(nucl, pointer)
82  int     nucl;
83  int     *pointer;
84  {
85          register  struct  ucl  *ucpt;
86          register  int  *pnt;
87          register  int  result;

88          ucpt = &ucl[nucl];
89          pnt = pointer;
90          if (ucpt->uc_flag & DELREAD) {
91                  bufv(pnt);
92                  if (ucpt->uc_rdcpt) goto repeat;
93                  ucpt->uc_flag =| ARREAD;
94                  return;
95          }
96          if (ucpt->uc_flag & BITQ) pnt->pk1_lng = 0177400;
97          else pnt->pk1_lng = 0;
98          if (ucpt->uc_fgs & STILLWR) pnt->pk1_lng =| 0377;
99          pnt->bf_to = &pnt->bf_buf[5];
100         result = pckx25(ucpt->uc_nvl, USR + 0, pnt);
101         if (result > 0) {
102                 taskin(&eread, 30, nucl, pnt);
103                 return;
104         }
105         if (result) return;
106   repeat :
107         taskin(&uread, priority[2], nucl, 128, 0);
```

```
108         return;
109 }
110 /*
111  * Routine ecmdin():
112  *      - traiter les commandes venant de l'utilisateur.
113  *      - dèmander éventuellement au niveau paquet de générer un paquet.
114  *      - agir en fonction des demandes de l'utilisateur.
115  */
116 ecmdin(nucl, act, arg)
117 int     nucl, act, arg;
118 {
119         register  struct  ucl  *ucpt;
120         register  int   mask;
121         register  int   *pnt;
122         int     offset;

123         ucpt = &ucl[nucl];
124         offset = nucl * 140;
125         mask = ucpt->uc_flag;
126         if (mask & OCCUP) {
127                 switch (act) {
128                   case CALL :
129                         if (mask & (LINING | CALLING | ONLINE)) return;
130                         if (mask & (LINED)) {
131                                 pnt = ucpt->uc_point;
132                                 if(pnt == 0) {
133                                         pnt = bufp();
134                                         if (pnt == -1) {
135                                                 upcmd(nucl, LIB, SATURATION);
136                                                 return;
137                                         }
138                                         ucpt->uc_point = pnt;
139                                         if (seek(fstat, offset, 0) < 0) goto err;
140                                         if (read(fstat, pnt, 12 + RQLNG) < 0) goto er
141                                 }
142                                 mask =| CALLING;
143                                 pnt->bf_to = pnt->bf_buf + RQLNG;
144                                 pnt->bf_cpt = 0;
145                                 taskin(&uread, priority[2], nucl, OPLNG, pnt);
146                         }
147                         break;
148                   case INT :
149                         if (mask & ONLINE) pckx25(ucpt->uc_nvl, USER + INT, arg);
150                         break;
151                   case LIB :
152                         if (mask & (ONLINE | CALLING)) {
153                                 if (ucpt->uc_nvl != -1) pckx25(ucpt->uc_nvl, USER + LI
154                         }
155                         mask =& ~(ONLINE | CALLING);
156                         ucpt->uc_nvl = -1;
157                   case VIDVIT :
158                         if (mask & LINED) {
159                                 pnt = ucpt->uc_bufin;
160                                 if (pnt) pnt->bf_cpt = 0;
161                                 while (pnt) {
162                                         bufv(pnt);
163                                         pnt = *pnt;
```

```
164                                     }
165                                     mask =¦ (STOPREAD ¦ DELREAD);
166                                     if (mask & ONLINE) pckx25(ucpt->uc_nvl, USER + VIDVIT
167                             } else {
168                                     return;
169                             }
170                             break;
171                     case  REP :
172                             if (nucl != 0) return;
173                             pckx25(0, USER + REP, arg);
174                             break;
175                     case RUN :
176                             mask =& ~(STOPREAD ¦ DELREAD  ¦ STOPWRITE ¦ ARREAD);
177                             if (mask & ARREAD)
178                                     taskin(&uread, priority[9], nucl, 128, 0);
179                             break;
180                     case BREAK :
181                             if (mask & (CALLING ¦ ONLINE)) ecmdin(nucl, LIB, arg);
182                             mask = 0;
183                             ucpt->uc_ppid = 0;
184                             if (ucpt->uc_point) {
185                                     bufv(ucpt->uc_point);
186                                     ucpt->uc_point = 0;
187                             }
188                             break;
189                     default :
190                             return;
191                 }
192         } else {
193                 if (act == BEGIN) {
194                         if (ucpt->uc_point) {
195                                 pnt = ucpt->uc_point;
196                                 ucpt->uc_point = 0;
197                         } else {
198                                 pnt = bufp();
199                                 if (pnt == -1) {
200                                         taskin(&ertrait, 29, 'EE', 33);
201                                         return;
202                                 }
203                         }
204                         mask =¦ (OCCUP ¦ LINING);
205                         pnt->bf_cpt = 0;
206                         pnt->bf_to = pnt->bf_buf;
207                         taskin(&uread, priority[9], nucl, RQLNG, pnt);
208                 } else {
209                         return;
210                 }
211         }
212         ucpt->uc_flag = mask;
213         if (seek(fstat, offset + 6, 0) < 0) goto err;
214         if(write(fstat, ucpt, 2) < 0) goto err;
215         return;
216   err :
217         taskin(&ertrait, 29, 'EE', 30);
218         return;
219 }
220 /*
```

```
221  * Routine elining():
222  *      - appelée pour établir la ligne entre l'utilisateur et le
223  *        moniteur réseau.
224  *      - traiter les informations que passe l'utilisateur (No de TTY, de
225  *        processus, d'utilisateur).
226  */
227 elining(nucl, pointer)
228 int     nucl;
229 char    *pointer;
230 {
231         register  struct  ucl  *ucpt;
232         register  char  *pnt;
233         register  int  offset;
234         int     result;

235         if (pointer->bf_cpt != RQLNG) {
236                 ecmdo(nucl, BREAK, 0);
237                 goto out;
238         }
239         ucpt = &ucl[nucl];

240         if (ucpt->uc_flag == 0) goto out;
241         offset = nucl * 140;
242         if (seek(fstat, offset, 0) < 0) goto err;
243         pnt = pointer->bf_buf;
244         ucpt->uc_ppid = pnt->rq_pid;
245         pnt->rq_uc = nucl;
246         ucpt->uc_flag = (OCCUP | LINED);
247         pnt->rq_fgs =| ucpt->uc_flag;
248         ucpt->uc_nvl = -1;
249         if (write(fstat, pointer, RQLNG + 12) < 0) goto err;
250         upcmd(nucl, RUN, 0);
251   out :
252         bufv(pointer);
253         return;
254   err :
255         taskin(&ertrait, 29, 'EE', 30, 0);
256         return;
257 }
258 /*
259  * Routine ecalling():
260  *      - appelée pour établir la liaison entre le moniteur réseau
261  *        et l'autre ETTD.
262  *      - analyser les paramètres fournis par l'utilisateur.
263  *      - demander au niveau paquet de générer un paquet d'appel.
264  */
265 ecalling(nucl,pointer)
266 int     nucl;
267 char    *pointer;
268 {
269         register  struct  ucl  *ucpt;
270         register  int  i;
271         register  char  *pnt;

272         if (pointer->bf_cpt != OPLNG) {
273                 upcmd(nucl, LIB, 0);
274                 return;
```

```
275         }
276         ucpt = &ucl[nucl];
277         if ((ucpt->uc_flag & CALLING) == 0) return;
278         pnt = pointer + 6;
279         (pnt->deb_op).op_uc = nucl;
280         for (i = 0; i <= 15; i++){
281                 if (pckx25(i, USER + CALL, pointer) >= 0) {
282                         ucpt->uc_nvl = i;
283                         ucpt->uc_vl = &vl[i];
284                         (pnt->deb_rq).rq_vl = i;
285                         (pnt->deb_op).op_vl = i;
286                         return;
287                 }
288         }
289         upcmd(nucl, LIB, SATURATION);
290         return;
291 }
292 /*
293  * Routine ecmdo():
294  *      - appelée par le niveau paquet lors de la réception d'un paquet
295  *        de commande; prendre les dispositions en fonction du problème
296  *        qui se pose.
297  *      - signaler a l'utilisateur ce qui se passe.
298  *      - bloquer si nécessaire les transferts avec l'utilisateur.
299  */
300 ecmdo(nucl, act, arg)
301 int     nucl, act, arg;
302 {
303         register  struct  ucl  *ucpt;
304         register  int  status;
305         register  int  offset;
306         char    *pnt;
307         int     i;

308         if (nucl < 0 && act != CALL) return;
309         ucpt = &ucl[nucl];
310         status = ucpt->uc_flag;
311         offset = nucl * 140;

312         switch(act) {
313           case LIB :
314                 if (status & (ONLINE | CALLING)) {
315                         status =& ~(ONLINE | CALLING);
316                         status =| (STOPREAD | DELREAD);
317                 } else {
318                         return(0);
319                 }
320                 break;
321           case VIDVIT :
322                 if (status & ONLINE) {
323                         status =| (STOPREAD | DELREAD);
324                 } else {
325                         return;
326                 }
327                 break;
328           case OKCALL :
329                 if (status & CALLING) {
```

```
330                     status =& ~CALLING;
331                     status =| ONLINE;
332                     act = RUN;
333                     if(ucpt->uc_point) {
334                             if(seek(fstat, offset, 0) < 0) goto err;
335                             if(write(fstat, ucpt->uc_point, 140) < 0) goto err;
336                             bufv(ucpt->uc_point);
337                             ucpt->uc_point = 0;
338                     }
339                     taskin(&uread, priority[2], nucl, 128, 0);
340             } else {
341                     return;
342             }
343             break;
344         case CALL :
345             nucl = -nucl;
346             ucpt = &ucl[1];
347             for(i = 1; i <= 5; i++){
348                     if(ucpt->uc_flag & (ONLINE | CALLING)) continue;
349                     if(ucpt->uc_point == 0) {
350                             offset = i * 140;
351                             ucpt->uc_point = bufp();
352                             if (ucpt->uc_point == -1) return(-1);
353                             if(seek(fstat, offset, 0) < 0) goto err;
354                             if(read(fstat, ucpt->uc_point, RQLNG + 6) < 0) goto e
355                     }
356                     pnt = ucpt->uc_point + 6;
357                     (pnt->deb_op).op_vl = (pnt->deb_rq).rq_vl = nucl;
358                     ucpt->uc_nvl = nucl;
359                     ucpt->uc_vl = &vl[nucl];
360                     (pnt->deb_op).op_uc = i;
361                     status = (OCCUP | CALLING | LINED);
362                     nucl = i;
363                     goto out;
364             }
365             return(-1);
366         case INT :
367             if(!(status & ONLINE)) return;
368             break;

369         default :
370             return;
371     }
372 out :
373     ucpt->uc_flag = status;
374     upcmd(nucl, act, arg);

375     if(seek(fstat, offset + 6, 0) < 0) goto err;
376     if(write(fstat, ucpt, 2) < 0) goto err;
377     return(i);
378 err :
379     taskin(&ertrait, 29, 'EE', 31);
380     return(i);
381 }
```

Annexe L: Routines de liaison avec le moniteur /dev/pck?

```
 1 #
 2 #include "param.h"
 3 #include "mnemo.h"
 4 #include "pckx25.h"
 5 #include "net25.h"
 6 #include "open.h"
 7 #include "task.h"
 8 #include "uclx25.h"
 9 #include "drx25.h"
10 #include "routine.h"
11 #include "buffer.h"


12 /*
13  * Routine upcmd():
14  *      - envoyer des données de contrôle vers l'utilisateur.
15  *      - lui signaler que des données ont ete envoyées.
16  *      - si il y a une erreur (non reception par l'utilisateur)
17  *        déclencher une procédure d'erreur.
18  */

19 upcmd(id, cmd, param)
20 int     id, cmd, param;
21 {
22         extern  int errno;
23         register  struct  opr  *pnt;
24         register  struct  ucl  *ptucl;

25         struct  opr {
26                 int     param;
27                 char    cmid;
28         } oprqst;

29         pnt = &oprqst;
30         id =& 017;
31         cmd =<< 4;
32         cmd =& 0360;
33         cmd =+ id;
34         ptucl = &ucl[id];
35         pnt->param = param;
36         pnt->cmid = cmd;

37         if (ptpck(uc_fwr, pnt, id) >= 0) {
38                 if (ptucl->uc_ppid == 0) return;
39                 if (kill(ptucl->uc_ppid, 1) >= 0) return;
40         }

41         taskin(&ertrait, 29, 'OU' + id, 2, errno);
42         return;
43 }


44 /*
```

```
45  * Routine uwrite():
46  *       - envoyer des données-informations vers l´utilisateur.
47  *       - si on ne peut mettre tout dens le canal de pck?, regénérer ce
48  *         travail pour plus tard.
49  *       - si tout est envoyé, appeler ewrite() de endend.c pour avoir un
50  *         autre buffer de donnees-informations.
51  */

52 uwrite(id, pointbf)
53 int     id;
54 char    *pointbf;
55 {
56         register  int  result;
57         register  struct  bf  *ptbf;
58         register  int   cpt;
59         extern  int  errno;

60         ptbf = pointbf;
61         if (ptbf->bf_cpt) {
62                 if (ptbf->bf_cpt > WMAX) cpt = WMAX;
63                 else cpt = ptbf->bf_cpt;
64                 result = wrpck(uc_fwr, ptbf->bf_to, cpt, id);
65                 if (result < 0) {
66                         taskin(&ertrait, 29, ´OU´ + id, errno);
67                         return;
68                 }
69                 ptbf->bf_to =+ result;
70                 ptbf->bf_cpt =- result;
71                 result = &uwrite;
72         } else {
73                 result = &ewrite;
74         }

75         taskin(result, priority[5], id, ptbf);
76         return;
77 }


78 /*
79  * Routine uread():
80  *       - effectuer des lectures sur le canal de pck? pour obtenir les
81  *         caractères venant de l´utilisateur.
82  *       - effectuer cette lecture tant que l´utilisateur est en train
83  *         d´écrire, ou tant que c´est permis.
84  *       - lorsqu´il faut arrêter (buffer plein, plus rien à lire,
85  *         transfert bloque), passer le buffer a eread() de ENDEND pour
86  *         traitement.
87  */

88 uread(id, lng, pointer)
89 int     id, lng;
90 char    *pointer;
91 {
92         extern  int  errno;
93         register  struct  ucl  *ptucl;
94         register  char  *pnt;
95         register  int  result;
```

```
 96          ptucl = &ucl[id];

 97          if (pointer == 0){
 98                  pnt = bufp();
 99                  if (pnt == -1) goto repeat;
100                  pnt->bf_to = pnt->bf_buf + 5;
101                  pnt->bf_cpt = 0;
102                  pointer = pnt;
103          } else {
104                  pnt = pointer;
105          };

106          result = rdpck(uc_frd, pnt->bf_to, lng, id);
107          if (result < 0) goto err;

108          pnt->bf_to =+ result;
109          pnt->bf_cpt =+ result;

110          if((ptucl->uc_flag & STOPREAD) || (!(ptucl->uc_fgs & STILLWR))) goto out;

111          if (result != lng) {
112                  lng =- result;
113                  goto repeat;
114          }

115     out :
116          result = &eread;
117          if(ptucl->uc_flag & LINING) result = &elining;
118          if(ptucl->uc_flag & CALLING) result = &ecalling;
119          taskin(result, priority[9], id, pnt);
120          return;
121     err :
122          taskin(&ertrait, 29, 'OU' + id, 4, errno);
123     repeat :
124          taskin(&uread, priority[5], id, lng, pointer);
125          return;
126 }


127 /*
128  * Routine ugtcmd():
129  *       - obtenir de pck? les commandes et données de contrôle placées
130  *         par l'utilisateur.
131  *       - remettre ces données en ordre.
132  *       - commençer a les ana lyser en repérant celles qui viennent du
133  *         super utilisateur.
134  *       - repérer toute rupture anormale de la ligne (close utilisateur
135  *         sans prevenir) et prendre les mesures qui s'imposent.
136  */

137 ugtcmd()
138 {
139          extern  int  errno;
140          static  int  curucl;
141          register  char  *ucpt;
142          register  int  nucl;
143          register  int   act;
```

```
144         int     param[3];
145         int     i;

146         taskin(&ugtcmd, priority[4]);
147         while (i++ < 16) {
148                 curucl++;
149                 curucl =& 017;
150                 ucpt = &ucl[curucl];
151                 if(gtpck(uc_fwr, param, curucl) < 0)
152                         taskin(&ertrait, 29, 'OU' + curucl, 1, errno);
153                 ucpt->uc_rdcpt = (param[2] & 0177400);
154                 ucpt->uc_rdcpt =>> 8;
155                 ucpt->uc_wrcpt = (param[2] & 0377);
156                 if(ucpt->uc_fgs & OPENED) {
157                         ucpt->uc_fgs = param[1];
158                         if(!(param[1] & OPENED)) {
159                                 act = BREAK;
160                                 nucl = curucl;
161                                 goto out;
162                         }
163                 }
164                 ucpt->uc_fgs = param[1];
165                 if(param[1] & NOTRANS) {
166                         act = (param[1] & 017);
167                         if(curucl == 0) {
168                                 nucl = (param[1] & 0360);
169                                 nucl =>> 4;
170                                 if(nucl)
171                                         ecmdo(nucl, act, param[0]);
172                         } else {
173                                 nucl = curucl;
174                         }
175                         goto out;
176                 }
177         }
178         return;
179 out :
180         taskin(&ecmdin, priority[6], nucl, act, param[0]);
181         return;
182 }
```

Annexe M: Création et contrôle des processus pour utilisateurs externes.

```
1 #define NEXTPROC        5       /* nombre de processus externes */
2 #define all             p = &itab[0] ; p < &itab[NEXTPROC] ; p++
3 #define ever            ;;

4 struct  tab  {
5         int     pid;
6         int     pckw;
7         int     pckr;
8         char    pckx;
9 } itab[NEXTPROC];

10 char    *pcknam;
11 int     fildr, fildw;

12 main() {
13        register  int  i;
14        register  struct  tab  *p;
15        register  char  c;

16        pcknam = "/dev/pckx";

17        for(i = 3; i < 15; i++) {
18                close(i);
19        }

20        c = '1';
21        for (all) {
22                p->pckx = c++;
23        }

24        for (all) {
25                pckopen(p);

26                p->pid = fork();
27                if (p->pid == -1) ertrait(1);
28                if (p->pid == 0) {
29                        for(i = 5; i <= 15; i++)
30                                close(i);
31                        i = execl("netin", "netin", p->pckx, 0);
32                        if ( i < 0 ) ertrait(2);
33                }
34                printf("creation : %d0, p->pid);
35                close(fildw);
36                close(fildr);
37        }

38        printf("0netinit : attente. 0);

39        for (ever) {
40                i = wait();

41                for (all) {
42                        if (p->pid == i) {
```

```
43                                      close(p->pckw);
44                                      close(p->pckr);
45                                      sleep(5);
46                                      printf("netinit : mort de %d.0, i);
47                                      pckopen(p);
48                                      p->pid = fork();
49                                      if(p->pid == -1) ertrait(3);
50                                      if(p->pid == 0) {
51                                              for(i = 5; i <= 15; i++);
52                                                      close(i);
53                                              i = execl("netin", "netin", p->pckx, 0);
54                                              if (i < 0) ertrait(4);
55                                      }
56                                      close(fildw);
57                                      close(fildr);
58                                      printf("creation : %d0, p->pid);
59                                      break;
60                              }
61                      }
62              }
63 }

64 pckopen(pointer)
65 char    *pointer;
66 {
67              register  char  *p;

68              p = pointer;

69              pcknam[8] = p->pckx;
70              fildr = open(pcknam, 0);
71              if (fildr != 3) ertrait(5);
72              fildw = open(pcknam, 1);
73              if (fildw != 4) ertrait(6);
74              p->pckw = dup(fildw);
75              if (p->pckw < 0) ertrait(7);
76              p->pckr = dup(fildr);
77              if (p->pckr < 0) ertrait(8);
78              return;
79 }

80 ertrait(no)
81 int     no;
82 {
83              printf("0netinit : panique pas de processus externes. 0);
84              printf(" numero : %d. 2 ", no);
85              exit();
86 }
```

Errata de la 2ème partie.

<numéro de page>L<nombre de ligne> : compter les lignes en remontant du bas de la page.
<numéro de page>l<nombre de ligne> : compter les lignes en commençant au sommet de la page.

A.14 l 8, A.14 L 8, A.15 L 15, A.16 l 11, A.18 l 10, A.18 L 8, A.19 l 9, A.19 L 11, A.20 l 6, A.20 l 21 et A.20 L 1 remplacer :

```
} *pnt;
```

par :

```
} reserv;
register char *pnt;

pnt = &reserv;
```

A.17 l 12, 13, 14 : Inverser cmd_cop et cmd_param2.

A.14 l 10 remplacer par :

```
rqpt->rq_fgs = (status ? EXT : INT);
```

D.8 l 24 :

```
register int c;
```

F.2 l 8 : remplacer par :

```
pid = fork();
if (pid == -1) {
        printf("0 PANIC, pas de saut d processus. 0);
        exit();
}
if (pid) {
        printf("0 Processus moniteur reseau numero: %d", pid);
        exit();
}
```

I.6 l 8 insérer :

```
signal(1, &biphip);
```

K.5 l 19 :
transposer cette ligne après la ligne numéro 13.

L.3 l 15 remplacer par :

```
if(ptucl->uc_flag & STOPREAD) goto out;
if((pnt->bf_cpt > 0) && (!(ptucl->uc_fgs & STILLWR))) goto out;
```