Département Informatique-Microélectronique-Automatique Polytech Lille

Rapport PFE

Partition HTTP/TLS Pour Pépin

Mageshwaran SEKAR

sous la tutelle de

Julien IGUCHI-CARTIGNY

IRCICA

Table des matières

Remerciements

Je souhaite tout particulièrement remercier mon tuteur, M. Julien IGUCHI-CARTIGNY, pour son accueil au sein de l'équipe 2XS, sa confiance et ses conseils pour la préparation de ce rapport et la soutenance.

Je tiens également à remercier les membres de l'équipe (Mahieddine YAKER, Quentin BERGOUGNOUX, Narjes JOMAA) pour leur aide et leur participation. J'ai été amené à travailler avec mon collègue, Mahieddine, qui m'a beaucoup aidé pour l'intégration de la solution dans la carte embarquée. Grâce à son intervention, j'ai pu débloquer plusieurs problèmes rencontrés lors du développement.

Je voudrais aussi remercier Frederik VAN SLYCKEN (développeur picotcp au sein d'Altran Intelligent Systems, Belgique) et Maxime VINCENT qui ont répondu à mes questions concernant picotcp (qui fournit la pile TCP/IP) sur GitHub.

Présentation du projet

Pendant ces 6 mois de projet de fin d'étude, j'ai travaillé dans l'équipe de ***2XS*** à l'***IRCICA***. Cette équipe, ***eXtra Small eXtra Safe*** intègre des projets de sécurité, des systèmes embarquées, etc. Le développement de Pépin est l'un de ces projets de l'équipe.

Actuellement, 2XS développe un proto-noyau, Pépin (ou *Pip* en anglais), qui a pour but de s'assurer de l'isolation complète entre les différentes partitions en tout point de l'exécution sur le système. Pépin est fait de manière à ce que l'on puisse le porter dans n'importe quel système. La partie prouvable du système est codée en Coq converti en code C freestanding (indépendant de l'architecture) lors de la compilation par l'intermédiaire de l'outil Coq2C. Cette partie de code est considérée comme la cœur de la sécurité de Pépin.

Au-dessus de Pépin se situe l'espace utilisateur (*user space*) où on peut implémenter les différentes applications. Dans le cadre de ce projet, je suis amené à développer un serveur web sécurisé au-dessus de Pépin. C'est à dire que le serveur doit pouvoir être porté dans un environnement baremetal où il n'y a pas de bibliothèques spécifiques fournies (comme *libc* dans Linux) pour la gestion de mémoire des applications. Il y a aussi d'autres contraintes de développement dans ce type d'environnement qui sera discuté plus tard.

1 Pépin et les différents types de noyaux

Le noyau est la partie essentielle d'un système car il fournit un mécanisme de communication entre l'espace utilisateur (couche haute) et le matériel (hardware). Il gère aussi le multiplexage de ces parties matériaux. Autrement dit, il se comporte comme un arbitre lorsque les différents outils de systèmes veulent utiliser la même ressource (ex : carte son) simultanément.

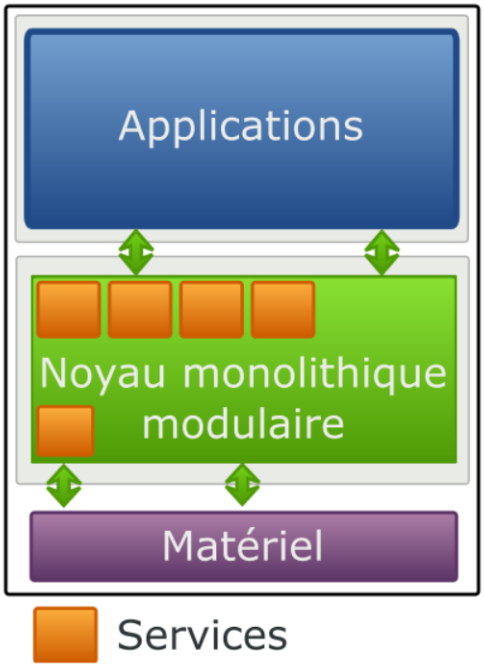


FIGURE 1 – Schéma de noyau monolithique modulaire [9]

Il existe plusieurs types de noyaux (ex : monolithique, micronoyau, exonoyau, etc.) qui ont été développés pour différents buts. Je vais détailler ces différents noyaux dans la section suivante.

1.1 Les différents noyaux

Dans le noyau dit ***monolithique***, seules les parties fondamentales du système sont regroupées dans un bloc de code unique (monolithique). Les autres fonctions, comme les pilotes matériels, sont regroupées en différents modules

qui peuvent être séparés tant du point de vue du code que du point de vue binaire.

Les systèmes à micro-noyaux cherchent à minimiser les fonctionnalités dépendantes du noyau en plaçant la plus grande partie des services du système d'exploitation à l'extérieur de ce noyau, c'est-à-dire dans l'espace utilisateur. Ces fonctionnalités sont alors fournies par de petits serveurs indépendants possédant souvent leur propre espace d'adressage.

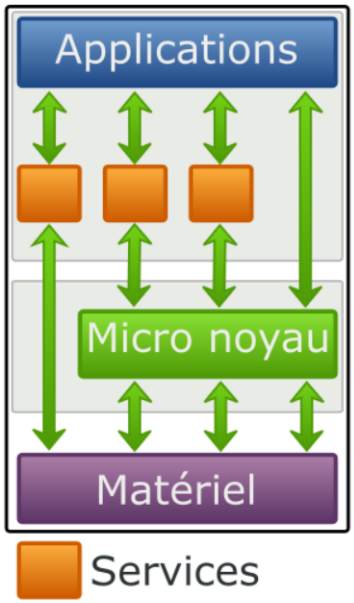


FIGURE 2 – Schéma de micronoyau [9]

Un ***exo-noyau*** est un type de noyau qui permet d'éliminer l'abstraction matérielle afin que l'utilisateur soit au plus près du matériel. De plus, le mécanisme de multiplexage des ressources permet à l'exo-noyau d'être souple et performant.

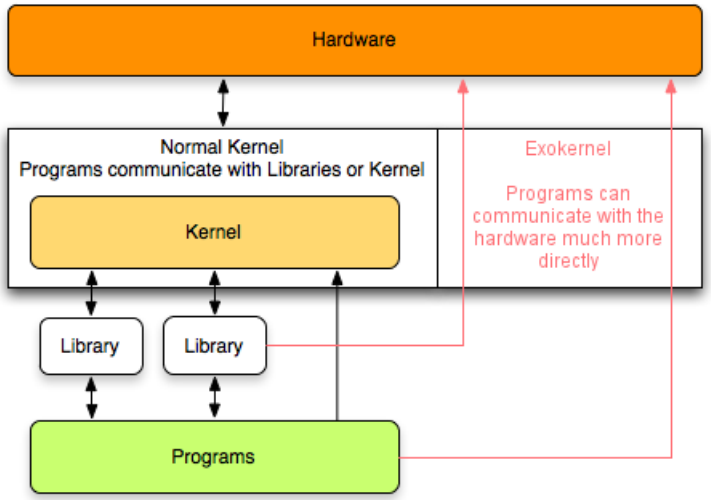


FIGURE 3 – Comparaison du noyau monolithique et de l'exo-noyau [5]

1.2 Architecture de Pépin

Pépin est tout à fait différent des autres noyaux car il a une architecture spécifique comme montre le schéma ci-dessous :

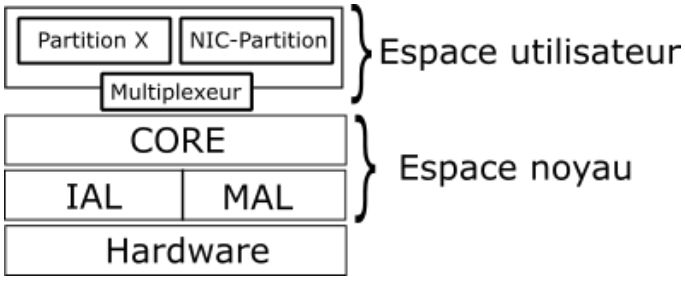


FIGURE 4 – Architecture de Pépin

Le noyau de Pépin se constitue de 4 parties comme :

- **IAL** : gestion de l'interruption (configurer, activer, désactiver, etc.)
- **MAL** : gestion de communication avec l'unité de gestion mémoire (*Memory Management Unit*, MMU)
- **BOOT** : initialiser les matériaux et démarrer Pépin
- **CORE** : contient le code principal de Pépin

Au-dessus du noyau (dans l'espace utilisateur), se situe le multiplexeur et les différentes partitions que l'on peut implémenter. Le multiplexeur est la couche de base de manipulation des partitions en mode utilisateur. Il fournit tous les composants systèmes non présents dans le noyau comme l'ordonnanceur.

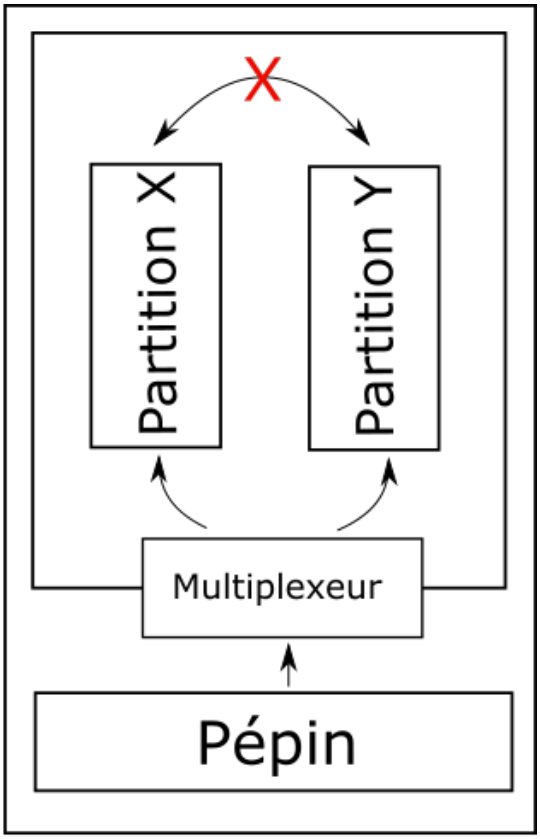


FIGURE 5 – Interaction des différentes parties dans Pépin

Le but du développement de Pépin est d'avoir un système de petite taille et sécurisé. Pour garantir le maximum de sécurité, les différentes partitions qui se trouvent dans l'espace utilisateur ont une hiérarchie de manière à ce que la communication entre elles soit limitée ou totalement interdite. De ce fait, une application qui tourne dans une partition ne peut pas modifier le comportement(ex : en modifiant les données dans l'espace mémoire) d'une autre application qui se trouve dans la différente partition.

2 Etude des serveurs web sécurisé

L'objectif principal de ce projet est de porter un serveur web dans Pépin. Pour cela, il a fallu identifier les différents types de serveur web qui sont déjà existants afin de simplifier la tâche. Il est inutile de développer soi-même un serveur web si on arrive à trouver un serveur web adapté.

Avant de pouvoir commencer, il y a des critères à considérer comme la légèreté, la sécurité par SSL/TLS surtout le support de mécanisme *TLS-PSK* et la dépendance minimale d'un système (Linux, Microsoft, etc.) pour faciliter le portage vers un autre système.

Tout d'abord, il est suffisant de traiter les requêtes HTTP (GET, POST, PUT, etc). On ne cherche pas un serveur complet avec traitement page dynamique (comme PHP). Il suffit de traiter les requêtes simples.

Dans cette section, je vais détailler les différents serveur web qui ont été considérés, le choix final et la raison de cette sélection.

2.1 Les différents serveurs web existants

Actuellement, il existe de nombreuses implémentation de serveur web qui sont sous la catégorie des logiciels libres. Pour en citer quelques-uns, on retrouve Apache, LigHTTPD, NGinx, etc. Je vais faire une comparaison de ces différents serveurs pour trouver celui qui est plus adapté.

lighttpd est un serveur HTTP léger et rapide du fait qu'il ait une plus petite empreinte mémoire que d'autres serveurs HTTP ainsi qu'une gestion intelligente de la charge CPU. Même s'il est vu comme plus léger que les autres serveurs, il n'est pas non plus adapté à notre système car il a besoin d'un système d'exploitation classique ou embarqué.

Mongoose, quant à lui, est un serveur web dédié pour systèmes embarqués. Il est de très petite taille et semble adapté à notre système. Pour aider le développement, il y a un API en C. Pourtant, l'implémentation de SSL/TLS n'est pas aussi complète, en effet, il manque le mécanisme TLS-PSK que l'on souhaitait.

SMEWS(Smart & Mobile Embedded Web Server) développé par l'équipe 2XS est un serveur web pour l'internet des objets (IoT). Il a sa propre pile TCP/IP dédiée pour le serveur web. Puisqu'on a une pile dédiée uniquement

pour le serveur HTTP, on ne peut pas l'utiliser pour d'autre service. De plus, il n'implémente pas de SSL/TLS.

2.2 Le serveur web retenu

Puisque les différents serveurs web déjà existants ne sont pas tout à fait adaptés à notre solution, il fallait trouver une autre méthode d'implémentation qui contienne la pile complète de TCP/IP puisqu'elle n'était pas encore développée dans Pépin.

2.2.1 Trame Ethernet

La pile de TCP/IP mentionnée contient plusieurs couches dont chacune d'elles résout une problématique différente que l'autre.

Couches hautes	Application
	Transport
	Réseau
Couches basses	Liaison de données
	Physique

Table 1 – Pile TCP/IP

La première couche (**physique**), a pour but de transmettre l'ensemble de la trame en faisant le codage (Manchester, etc.) avant de l'envoyer sur la moyenne de transmission (câble RJ45, fibre, etc.). La couche **liaison de données** permet de communiquer avec l'équipement voisin dans un réseau local. Ces deux couches sont considérées comme les couches basses parce qu'elles se trouvent au plus près du matériel (carte réseau).

Dans la plupart des cas, ces couches sont gérées par le système. Dans le cas de Pépin, nous avons une partition dans l'espace utilisateur (NIC-Partition) qui fait l'abstraction du matériel. Les applications doivent pouvoir communiquer avec cette partition si elles souhaitent envoyer des paquets vers la carte réseau et ensuite vers le réseau. Puisque ces couches basses sont déjà prises en charge par Pépin, on s'intéresse plutôt aux couches hautes.

La couche **réseau** est la troisième couche dans cette pile TCP/IP. Il résout le problème d'acheminement de paquets entre une source vers sa destination au travers du même ou d'un autre réseau. La couche **transport** assure la communication bout-en-bout et en même temps fiabilise les échanges et vérifie

que les données (trames) arrivent dans une bonne ordre. La dernière couche (**application**), est utilisée pour communiquer avec les différentes applications d'un système pour fournir les services demandés aux clients (ex :*HTTP*, *DNS*, etc.).

Il faut qu'on soit capable de traiter toutes les couches hautes ...Pour cela, j'ai étudié les différentes solutions de pile TCP/IP pour le système embarqué. Je devrais choisir une solution qui ait le moins de dépendance de système afin de pouvoir utiliser dans n'importe quelle architecture.

2.2.2 Solution de pile TCP/IP

Pour avoir une pile TCP/IP dans Pépin, j'ai trouvé que **picoTCP** est la solution la mieux adaptée. picoTCP est capable de gérer la pile TCP/IP de couche liaison de données jusqu'à la couche application sachant que la couche physique est gérée par le matériel lui-même.

Il est de petite taille ce qui est un élément essentiel pour un système embarqué et aussi déjà testé dans plusieurs types de systèmes. Et, il a été développé en prenant en compte l'indépendance de l'architecture ce qui facilite le portage vers différents systèmes.

Cependant, picoTCP ne fournit pas de mécanisme de chiffrement de données. Pour cela, il faut trouver une solution qui s'adapte avec picoTCP. Avant d'en trouver une, je vais expliquer les différents types de chiffrement fournit par SSL/TLS.

2.3 Sécurisation d'échange de données par SSL/TLS

La sécurisation dite SSL/TLS (Secure Socket Layer / Transport Layer Security) se constitue au chiffrage des données avant la transmission de ces derniers sur le réseau local ou sur Internet dans le but de protéger les informations transmises. Le principe de SSL/TLS est catégorisé sur 2 principe :

- ***Chiffrement des informations*** : Rendre incompréhensible les données échangées entre tous les équipements au travers de réseau sauf l'émetteur et le destinataire. Autrement dit, seuls ces deux parties peuvent déchiffrer et décoder les informations envoyées.
- ***Authentification***(dans le cas asymétrique) : Permet de s'assurer que le client communique avec l'équipement demandé (et non pas avec un équipement indésirable)

Il existe 2 grands types de chiffrements : ***asymétrique*** et ***symétrique*** qui sont utilisés actuellement.

Les algorithmes de ***chiffrement asymétrique*** sont basés sur le partage entre les différents utilisateurs d'une clé publique ou l'utilisation d'un certificat signé qui est utilisé dans la plupart des cas. La figure ci- dessous explique l'utilisation d'un certificat pour chiffrer les données entre le serveur et le client HTTP.

FIGURE 6 – Schéma d'établissement de connexion sécurisée [2]

Parfois, dans les systèmes embarqués, la ressource de calcul est très limitée. De ce fait, on préfère utiliser un chiffrement symétrique qui a un mécanisme très différent du chiffrement asymétrique.

Le ***chiffrement symétrique*** consiste à utiliser une clé (ex : TLS-PSK) pour rendre un message incompréhensible aux tierces parties. Elle est dite symétrique car cette même clé permet à ceux qui en ont connaissance de déchiffrer le message et ainsi d'accéder à son contenu. Celle-ci valide la contrainte de confidentialité, tant que la clé reste secrète.

2.4 SSL/TLS avec wolfSSL

En cherchant une solution qui pourrait fournir ces différents types de chiffrement, j'ai choisi ***wolfssl***. C'est une bibliothèque de SSL/TLS orientée système embarqué grâce à sa taille et la portabilité au contraire de *openSSL*. wolfSSL utilise les algorithme de cryptographie fourni par wolfCrypt (ex : RSA, Diffie-Hellman, AES, etc.).

2.5 La solution complète

La table ci-dessous résume la solution que j'avais choisi pour résoudre le choix d'un serveur web et la couche de TCP/IP.

Application	Serveur web
Transport	wolfSSL
	picoTCP
Réseau	

TABLE 2 – Pile TCP/IP avec picoTCP et wolfSSL

Le serveur web implémenté au-dessous de wolfSSL est possible grâce aux fonctions fournies par picoTCP.

3 Tester l'intégration de picoTCP et wolfSSL

Avant de porter la solution dans Pépin, je l'ai d'abord testé dans Linux. Le premier test a été d'implémenter picoTCP tout seul sans wolfSSL pour vérifier son fonctionnement. Au préalable de ces tests, il fallait configurer l'interface TUN/TAP sur Linux

3.1 Interface TUN/TAP

Pour utiliser picoTCP sous Linux, il faut créer un tunnel TUN/TAP afin que le système d'exploitation (Linux) ne modifie pas la trame ethernet. C'est parce que l'interface de carte réseau va enlever l'en-tête Ethernet et va passer le payload (normalement les paquets IPs) au système. Cela explique pourquoi l'utilisation de TUN/TAP est utile. Un dispositif TUN/TAP peut être vu comme une interface réseau qui communique avec un programme utilisateur (dispositif logiciel) au lieu d'une vraie carte matérielle (TUN pour mimer un périphérique point à point, TAP pour mimer un périphérique Ethernet).

La création d'une interface TAP doit se faire par l'intermédiaire d'un programme de l'espace utilisateur. En fonction des usages, plusieurs outils permettent de manipuler ces interfaces. On peut citer *tunctl* qui fait partie du paquet *uml-utilities*.

```
sudo tunctl #add tap0 interface
sudo ifconfig tap 10.0.0.1 #assign IP to tap0
```

Afin que la configuration de TUN/TAP reste inchangée, j'ai ajouté les lignes suivantes dans le fichier */etc/network/interfaces* :

```
  auto tap0
  iface tap0 inet static
    pre-up ip tuntap add tap0 mode tap user root
    address 10.0.0.1
    netmask 255.255.255.0
```

3.2 picoTCP seul

Pour tester le fonctionnement de picoTCP, on peut utiliser le ping (ICMP). La réponse du ping est considérée comme un bon (ou mauvais) fonctionne-

ment de picoTCP. Dans ce test, l'interface tap0 a l'IP de 10.0.0.1 et le côté picoTCP, on a 10.0.0.2 comme IP.

```
struct pico_ip4 ipaddr, netmask;
struct pico_device* dev;

dev = pico_tap_create("tap0");

/* attribute ip and netmask to a new interface in tap0 */
pico_string_to_ipv4("10.0.0.2", &ipaddr.addr);
pico_string_to_ipv4("255.255.255.0", &netmask.addr);

pico_ipv4_link_add(dev, ipaddr, netmask);

/* ping the ip address NUM_PING times */
pico_icmp4_ping("10.0.0.1", NUM_PING, 1000, 10000, 64,
ping_cb);
```

3.3 Glue picoTCP et wolfSSL

Maintenant, il fallait implémenter la glue qui permet de faire fonctionner wolfSSL au-dessus de picoTCP et ensuite d'ajouter des fonctions pour traiter les paquets HTTPS. Ensuite, on utilise une fonction callback qui permet d'initialiser le serveur web avec TLS-PSK. Il est aussi important de préciser la clé partagée (PSK) qui sera utilisée côté client pour communiquer avec le serveur.

On peut tester la connexion et l'échange de données avec le serveur en utilisant l'utilitaire fourni avec *openssl* sous Linux :

```
openssl s_client -connect <server_ip>:<port> -psk <clé_psk>
```

3.4 Serveur HTTP

La partie serveur HTTP implémentée au-dessus de wolfSSL va recevoir la requête envoyée par le client et la décoder afin d'envoyer la réponse à ce dernier. Pour cela, j'ai ajouté la fonction ci-dessous :

```
void serverWakeup(uint16_t ev, uint16_t conn){
  ...
  if(ev & EV_HTTPS_REQ){ /* new header received */
    char *resource = pico_https_getResource(conn);
    pico_https_respond(conn, HTTPS_RESOURCE_FOUND);
    pico_https_submitData(conn, data, dataSize);
    pico_https_close(conn);
  }
}
```

On peut, maintenant tester la solution complète (sous Linux), avec l'outil que l'on a utilisé avant, qui est *openssl*. Apres avoir lancé la commande (et après l'échange de clé), on lui passe la requête HTTP par la méthode GET :

```
GET /test HTTP/1.1
```

On trouve le résultat suivant :

```
HTTP/1.1 404 Not Found
Host: localhost
Connection: close
```

Le serveur a retourné une erreur 404 parce que la ressource demandée « /test » n'était pas disponible. Même s'il a retourné cette erreur, le serveur est complètement fonctionnel. Maintenant que la solution marche bien, je vais commencer à étudier la méthode de portage vers Pépin.

4 Adaptation de code de serveur web pour Pépin

Les bibliothèques de picoTCP et wolfSSL sont faites pour être portées dans n'importe quel système mais il faut d'abord réécrire certaines fonctions car elles sont dépendantes du système Linux en ce moment et que ces bibliothèques ne sont pas disponibles dans Pépin.

4.1 Allocation mémoire

Quelques-unes des fonctions principales à porter vers Pépin sont celles concernant l'**allocation dynamique** de mémoire.

La plupart des fonctions ayant des besoins en mémoire dépendent de l'usage que l'on en fait, il est nécessaire de pouvoir, à des moments arbitraires de l'exécution, demander au système l'allocation de nouvelles zones de mémoire, et de pouvoir restituer au système ces zones (libérer la mémoire). C'est pour cette raison que l'on a besoin des fonctions comme :

- **malloc** : qui alloue une espace de mémoire (dans le tas) avec une taille demandée
- **zalloc** : rassemble à malloc mais avec une étape supplémentaire d'initialiser l'espaces alloués avec zéro (équivalent de calloc sous Linux)
- **realloc** : sert à redimensionner (normalement agrandir) les espaces mémoire allouées
- **free** : libérer les espaces allouées dynamiquement

Pour réaliser ces fonctions, il est essentiel de réserver un bloc de mémoire. Puis, il faut une structure qui contient la taille de mémoire à allouer, le pointeur vers la prochaine bloc de métadonnée, une variable pour indiquer si l'espace est allouée (ou est libérée) et un pointeur vers le début de tableau data. Cette structure est décrit ci-dessous :

```
struct metadata {
        size_t size;
        struct s_block *next;
        int free;
        void *ptr;
        char data[1];
};
```

La schéma indique le bloc de mémoire et la structure métadonnée qui est utilisée pour l'allocation mémoire.

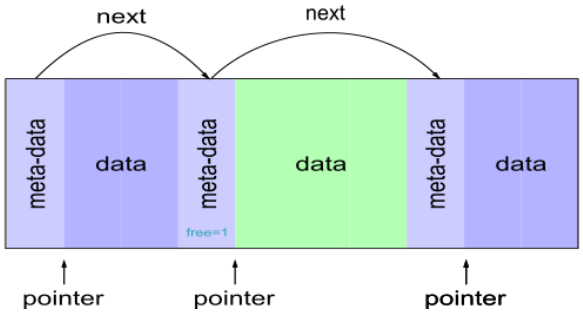


FIGURE 7 – Mécanisme de l'allocation mémoire

4.2 Délai

Nous avons aussi besoin d'une fonction qui est capable de calculer la date (ou le temps écoulé depuis le démarrage de système). Celle-ci est utile pour calculer le délai entre deux itérations consécutives dans une boucle. On peut imaginer une fonction qui rassemble ***sleep***.

Pépin, en fait, fournit une fonction, *pip_time* qui retourne le nombre de tick depuis le démarrage. J'utilise celle-ci pour remplacer la fonction sleep/usleep.

4.3 Bibliothèque math

Nous avons besoin de certaines fonctions de la bibliothèque mathématique, ***libmath*** comme ***pow*** (qui calcule la puissance : x^y) et *log* qui sont utilisées (par wolfSSL) dans l'algorithme de cryptographie de Diffie-Hellman.

Pour satisfaire cette condition, j'ai choisi ***openlibm*** qui est une bibliothèque mathématique en C qui est portable et indépendant de système. Il fournit les mêmes fonctions que dans ***libm***, notamment pow et log.

4.4 Générateur de nombre aléatoire

Par défaut, wolfSSL utilise la ressource */dev/random* ou */dev/urandom* du système (notamment Linux) pour générer les nombres aléatoires qui sont utilisés dans la création de clés (symétrique ou asymétrique). Puisque cette

ressource n'est pas disponible dans notre système, il faut réécrire une fonction qui soit capable de la remplacer. Vu que c'est un algorithme plutôt complexe, en ce moment, on génère des nombres aléatoires qui ont une faible entropie (qui pourrait être estimé par quelqu'un qui essaie d'attaquer le réseau) comme montre le code :

```
uint32_t rand(uint32_t feed){
  static uint32_t seed;

  if (!feed)
    return seed;

  seed *= 1664525;
  seed += 1013904223;
  seed ^= ~(feed);

  return seed;
}
```

Il faudrait réécrire, dans le futur, un générateur aléatoire plus performant que celui que nous avons en ce moment.

5 Implémentation dans Galileo

Dans cette partie, je vais détailler comment le serveur web (qui tourne dans la partition de Pépin) a été porté dans la carte Intel Galileo Gen. 2.

5.1 Spécification de Galileo

La carte Intel Galileo 2 est équipée d'un SoC Intel Quark X1000 de 32-bit cadencé à 400MHz. Au niveau de mémoire vice, elle est fournie avec 256Mo de RAM. Il a aussi un logement pour une carte SD jusqu'à 32Go. Les autres caractéristiques de cette carte sont : 10/100 Mbit Ethernet, USB 2.0, UART, port JTAG, etc. La vue du dessus de cette carte est représentée dans la figure ci-dessous :

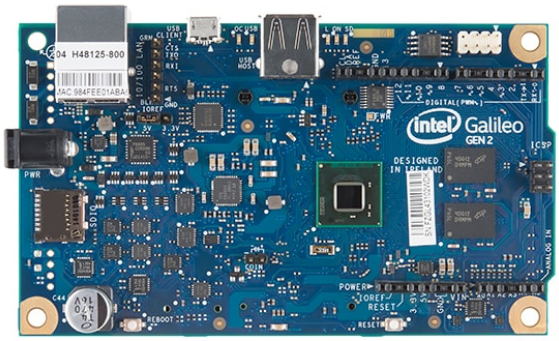


FIGURE 8 – Carte Intel Galileo Gen. 2 [7]

5.2 Interfaçage avec le driver de la carte réseau

Pour compléter et pouvoir utiliser Pépin sans soucis dans Galileo, je devais interfacer notre système avec le driver de carte réseau qui est en développement (par mon collègue, Mahieddine). De ce fait, j'ai travaillé en collaboration avec lui afin de familiariser avec la carte.

Les fonctions de bases nécessaires pour le bon fonctionnement du programme sont :

- ***eth_init*** : Initialiser toutes les variables/registres qui gèrent la réception et l'émission de paquets par le matériel
- ***getMacAddr*** : Récupérer l'adresse MAC de la carte réseau

— ***send*** : Envoyer les paquets générés par picoTCP vers le réseau
— ***poll*** : Récupérer les paquets qui arrivent sur la carte réseau

En réalisant les tests (envoie de paquets vers le réseau utilisant le driver), Mahieddine et moi, nous nous sommes rendu compte qu'il y avait des problèmes avec le driver de la carte réseau. De ce fait, je devais trouver une autre solution pour tester l'intégration du serveur dans Galileo. Suite à sa suggestion, je commence à étudier la méthode d'implémentation du protocole SLIP (Serial Line IP).

Ce protocole permet d'envoyer les paquets IP vers la liaison série. En connectant un PC avec cette liaison série, on pourrait récupérer ces paquets et envoyer vers la destination dans le réseau. Ce protocole modifie la trame en ajoutant ou modifiant certains octets comme décrit ci-dessous :

— Ajout d'un octet ***END*** pour indiquer la fin de la trame
— Si l'octet ***ESC*** est présent dans la trame, ajouter à la suite l'octet (***ESC_ESC***)
— Si l'octet ***END*** est présent dans la trame, remplacer celle-ci par une séquence de ***ESC*** et ***ESC_END***

Ces différents octets sont décrits dans le tableau :

Valeur hexadécimale	Octet	Description
0xC0	END	Fin de trame
0xDB	ESC	Caractère d'échappement
0xDC	ESC_END	Transposition du caractère END
0xDD	ESC_ESC	Transposition du caractère ESC

TABLE 3 – Descriptions des différents octet du protocole SLIP

Afin qu'un PC (sous Linux) puisse récupérer les trames du protocole SLIP sur la liaison série, on associe une interface réseau à une liaison série avec la commande *slattach* puis ajouter l'adresse IP pour cette interface.

```
#!/bin/sh

slattach -L -s 115200 -p slip /dev/ttyUSB0 -d &
sleep 1
ifconfig sl0 10.0.0.1 dstaddr 10.0.0.2 mtu 1500

echo 1 > /proc/sys/net/ipv4/ip_forward
echo 1 > /proc/sys/net/ipv4/conf/all/proxy_arp
```

```
socat tcp-l:443,reuseaddr,fork file:/dev/ttyUSB0,nonblock,\
raw,echo=0,waitlock=/var/run/tty &
```

Maintenant, le PC est capable de récupérer les trames envoyées vers la liaison série par Galileo et de les traiter.

5.3 Amorçage de Pépin dans Galileo

Avant de pouvoir faire des tests sur la liaison série, il faut d'abord amorcer Pépin dans Galileo. Pour cela, je compile Pépin ce qui génère un fichier binaire. Ensuite, on doit créer l'arborescence ci-dessous (dans une carte SD à mettre dans la carte Galileo) pour que l'outil d'amorçage GRUB reconnaisse bien le nom du fichier (dans ce cas, le binaire) à booter.

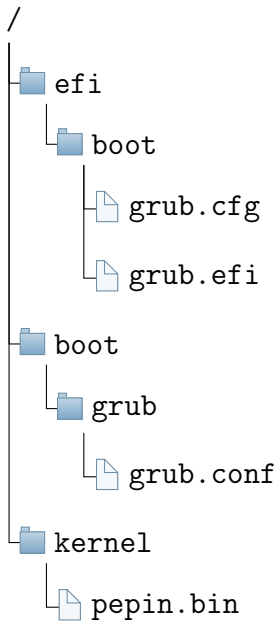


FIGURE 9 – L'arborescence de fichier dans la carte SD

Dans *grub.conf*, on lui indique le fichier à utiliser pour le chainloader. Dans notre cas, on a besoin d'utiliser le chainloader afin de pouvoir booter un système non-supporté avec un autre *bootloader*. Le fichier *grub.efi* sera utilisé par le chainloader.

```
default 0
timeout 5

title Multiboot GRUB
  root (hd0,0)
  chainloader /efi/boot/grub.efi
```

Il est aussi important de préciser le fichier binaire à utiliser pour booter le système. De ce fait, on modifie *grub.cfg* afin d'avoir les modifications montrées ci-dessous :

```
set default=0
set timeout=5

menuentry 'Pepin' {
  set root='hd0,msdos1'
  multiboot /kernel/pepin.bin ro
}
```

À ce stade, on devrait être capable de booter Pépin sur Galileo. Il est alors temps de tester le fonctionnement du serveur web porté sur ce système.

5.4 Débogage dans Galileo

Pour faire le débogage, on a besoin d'une fonction pour écrire et envoyer des caractères sur le port série. Nous avons déjà une fonction *printf* disponible dans le système qui est utilisé pour la plupart de débogage.

Un autre méthode de débogage est de récupérer l'adresse mémoire de l'instruction (EIP) qui cause l'erreur lors de l'exécution.

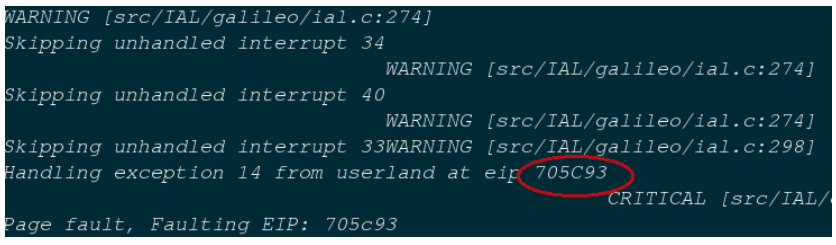


FIGURE 10 – Affichage de message d'erreur de Galileo

Après avoir récupéré cette adresse, je lance la commande ***objdump*** pour désassembler le binaire (ELF) qui a été généré lors de la compilation.

```
objdump -D web-server.elf
```

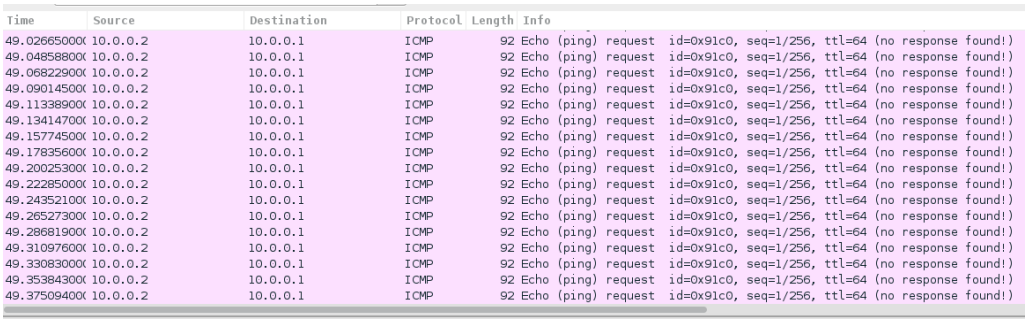
Cet outil affiche l'adresse et l'instruction correspondante. En regardant l'affichage erreur de Galileo, l'erreur s'est produite à l'adresse 705C93 et ça corresponde à la fonction *check_ dev_ serve_ interrupt*

```
00705c8a <check_dev_serve_interrupt>:
  705c8a: 55        push  %ebp
  705c8b: 89 e5     mov   %esp,%ebp
  705c8d: 83 ec 08  sub   $0x8,%esp
  705c90: 8b 45 08  mov   0x8(%ebp),%eax
  705c93: 8b 40 3c  mov   0x3c(%eax),%eax
  705c96: 85 c0     test  %eax,%eax
  ...
```

Donc, avec cet outil j'arrive à récupérer plupart des erreurs et à les résoudre. Maintenant il est temps de tester le fonctionnement du serveur web dans Galileo.

5.5 Tester le fonctionnement de la solution

Vu que j'utilise la liaison série pour envoyer les paquets et pour ne pas gêner l'envoie de la trame SLIP sur cette liaison série, j'ai été obligé de désactiver tout type de message de débogage parce qu'il utilise aussi le port série. Pour tester le fonctionnement de la solution implémentée, il a fallu écouter les trames reçues sur l'interface SLIP, ***sl0***, qui a été configurée sur le PC auparavant. Pour cela, j'ai utilisé outil ***wireshark*** qui écoute, en mode promiscuité, sur l'interface réseau (dans notre cas, c'est l'interface SLIP, sl0).



Time	Source	Destination	Protocol	Length	Info
49.02665000(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.04858800(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.06822900(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.09014500(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.11338900(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.13414700(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.15774500(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.17835600(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.20025300(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.22285000(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.24352100(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.26527300(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.28681900(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.31097600(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.33083000(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.35384300(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)
49.37509400(	10.0.0.2	10.0.0.1	ICMP	92	Echo (ping) request id=0x91c0, seq=1/256, ttl=64 (no response found!)

FIGURE 11 – Ecoute de l'interface sl0 avec Wireshark

Mais, il semble que le test ping ne marche pas dû au fait que la réception de paquet avec la protocole SLIP pose des problèmes. En effet, je suis en train de débugger pour trouver une solution à ce bogue.

Synthèse du travail

Au début de projet, j'ai essayé de comprendre l'architecture de Pépin parce que j'allais travailler avec ce noyau tout au long de projet.

Ensuite, j'ai étudié les différents serveur web existants afin de trouver une solution adapté pour Pépin. Mais, il manquait la pile TCP/IP et/ou la sécurisation de données par SSL/TLS. Alors, j'ai décidé d'utiliser picoTCP qui fournit cette pile et ensuite ajouter wolfSSL en dessous de picoTCP. Ensuite, j'ai implémenté un serveur web au-dessus de wolfSSL. J'ai d'abord testé cette solution sous Linux en envoyant une requête HTTPS au serveur en utilisant l'outil openssl. La solution fonctionnait sous Linux.

A la suite, j'ai porté cette solution vers Pépin pour pouvoir intégré dans la carte Galileo. Les fonctions que j'ai porté inclus l'allocation mémoire, la bibliothèque mathématique et une fonction de générateur de nombre pseudo aléatoire.

J'ai dû aussi interfacer picoTCP avec le driver de la carte Galileo afin de pouvoir envoyer les paquets sur le réseau. Puisque j'avais rencontré des problèmes d'envoie et réception de paquets avec cette méthode, j'ai choisi d'utiliser le protocole IP sur sérial (SLIP) pour l'émission et réception de paquet.

Je rencontre toujours de petits problèmes lors de réception de la trame par liaison série même si l'envoie de trame est fonctionnelle. En ce moment, il n'est pas possible de tester le comportement de serveur web dans Galileo dû au problème de réception de paquets.

Conclusion

Pendant ces 6 mois de projet, je me suis familiarisé avec le système de Pépin qui n'était pas facile à comprendre au début. Cependant, l'apprentissage de ce type de système ramène à une compréhension globale du fonctionnement d'un noyau. Il est intéressant de savoir comment fonctionnent les noyaux pour pouvoir développer les applications.

J'ai eu aussi l'occasion de revoir les fonctions fondamentales comme l'allocation mémoire ce qui m'a permis d'approfondir les connaissances de gestion mémoire.

Durant cette période, j'ai affronté des problèmes lors de portage de serveur web vers Pépin (dans la carte Galileo). Grâce aux méthodes expliquées dans la section 5.4, j'ai pu identifier les problèmes survenus.

Pour conclure, ce projet a été enrichissant et m'a permis d'approfondir les connaissances en système informatique et aussi ses fonctionnalités. Je tiens, encore une fois, à remercier l'équipe 2XS d'avoir proposé ce sujet en tant que PFE.

Références

[1] Alternative Web Servers Compared : Lighttpd, Nginx, LiteSpeed and Zeus. http://royal.pingdom.com/2008/04/17/alternative-web-servers-compared-lighttpd-nginx-litespeed-and-zeus/.

[2] Comment fonctionne le certificat SSL OVH ? https://www.ovh.com/fr/ssl/fonctionnement-ssl.xml.

[3] Dynamic Storage Allocation. http://www.cs.virginia.edu/~son/cs414.f05/lec11.slides.pdf.

[4] Exokernel. http://wiki.osdev.org/Exokernel.

[5] Exokernel. https://en.wikipedia.org/wiki/Exokernel.

[6] Fonction TUN/TAP du noyau Linux. https://www.inetdoc.net/guides/vm/vm.network.tun-tap.html.

[7] Intel Galileo2 vs. Raspberry Pi2. http://eu.mouser.com/applications/galileo-2-raspberry-pi-2/.

[8] Memory Allocators 101. http://jamesgolick.com/2013/5/15/memory-allocators-101.html.

[9] Noyau de système d'exploitation. https://fr.wikipedia.org/wiki/Noyau_de_syst%C3%A8me_d%27exploitation.

[10] picoTCP wiki. https://github.com/tass-belgium/picotcp/wiki.

[11] RFC : Serial Line IP. https://tools.ietf.org/html/rfc1055.

[12] SMEWS : Smart & Mobile Embedded Web Server. http://www.cristal.univ-lille.fr/2XS/smews/.

[13] Testing HTTPS with openssl. https://blog.yimingliu.com/2008/02/04/testing-https-with-openssl/.

[14] The PIP Protokernel. http://pip.univ-lille1.fr/.

[15] wolfSSL Porting Guide. https://www.wolfssl.com/wolfSSL/Docs-wolfssl-porting-guide.html.