Mise à niveau en C

Sockets TCP sous linux

Enseignant: P. Bertin-Johannet

Objectif

- Ce cours se concentre sur la communication réseau en C sous **linux**.
- Pour cela nous utiliserons des sockets de type **unix**.
- Les interfaces utilisées fonctionnent aussi sous MacOS et BSD mais le fonctionnement en interne peut différer de ce qui est présenté dans le cours.

Comunication réseau sous linux

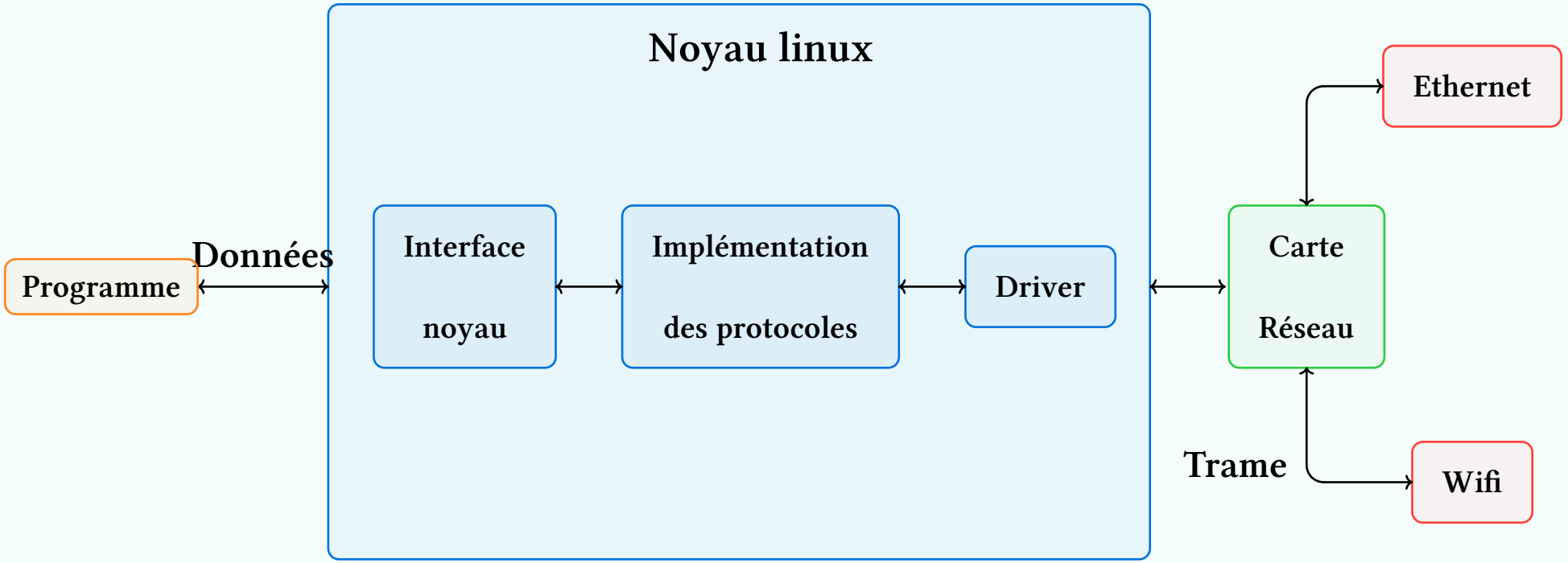
- Pour des raisons de sécurité et de stabilité, un processus n'est pas authorisé à utiliser directement les périphériques connectés (clavier, carte réseau, clé usb, carte son…).

- Le **noyau** du système d'exploitation se charge alors de la communication avec les périphériques et les processus communiquent avec le noyau via des **appels systèmes** (c'est ce que fait printf par exemple).

- La plupart du temps, le noyau utilise un **driver** spécifique au périphérique concerné pour communiquer avec lui.

- Le noyau linux contient aussi une implémentation de certains protocoles.

Communication réseau - schéma simplifié



Utilisation persistante d'un port/addresse

- Souvent un programme souhaitera effectuer plusieurs échanges sur un même couple *port/addresse*, par exemple:
  - Pour envoyer des requêtes et recevoir des données.
  - Pour envoyer plusieurs requêtes.
  - Pour effectuer un handshake ou une étape d'identification préalable à l'envoi de données.

- Afin de réserver le couple *port/addresse* pour un programme et éviter de le reconfigurer à chaque requête, le noyau nous propose d'utiliser une **socket** ("prise" en anglais).

- Une socket est une structure associée à un processus qui contient les informations nécéssaires à l'envoi de données.

Identification des sockets

- En s'inspirant de la philosophie *unix*, le noyau linux tente quand possible de traiter une socket comme un fichier.
- On peut donc faire le parallèle entre envoyer des données sur le réseau et écrire dans un fichier (même chose pour recevoir des données et lire le contenu d'un fichier).
- Chaque socket créée par un processus est donc identifiée par un descripteur de fichier, il s'agit d'un entier unique au procéssus.
- À l'intérieur du noyau, les sockets sont identifiées par un inode, le noyau maintient une correspondance entre les inodes et les descripteurs de fichiers de chaque processus.

Exemple

Le code ci-dessous est simplifié, seuls les arguments pertinents sont indiqués.

```
// la fonction socket demande au noyau de créer une socket et nous renvoie son numéro
int numero_socket_a = socket(...);
int numero_socket_b = socket(...);
// On peut ensuite utiliser ce numéro pour connecter la socket à un couple port/
addresse
bind(numero_socket_a, ..., "127.0.0.1", 8080, ...);
bind(numero_socket_b, ..., "172.21.200.15", 5512, ...);
// On peut ensuite envoyer ou lire des données
send(numero_socket_b, ...) // on envoie des données sur la socket b
read(numero_socket_a, ...) // on attend de recevoir des données sur la socket a
// NOTE: Les fonctions socket, bind, send et read effectuent toutes un appel système
```

Sockets TCP

TCP - Rappels

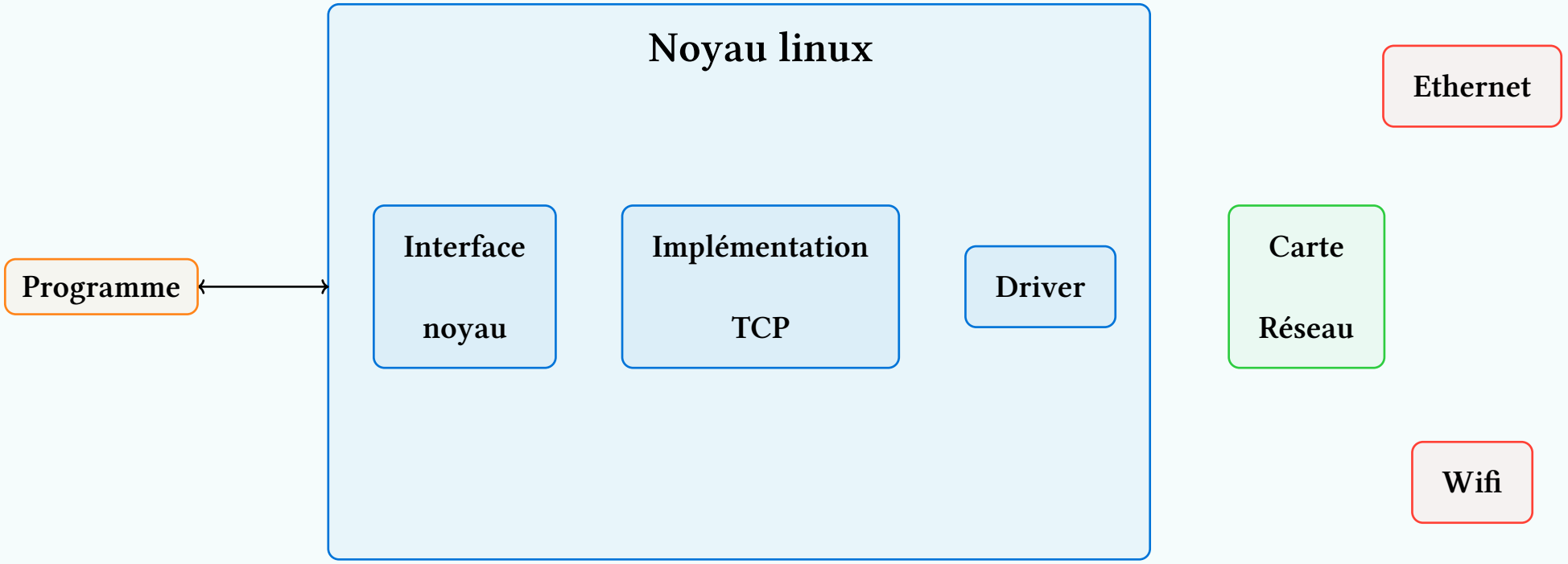
- Le protocole TCP (pour transmission control protocol) est un protocole en mode **Connecté**.
- Cela permet, entre autres, de s'assurer que tous les paquets échangés arrivent dans l'ordre.
- Ainsi, avant de pouvoir échanger des données, un client et un serveur doivent effectuer un **Handshake**:
  - Le serveur doit être en attente de connections.
  - Un client envoie un paquet **SYN**.
  - Le serveur répond avec un paquet **SYN ACK**.
  - Le client envoie un paquet **ACK**.
  - Le client comme le serveur peuvent tous les deux commencer à s'échanger des données.
- Les prochains paquets qui seront échangés contiendront l'identifiant de la connection, permettant ainsi au serveur de les diriger vers la bonne socket.
- Une suite de paquet similaire est aussi envoyée lors de la déconnexion.

TCP sous linux

- Le noyau linux contient une implémentation du protocole TCP.
- Si on souhaite pouvoir échanger avec plusieurs procéssus qui initieront eux même la connection, il faut créer une socket **d'écoute** qui attendra des paquets **SYN**.
  - Le système effectuera alors l'étape de **handshake** et créera une socket pour la connection souvellement créée.
  - Pour échanger des données sur cette connection, nous utiliserons l'identifiant de la nouvelle socket.
- Si on souhaite simplement envoyer des données à un serveur, on ne crée qu'une socket en précisant le port et l'addresse et on demande au système d'initier le handshake.

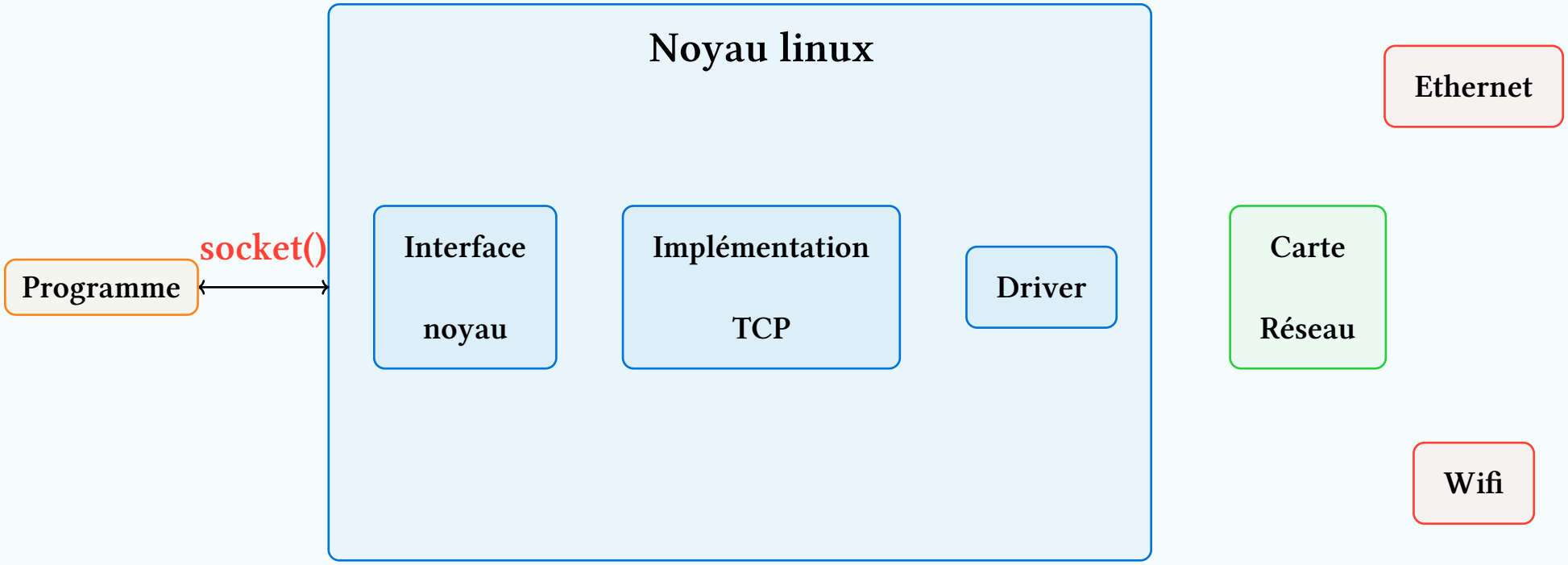
TCP sous linux - Schéma simplifié

On considère ici un serveur qui souhaite accepter des connections de plusieurs sources possibles.



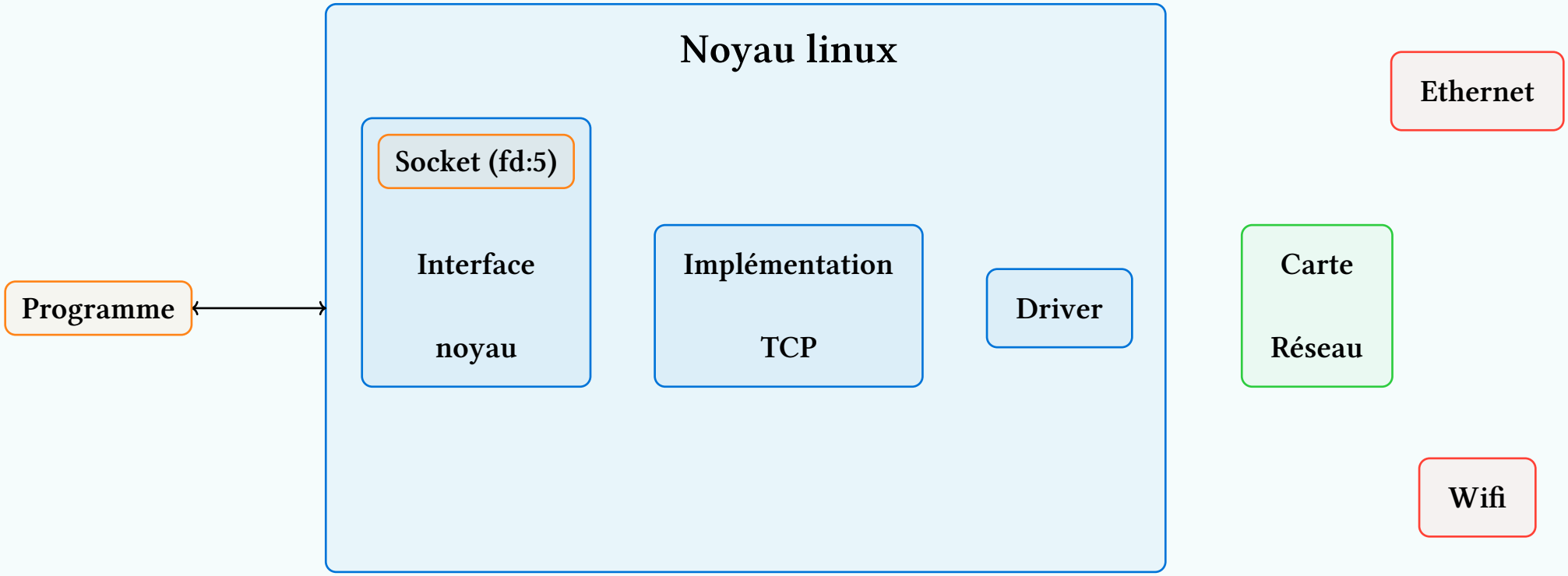
TCP sous linux - Schéma simplifié

Le programme va alors créer une socket et la configurer pour utiliser TCP.



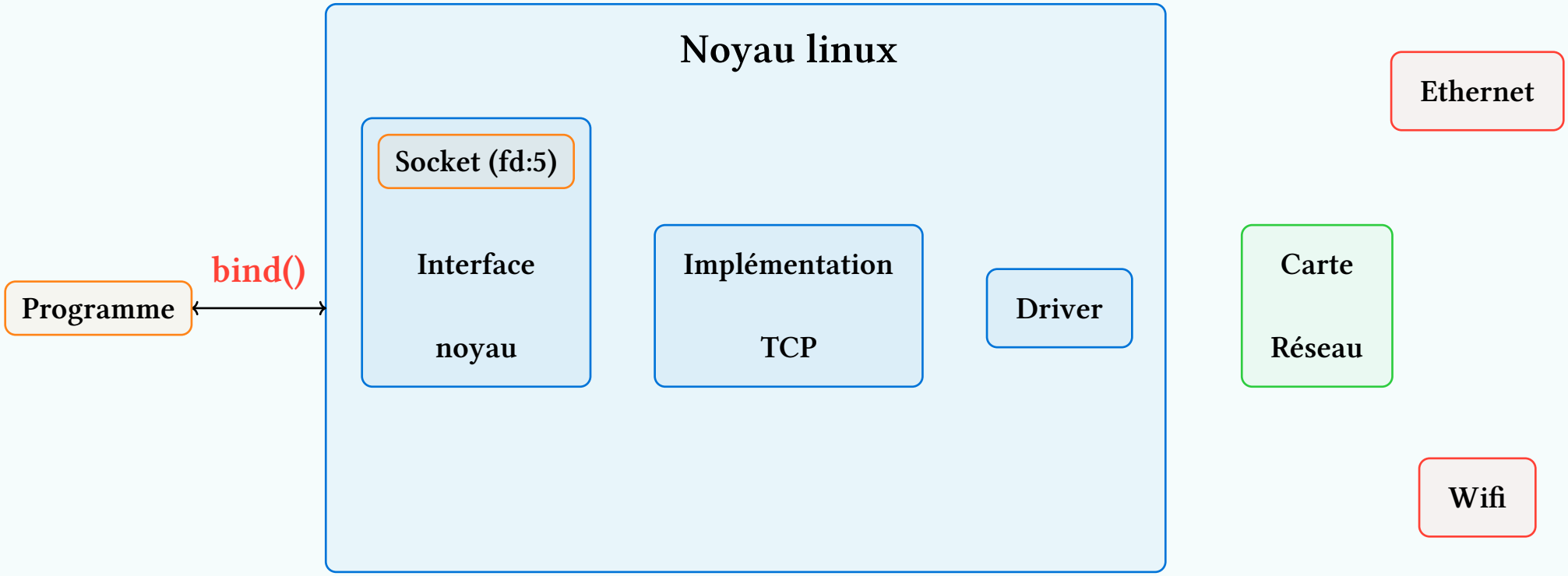
TCP sous linux - Schéma simplifié

Le noyau crée la socket et lui associe un identifiant (ici 5) qui permettra au processus de l'utiliser.



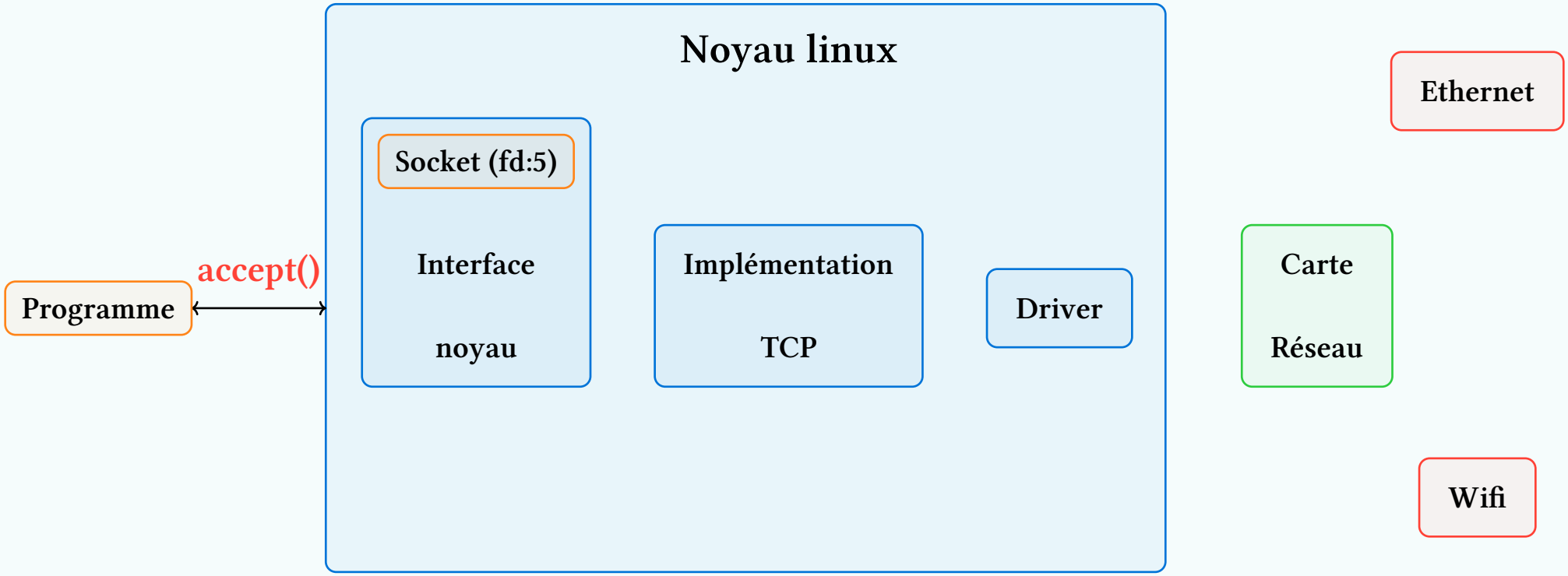
TCP sous linux - Schéma simplifié

Le programme attache ensuite la socket à un port et précise qu'elle devra écoutera toutes les addresses.



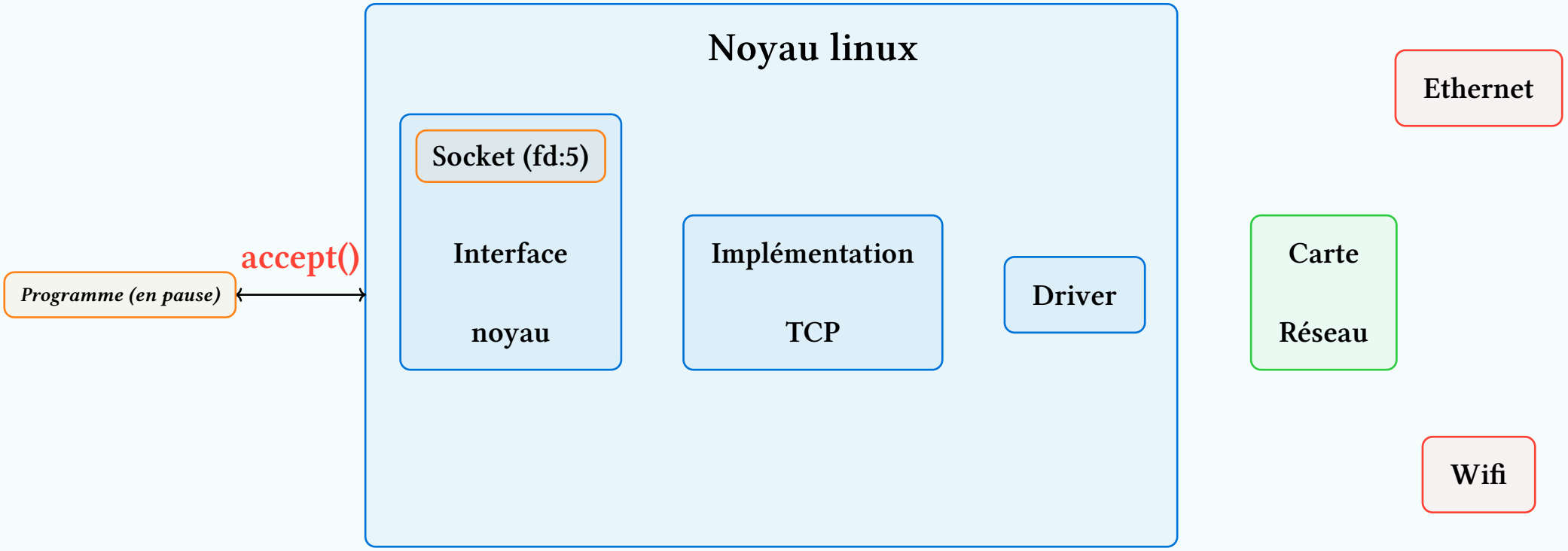
TCP sous linux - Schéma simplifié

Le programme demande ensuite au système de le mettre en pause en attendant une connection.



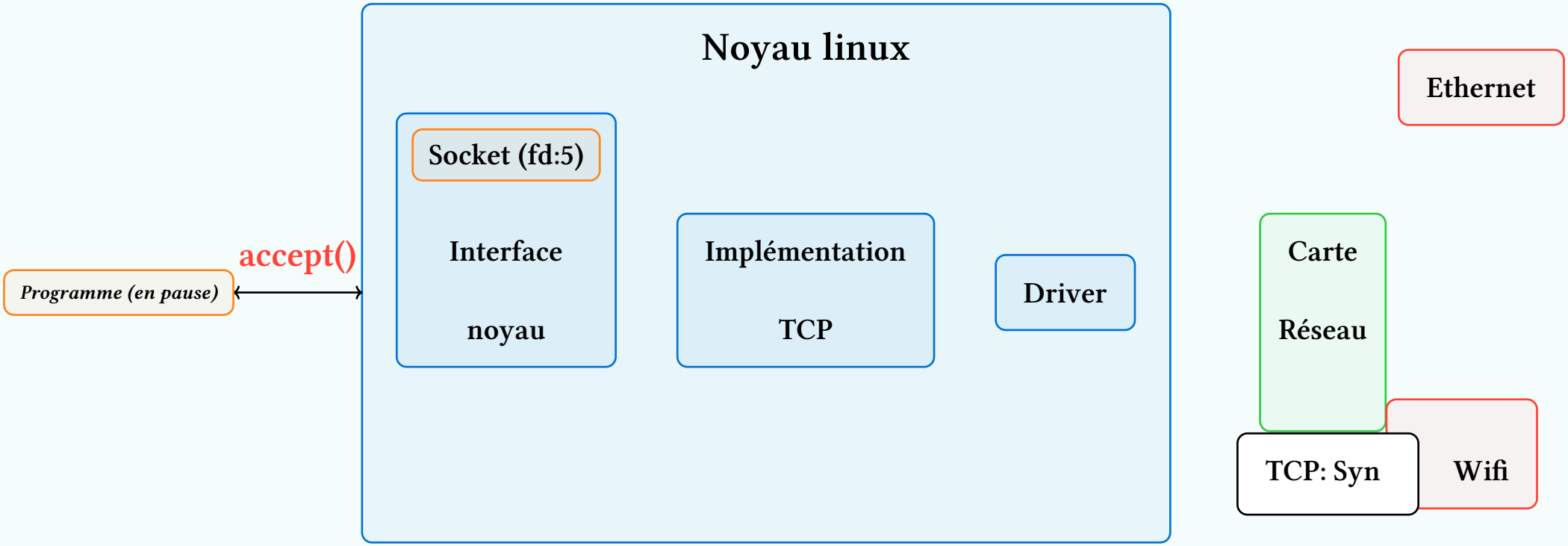
TCP sous linux - Schéma simplifié

Le noyau met le programme en pause et se configure pour réagir à l'arrivée de paquets **SYN** sur le port défini.



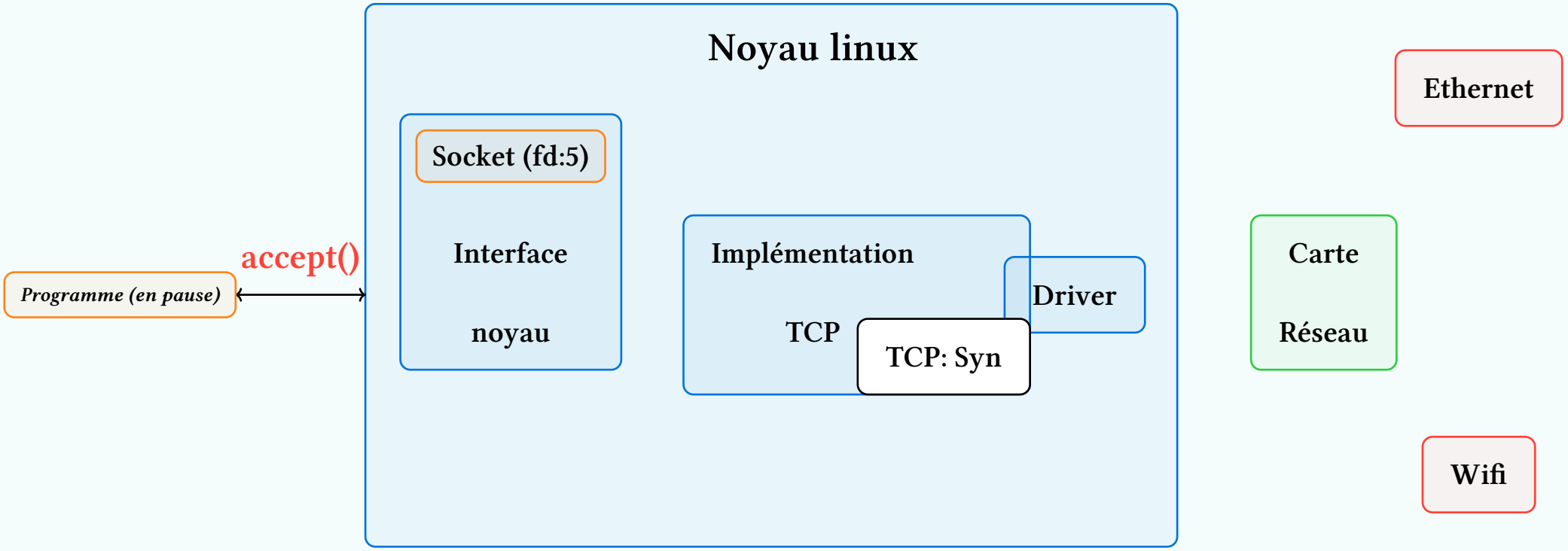
TCP sous linux - Schéma simplifié

Un paquet **SYN** arrive sur le réseau.



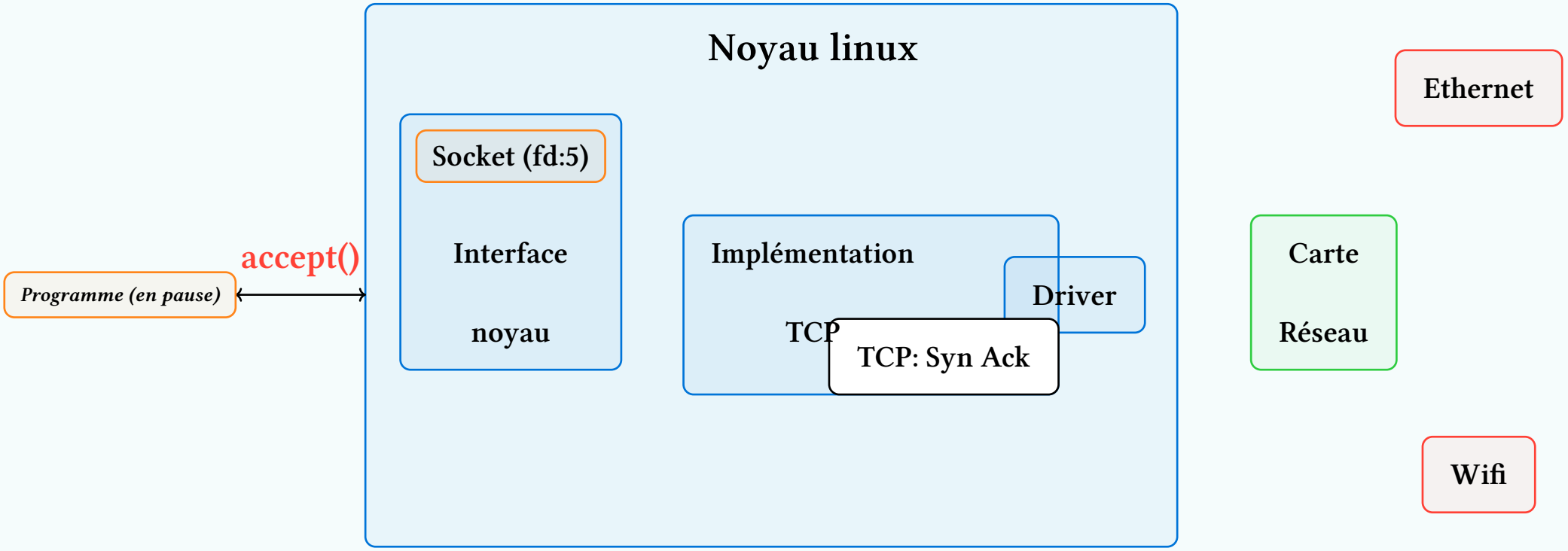
TCP sous linux - Schéma simplifié

Le paquet est transmis au noyau via la carte réseau et le driver.



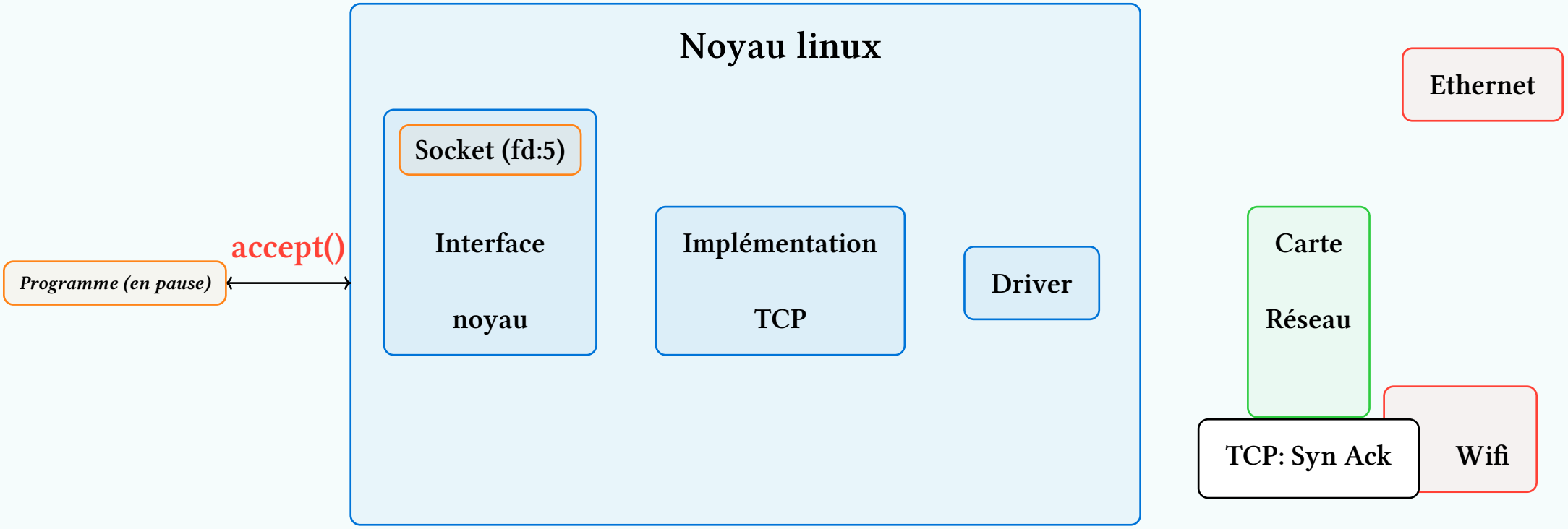
TCP sous linux - Schéma simplifié

Le noyau traite le paquet et prépare un paquet **SYN ACK**.



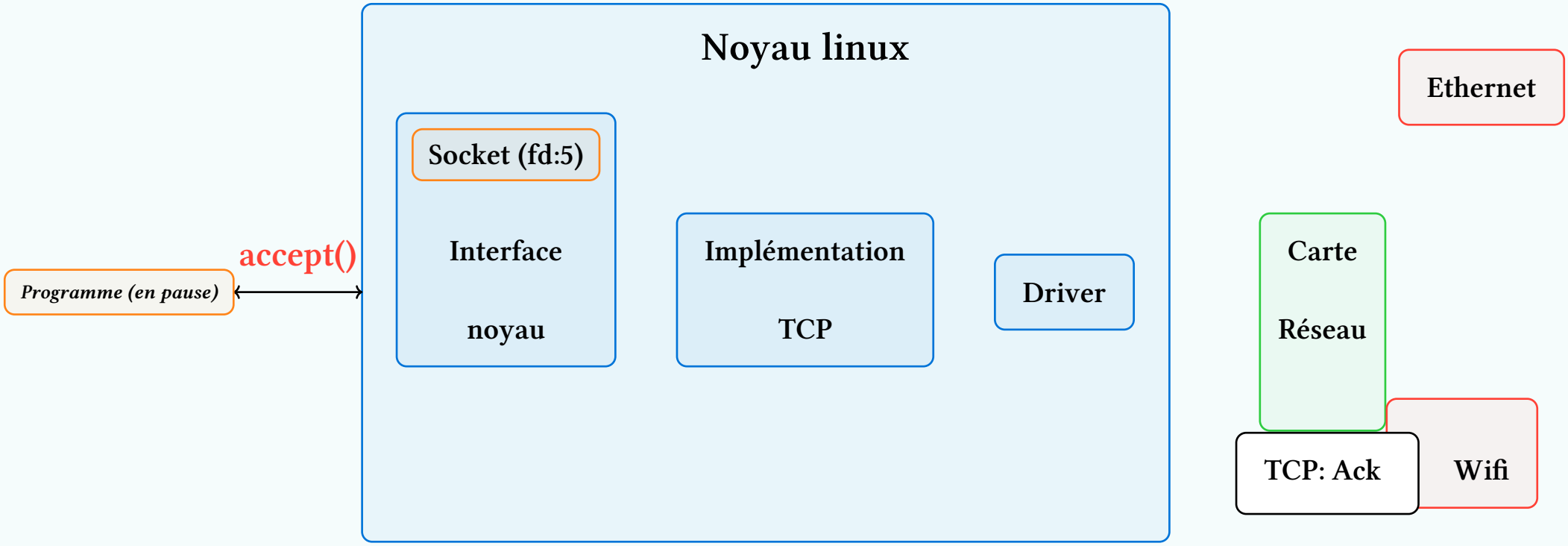
TCP sous linux - Schéma simplifié

Le noyau envoie, via la carte réseau, le paquet **SYN ACK** au client.



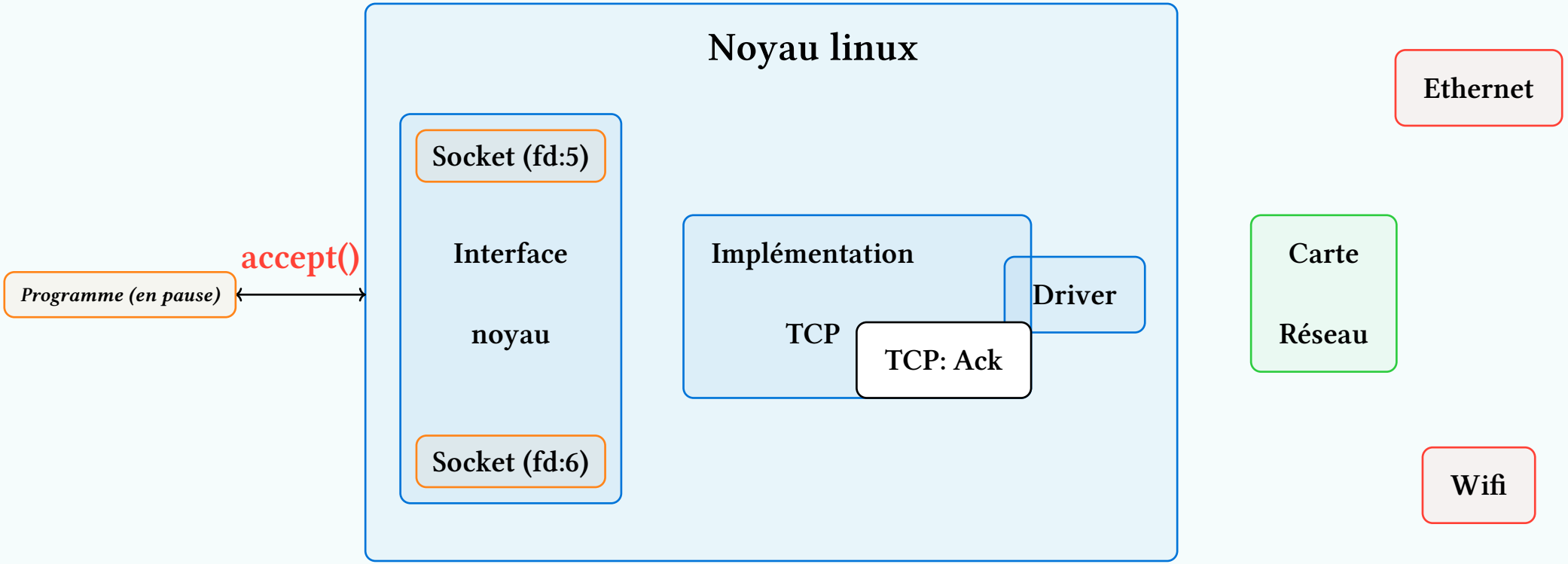
TCP sous linux - Schéma simplifié

Le client renvoie un paquet **ACK**.



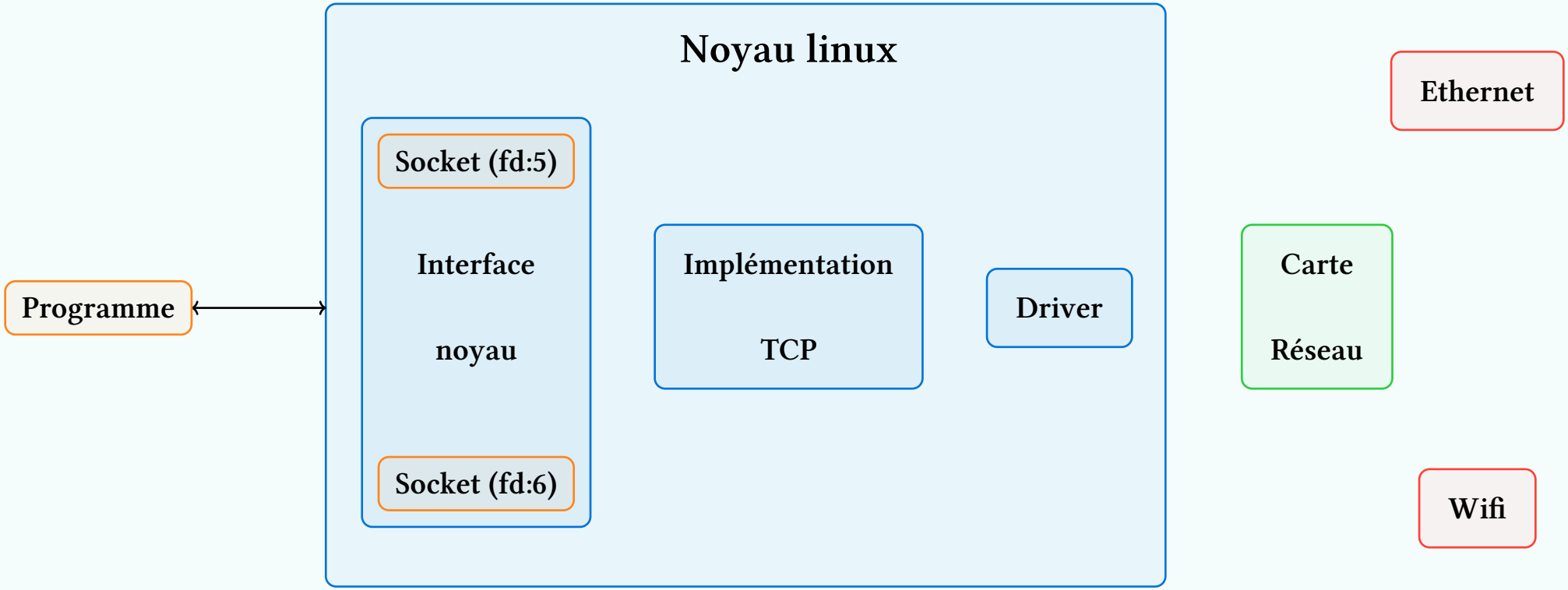
TCP sous linux - Schéma simplifié

À la récéption du **ACK**, le handshake est réussi et le noyau crée une socket spécifique pour cette connexion.



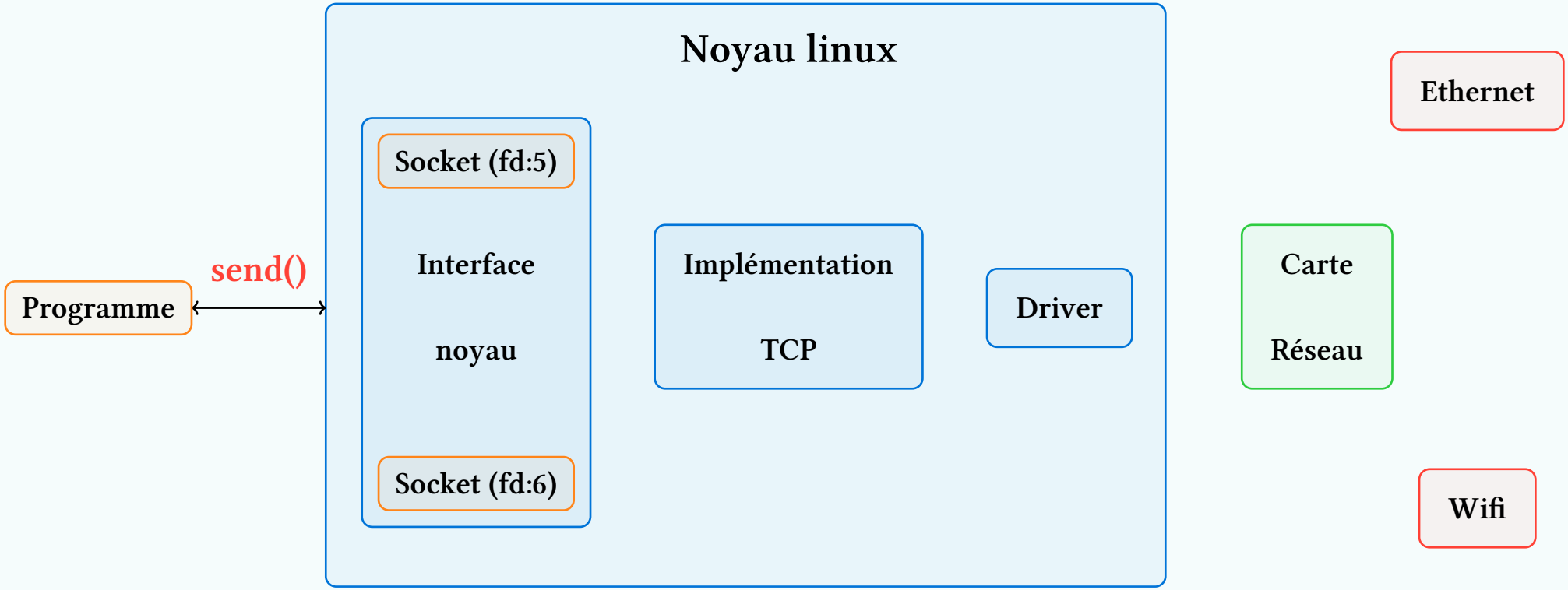
TCP sous linux - Schéma simplifié

Le noyau réveille alors le processus qui était en pause et lui indique le numéro de la socket créée (ici: 6).



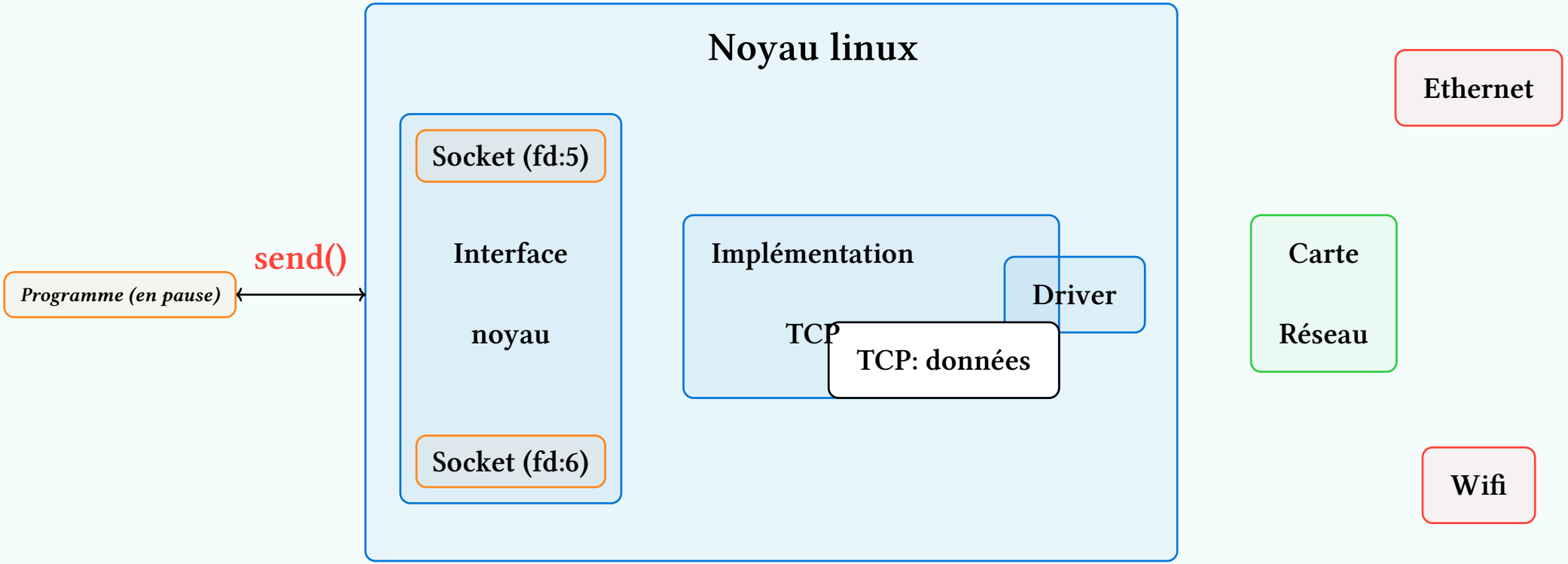
TCP sous linux - Schéma simplifié

Le programme peut ainsi envoyer des données au client via la connection TCP avec le numéro de socket **6**.



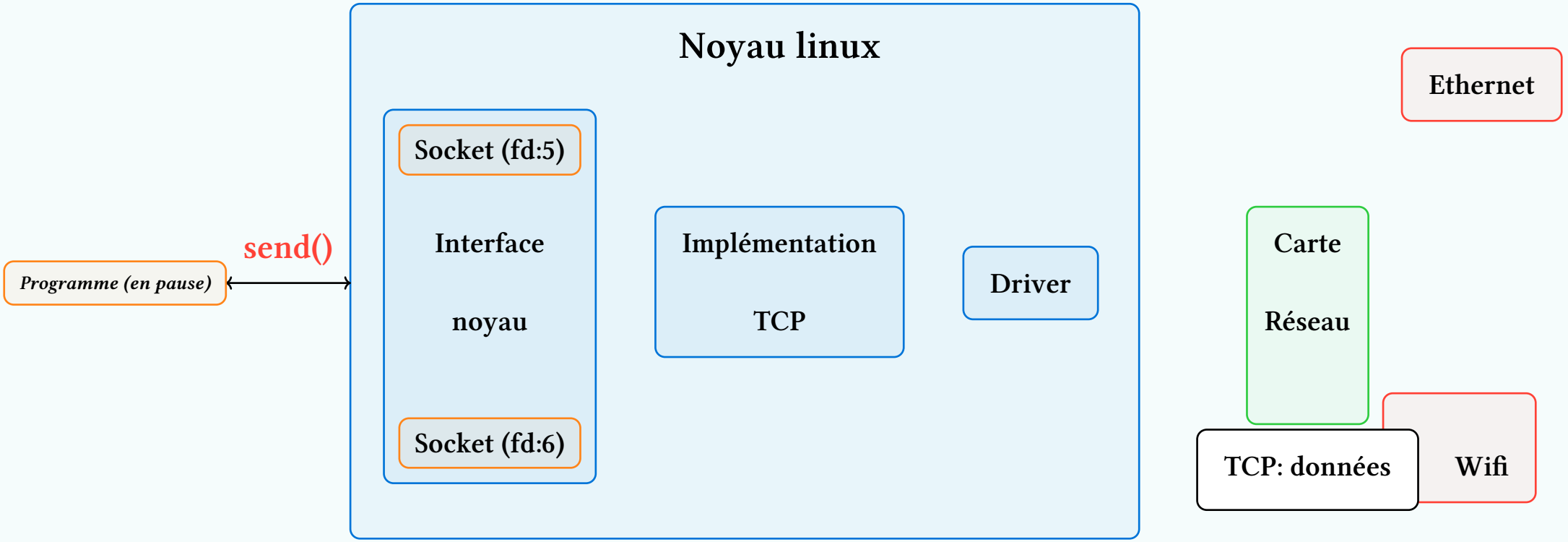
TCP sous linux - Schéma simplifié

Le noyau créera alors un paquet TCP et l'enverra automatiquement au client.



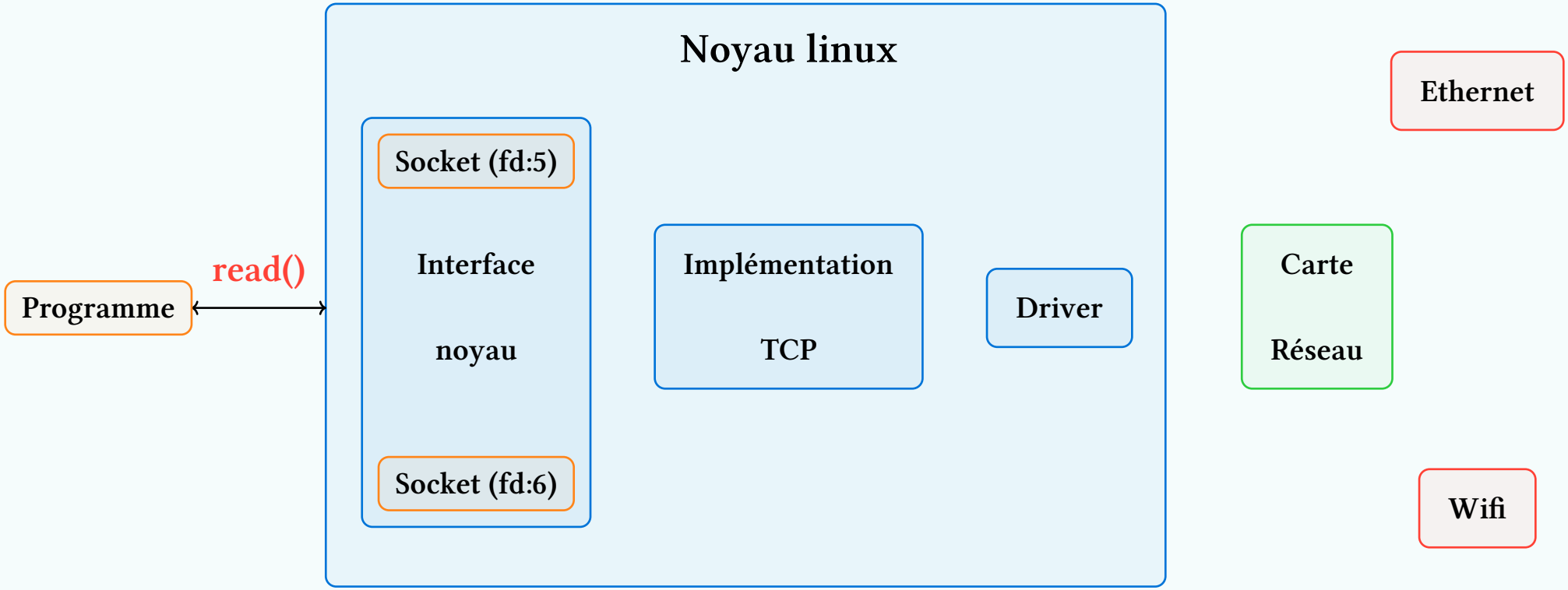
TCP sous linux - Schéma simplifié

Le noyau créera alors un paquet TCP et l'enverra automatiquement au client.



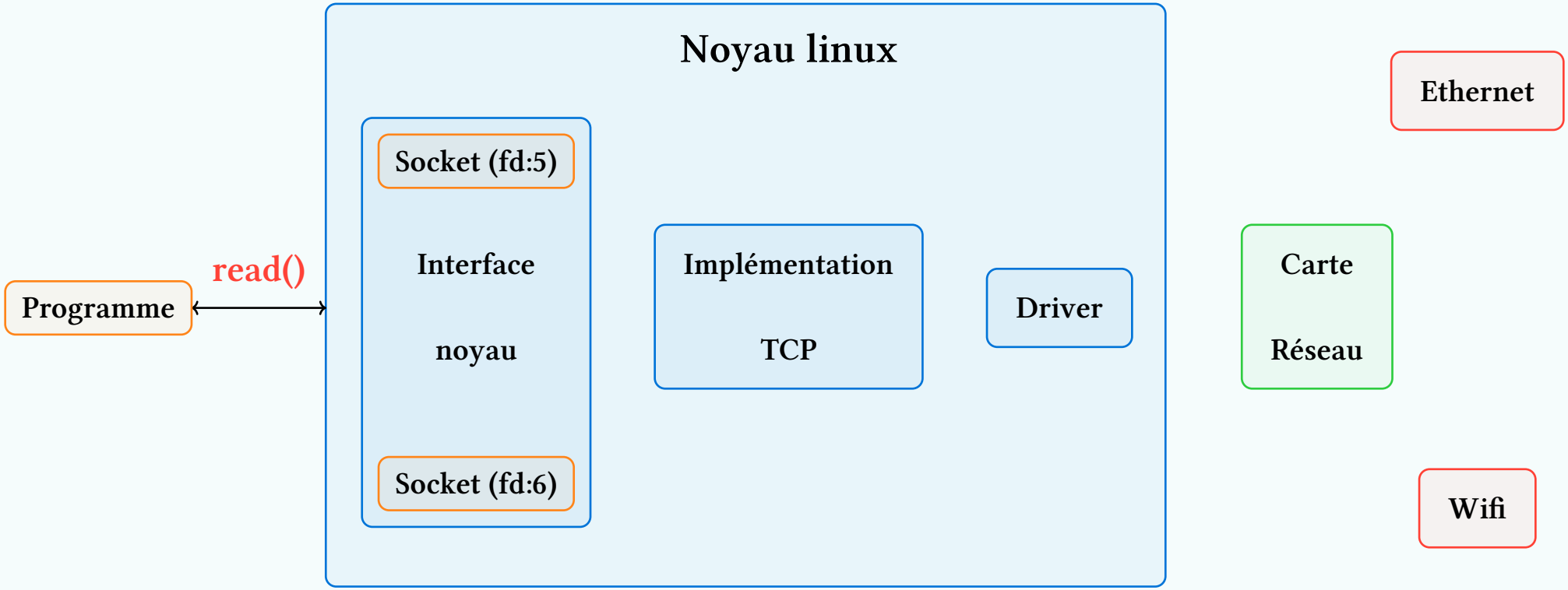
TCP sous linux - Schéma simplifié

Le programme peut aussi recevoir des données du client via la connection TCP avec le numéro de socket **6**.



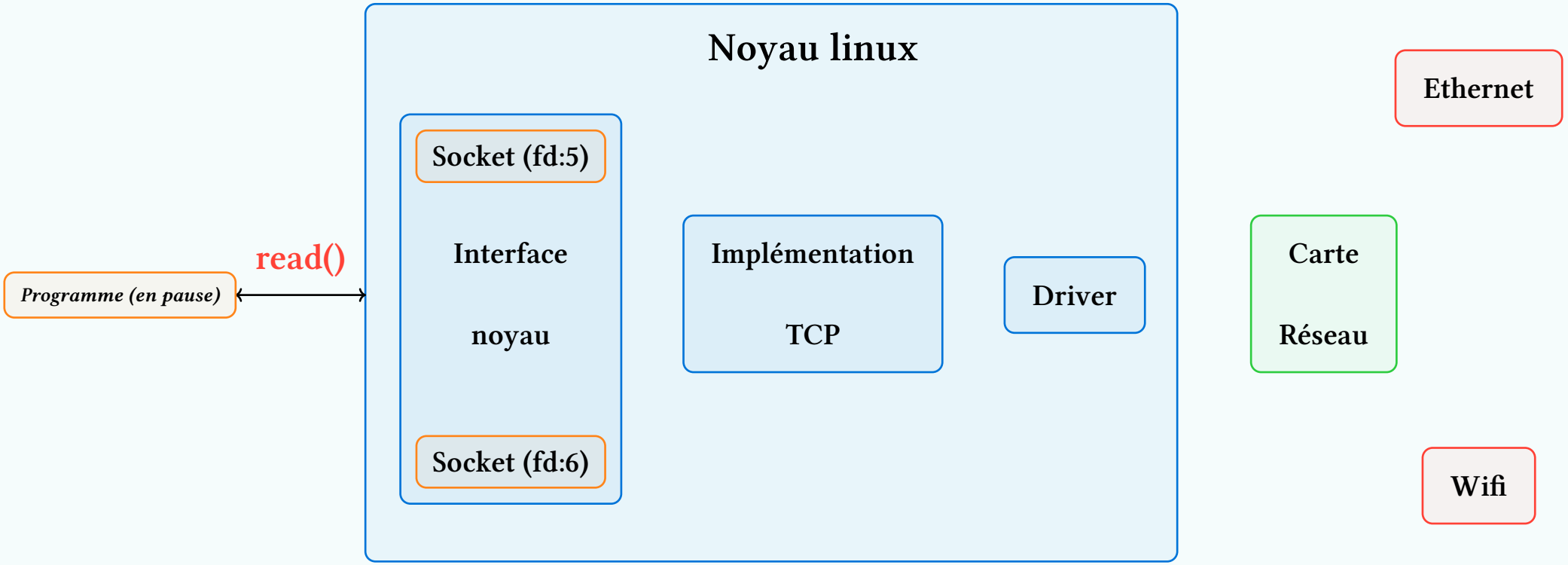
TCP sous linux - Schéma simplifié

Si le noyau a recu des paquets du client, il renverra les données au programme.



TCP sous linux - Schéma simplifié

Sinon il mettra le programme en pause en attendant de recevoir des paquets.



Autres types de socket

Choix du protocole

La fonction `socket()` accepte trois argument qui permettent de configurer le protocole utilisé.

- La famille de protocoles à utiliser (pour ipv4 par exemple, on passe **PF_INET** (*protocol family internet*)).

- Le type de socket:
  - si on souhaite échanger des données de manière fiable en utilisant une connection: **SOCK_STREAM**.
  - si on souhaite échanger des paquets de données sans vérification: **SOCK_DGRAM**.
  - etc…

- Le protocole à utiliser ou la valeur `0` si on souhaite laisser le noyau décider automatiquement du protocole.
  - Si on a renseigné **PF_INET** et **SOCK_STREAM**, le protocole sera TCP.
  - Si on a renseigné **PF_INET** et **SOCK_DGRAM**, le protocole sera UDP.

Choix du protocole - Exemple

```
int socket_tcp = socket(PF_INET, SOCK_STREAM, 0);
int socket_udp = socket(PF_INET, SOCK_DGRAM, 0);
int socket_ddp = socket(PF_APPLETALK, SOCK_DGRAM, 0);
int socket_bluetooth = socket(PF_BTH, SOCK_STREAM, 0);
// ici on ne définit pas de protocole, on écrit directement les
octets à envoyer sur le réseau
int socket_no_protocol = socket(PF_PACKET, SOCK_RAW, 0);
```

Sockets brutes

- Si on ne veut pas utiliser l'implémentation d'un protocole dans le noyau (parce qu'elle n'existe pas par exemple), on peut préciser les options **PF_PACKET** et **SOCK_RAW**.
- On précise ensuite en troisième argument le type de données que l'on souhaite échanger.

```
// on passe ETH_P_ALL pour tous les protocoles possibles
int raw_socket = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
// ETH_P_ARP pour les paquets ARP
int arp_socket = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ARP));
// ETH_P_IP pour les paquets IP
int ip_socket = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP))
// etc... les constantes disponibles sont consultables dans if_ether.h
```

Échange de données sur une socket brute

- Lorsqu'on échange des données sur une socket brute, il faut envoyer les octets demandés par le protocole.
- Pour cela on prépare les octets dans la mémoire et on passe l'addresse du premier élément à la fonction `send` (ou `sendto`).

```
// on imagine ici un protocole qui demanderait d'envoyer la requête suivante:
// ---- HEADER ----
// 2 octets qui contient la valeur 1290 (5 * 256 + 10)
// 8 bits suivants contiennent le code 60
// 6 bits suivants contiennent la valeur 5
// 2 bits suivants contiennent la valeur 4
// ----  DATA  ----
// 32 bits suivants: payload (ici "abc2" en ASCII²)
char requete[8] = {5, 10, 60, 22, 97, 98, 99, 50};
```

Échange de données brutes - Exemple avec une structure

```
typedef struct Req{
  // On peut utiliser une structure pour facilement remplir les données demandées par le protocole.
  uint16_t premiere_val;
  char seconde_val;
  char troisieme_val;
  char payload[4];
} Req;
// On crée la socket et une structure dest_addr qui contient l'addresse
Req request;
request.premiere_val = 1290;
request.seconde_val = 60;
request.troisieme_val = (5 << 2) | 4;
request.payload = {'a', 'b', 'c', '2'};
// on passe un pointeur vers la struct à la fonction sendto
sendto(socket_id, &request, sizeof(Req), 0, &dest_addr, sizeof(dest_addr));
```

Mise en pratique

```
printf("

          _____
        < TP 6 >
          -----
                \   ^__^
                 \  (oo)\_______
                    (__)\       )\/\
                        ||----w |
                        ||     ||
        ")
```