Réseaux - Cours 2

UDP et TCP : Protocoles de Transport de TCP/IP

Cyril Pain-Barre

IUT Informatique Aix-en-Provence

Semestre 2 - version du 22/3/2011

Services et Limitations d'IP

- Services d'IP :
  - interconnexion de réseaux
  - remise de datagrammes à des hôtes (adresses IP)
  - adaptation aux MTU des réseaux
  - durée de vie limitée des datagrammes
  - détection des erreurs sur l'en-tête
  - signalisation de certaines erreurs via ICMP
- Limitations d'IP :
  - pas d'adressage des applications (client/serveur Web, client/serveur FTP, etc.)
  - livraison des datagrammes non garantie
  - duplication possible des datagrammes !
  - déséquencement possible des datagrammes
  - erreurs possibles sur les données
  - pas de contrôle de flux

Rôle du Transport

- Aller au-delà des limites d'IP
- Assurer, si possible, la correction d'erreurs :
  - signalées par ICMP
  - non signalées
- 2 protocoles de transport disponibles dans TCP/IP :
  - UDP : transport rapide, non connecté, permettant la multi-diffusion
  - TCP : transport fiable en mode connecté point-à-point

$\Rightarrow$ distinguent les applications au sein d'un même hôte

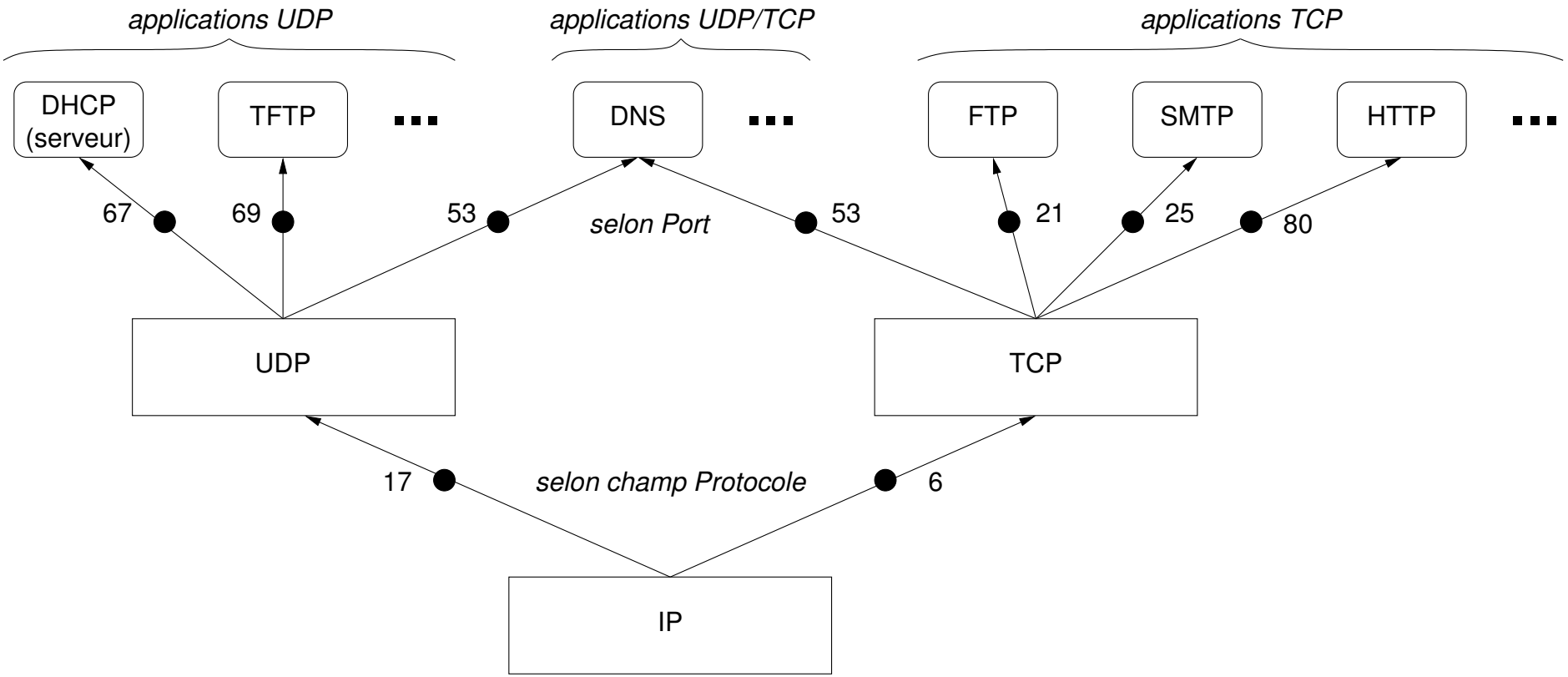
$\Rightarrow$ garantissent l'indépendance des communications

Adressage des applications

- Plusieurs applications réseaux peuvent s'exécuter en parallèle sur un ordinateur
- **Problème :** comment un émetteur peut-il préciser à quelle application est adressé un message ?
- La solution retenue sur Internet est l'utilisation de destinations abstraites : les **ports** (ne pas confondre avec les ports physiques des *hubs*/*switchs*)
  - entiers positifs sur 16 bits
  - UDP et TCP fournissent chacun un ensemble de ports indépendants : le port *n* de UDP est indépendant du port *n* de TCP
  - le système permet aux applications de se voir affecter un port UDP et/ou TCP (choisi ou de manière arbitraire)
  - certains numéros de port sont réservés et correspondent à des services particuliers

L'adresse d'une application Internet est le triplet :
(adresse IP, protocole de transport, numéro de port)

Démultiplexage des ports



Le protocole UDP : *User Datagram Protocol*

(RFC 768)

Services d'UDP

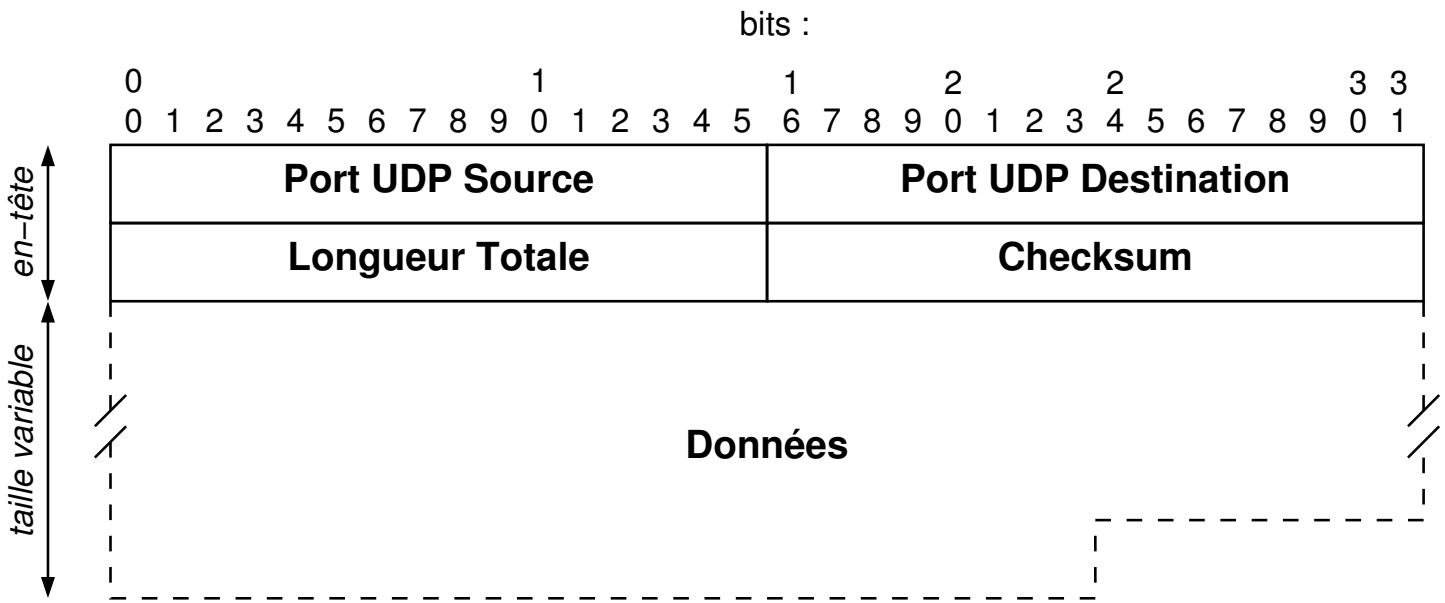
Utilise IP pour acheminer les messages d'un ordinateur à un autre.

- Service rendu :
  - adressage des applications par numéro de port
  - multiplexage/démultiplexage par numéros de port
  - contrôle facultatif de l'intégrité des données
- Même type de service non fiable, non connecté que IP :
  - possibilité de perte, duplication, déséquencement de messages
  - pas de régulation de flux

Un programme utilisant UDP doit gérer lui-même ces problèmes !

Format des datagrammes UDP

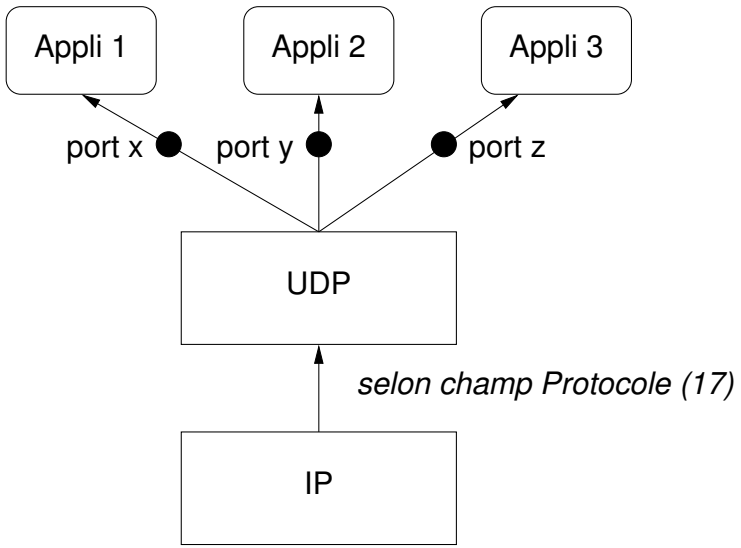
- En-tête : pas d'option possible, nombre fixe d'octets = 8
- Données : nombre variable d'octets ($\leq$ 65 535)



Champs UDP : ports source et destination

- *Port UDP Source* : indique le numéro de port de l'émetteur. Peut être à 0 si aucune réponse n'est attendue.
- *Port UDP Destination* : numéro de port du destinataire. Si ce port n'a été alloué à aucun processus, UDP renverra un message ICMP de destination inaccessible car port non alloué (type 3, code 3) et détruit le datagramme

Démultiplexage UDP selon le port destination :



Serveurs et ports réservés UDP

Voir `http://www.iana.org/assignments/port-numbers`

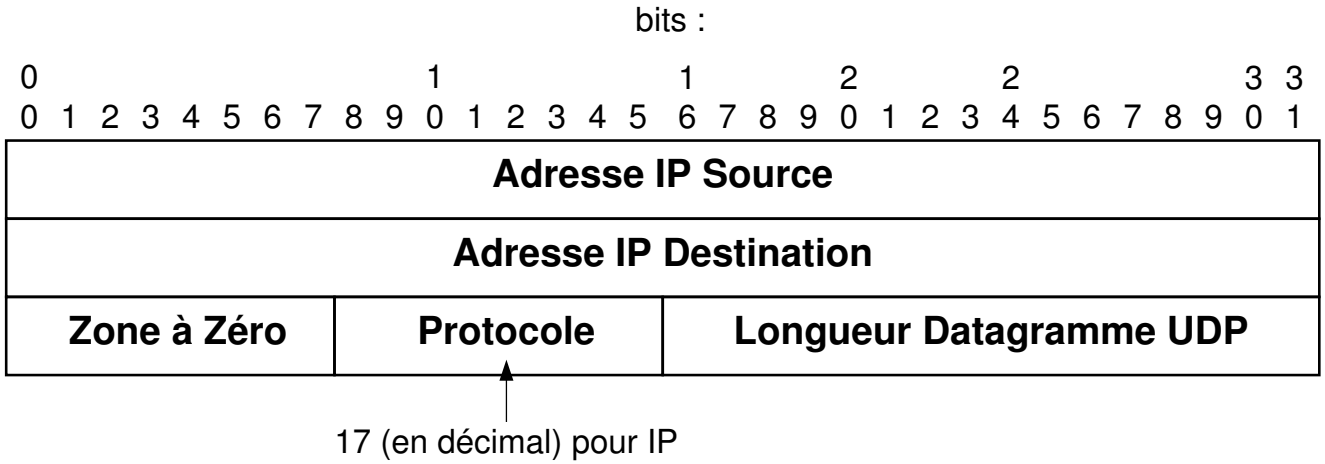
- *Well known Port Assignment* : certaines applications bien connues ont des ports UDP **réservés** $[0, 1023]$
  Exemples :

Num (décimal)	Application
7	Serveur echo
13	Serveur daytime
19	Serveur chargen
53	Serveur DNS
67	Serveur BOOTP/DHCP
68	Client BOOTP/DHCP
69	Serveur TFTP
123	Serveur NTP

- les ports $[1024, 49151]$ sont **enregistrés** (mais peuvent être utilisés)
- les ports $[49152, 65535]$ sont dits **dynamiques** et/ou à **usage privé**

Champs UDP : Checksum

- Facultatif : tout à 0 si non calculé
- Vérifie la totalité du datagramme + Pseudo en-tête UDP. Permet de s'assurer :
  - que les données sont correctes
  - que les ports sont corrects
  - *que les adresses IP sont correctes*
- Même calcul que IP sur tout le datagramme UDP (bourrage éventuel 1 octet à 0) + pseudo en-tête UDP
- Pseudo en-tête UDP (interaction avec IP) : (12 octets)



Interface socket BSD (Unix) pour UDP

- **int socket(int domain, int type, int protocol)** : retourne un Service Access Point (SAP) auprès d'UDP, utilisé en premier paramètre des primitives ci-après
- **int bind(int sock, const struct sockaddr *addr, socklen_t addrlen)** associe au SAP l'adresse d'application *addr*
- **ssize_t sendto(int sock, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)** demande l'envoi de *len* octets contenus dans *buf*, à l'adresse *to*
- **ssize_t recvfrom(int sock, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)** reçoit dans *buf* un message d'au plus *len* octets ; l'adresse de l'émetteur est placée dans *from*
- **int close(int sock)** : libère le SAP

Modèle Client/Serveur utilisant UDP

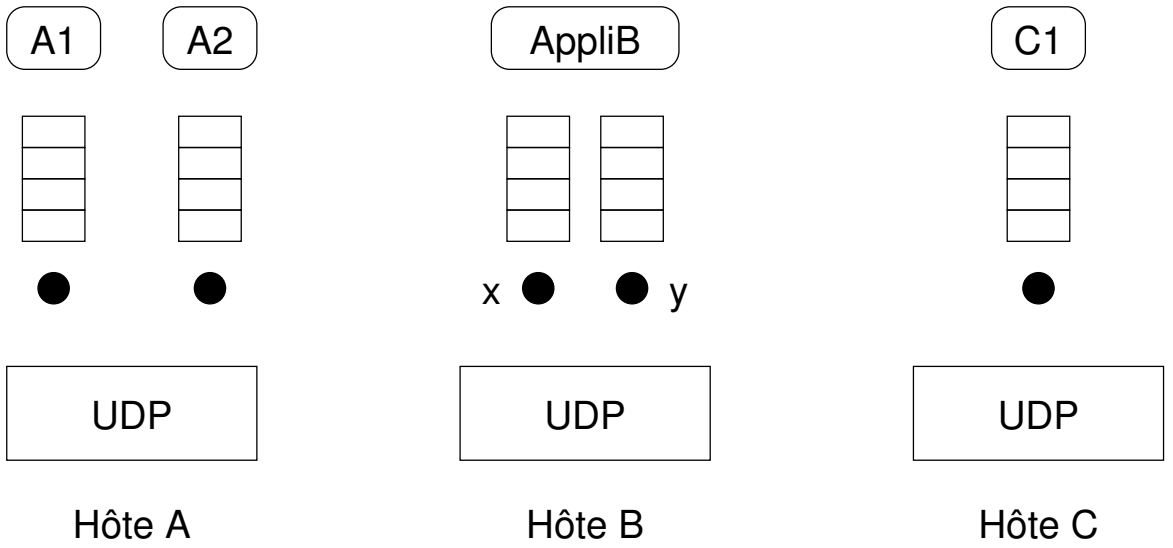
- Le serveur est démarré sur un ordinateur **en écoute** sur un port. Son adresse est le triplet : (*adresse IP*, UDP, *port*)
  L'écoute consiste à attendre qu'un message parvienne à ce port ; les clients devront connaître le port du serveur (intérêt des ports réservés)
- Le client envoie une requête au serveur (à son adresse) :
  - pour recevoir une réponse du serveur, il doit avoir acquis un numéro de port UDP. Le plus souvent, ce port est quelconque. Rares sont les clients (comme BOOTP/DHCP) qui nécessitent un port précis.
  - la requête est un **message applicatif** : suite d'octets constituant un PDU (Protocol Data Unit) du protocole qu'implémentent le client et le serveur.
  - UDP fabrique un datagramme UDP avec pour champ *Données* ce message, et pour champ *Port Destination* celui du serveur. Le *Port Source* est celui du client.
  - Le datagramme UDP est ensuite envoyé via un datagramme IP avec les adresses IP du serveur (destination) et du client (source).
- Le serveur reçoît le message du client, ainsi que son adresse. Il peut alors traiter le message et répondre au client.
- Selon le protocole, la discussion peut se poursuivre, ou s'arrêter là.

Files d'attente et réception

- Attribution de port $\Rightarrow$ allocation file d'attente
- Réception d'un datagramme :
  - port destination non attribué : destruction + message ICMP
  - port alloué : si file non pleine, ajouté dans la file,
    sinon détruit (pas de message ICMP !)
- Appli consomme les éléments de la file, par accès synchrone :
  - si aucun datagramme, processus placé en attente
  - sinon, consommation du premier datagramme (FIFO)
- Une application peut demander plusieurs ports

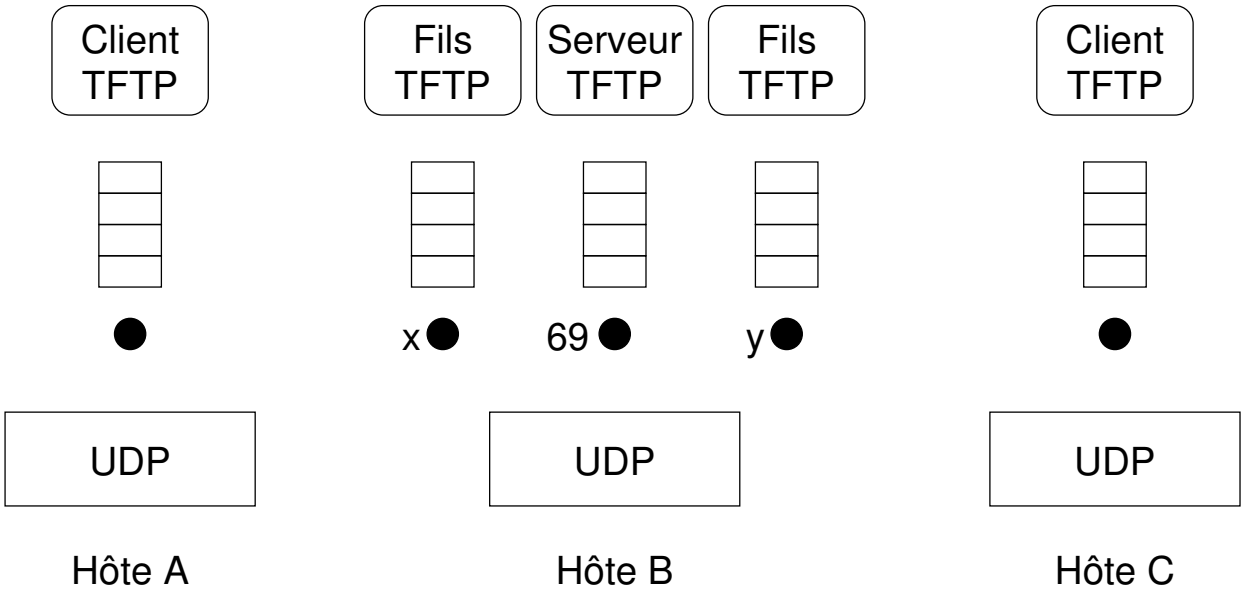
Utilisation UDP : modèle requête/réponse

- Soit AppliB qui utilise 2 ports :
  - si reçoit datagramme sur port x, renvoie les données en majuscules
  - si reçoit datagramme sur port y, renvoie les données à l'envers
- pour ce genre de service, AppliB peut traiter les datagrammes au fûr et à mesure



Utilisation UDP : suivi de la discussion

- Si la discussion est longue et complexe, il est plus simple de déléguer
- Exemple : serveur TFTP
  - reçoit un premier datagramme d'un client sur le port 69
  - demande un nouveau port
  - crée un fils pour traiter ce client sur ce nouveau port
  - renvoie un datagramme au client indiquant sur quel port poursuivre la discussion (transfert de fichier)
  - retourne à l'écoute de l'arrivée de nouveaux clients
  - la discussion se poursuit entre le client et le fils TFTP "dédié"



Utilisation UDP : multidiffusion

- UDP offre un service en mode non connecté
- Une application peut ainsi exploiter la multidiffusion IP et émettre un seul datagramme qui sera reçu par un ensemble de stations, en utilisant comme destination, une adresse :
  - broadcast : diffusion à toutes les stations d'un réseau, ou du réseau local pour l'adresse de diffusion limitée (255.255.255.255)
  - multicast : adresses de classe D pour diffuser à un ensemble de stations se trouvant éventuellement sur des réseaux différents (les routeurs qui les séparent doivent être configurés pour cela). Utilisées notamment pour la diffusion de médias audio/vidéo.

Exemple

Envoi par un client DHCP d'un message à tous les serveurs DHCP du réseau local. L'adresse destination du message DHCP est (255.255.255.255, UDP, 67)

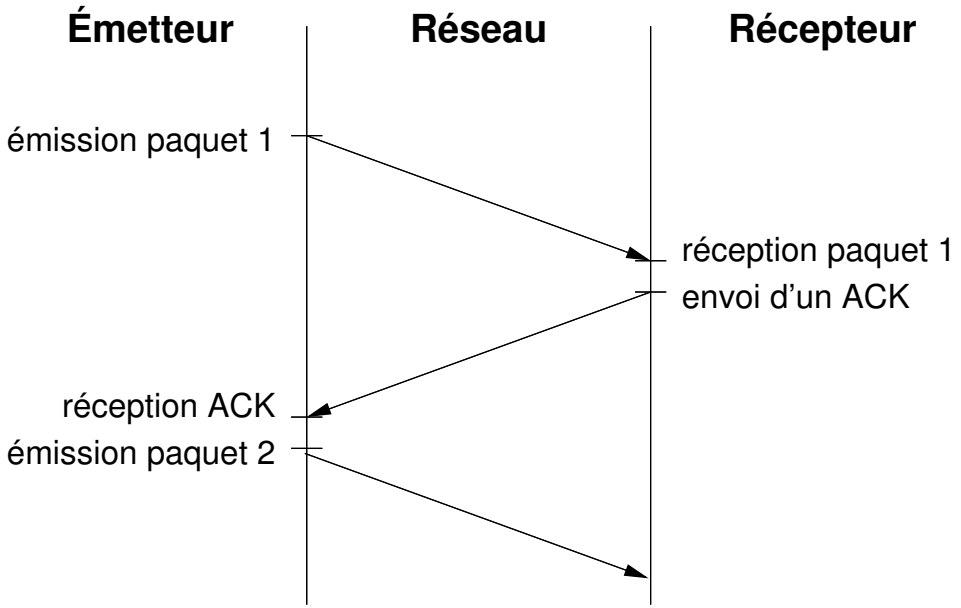
Mise en œuvre du transfert fiable

Transfert fiable en mode connecté

- Bien souvent les applications ont besoin d'échanger de gros volumes de données de manière fiable
- Exemples : FTP, SMTP, HTTP, ...
- Il est toujours possible d'offrir ce service en s'appuyant sur un service non fiable non connecté comme IP ou UDP
- Pour cela, on a besoin de quelques éléments essentiels :
  - les accusés de réception (ACK)
  - des temporisateurs : alarmes qui expirent (timeout)
  - la numérotation des paquets (ou données)

Accusés de réception (ACK)

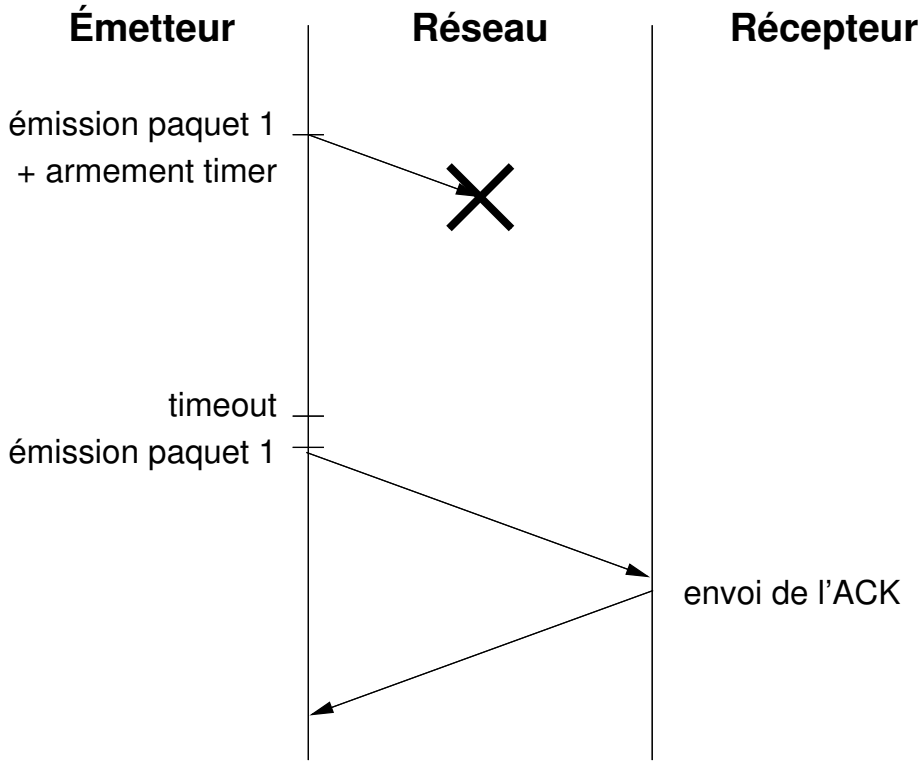
- L'émetteur d'un paquet attend une confirmation de réception de la part du récepteur avant d'envoyer un autre paquet
- Le récepteur accuse réception d'un paquet en envoyant un ACK



Protocole de type **envoyer et attendre**

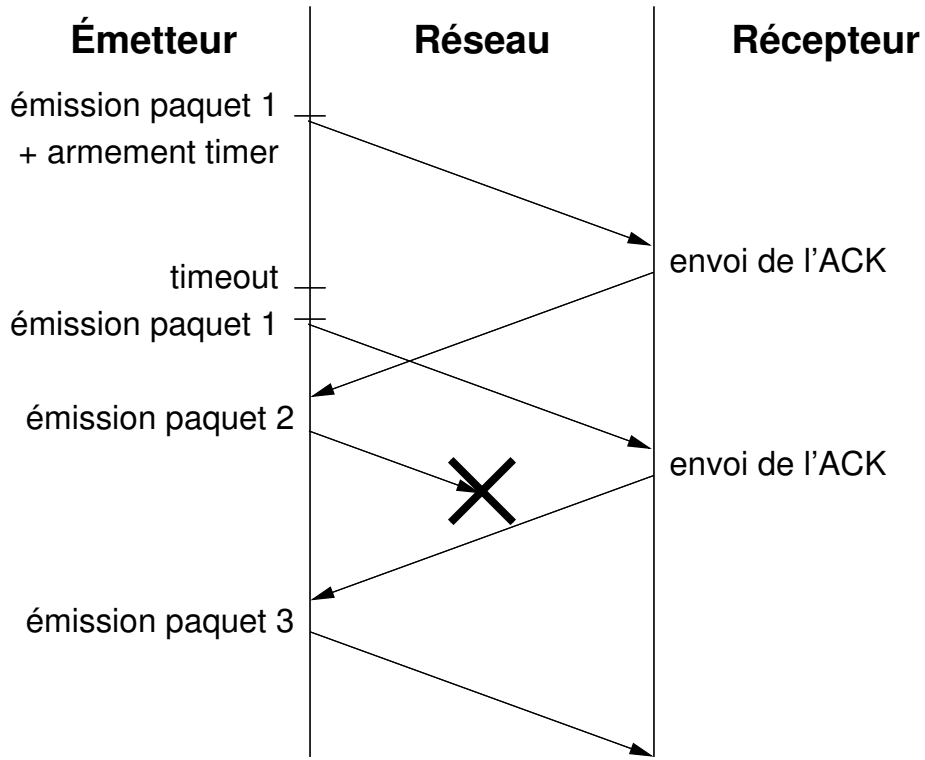
Temporisateurs

- Des paquets peuvent être perdus dans le réseau
- Si paquet "perdu", pas d'ACK donc blocage
- Utilisation d'un *timer* armé lors de l'émission du paquet :
  - si expire, renvoie le paquet
  - si réception ACK avant expiration, désactivation du timer



Temporisateurs : problème de réglage

Scénario du timer trop court et mise en cause fiabilité :

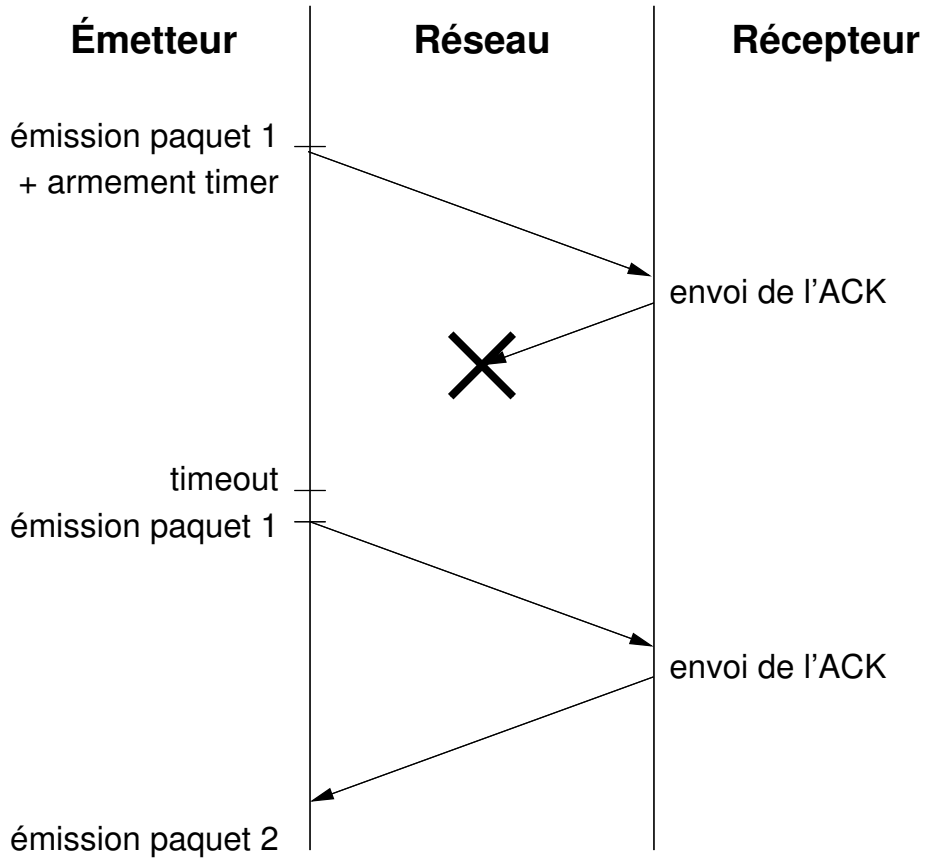


- **Le paquet 1 est accepté deux fois par le récepteur !**
- **Le deuxième ACK est pris pour celui du paquet 2 !**

Mais si timer trop long, on perd en efficacité. . .

ACK : problème de l'ACK perdu

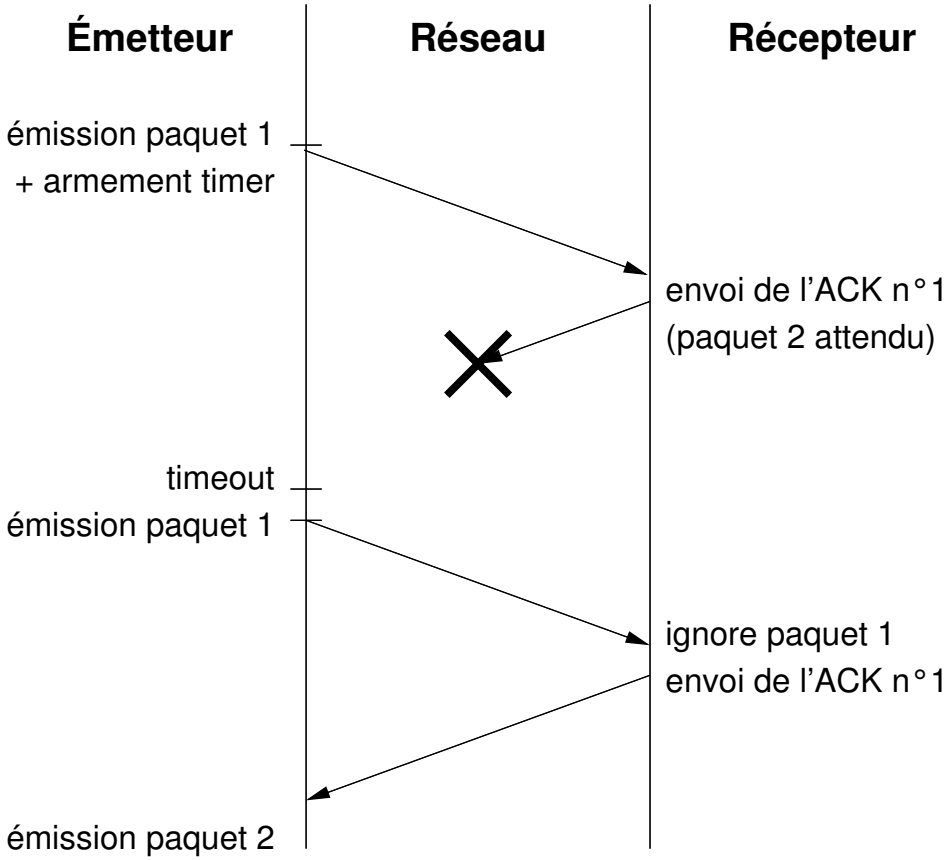
Scénario de l'ACK perdu et mise en cause fiabilité :



Le paquet 1 est accepté 2 fois par le récepteur !

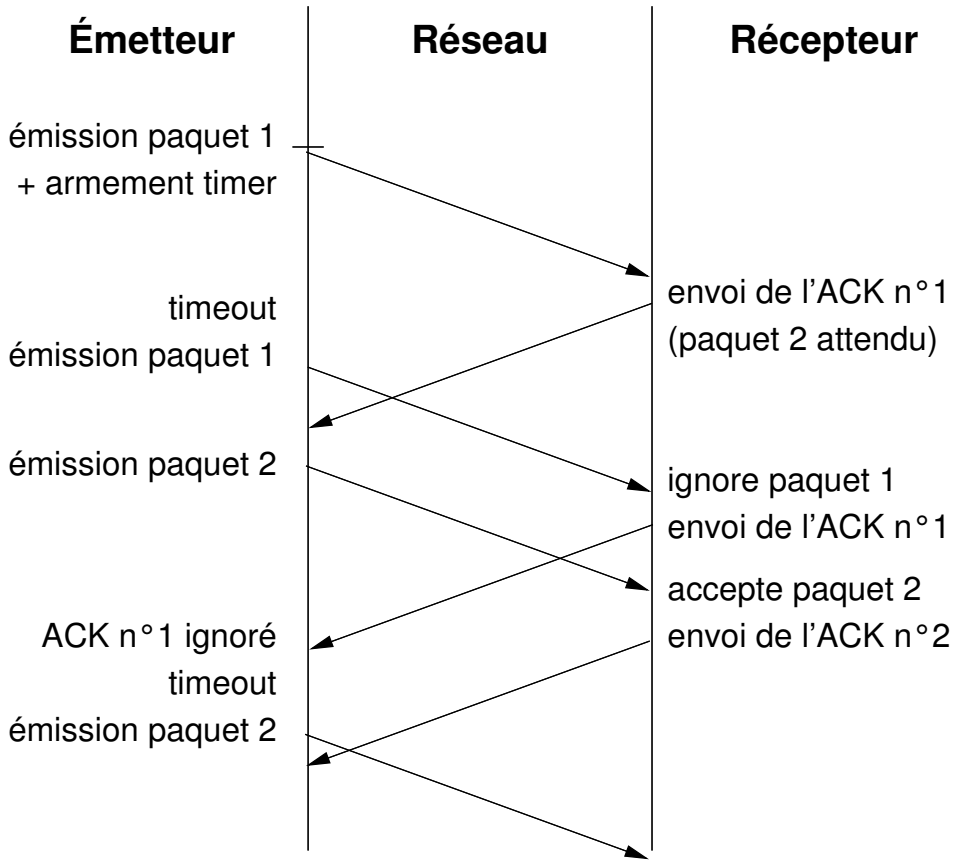
Numérotation des paquets et des ACK

Les paquets et les ACK portent des numéros : résoud les problèmes d'**ACK perdu** et d'alarme trop courte.



Numérotation des paquets et des ACK

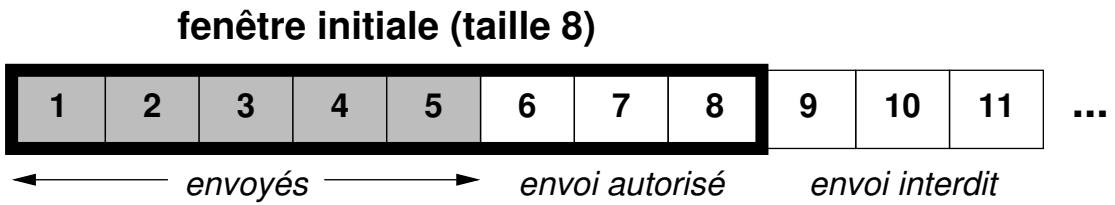
Les paquets et les ACK portent des numéros : résoud les problèmes d'ACK perdu et d'**alarme trop courte**.



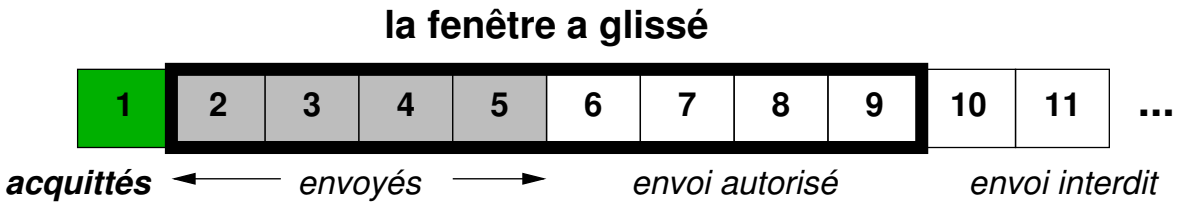
Fenêtre glissante (ou à anticipation)

Principe :

- émettre n paquets sans attendre d'ACK (n est la taille de la fenêtre)

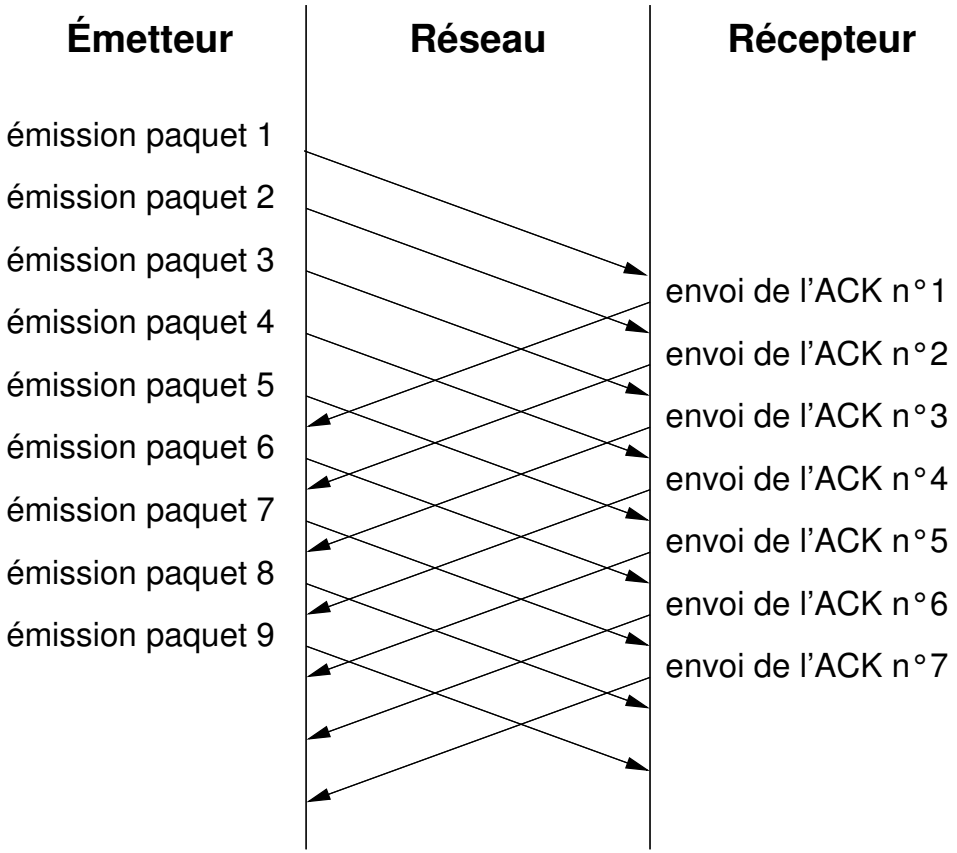


- un temporisateur par paquet émis
- la réception du ACK du premier paquet de la fenêtre la fait glisser :



Efficacité de la fenêtre glissante

La fenêtre glissante permet d'exploiter au mieux le réseau :



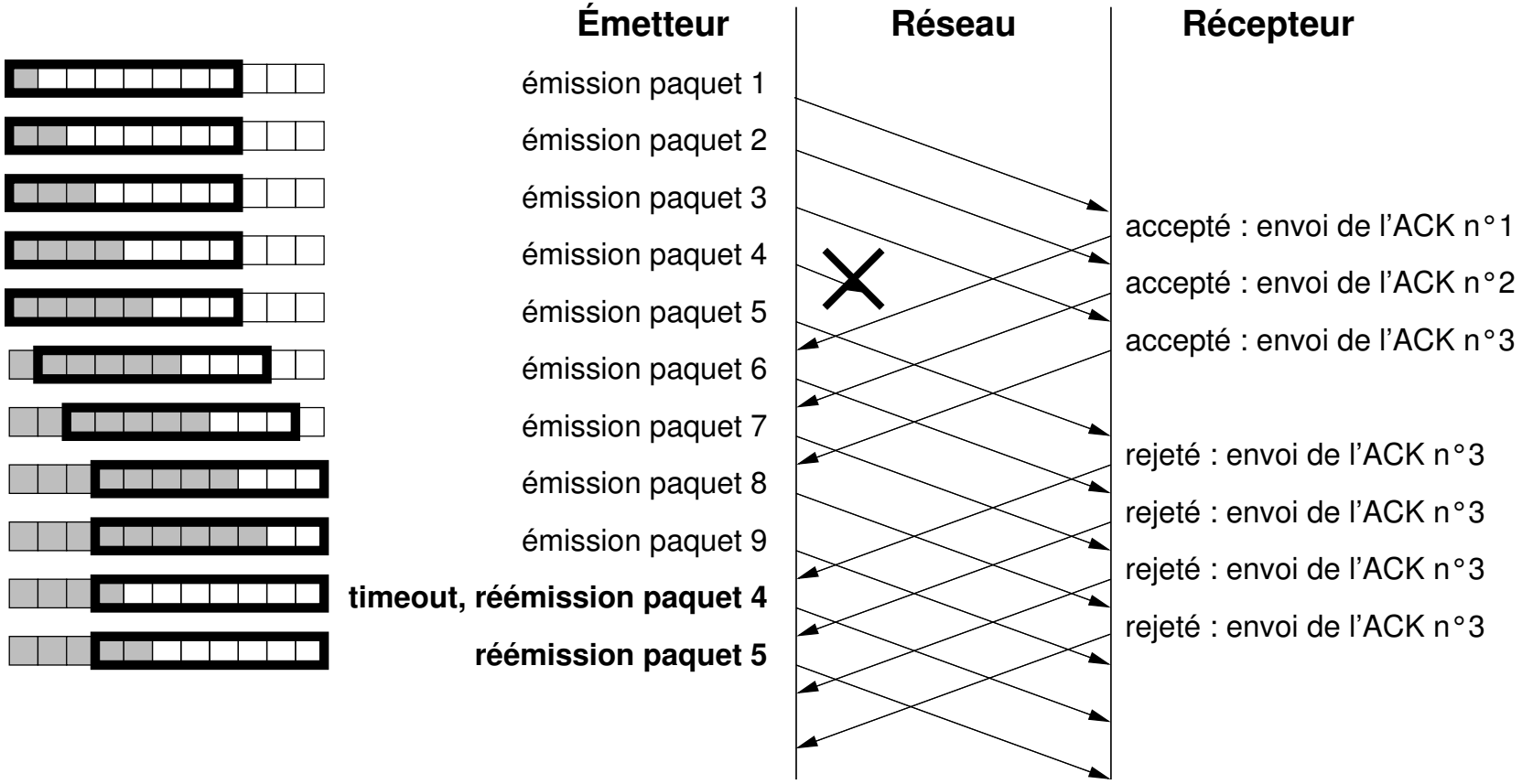
Fenêtre glissante et gestion des acquittements

Si un paquet de données est perdu, deux possibilités :

- rejet total (ou global)
- rejet sélectif

Fenêtre glissante et gestion des acquittements

Rejet total : aucun paquet suivant celui perdu n'est acquitté



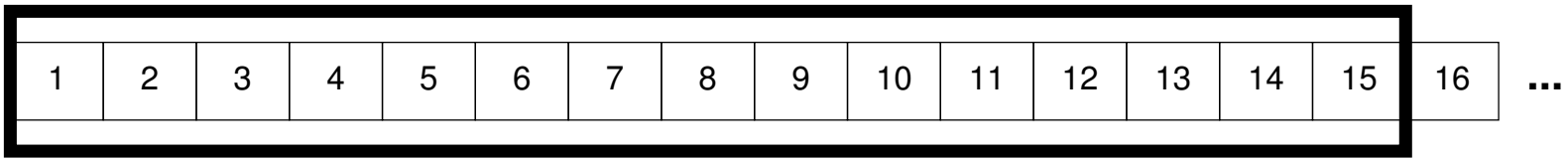
⇒ la fenêtre en réception a une taille de 1

Fenêtre glissante et gestion des acquittements

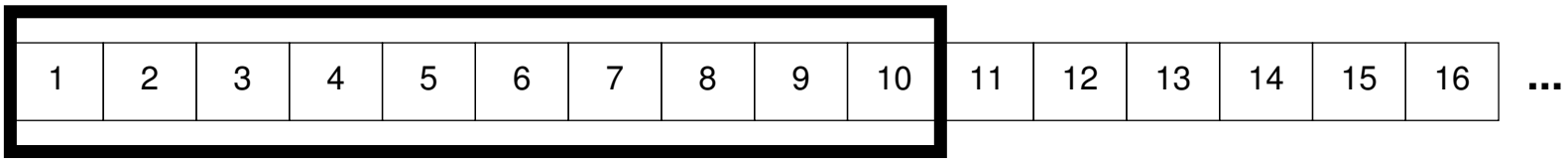
Rejet sélectif : paquet de données i perdu

- la fenêtre en réception a une certaine taille m (de préférence $m = n$)
- ceux qui entrent dans la fenêtre sont gardés, les autres sont ignorés
- mais les ACK pour ces paquets ont pour numéro $i - 1$
- si timeout, l'émetteur n'envoie que le premier paquet non acquitté
- lorsque paquet i réémis et reçu, l'ACK renvoyé est celui du dernier paquet reçu (ou celui avant un autre paquet éventuellement perdu)

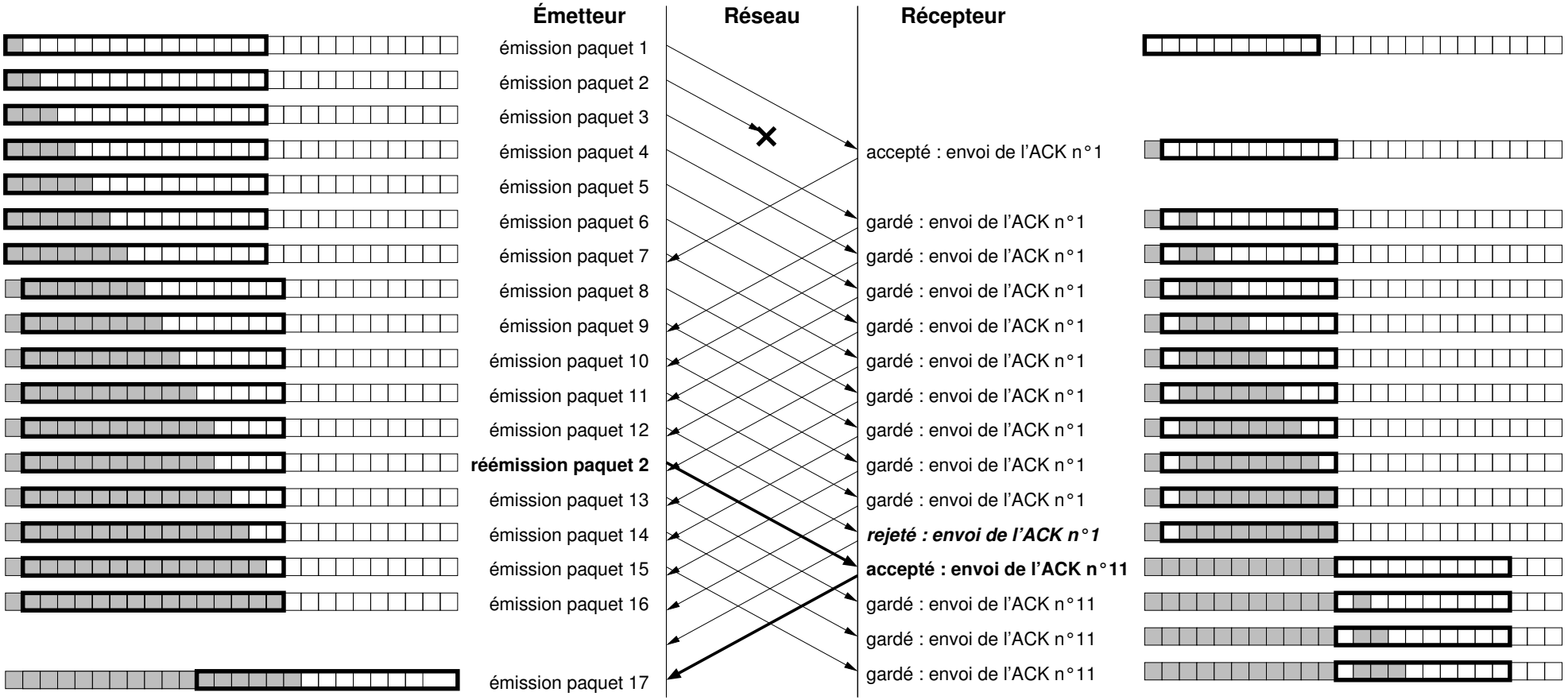
Fenêtre d'émission (taille 15) :



Fenêtre de réception (taille 10) :

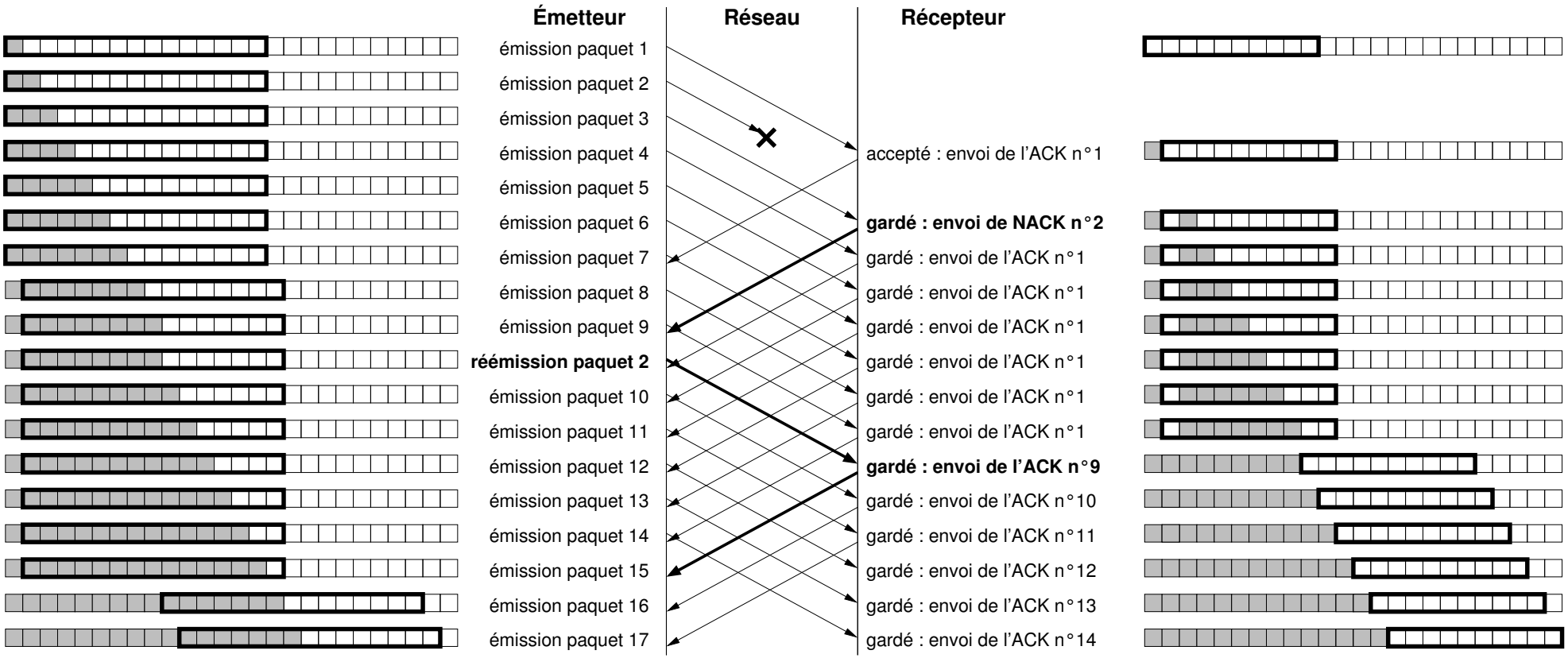


Rejet sélectif : exemple



Rejet sélectif : amélioration par NACK

Envoi d'un (seul) NACK si récepteur se rend compte d'une erreur :



Transmission bidirectionnelle et superposition

Pour une transmission bidirectionnelle :

- équiper les deux côtés de fenêtres d'émission et de réception

  pas forcément de même taille de chaque côté !

- amélioration par **superposition** ou **piggybacking** :
  les paquets de données contiennent un indicateur ACK (oui ou non le paquet contient aussi un ACK) et un champ n° ACK :

Le protocole TCP : *Transmission Control Protocol*

(RFC 793 corrigée par RFC 1122 et 1323)

Propriétés du service de TCP

- **Orienté connexion :** transfert de flots d'octets. La suite d'octets remise au destinataire est la même que celle émise
- **Circuits virtuels :** une fois une connexion demandée et acceptée, les applications la voient comme un circuit dédié
- **Transferts tamponnés :** quelle que soit la taille des blocs de données émis par les applications, TCP est libre de les découper ou de les regrouper
- **Connexions non structurées :** pas de frontière placée par TCP entre les messages émis par les applications
- **Connexions full-duplex :** les données s'échangent dans les deux sens mais un sens de transmission peut être libéré par l'émetteur

Adresse d'application, port et connexion

- Comme pour UDP, l'adresse d'une application est un triplet (*adresse IP*, TCP, *port*) : le serveur et le client doivent en posséder une. Le port du client est généralement quelconque.
- Mais à la différence d'UDP, on ne peut envoyer un message directement à une adresse : il faut que le client établisse une connexion avec le serveur. Ils ne peuvent échanger des messages que via une connexion
- **Établissement d'une connexion :**
  - Serveur : effectue une **ouverture passive** en écoutant sur un port, c'est à dire en demandant un port et en attendant qu'un client s'y connecte
  - Client : effectue une **ouverture active** en demandant l'établissement d'une connexion entre son adresse et celle du serveur. Le serveur doit être en écoute. Les modules TCP du client et du serveur intéragissent pour établir cette connexion.
- Une fois la connexion établie, le serveur et le client doivent l'utiliser pour envoyer/recevoir des messages. TCP est chargé d'assurer la fiabilité de la connexion (notamment s'occupe des acquittements/ retransmissions)

Serveurs et ports réservés TCP

Voir `http://www.iana.org/assignments/port-numbers`

- *Well known Port Assignment* : certaines applications bien connues ont des ports UDP **réservés** $[0, 1023]$. Exemples :

Num (décimal)	**Application**
7	Serveur echo
13	Serveur daytime
20	Serveur FTP (données)
21	Serveur FTP (commandes)
22	Serveur SSH
23	Serveur TELNET
25	Serveur SMTP (transfert de mail)
53	Serveur DNS
80	Serveur HTTP (www)
119	Serveur NNTP (news)

- les ports $[1024, 49151]$ sont **enregistrés** (mais peuvent être utilisés)
- les ports $[49152, 65535]$ sont dits **dynamiques** et/ou à **usage privé**

Interface socket BSD (Unix) pour TCP

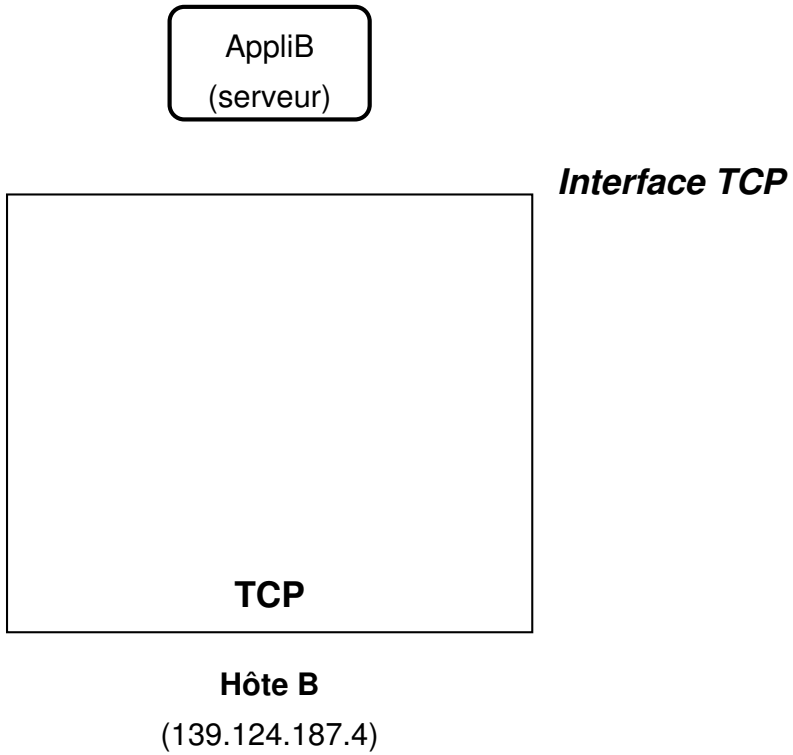
- **socket()** et **bind()** ont la même fonction que pour UDP
- **int listen(int sock, int backlog)**
  (serveur) prépare le SAP pour une ouverture passive
- **int accept(int sock, struct sockaddr *addr, socklen_t *addrlen)**
  (serveur) attend une demande de connexion (ouverture passive). Renvoie un SAP pour la connexion établie et *addr* contient l'adresse du client
- **int connect(int sock, const struct sockaddr *addr, socklen_t addrlen)**
  (client) demande l'établissement d'une connexion (ouverture active) au serveur d'adresse *addr*
- **ssize_t send(int sock, const void *buf, size_t len, int flags)**
  envoie *len* octets contenus dans *buf* à travers la connexion *sock*
- **ssize_t recv(int sock, void *buf, size_t len, int flags)**
  reçoit dans *buf* au plus *len* octets de la connexion *sock*
- **int close(int sock)** : libère un SAP (socket d'écoute ou connexion)

Ports et connexions

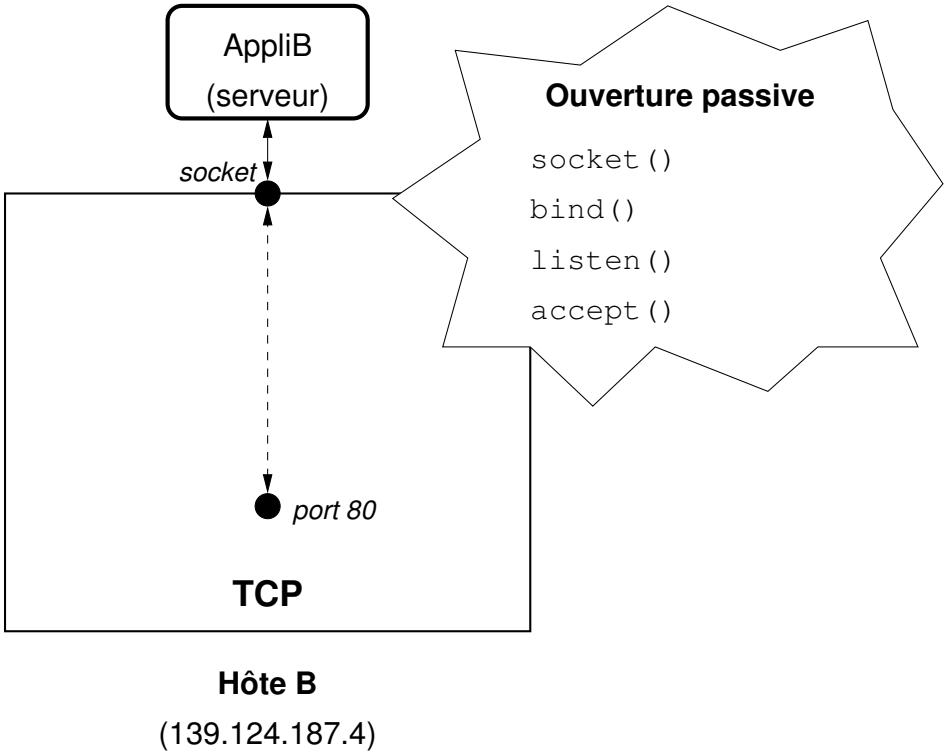
- Plus complexe qu'UDP car **un port peut être utilisé pour plusieurs connexions simultanément** :
  - un serveur peut accepter plusieurs clients à la fois : chaque appel d'accept() retourne une nouvelle connexion utilisant le port du serveur
  - plus rare, un client peut aussi utiliser son port pour établir plusieurs connexions (mais pas vers la même adresse serveur)
- En dehors des SAP d'ouverture passive, TCP gère surtout des "objets" connexion
- Une connexion est identifiée par le quadruplet formé avec l'adresse de ses deux extrémités :

  (adresse IP locale, port local, adresse IP distante, port distant)
- Les connexions sont gérées indépendamment les unes des autres
- Chaque connexion dispose de ses propres tampons en émission/réception et de chaque côté

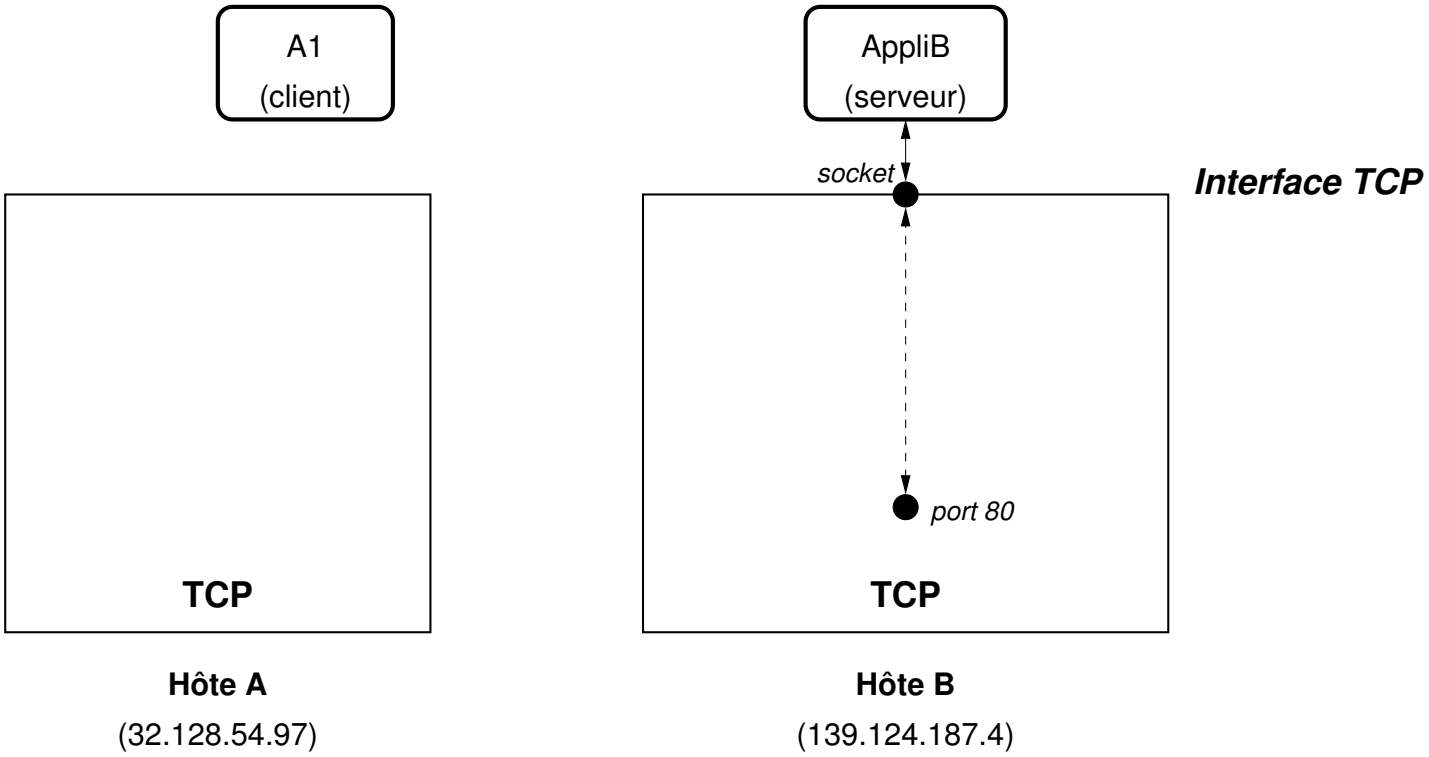
Ports et connexions : exemple



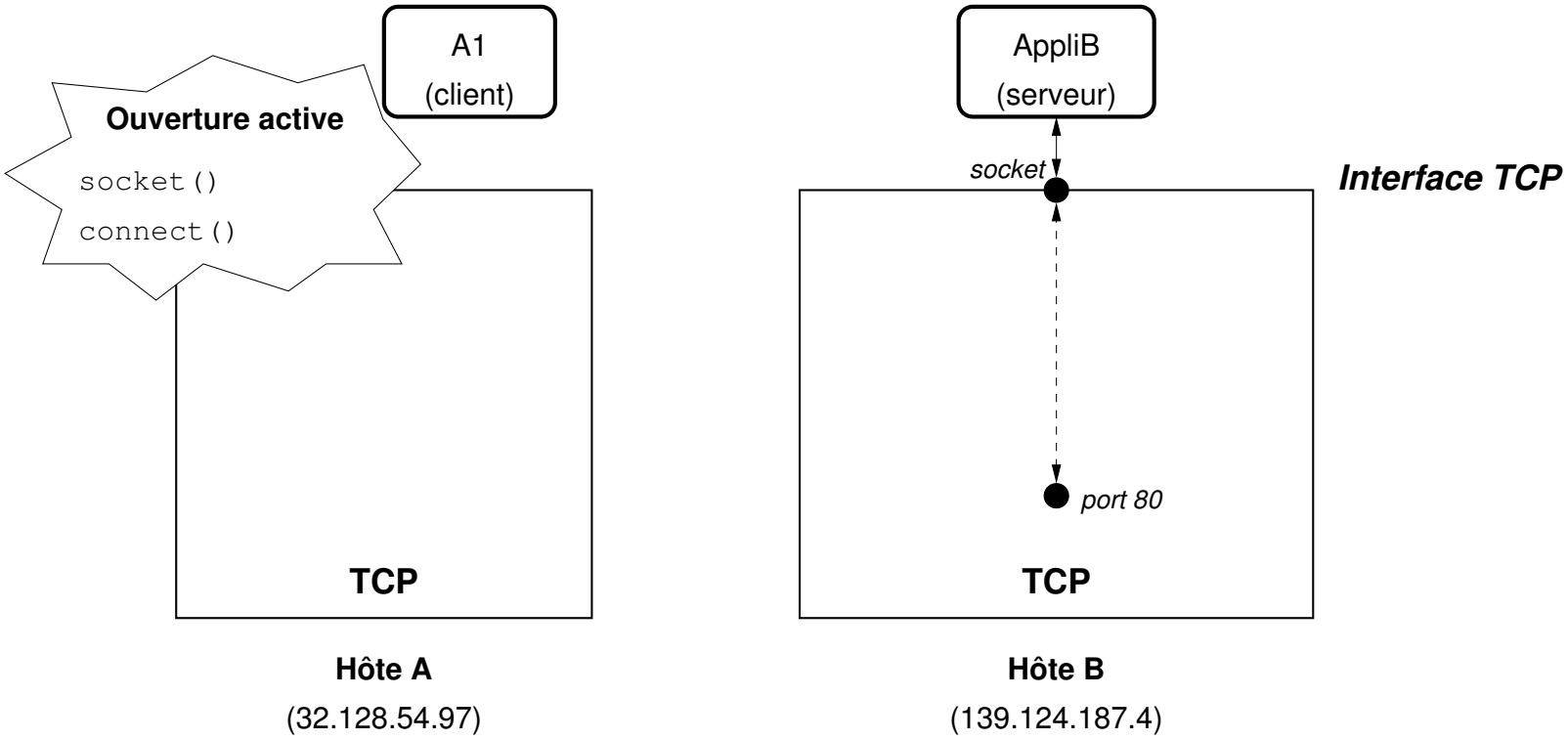
Ports et connexions : exemple



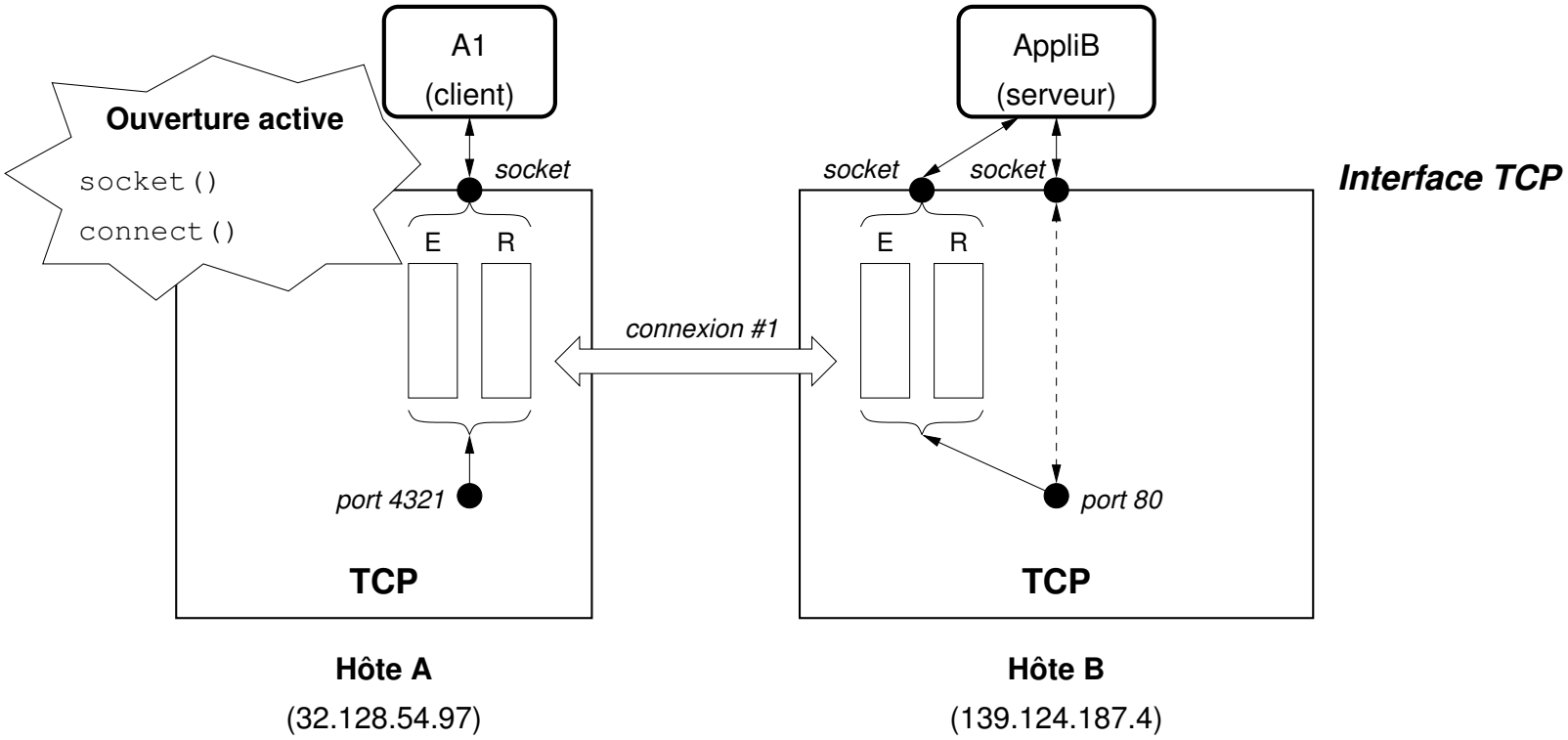
Ports et connexions : exemple



Ports et connexions : exemple

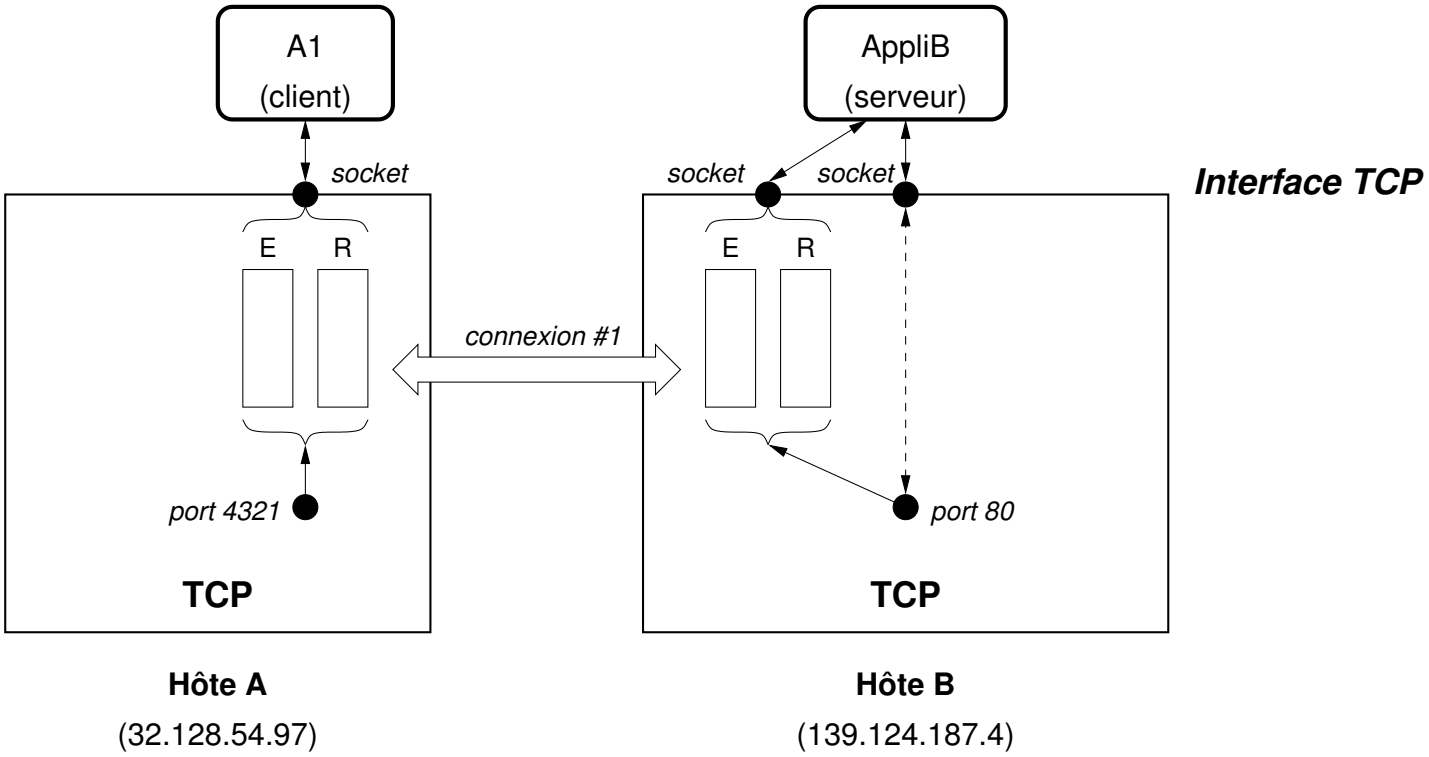


Ports et connexions : exemple



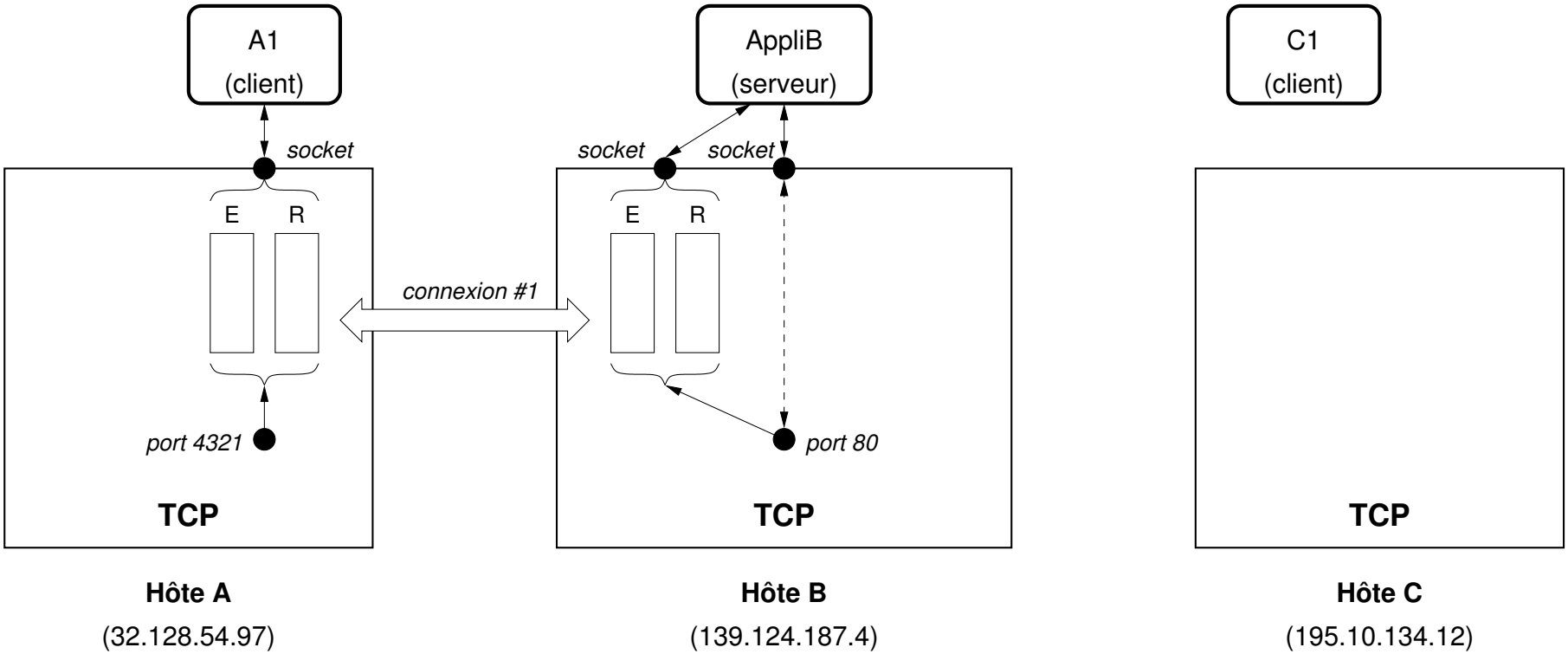
Connexion #1 : (139.124.187.4, 80) et (32.128.54.97, 4321)

Ports et connexions : exemple



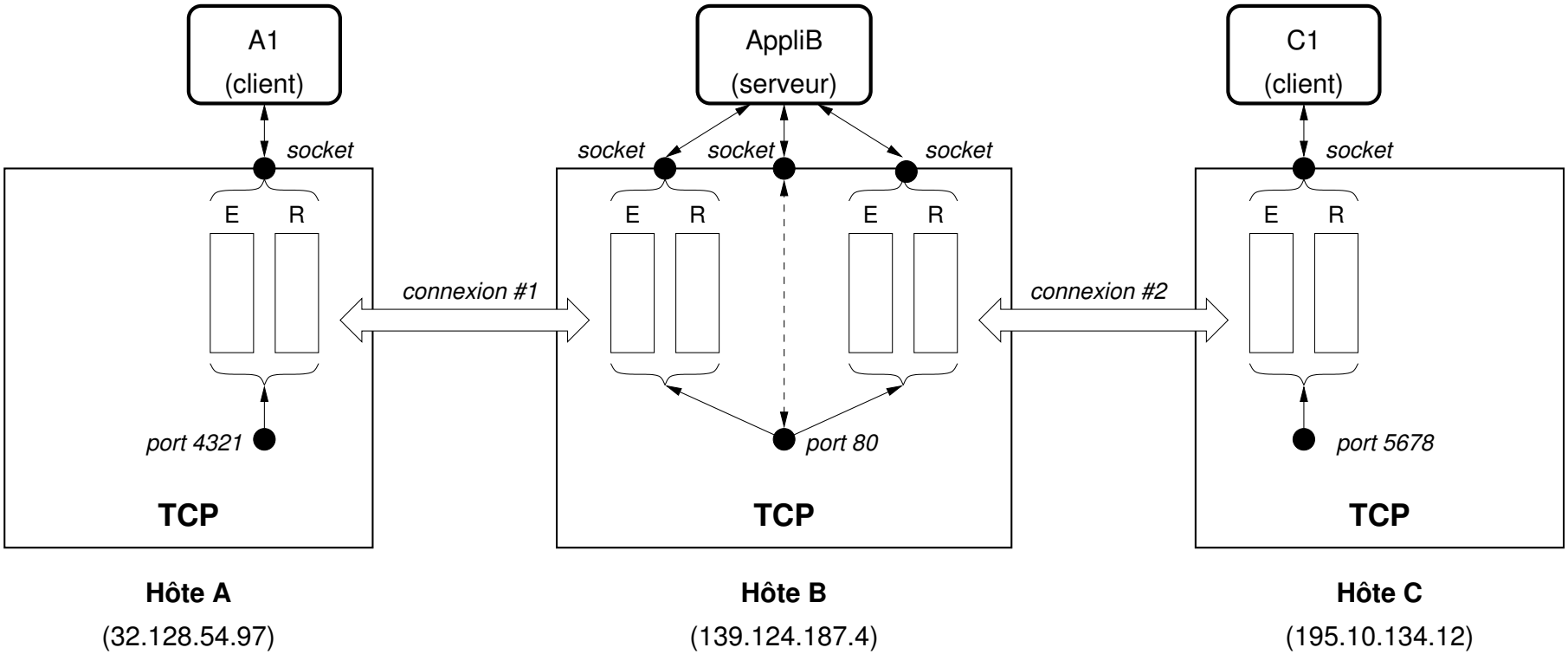
Connexion #1 : (139.124.187.4, 80) et (32.128.54.97, 4321)

Ports et connexions : exemple



Connexion #1 : (139.124.187.4, 80) et (32.128.54.97, 4321)

Ports et connexions : exemple



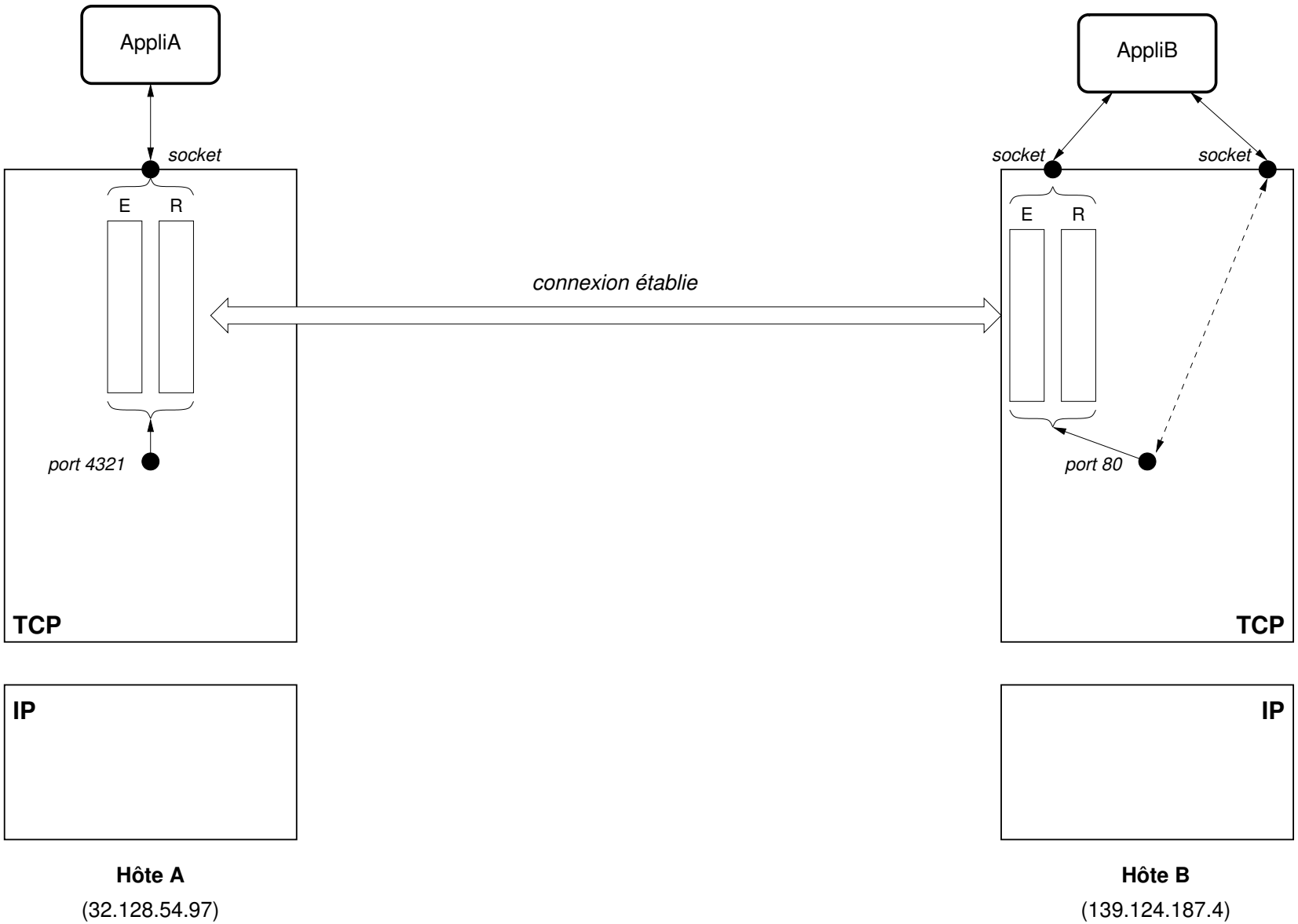
Connexion #1 : (139.124.187.4, 80) et (32.128.54.97, 4321)

Connexion #2 : (139.124.187.4, 80) et (195.10.134.12, 5678)

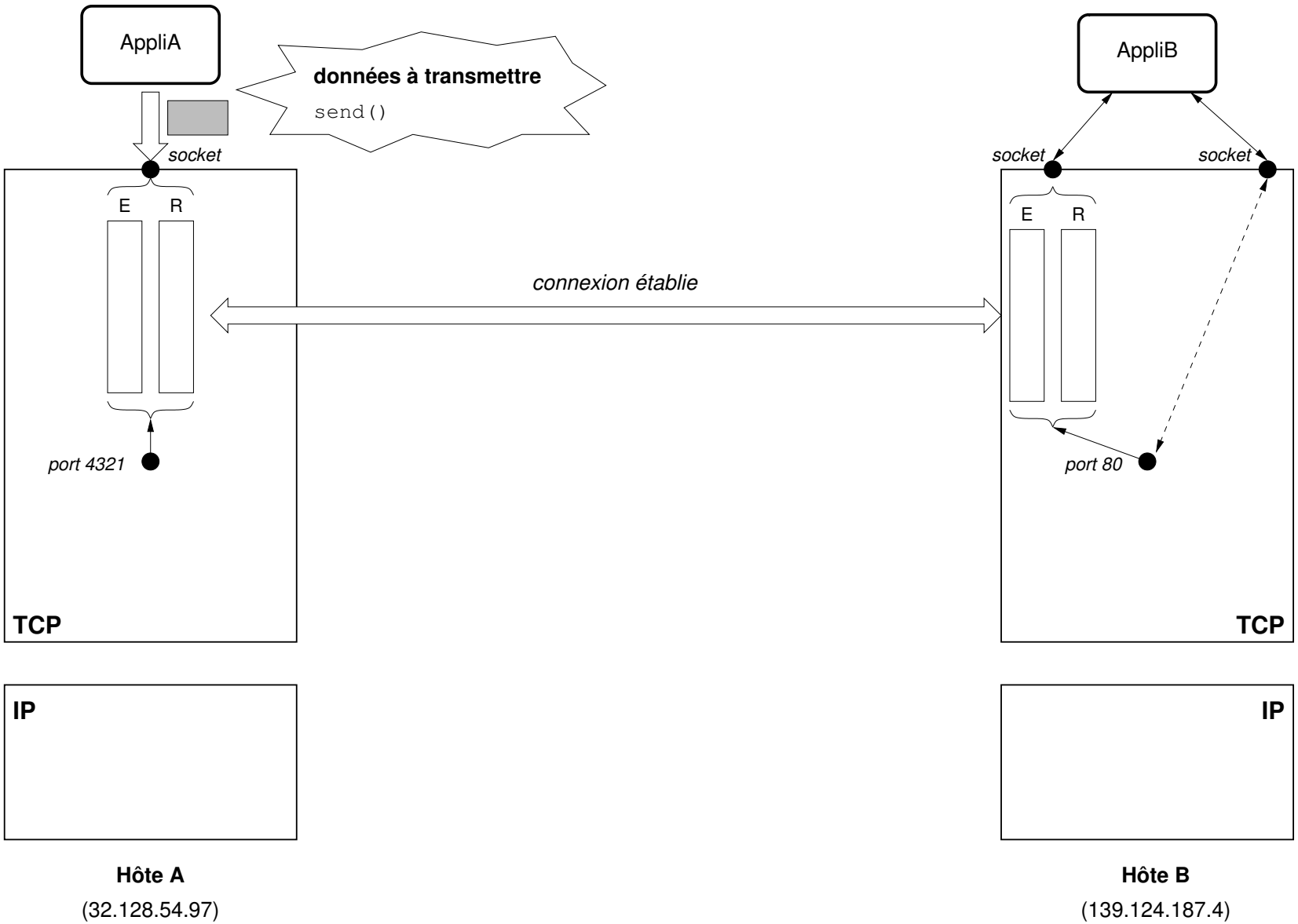
Flots d'octets et segments

- pour TCP une connexion sert à transmettre des flots d'octets dans les deux sens
- les flots sont transmis par des **segments** (PDU de TCP)
- un segment est transmis par un seul datagramme IP (sauf fragmentation pendant l'acheminement)
- l'émetteur transmet à TCP des blocs de données de taille quelconque
- le récepteur récupère des blocs de données de taille quelconque
- mais **le nombre d'octets transportés par un segment est décidé par TCP** :
  - pour des raisons d'efficacité
  - pour la régulation de flux

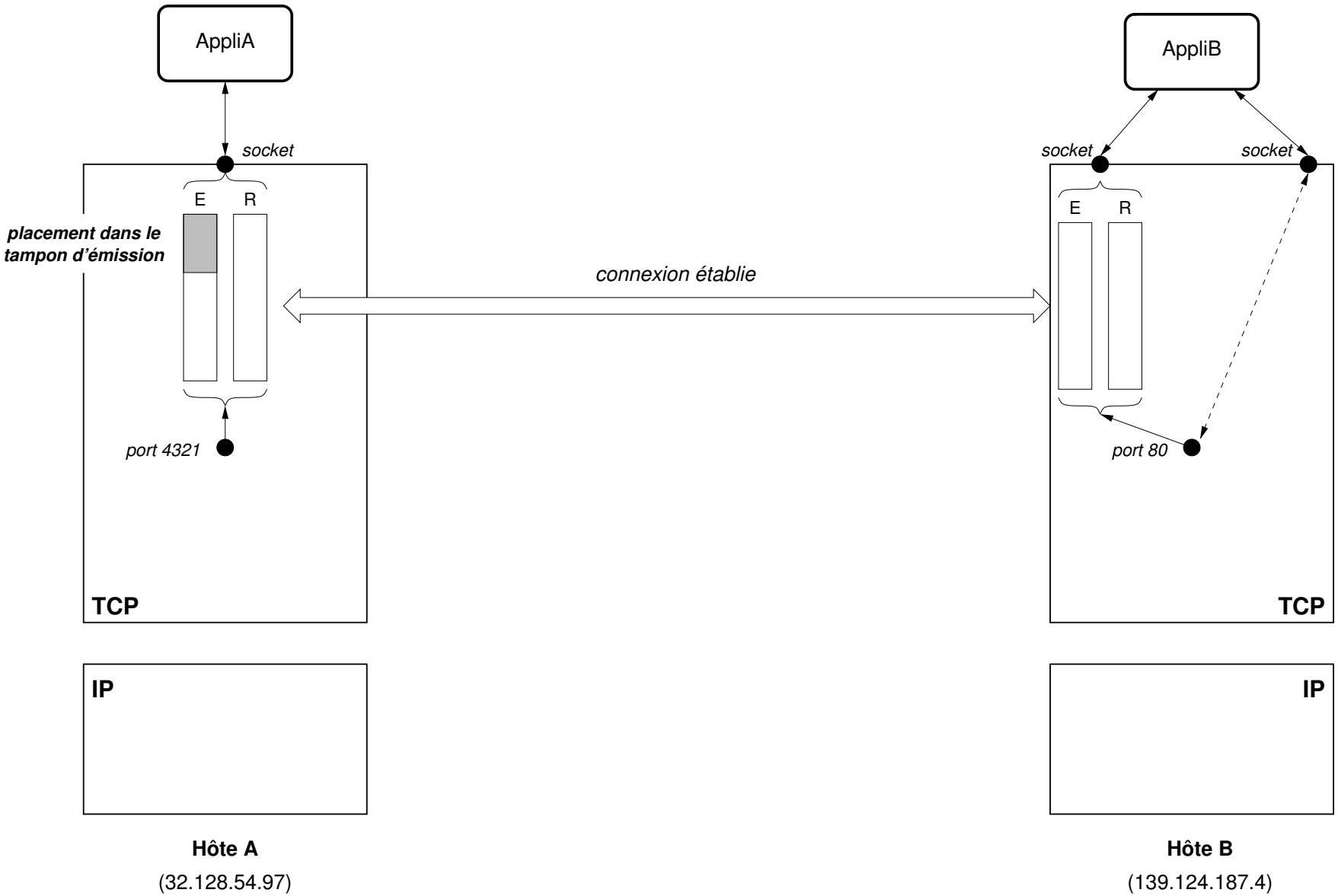
Flots d'octets et segments : exemple



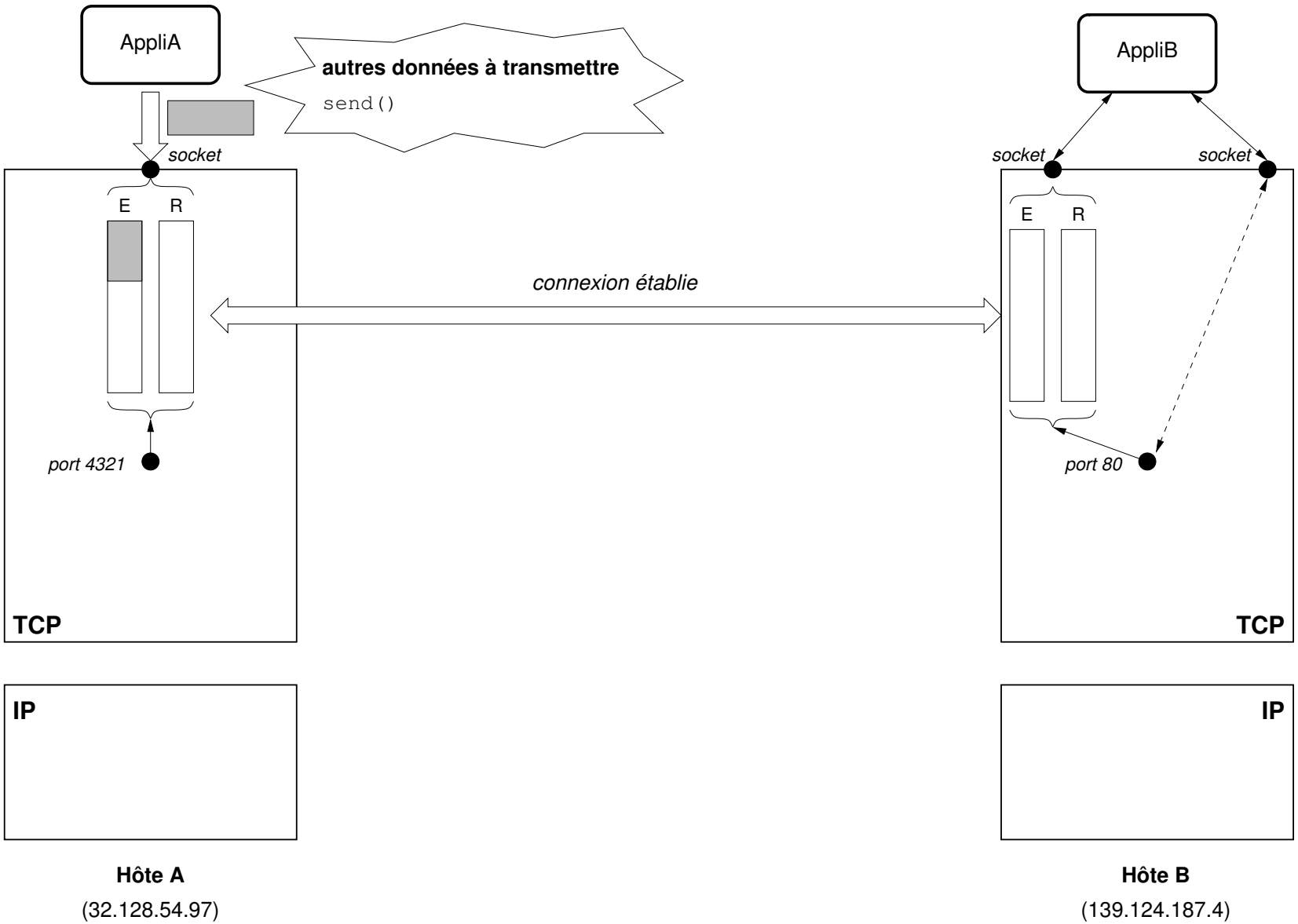
Flots d'octets et segments : exemple



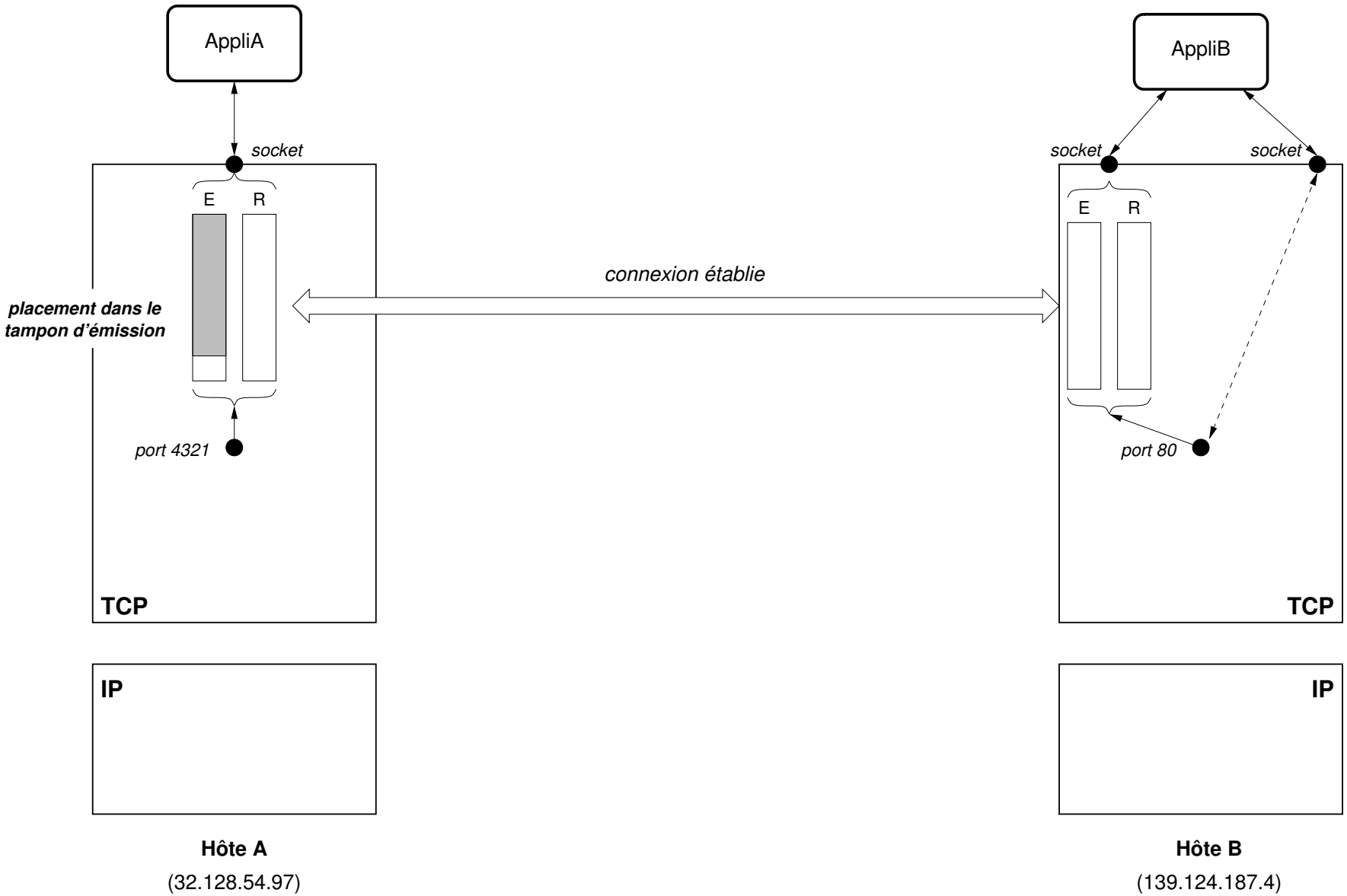
Flots d'octets et segments : exemple



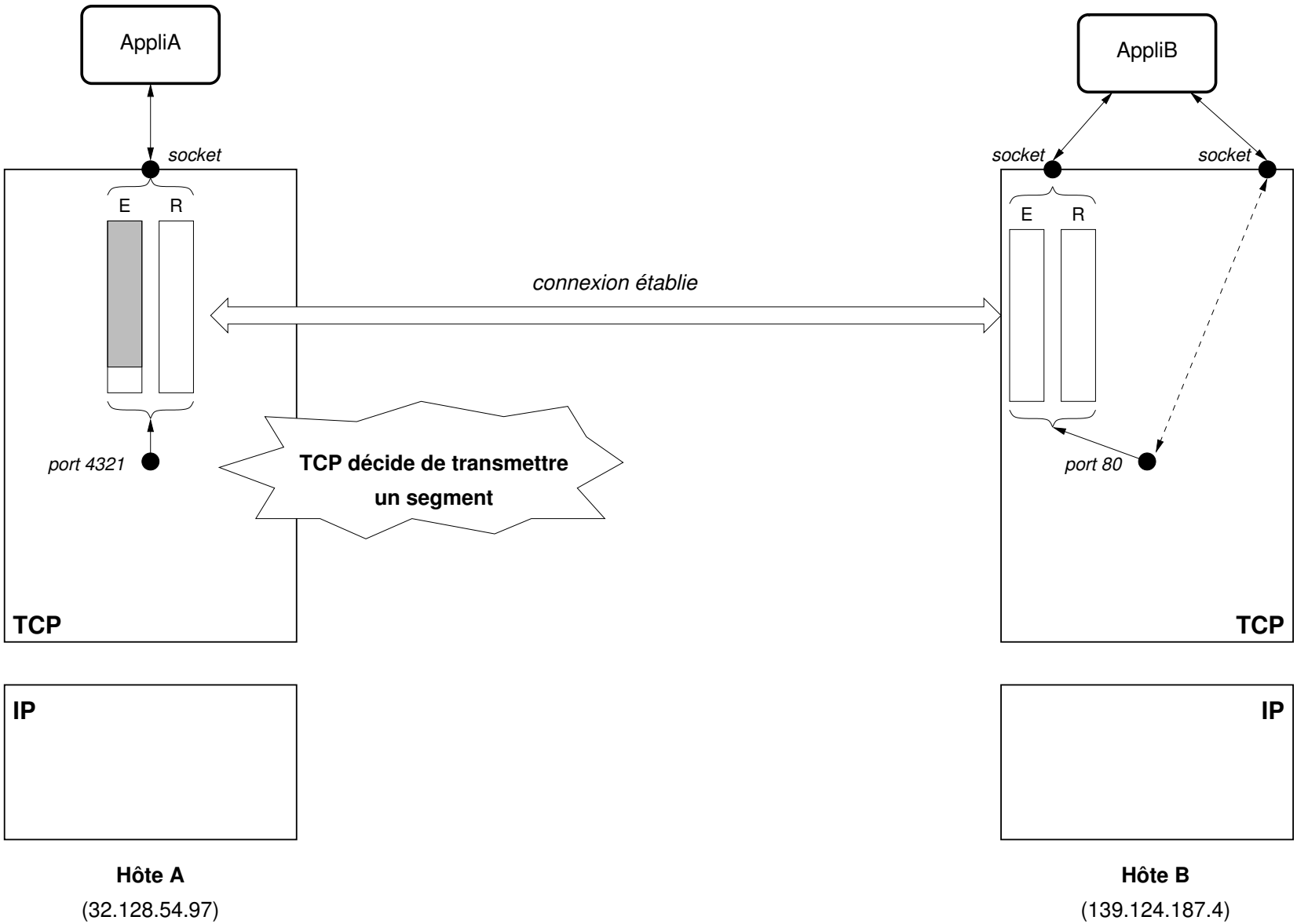
Flots d'octets et segments : exemple



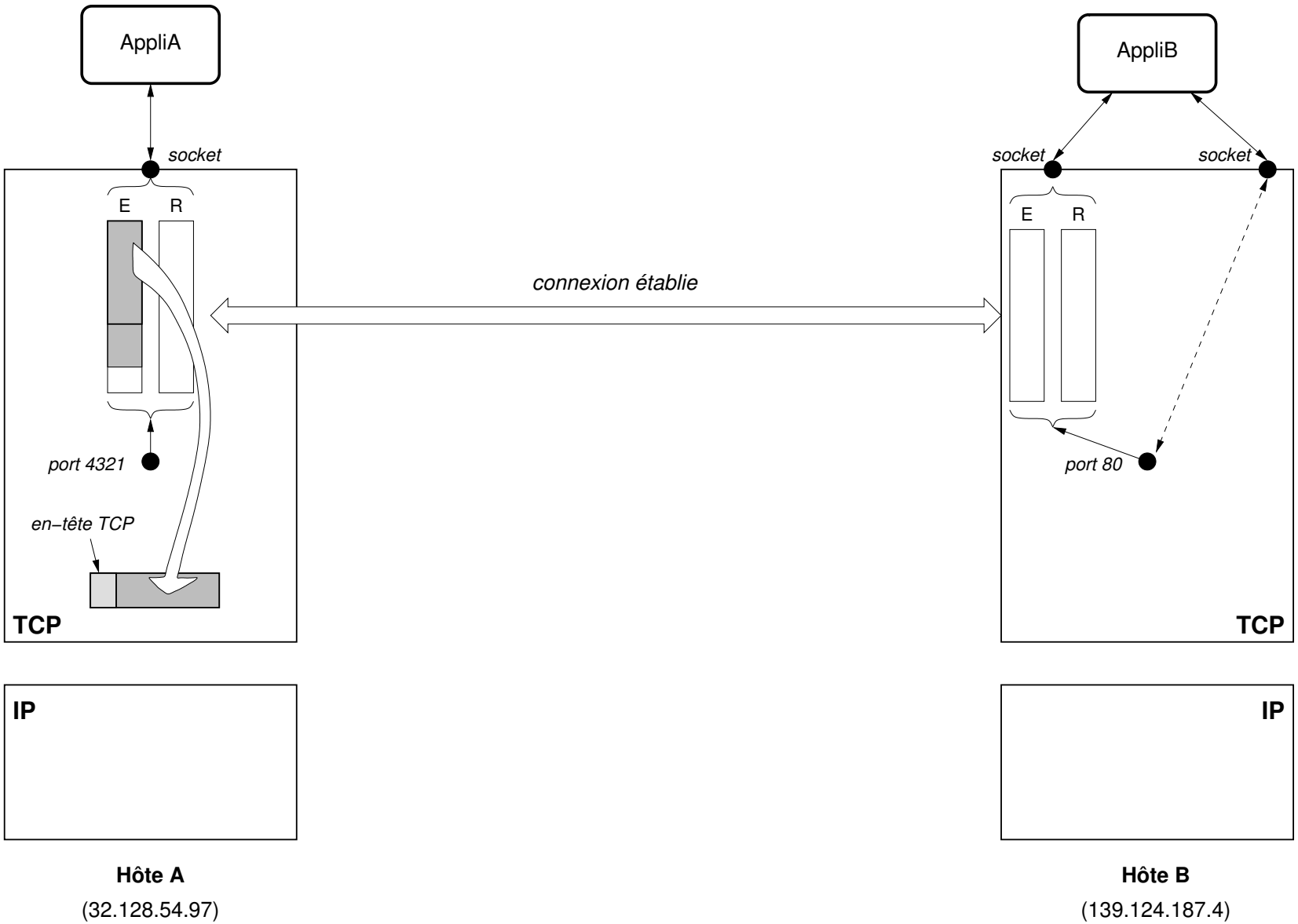
Flots d'octets et segments : exemple



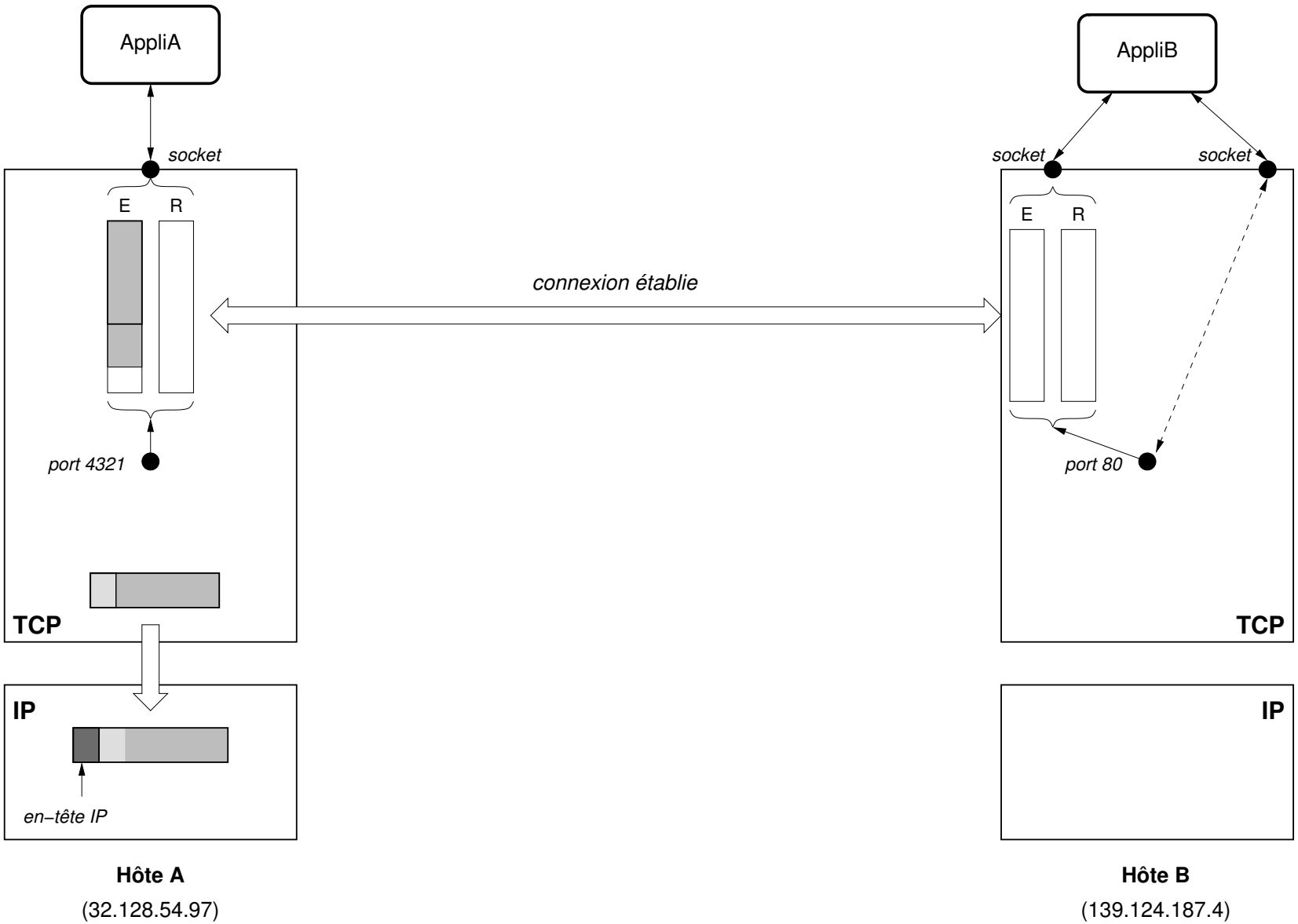
Flots d'octets et segments : exemple



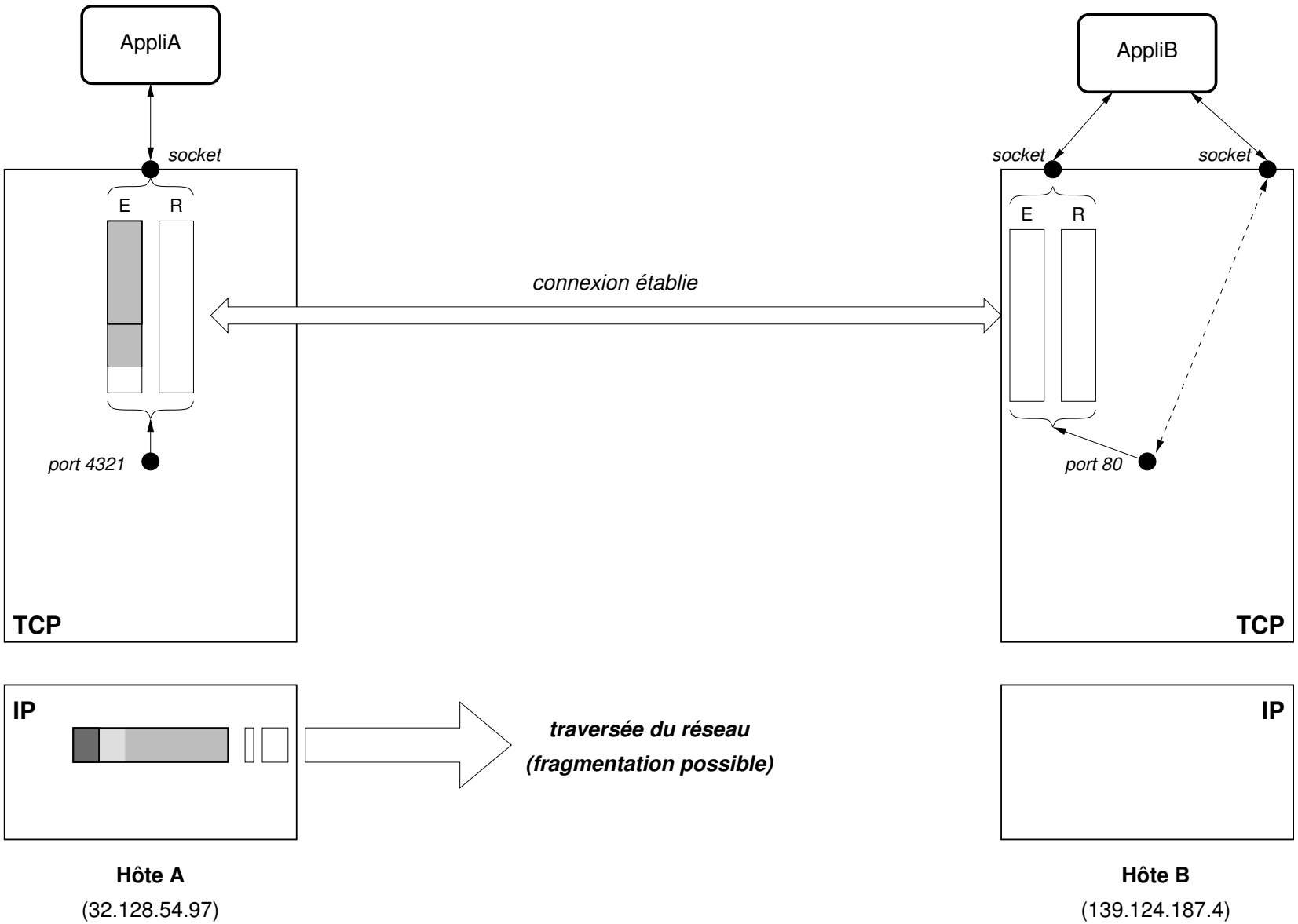
Flots d'octets et segments : exemple



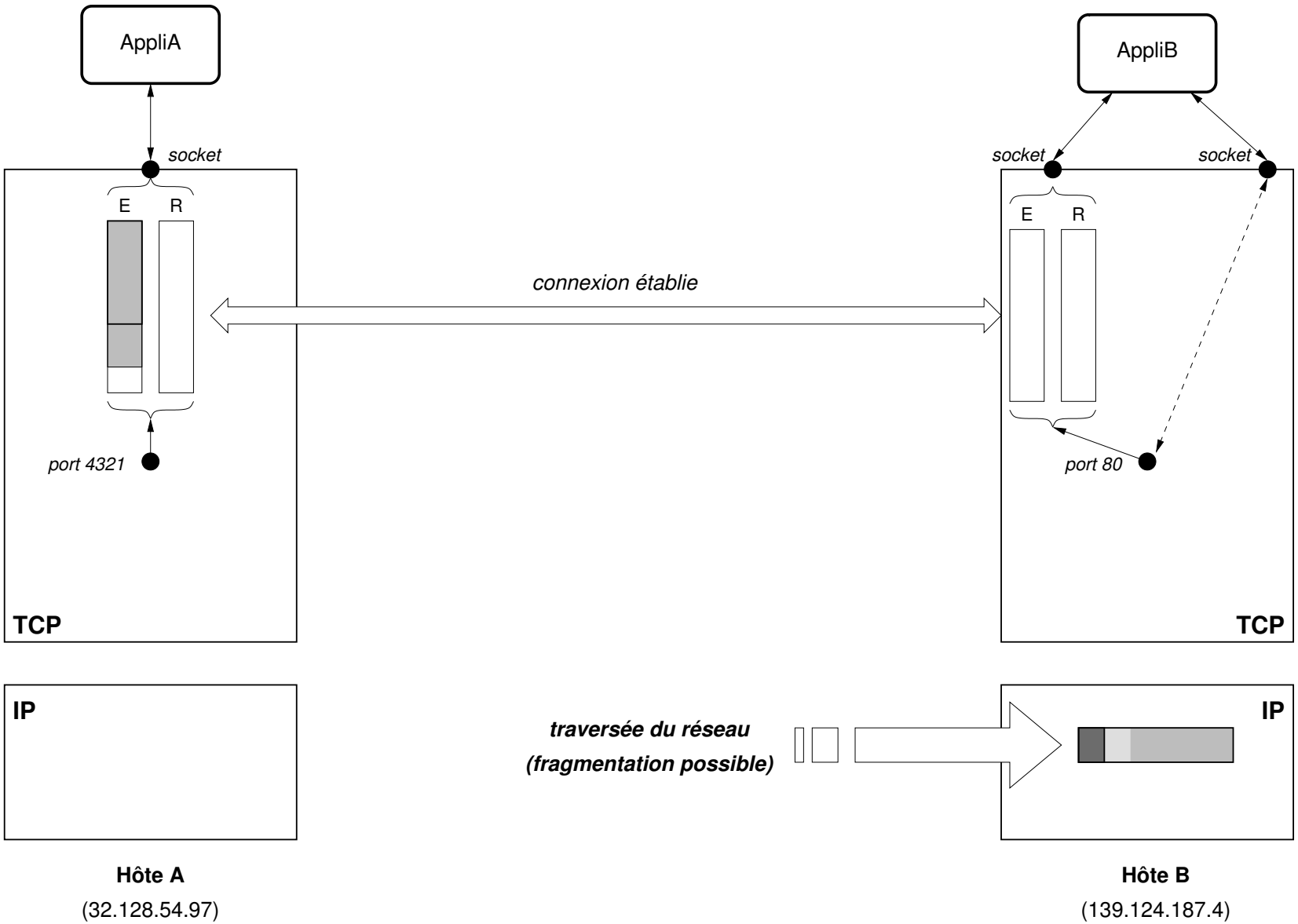
Flots d'octets et segments : exemple



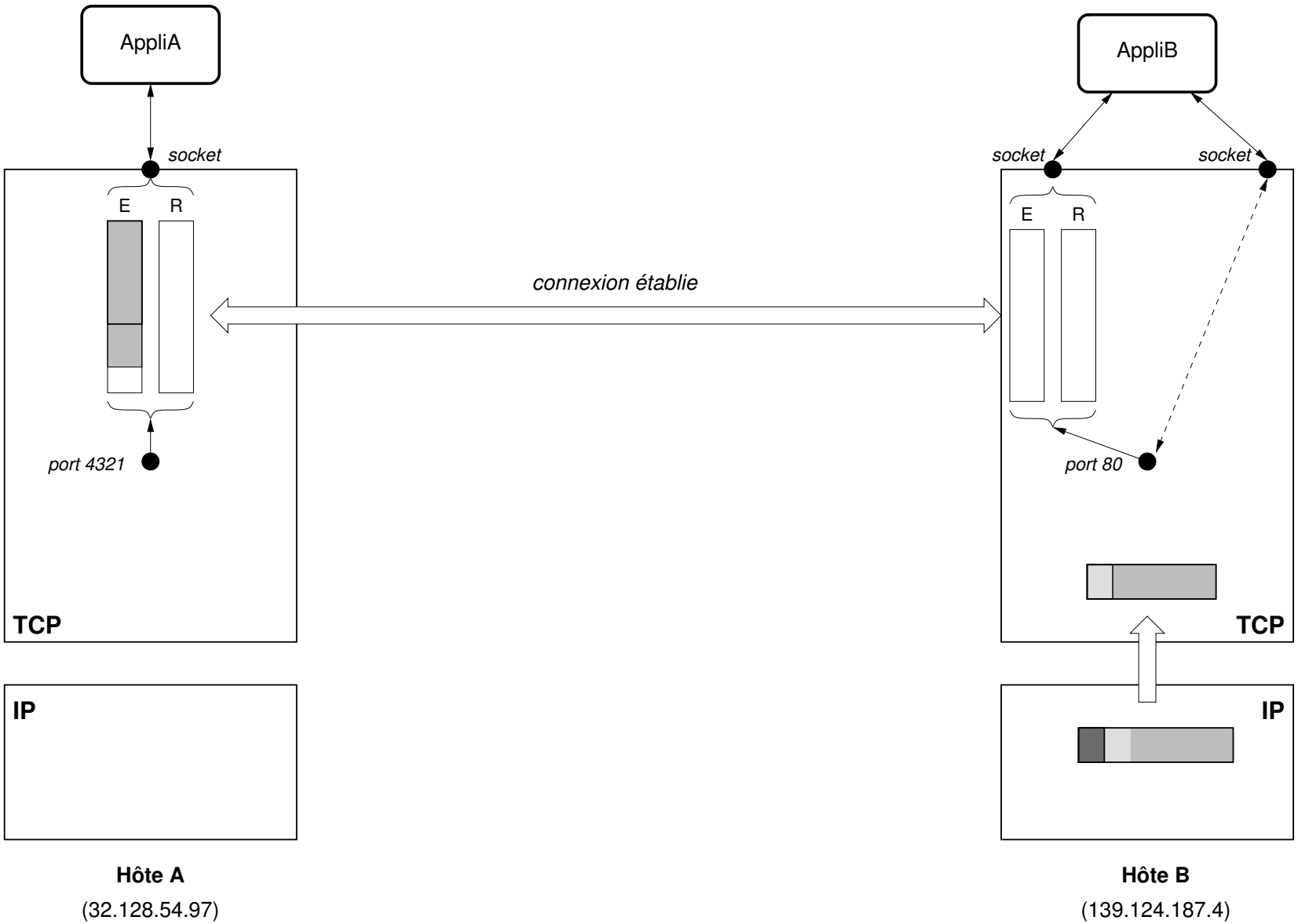
Flots d'octets et segments : exemple



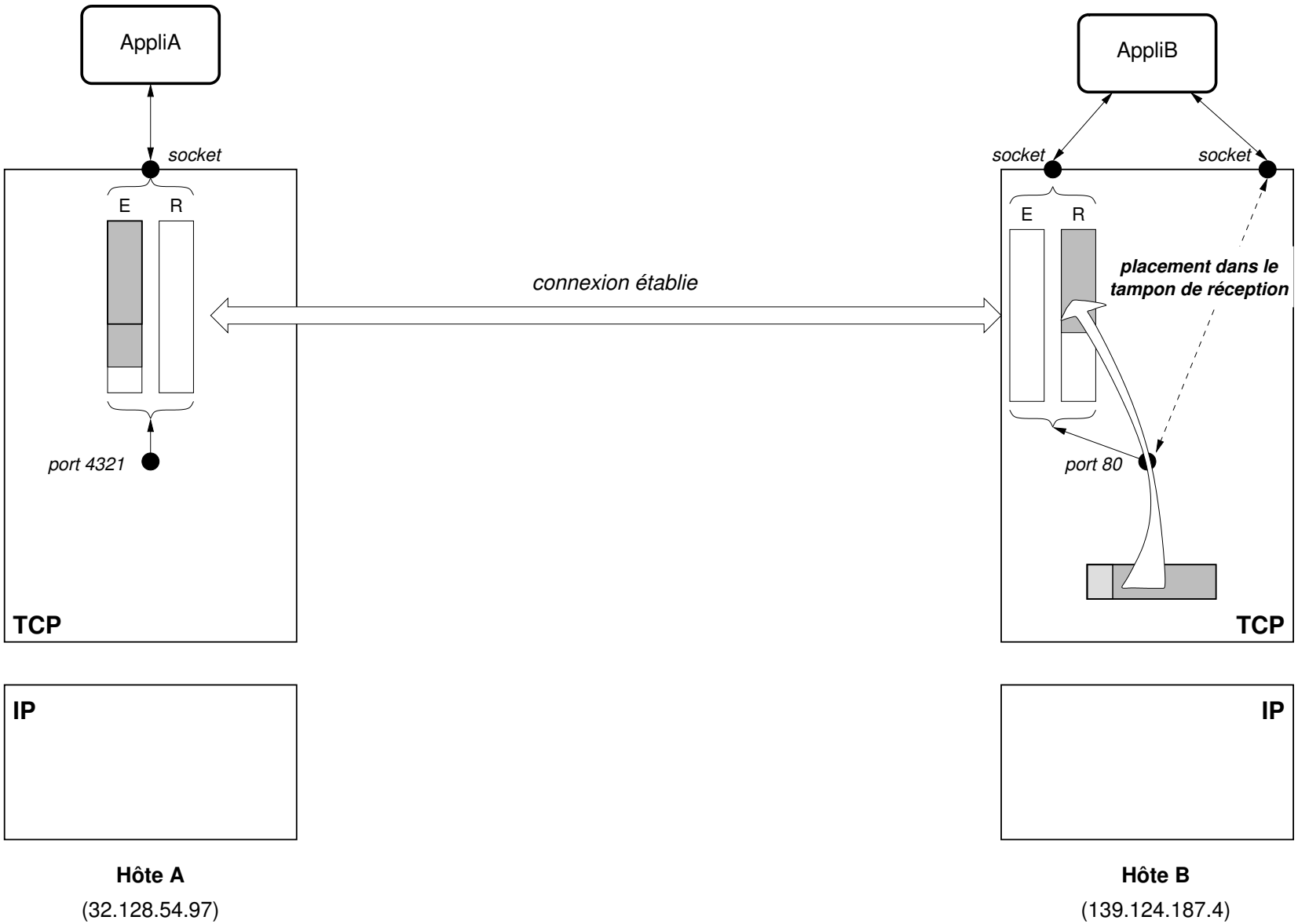
Flots d'octets et segments : exemple



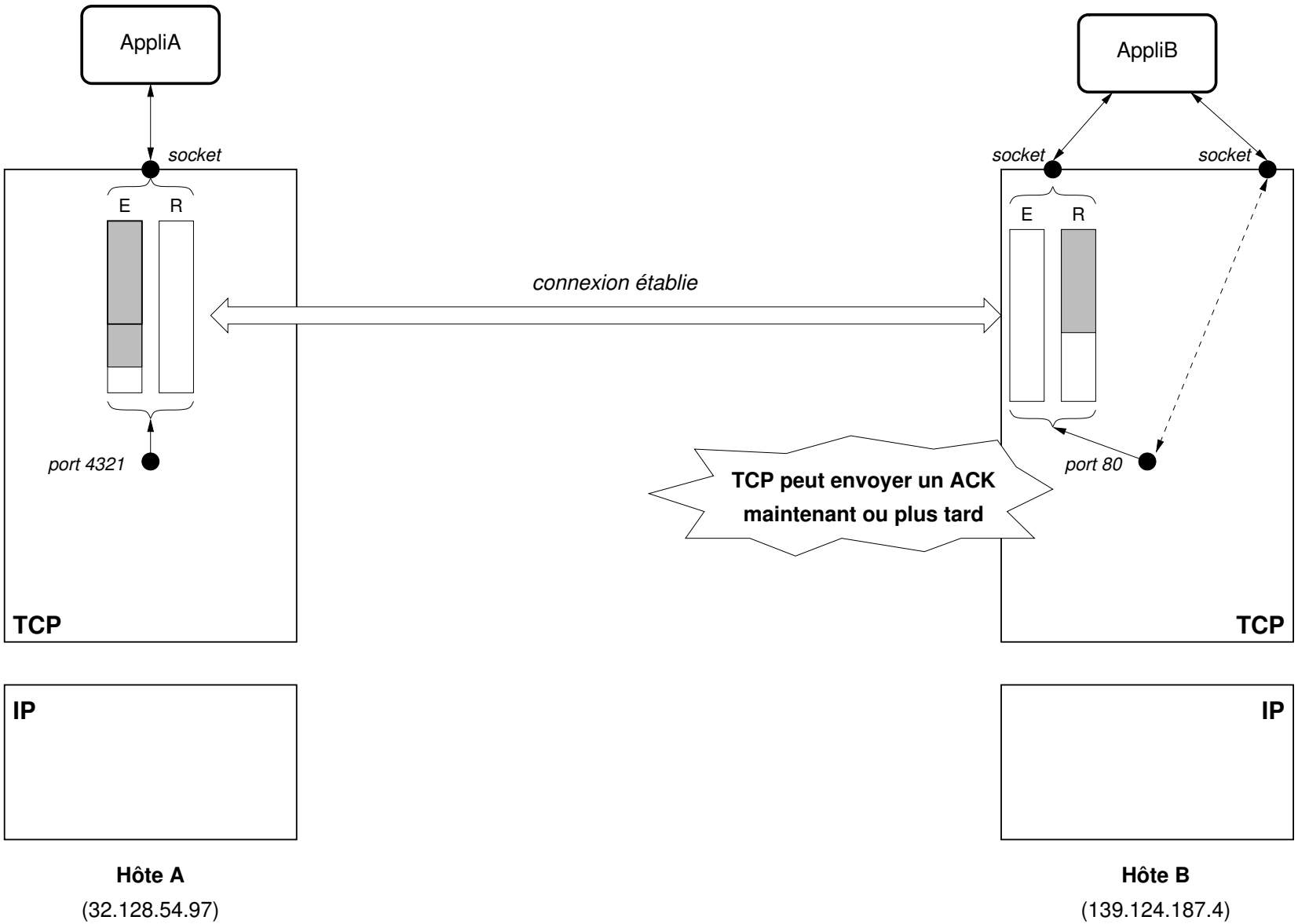
Flots d'octets et segments : exemple



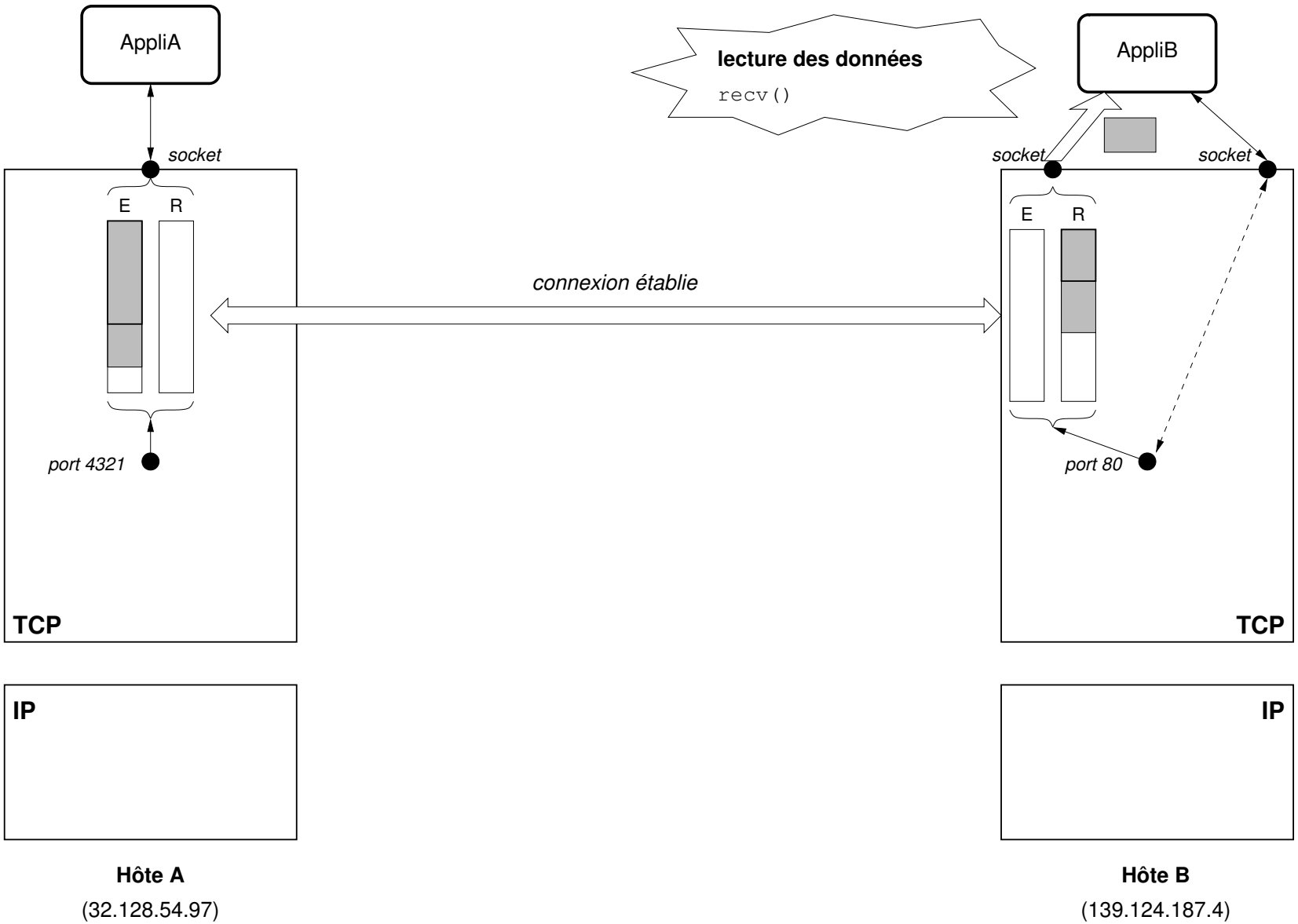
Flots d'octets et segments : exemple



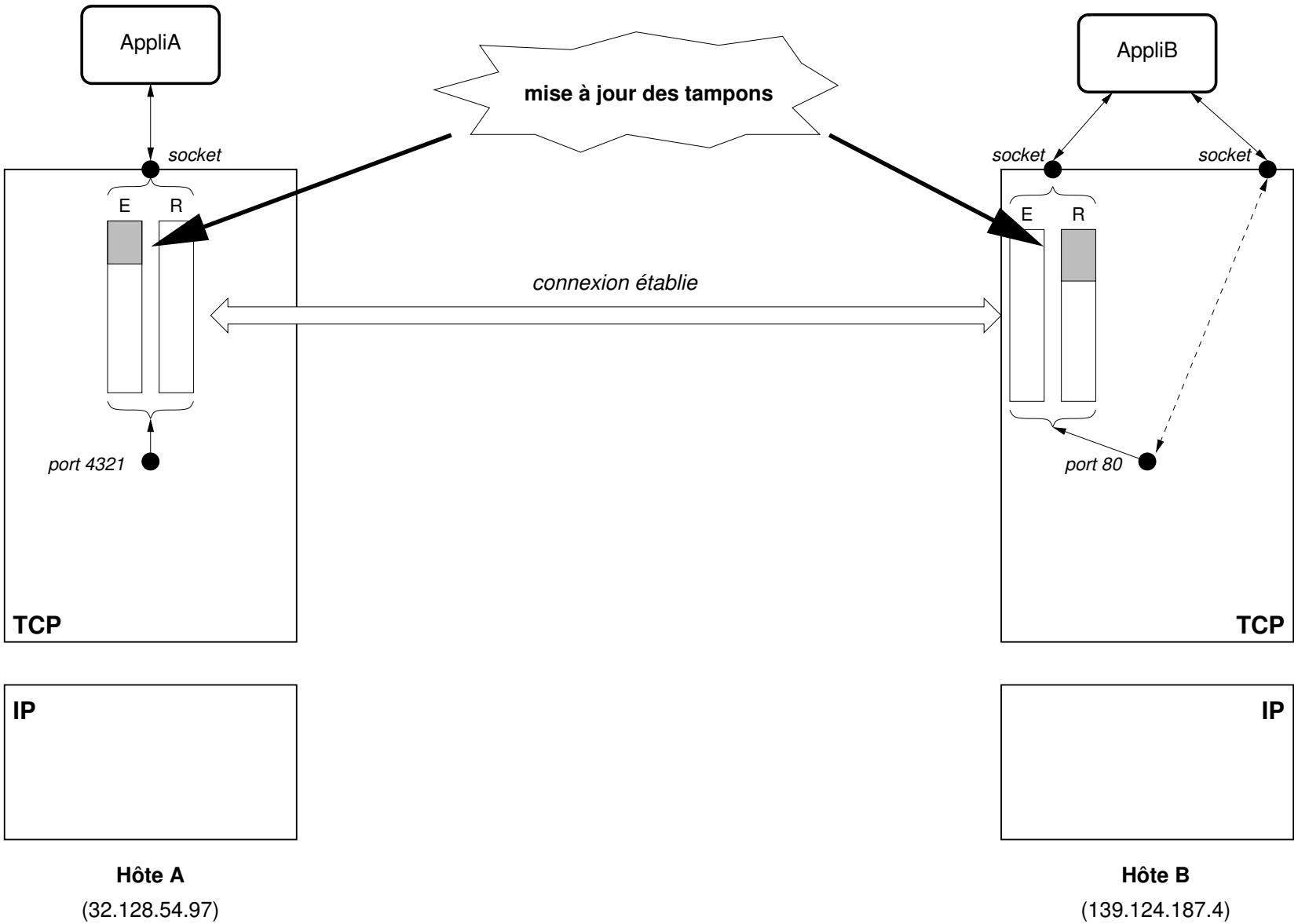
Flots d'octets et segments : exemple



Flots d'octets et segments : exemple

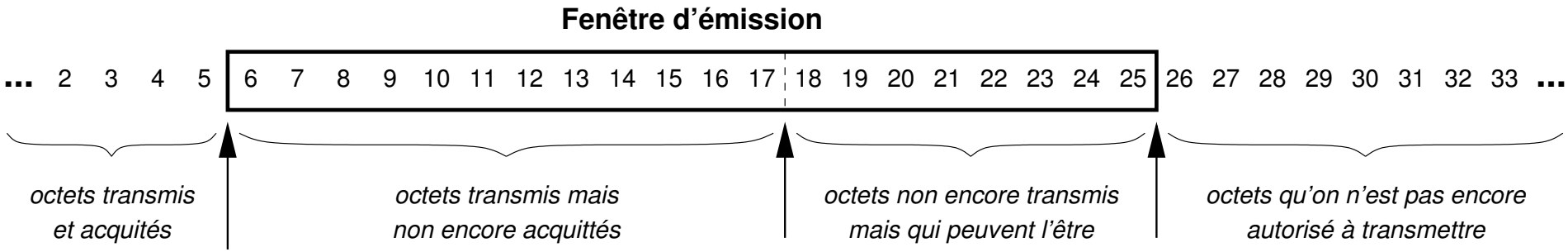


Flots d'octets et segments : exemple



Numéro de séquence et fenêtre glissante

- Pour une transmission efficace, TCP utilise une fenêtre glissante
- **TCP n'acquitte pas les segments mais les octets reçus**
- Tous les octets de données transmis portent un numéro : le numéro de séquence
- Les acquittements indiquent le numéro du prochain octet attendu
- La fenêtre glissante (émission) comporte alors 3 pointeurs :

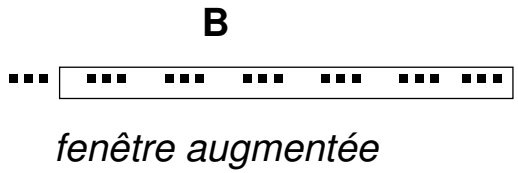


- Chaque côté de la connexion possède une fenêtre d'émission et une fenêtre de réception
- Les acquittements peuvent transiter avec les données (superposition)

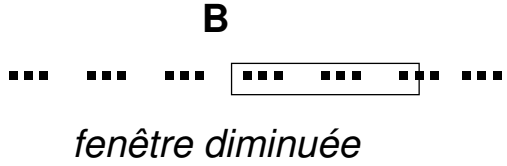
Taille de fenêtre variable et contrôle de flux

- La fenêtre glissante n'a pas une taille fixe
- Les segments (en particulier ACK) contiennent une information **taille de fenêtre** :

  A indique à B la place disponible dans son tampon de réception
- La réaction de B dépend de la taille annoncée :
  - augmentation : B augmente sa fenêtre et envoie les octets supplémentaires qu'elle comprend

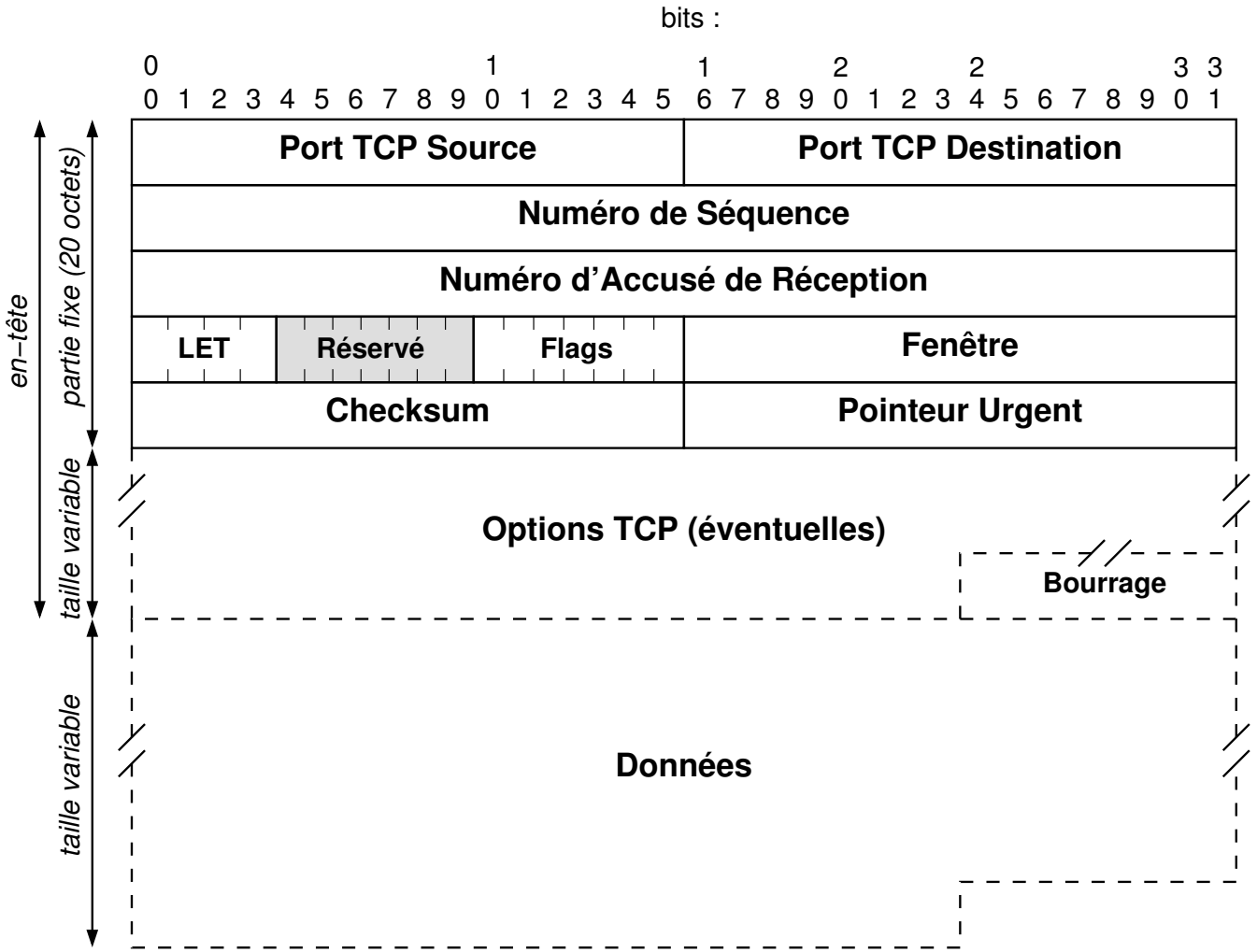


  - diminution : lors du glissement, B diminue sa fenêtre (sans exclure les octets qui y étaient déjà)



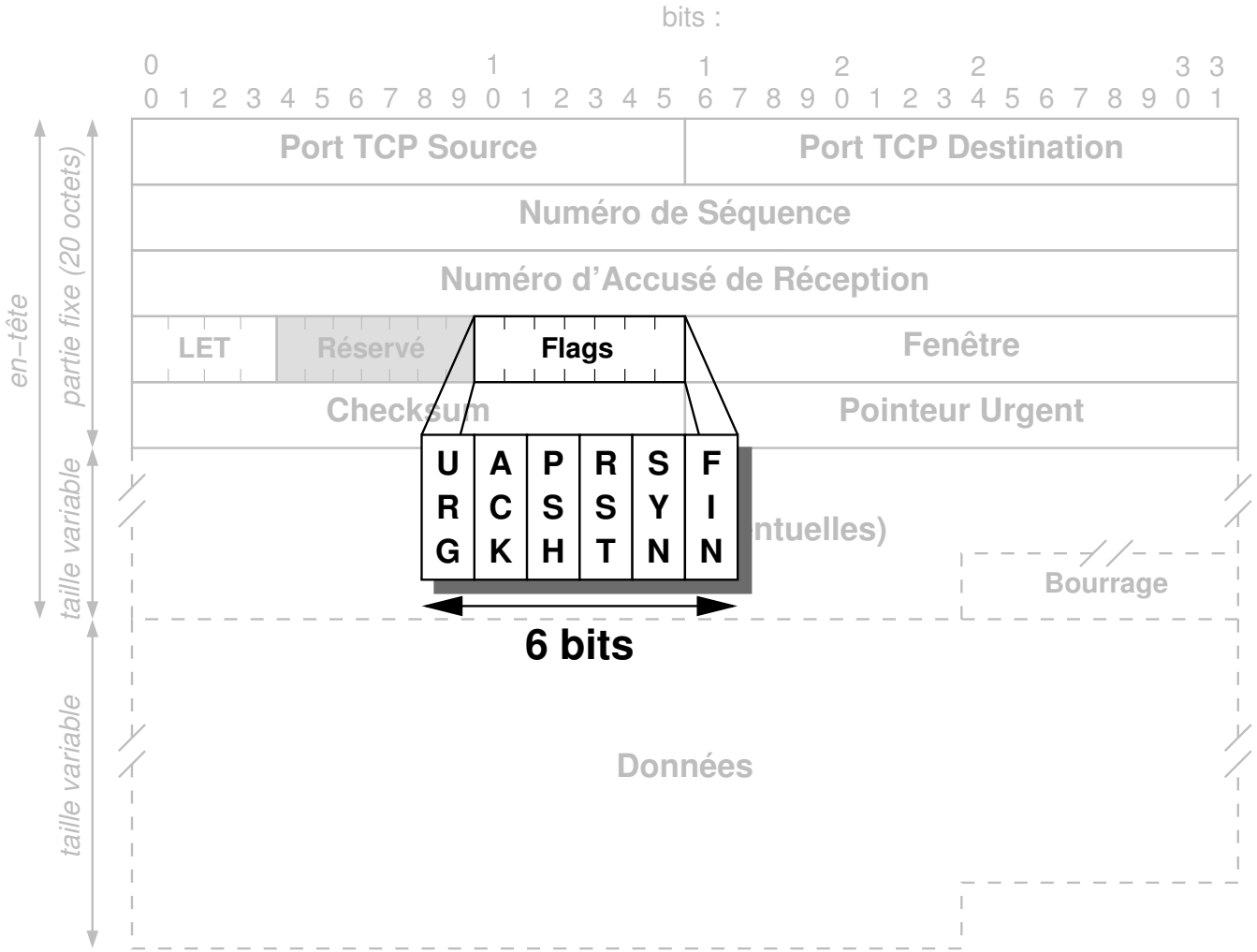
Format des segments TCP

Unité de données de protocole (PDU) échangée pour établir/libérer une connexion et transférer/acquitter des données

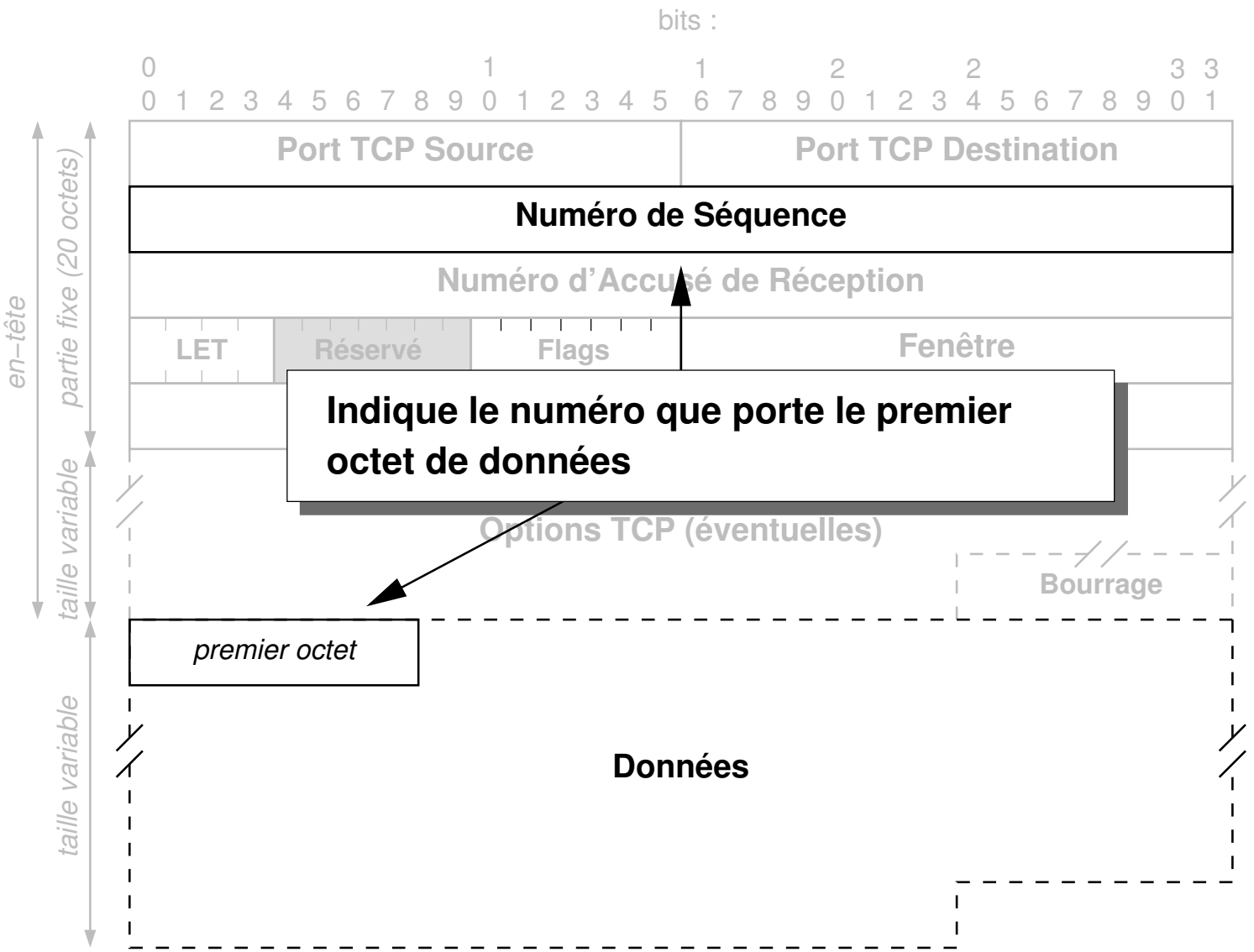


Format des segments TCP

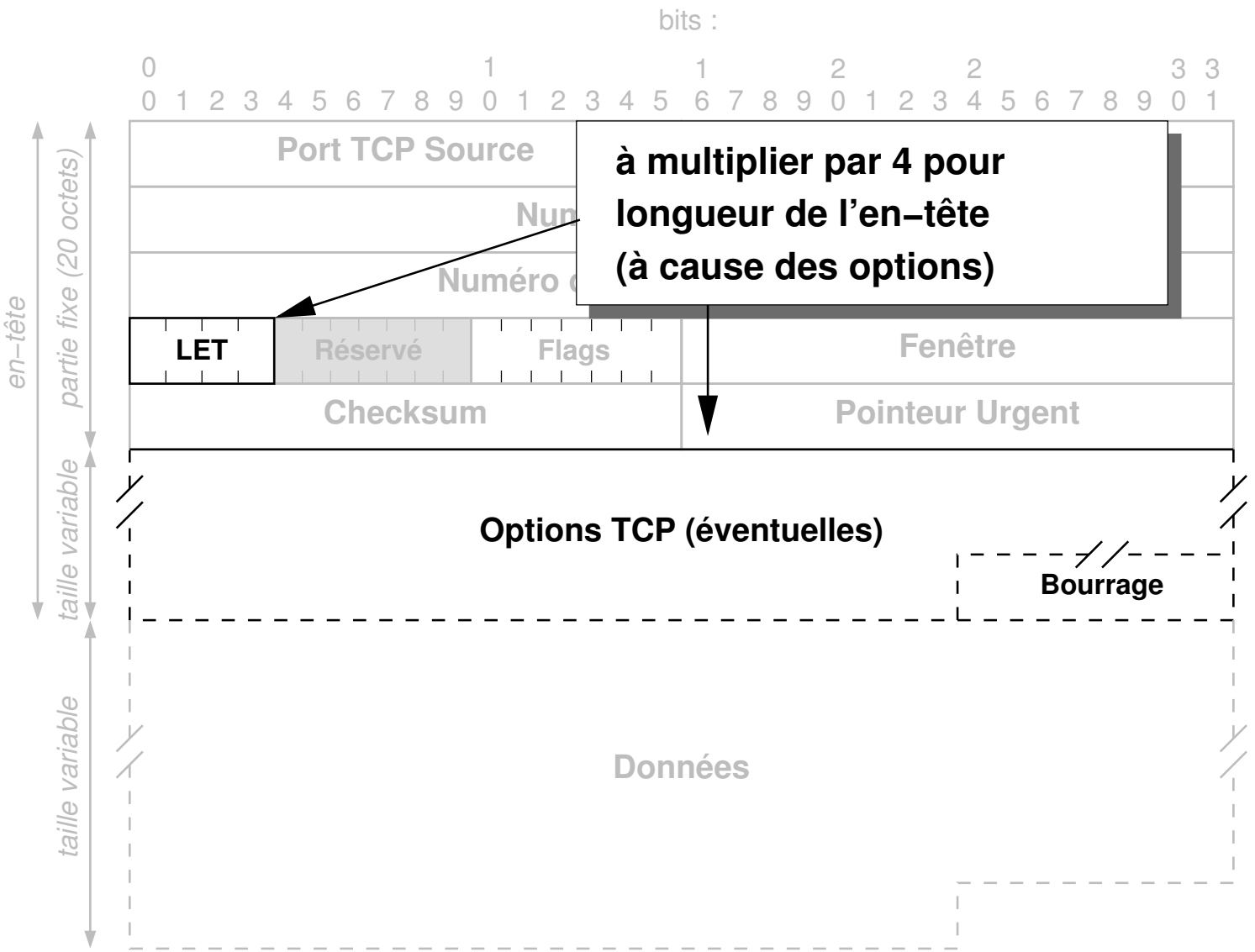
Unité de données de protocole (PDU) échangée pour établir/libérer une connexion et transférer/acquitter des données



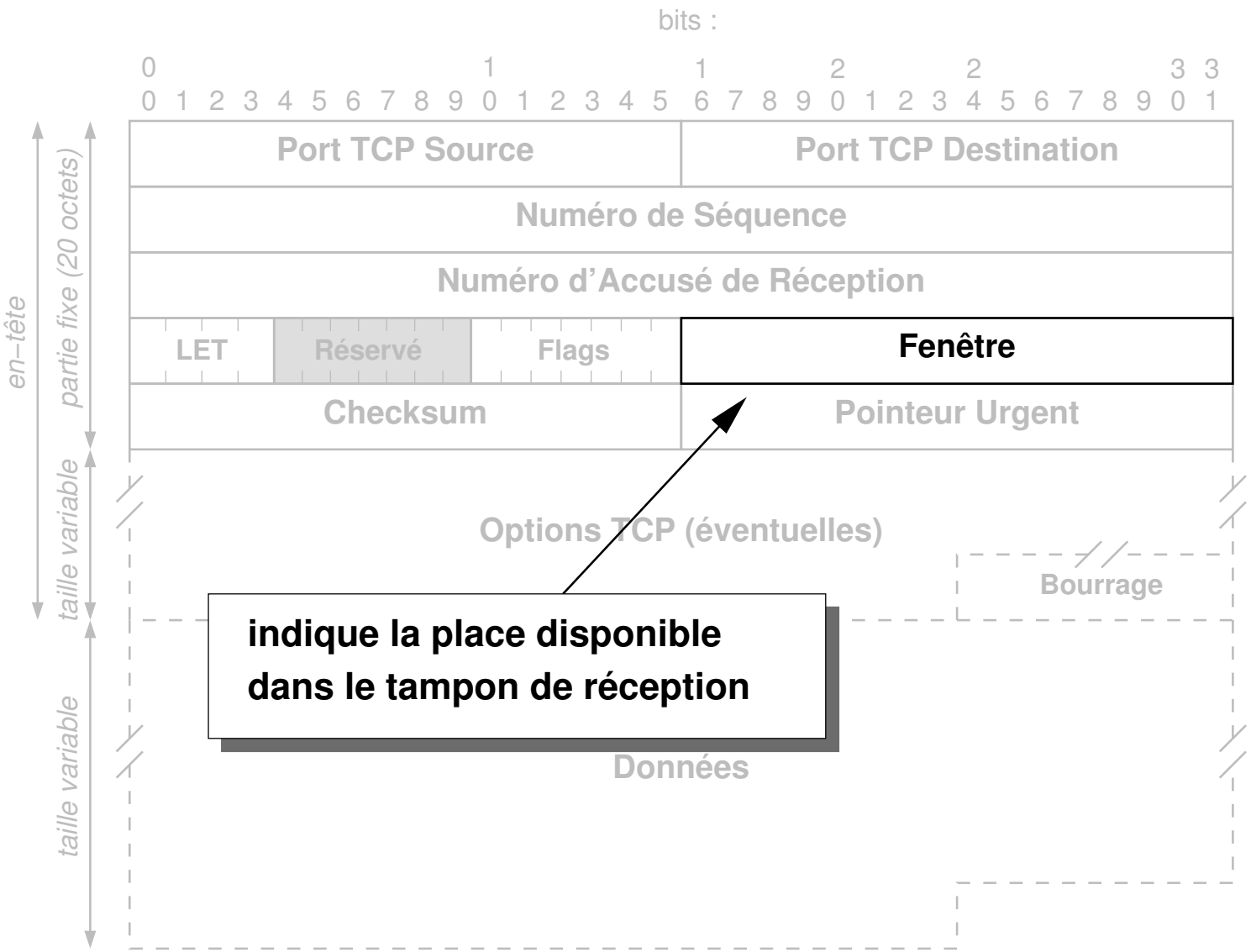
Segment TCP : champ Numéro de séquence



Segment TCP : champ LET

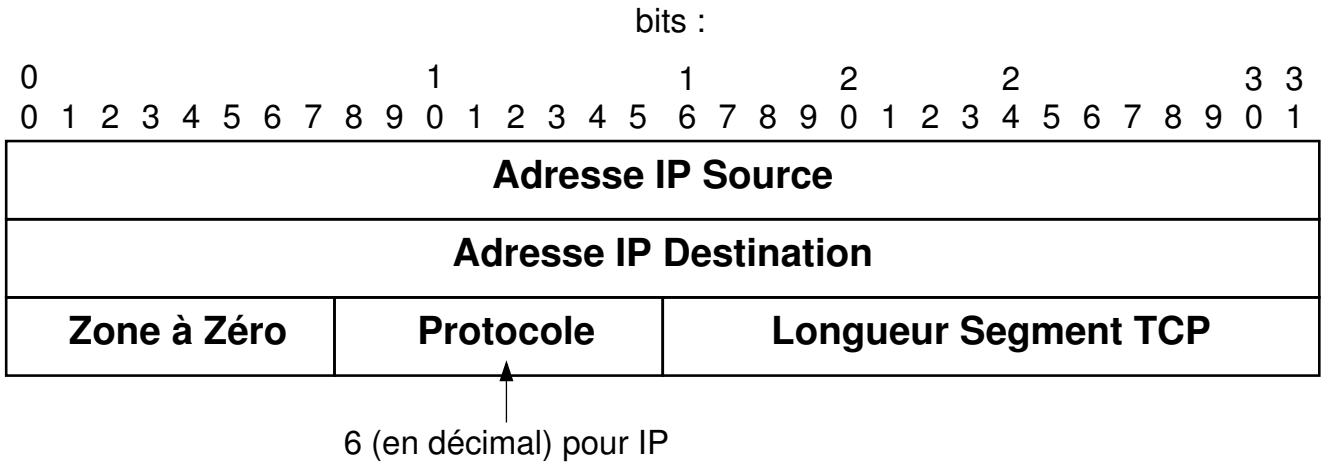


Segment TCP : champ Fenêtre



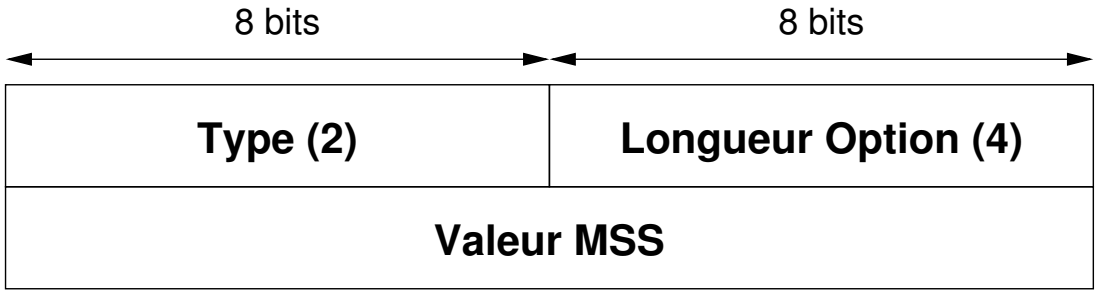
Segment TCP : Checksum

- Obligatoire
- Vérifie la totalité du segment + **Pseudo en-tête TCP**. Comme pour UDP, permet de s'assurer :
  - que les données sont correctes
  - que les ports sont corrects
  - *que les adresses IP sont correctes*
- Même calcul que IP/UDP (bourrage éventuel 1 octet à 0) + pseudo en-tête TCP
- Pseudo en-tête TCP (interaction avec IP) : (12 octets)



Segment TCP : option MSS

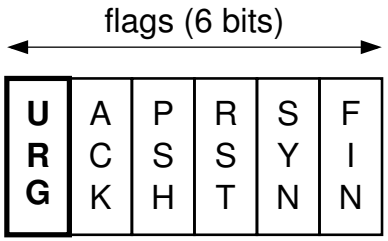
- Utilisée éventuellement pendant la phase de connexion (uniquement)
- Format (4 octets) :

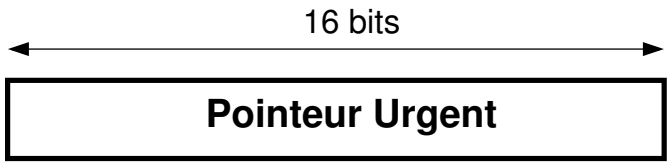


- Chaque côté indique la taille max **des données** des segments qu'il veut recevoir, appelée MSS (*Maximum Segment Size*) :
  - souvent MTU - 40 (en-têtes IP et TCP sans option)
  - dépendant de la taille des buffers de reception
  - par défaut 536 (576 octets pour le datagramme IP)
- Difficile à choisir pour l'Internet :
  - si trop petit, perte d'efficacité
  - si trop grand, risque de fragmentation
- L'idéal est le plus grand tel qu'aucun datagramme n'est fragmenté

Données Urgentes (Hors Bande)

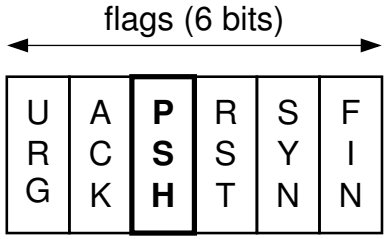
- Émetteur veut envoyer des données en urgence, sans attendre que le récepteur ait lu les données précédentes
- Exemple : envoi de Ctrl-C pour arrêter le traitement de l'application destinataire
- Le TCP émetteur place les données urgentes et envoie immédiatement le segment
- Le TCP récepteur interrompt l'application destinataire (sous Unix, signal SIGURG)
- Mise en œuvre par bit *URG* et *Pointeur Urgent* :
  - si bit URG = 1 : données urgentes présentes et *Pointeur Urgent* indique leur fin dans le segment (mais pas leur début)
  - si bit URG = 0 : pas de données urgentes, *Pointeur Urgent* ignoré





Remise forcée

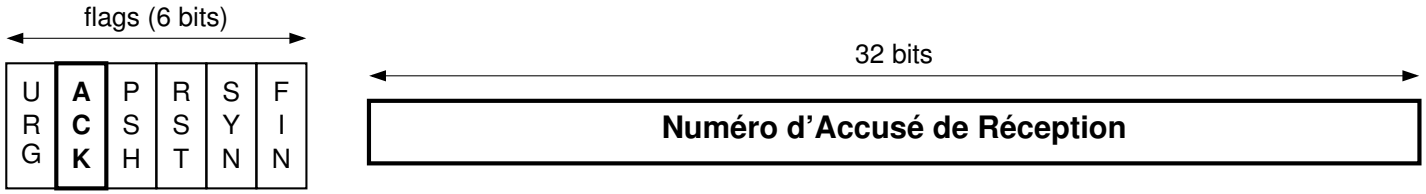
- Application émettrice demande de ne pas retarder l'émission des données, plutôt que de tamponner
- Particulièrement utile dans le cas d'un terminal virtuel : après le retour à la ligne, il faut envoyer, voire même pour chaque caractère tapé ou déplacement de souris (X-Window)
- TCP émetteur place le bit PSH à 1 :



- Le TCP récepteur doit remettre les données au plus vite (ne pas tamponner dans sa mémoire)

Acquittements et Retransmissions

- Le bit *ACK* à 1 indique que le champ *Numéro d'Accusé de Réception* doit être utilisé

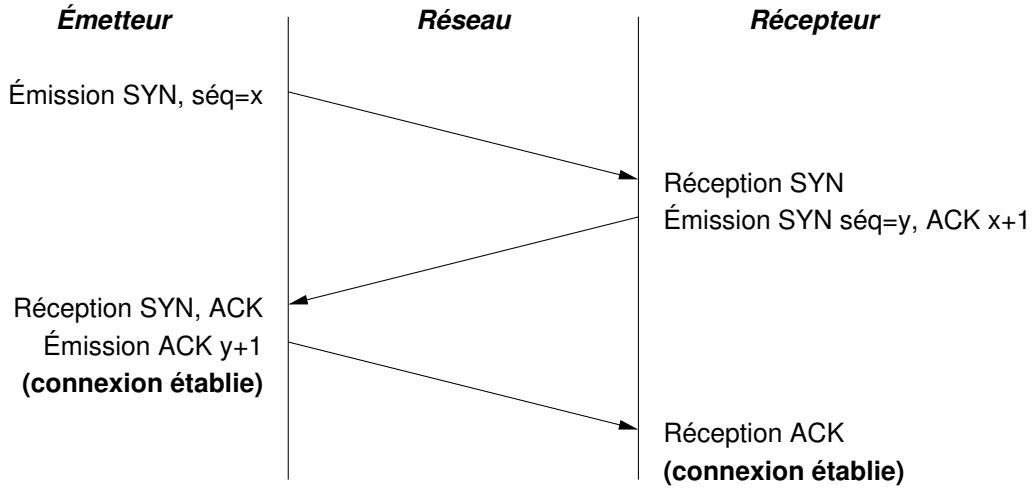


- Un segment non acquitté est retransmis après *timeout*
- Un segment retransmis peut contenir plus de données que le précédent
- Numéro ACK n'acquitte pas le segment : indique le numéro du prochain octet attendu
- L'ACK est *cumulatif* : a des avantages et des inconvénients
- Rejet global ou sélectif (le plus courant)
- En pratique l'émetteur ne renvoie que le premier segment non acquitté

Établissement d'une connexion

URG	**ACK**	PSH	RST	**SYN**	FIN

- En 3 temps, avec bits SYN et ACK

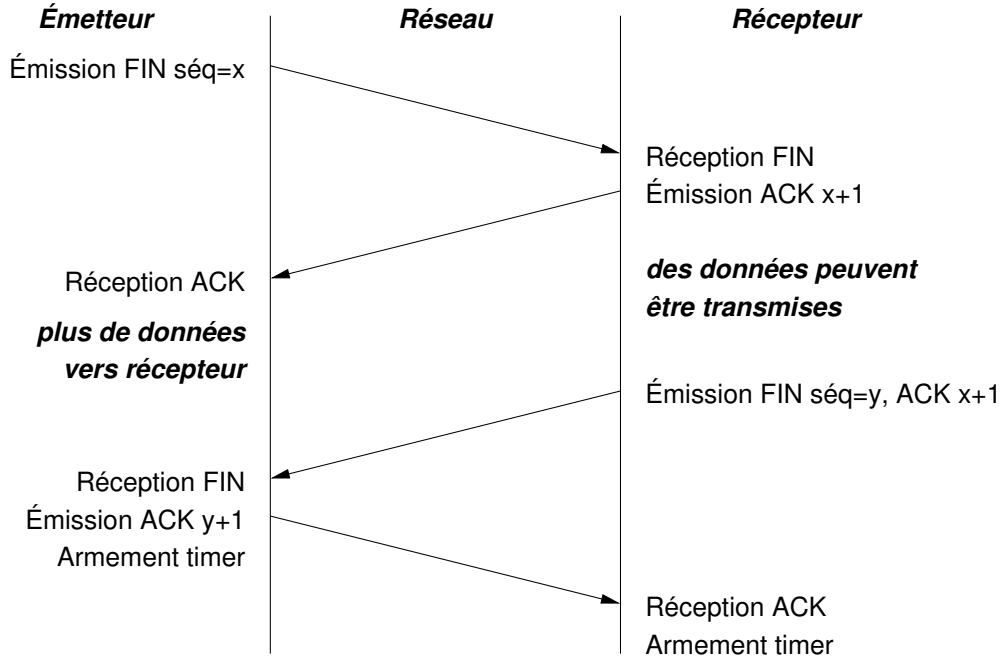


- Permet d'ignorer des demandes retardées
- Résoud aussi les connexions simultanées des deux côtés
- Numéro de Séquence choisi (presque) aléatoirement dans chaque sens

Libération d'une connexion

URG	**ACK**	PSH	RST	SYN	**FIN**

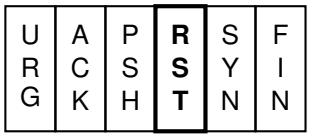
- Processus en trois temps modifié, avec bits FIN et ACK
- Connexion libérée lorsque chaque côté a indiqué qu'il n'avait plus de données à émettre :



- À la fin, un temporisateur est utilisé pour laisser le temps aux segments retardés d'arriver ou d'être détruits
- Puis les données relatives à la connexion sont détruites

Fermeture brutale d'une connexion

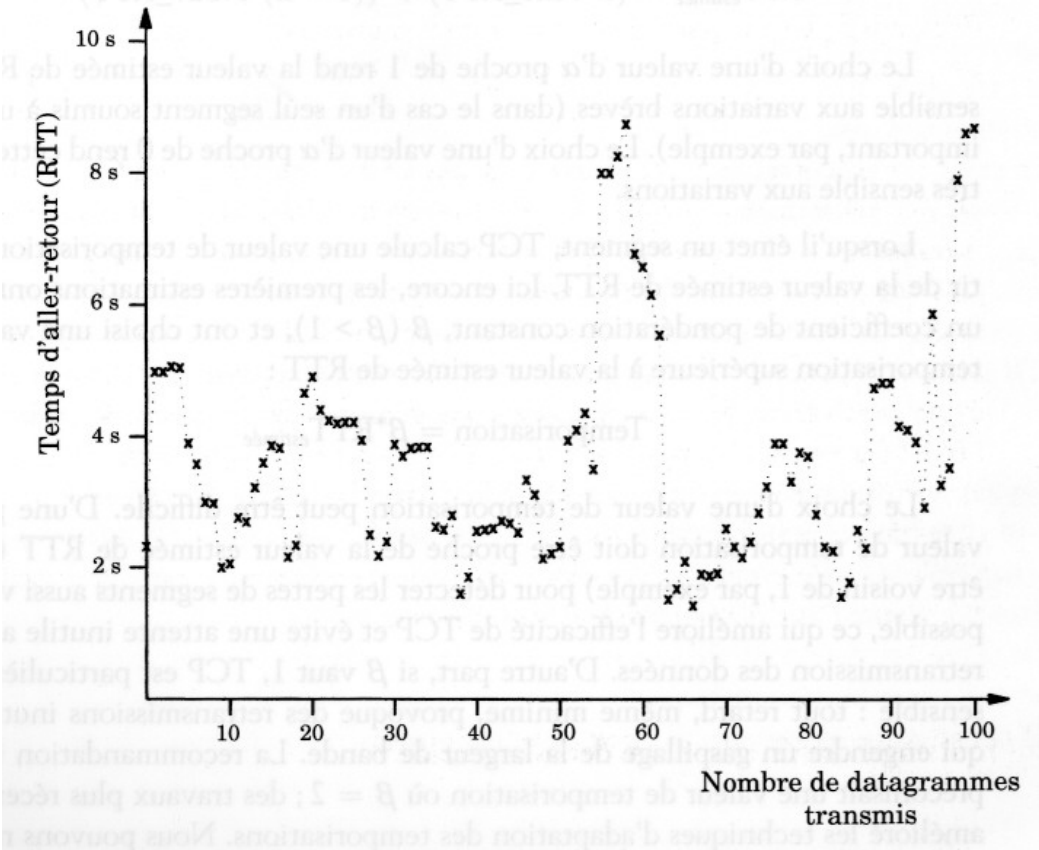
- Bit RST à 1 :



- Signifie qu'il a y eu un problème grave
- Connexion libérée immédiatement
- Données non traitées/segments retardés sont perdus
- Applications sont informées
- Utilisé aussi pour refuser une demande de connexion

Temporisation et retransmission

- Un segment (données) non acquitté doit être retransmis
- Problèmes importants sur Internet :
  - comment calculer/estimer le RTT (*Round Trip Time*) ?
  - quelle durée du temporisateur ?



source : TCP/IP Architecture, protocoles, applications. D. Comer, Edts Dunod

Ajustement dynamique du RTT estimé

- Pour chaque connexion, TCP gère une variable RTT
- Soit M = temps mis pour retour de l'ACK d'un segment
- RTT est mis à jour selon la formule :

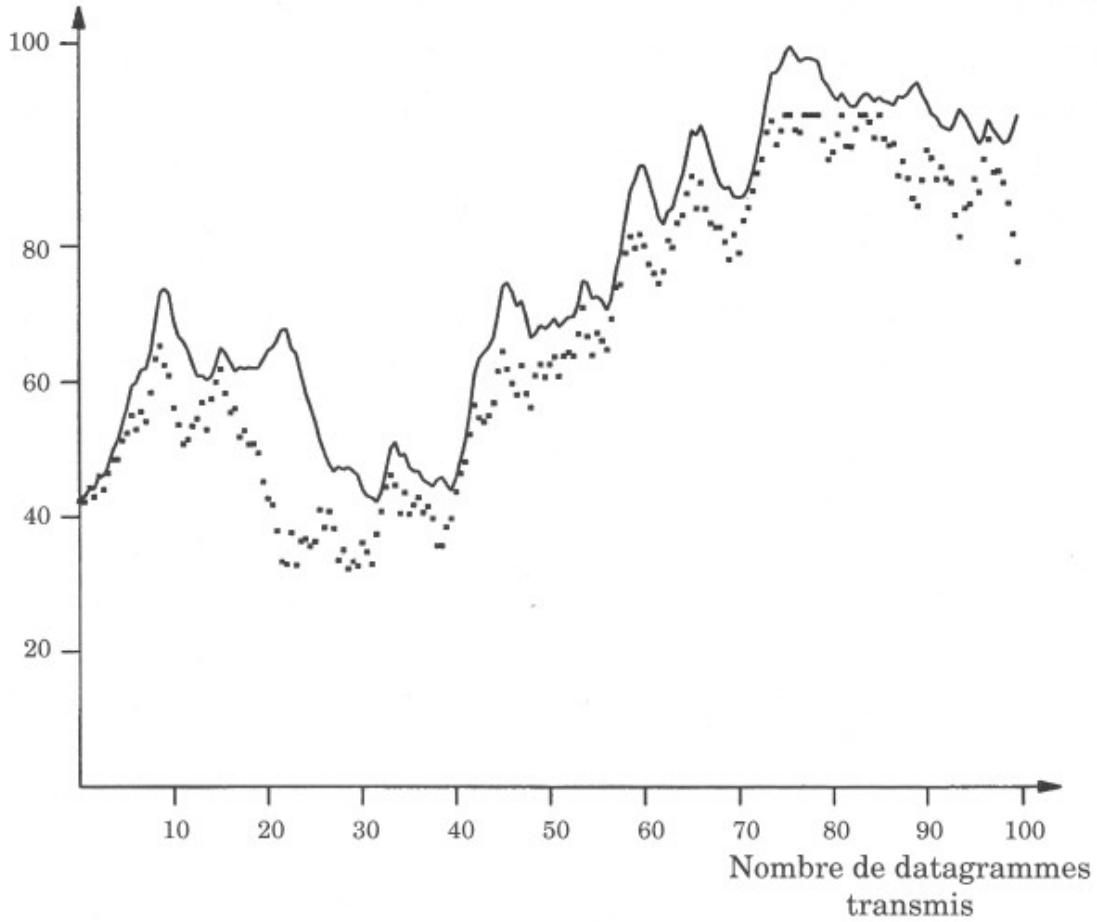
$$RTT = \alpha RTT + (1 - \alpha)M$$

où $0 \leq \alpha \leq 1$ (généralement 7/8) est le poids attribué à l'anciennce valeur de RTT

- Valeur du timer $T = \beta RTT$ (à l'origine du standard $\beta = 2$)
- Jacobson (1988) a proposé une amélioration du calcul de RTT et de T, pour tenir compte de la variance. Devenu la règle depuis 1989 :
  - $Diff = M - RTT$
  - $RTT = RTT - \delta \times Diff$, où généralement δ vaut $1/2^3$
  - $Dev = Dev + \rho \times (|Diff| - Dev)$, où généralement ρ vaut $1/2^2$
  - $T = RTT + \eta \times Dev$, où généralement η vaut 4

Ajustement dynamique du timer : exemple

En pointillés, des valeurs aléatoires de RTT. En trait plein, le timer calculé :



source : TCP/IP Architecture, protocoles, applications. D. Comer, Edts Dunod

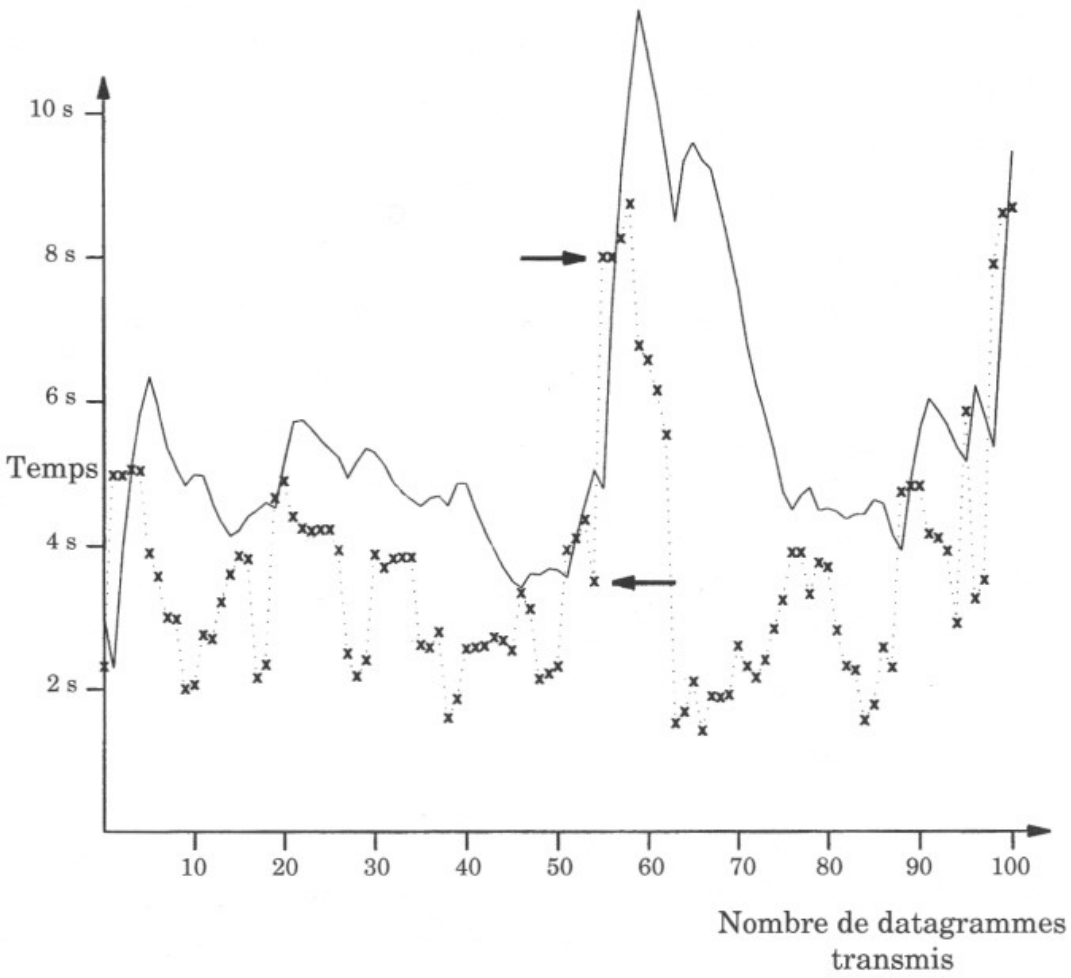
RTT et Algorithme de Karn

- **Problème :** si un segment est retransmis et un ACK reçu, doit-il être pris pour le premier ou le dernier segment ?
- Selon le choix, le calcul de RTT et de T peut être grandement faussé !
- Algorithme de Karn (radio amateur) :
  - Ne pas mesurer M pour un segment retransmis
  - Mais augmenter T, selon la formule :

$$T = \gamma \times T$$

où généralement γ vaut 2.

Timer et grandes variations RTT



source : TCP/IP Architecture, protocoles, applications. D. Comer, Edts Dunod

Il reste des situations impossibles à prévoir mais la réaction est généralement très bonne.

Syndrome de la fenêtre stupide

- Soit une application qui ne lit les données qu'octet par octet
- L'émetteur envoie suffisamment d'octets pour remplir le tampon de réception
- Le TCP récepteur envoie une taille de fenêtre nulle
- Lorsque l'application lit un octet, TCP enverra une taille de fenêtre de 1
- L'émetteur peut alors envoyer 1 octet !
- Puis taille de fenêtre 0
- etc.

C'est pas mieux si c'est l'émetteur qui envoie de lui-même des petits segments !

Éviter la fenêtre stupide

- Côté récepteur : (Algorithme de Clark) n'annoncer une réouverture de la fenêtre que lorsque sa taille est :
  - soit égale à la moitié du tampon de réception
  - soit égale au MSS

- Côté émetteur : (Algorithme de Nagle) retarder le plus possible l'envoi :
  - si des données n'ont pas été acquittées, placer les nouvelles données en tampon
  - ne les envoyer que si atteint taille maximale du segment ou la moitié de la fenêtre d'émission
  - lorsqu'un ACK arrive, envoyer ce qu'il y a (même si le segment n'est pas plein)

  ceci même en cas de PUSH !

Il est parfois nécessaire de désactiver l'algo de Nagle, notamment pour X-Window (il faut envoyer les mouvements de la souris de suite et non les tamponner...).

TCP et la congestion

- La congestion produit des retards importants et des pertes de datagrammes
- Retransmettre des segments retardés/détruits aggrave la congestion jusqu'à l'**effondrement congestif**
- TCP en tient compte et cherche à éviter la congestion et réagit en cas de congestion :
  - démarrage lent
  - diminution dichotomique
- Nécessite la gestion d'une **fenêtre de congestion**
- Fenêtre utilisée = min(Fenêtre récepteur, Fenêtre congestion)

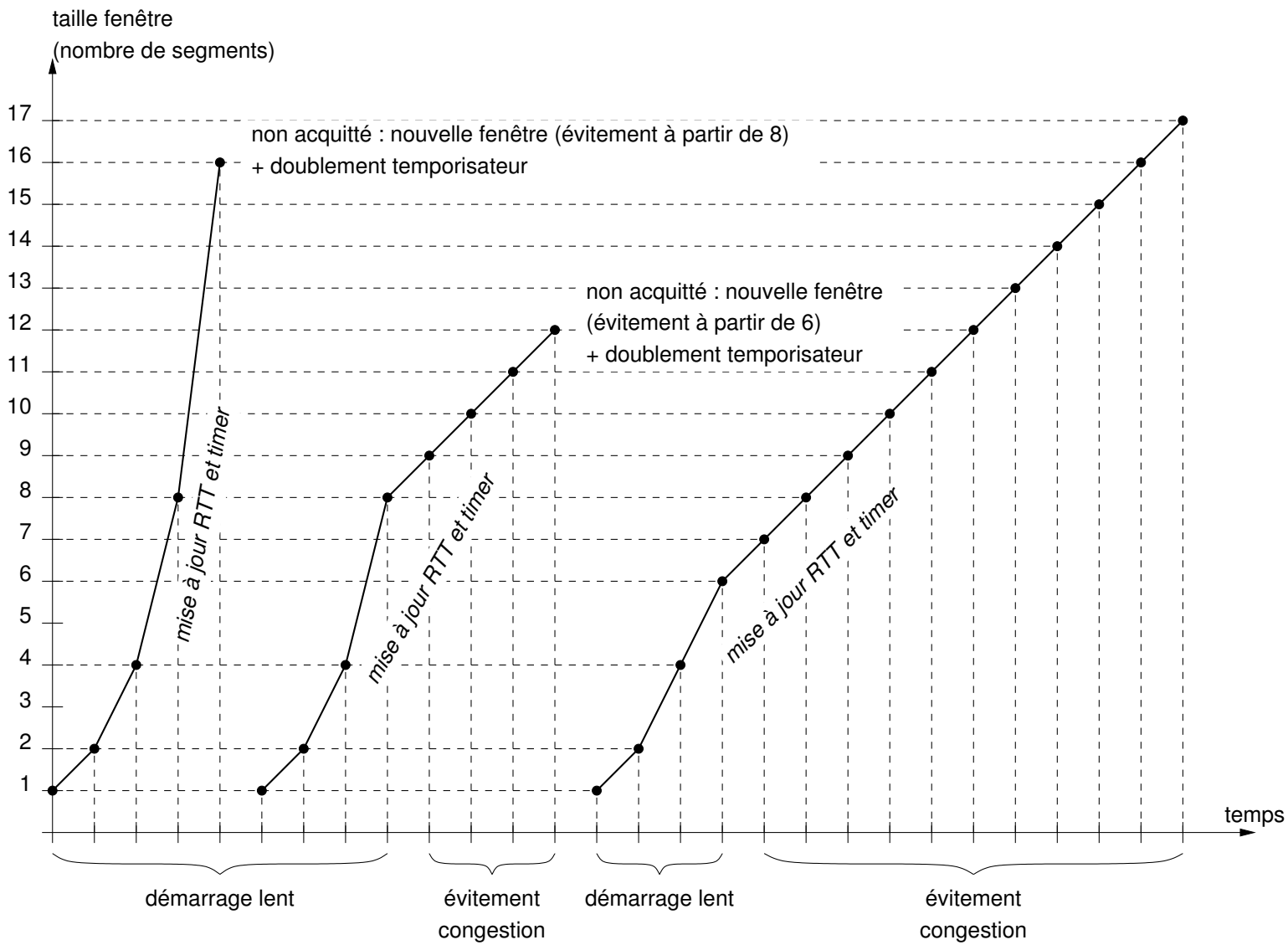
TCP et la congestion : démarrage lent

- Au début de la connexion ou suite à une congestion, effectuer un démarrage lent avec fenêtre congestion = 1 segment
- Démarrage lent : pour chaque segment acquitté, augmenter de 1 segment la fenêtre de congestion
- **lent mais croissance exponentielle !**

TCP et la congestion : diminution dichotomique

- TCP estime que la perte d'un segment est due à la congestion
- Si segment perdu, fixer un seuil à la moitié de la fenêtre de congestion (taille mini = 1 segment)
- Augmenter la temporisation (algo de Karn)
- Recommencer un démarrage lent jusqu'à atteindre le seuil
- phase d'**évitement de congestion** : à partir du seuil, n'augmenter que de 1 segment pour l'ensemble des segments acquittés (croissance linéaire)

TCP et la congestion : illustration



Automate d'états fini de TCP

- transitions d'états : **événement / action**, où :
  - événement :
    - *italiques* : primitive appelée par le programme
    - ACK, SYN, etc. : réception d'un segment avec ce bit positionné
  - action :
    - ACK, SYN, etc. : envoi segment
    - - : rien
    - tempo : temporisation de 2 fois la durée de vie d'un segment
- en pointillés, les événements exceptionnels