RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Servir du contenu embarqué via des applications Web : modèle, conception et expérimentation

Simon Duquennoy[1] Gilles Grimaud[1] Jean-Jacques Vandewalle[2]

[1] LIFL, CNRS UMR 8022, Univ. Lille 1,
INRIA Lille - Nord Europe, France
{*simon.duquennoy, gilles.grimaud*}*@lifl.fr*
[2] Gemalto Technology & Innovation, France
jean-jacques.vandewalle@gemalto.com

Résumé
Les systèmes embarqués tels les cartes à puce, les étiquettes électroniques ou les capteurs de terrain sont aujourd'hui très répandus, mais constituent pour la plupart des systèmes fermés (accédés uniquement par des terminaux dédiés, ils fonctionnent en vase clos). Une solution consiste à embarquer des serveurs Web dans ces systèmes enfouis afin d'en améliorer l'accessibilité. Dans cet article, nous proposons un modèle de performances de TCP dans le contexte des serveurs Web embarqués, ainsi qu'une taxinomie des contenus susceptibles d'être servis par une application Web embarquée. L'idée principale de cet article consiste à adapter le comportement de la pile de communication en fonction des caractéristiques des contenus applicatifs. Nous proposons pour cela un ensemble de stratégies adaptées à chaque type de contenu. Notre modèle permet d'estimer les bénéfices de ces stratégies, en terme de performances ainsi que de charge mémoire. Enfin, nous mettons à l'épreuve nos propositions grâce à une expérimentation sur carte à puce, afin d'en mesurer les réels bénéfices et de valider notre modèle. Notre prototype, nommé *Smews*, s'exécute sur des matériels très contraints et se comporte sensiblement mieux que les solutions de l'état de l'art, tant en terme de performances que de charge mémoire.

Mots-clés : Modélisation TCP, Serveur Web embarqué, Conception transversale

1. Introduction

Les systèmes embarqués tels les cartes à puce, les étiquettes électroniques ou les capteurs de terrain constituent une informatique enfouie de plus en plus importante. Le plus souvent, ce sont des systèmes fermés accédés via des terminaux dédiés. Une nouvelle solution appelée *Web of Things* [12, 11] consiste à embarquer des serveurs Web dans ces systèmes contraints afin de transformer ces derniers en supports d'applications Web. Ils peuvent ainsi être accédés, consultés et manipulés à partir d'un ordinateur (ou d'un téléphone portable) simplement muni d'un navigateur Web. Cela permet d'éviter le développement et le déploiement de clients dédiés, tout en ouvrant ces systèmes à leurs usagers et en leur permettant d'interagir avec l'Internet.
La carte à puce constitue un bon exemple d'équipement largement diffusé et enfoui : il s'en est vendu environ 5 milliards en 2008 [1] (en comparaison des 0.15 milliards d'ordinateurs portables [2]). Le standard industriel Smart Card Web Server [20] permet de déployer un serveur Web embarqué (EWS) dans des cartes à puce. Ainsi, un utilisateur peut accéder à sa carte SIM à partir du navigateur Web de son téléphone portable. En utilisant du *mashup* Web [18, 12, 11], les informations privées de la carte (*e.g.*, carnet de contacts, historique d'opérations) peuvent être fusionnées avec des données accessibles uniquement sur Internet (*e.g.*, offres commerciales, services en ligne).
Le fait de fournir un accès aisé à des systèmes enfouis déjà largement répandus constitue un véritable défi auquel nous proposons d'apporter des solutions dans cet article. Ces équipements sont si contraints

[1] http://www.researchandmarkets.com/
[2] http://www.idctracker.com/

(une carte à puce typique est basée sur un processeur 8 bit cadencé à quelques MHz accompagné d'un coprocesseur cryptographique, possède environ 1 ko de RAM et 16 ko d'EEPROM adressable) qu'ils rendent inappropriées les solutions basées sur le déploiement d'un serveur Web au dessus d'un système d'exploitation. Un serveur Web pensé comme étant son propre système d'exploitation est plus adapté à de tels matériels. De nombreux serveurs Web embarqués utilisent ce modèle [1, 5, 17, 7].

L'idée principale de cet article consiste à adapter le comportement de la pile de communication (HTTP/TCP/IP) en fonction de la nature des contenus Web pris en charge. Pour atteindre cet objectif, nous décrivons une taxinomie des ressources composant une application Web, puis nous proposons un ensemble de stratégies adaptées à chaque type de ressource. Nous justifions la pertinence de ces stratégies (i) à l'aide d'un modèle de trafic dédié que nous proposons et (ii) via une implémentation et une expérimentation sur carte à puce.

La Section 2 présente l'état de l'art des piles TCP/IP et des serveurs Web embarqués. Dans la Section 3, nous présentons un modèle de performance de TCP dans le contexte de petits serveurs accédés depuis un réseau local. Dans la Section 4, nous établissons une taxinomie des contenus d'applications Web. Sur la base de notre modèle de trafic, nous proposons un ensemble de stratégies pour servir ces contenus dans le contexte d'EWS. En Section 5, nous expérimentons nos propositions sur carte à puce, afin de valider notre modèle et nos stratégies. Enfin, nous concluons en Section 6.

2. État de l'art

Cette section présente l'état de l'art en matière de piles TCP/IP et de serveurs Web embarqués.

2.1. Piles TCP/IP embarquées

TinyTCP [4] et mIP [24] montrent qu'il est possible de communiquer avec des piles TCP/IP standard en n'implémentant qu'un sous ensemble des spécifications de TCP/IP.

Dans [6], Adam Dunkels présente deux piles TCP/IP embarquées. LwIP utilise une implémentation par couches, tandis que uIP suit une approche monolithique. uIP n'implémente qu'un sous-ensemble de TCP/IP mais nécessite moins de mémoire que lwIP. uIP utilise un modèle événementiel basé sur les protothreads [8], qui permettent d'écrire des routines bloquantes sans utiliser de pile d'exécution (avec un coût de 2 à 3 octets par protothread).

En raison de sa petite empreinte mémoire, nous avons sélectionné uIP comme notre première référence pour les expériences menées en Section 5. Le serveur Web fourni avec uIP est capable de servir des contenus statiques et dynamiques. Les fichiers sont pre-traités puis embarqués avec le serveur. Les contenus dynamiques utilisent des marqueurs dédiés dans des fichiers, à la manière de JSP/ASP/PHP.

2.2. Serveurs Web embarqués

Contrairement aux serveurs Web usuels, les serveurs Web embarqués ne sont pas contraints à être exécutés au dessus d'un système d'exploitation. Pour la plupart, ils utilisent leur propre pile TCP/IP dédiée et accèdent directement leurs pilotes de matériel. Ils peuvent donc être optimisés à tout niveau logiciel.

De nombreux travaux [1, 14, 23, 25, 13, 5, 17] montrent qu'il est possible d'embarquer un serveur Web dans des systèmes extrêmement contraints. La plupart de ces serveurs utilise un outil dédié pour prétraiter les contenus Web avant de les embarquer puis de les servir.

Le serveur Miniweb [7] a particulièrement retenu notre attention. Il a pour objectif de fournir une implémentation d'EWS à l'empreinte mémoire minimale, simplement capable de servir quelques contenus statiques. Il utilise sa propre pile TCP/IP, entièrement dédiée et liée au serveur Web. Nous avons choisi Miniweb comme notre seconde référence pour nos expériences, car il représente une borne minimale en terme de besoins matériels.

Lorsque Miniweb reçoit un paquet, il les octets entrants au fur et à mesure, sans les conserver en mémoire. Ses pilotes sont capables de fonctionner sans tampon, ou avec un tampon dont la taille est indépendante de la MSS de TCP[3]. Cela permet de réduire sa consommation mémoire à seulement quelques dizaines d'octets.

[3] La MSS (Maximal Segment Size) est la taille maximale des segments d'une connexion TCP.

Dans Miniweb, les paquets IP sont entièrement construits hors-ligne (incluant IP, TCP et HTTP). Les paquets d'ouverture de connexion, d'envoi du contenu puis de fermeture sont tous chainés en mémoire, prêts à être envoyés. Cela permet de simplifier grandement la gestion de TCP, et ainsi de réduire la taille du code de Miniweb.

3. Un modèle de TCP pour serveurs locaux et contraints

Dans cette section, nous proposons un modèle de TCP dédié aux serveurs accédés depuis un réseau local, comme le sont les serveurs Web embarqués.

3.1. Remarques préalables

Nous présentons ici une analyse du comportement des protocoles HTTP/TCP/IP lorsqu'ils interagissent pour servir une application Web.

3.1.1. HTTP : un modèle requête-réponse au dessus de TCP

HTTP est basé sur un modèle requête-réponse simple. Chaque requête se déroule sur une connexion TCP, et est encadrée d'une ouverture et d'une fermeture de connexion. Afin d'offrir une meilleure interaction avec TCP, HTTP 1.1 incite à l'utilisation de connexions persistantes, permettant à plusieurs requêtes consécutives de se dérouler sur une même connexion.
Lors d'une session HTTP, le comportement au niveau TCP est très simple. A chaque instant, un seul des deux hôtes envoie des données TCP, pendant que l'autre ne fait qu'accuser réception de ces données [4].

3.1.2. Impact des accusés différés TCP

Les accusés différés TCP [3] sont une politique introduite pour réduire le nombre d'accusés vides (segments TCP sans données). La plupart des systèmes d'exploitation implémente cette politique (*e.g.*, Windows et MacOs). Une pile TCP qui implémente les accusés différés n'accuse un segment que (i) 200 ms après sa réception ou (ii) dès qu'il reçoit un second segment.
De nombreuses piles TCP/IP embarquées n'ont jamais plus d'un paquet en vol (envoyé mais non accusé), ce qui permet une implémentation sans état simple et légère. Ce choix est particulièrement contre-performant face aux accusés différés, impliquant une limitation à 5 paquets par seconde (le délai de 200 ms est toujours déclenché chez l'hôte distant).

3.1.3. Trafic typique des applications Web

La méthodologie AJAX [10] est maintenant très utilisée pour l'élaboration d'applications Web. Une analyse que nous avons conduit précédemment [9] permet de caractériser le trafic produit par des applications AJAX largement utilisées sur Internet.
Dans un premier temps, le client reçoit un ensemble de fichiers structurant l'application (XHML, CSS, JavaScript). Ensuite, l'application s'exécute : des données applicatives sont échangées entre le client et le serveur. On constate que les fichiers statiques sont majoritairement plus grands (moyenne de 8 ko) que les contenus dynamiques (moyenne de 0.6 ko).

3.2. Modèle de trafic TCP/IP dédié

La modélisation du trafic de TCP est un sujet de recherche actif de cette dernière décennie. De nombreux de travaux ont pour objectif de caractériser le débit maximal obtenu en envoi de données continu, en fonction du taux de pertes de paquets et du RTT[5] [2, 16, 19, 21, 22]. La complexité de ces modèles est principalement due aux mécanismes de contrôle de congestion TCP et à la modélisation de la congestion du réseau. Le modèle le plus précis a été proposé dans [2], où les auteurs décrivent et s'appuie sur un modèle de congestion réseau très précis.

3.2.1. Présentation du modèle

Nous proposons un modèle adapté au cas de serveurs embarqués accédés via un réseau local. Plutôt que d'estimer un taux d'envoi maximal asymptotique, il mesure le temps requis par un hôte pour envoyer

[4] Si le *pipelining* HTTP est utilisé, les envois peuvent être bidirectionnels. Pour plusieurs raisons, les navigateurs Web actuels n'utilisent pas le *pipelining* par défaut.
[5] Le RTT est le temps d'aller retour entre deux hôtes.

une quantité de données bornée. Dans ce contexte, nous supposons que le serveur ne causera pas de congestion sur le réseau, en raison de la lenteur de son interface de communication en comparaison aux capacités du réseau local. Dans ce cas, la fenêtre d'envoi est limitée par la mémoire du serveur, car chaque segment en vol doit être conservé en tampon tant qu'il n'a pas été accusé. En pratique, la fenêtre maximale annoncée par Windows et linux est de respectivement 8 ko et 5 ko, ce qui est largement supérieur à la quantité de mémoire vive présente dans les petits systèmes embarqués.
Ce modèle considère le cas où la politique des accusés différés est utilisée (la situation la plus courante, car les piles TCP de Windows et MacOS implémentent cette politique). Nous supposons que le lien de communication est *half-duplex*, ce qui est le cas de la plupart des protocoles utilisés par les systèmes embarqués (*e.g.*, USB, ZigBee, APDU, Wifi, Ethernet). Nous notons BW le débit maximal du lien entre les deux piles TCP, tandis que le temps d'aller retour est noté RTT.

3.2.2. Définition du modèle

Notre modèle définit $ST(cs, wnd, n_{max})$ où ST est le temps d'envoi en fonction de cs (la taille du contenu à envoyer), wnd (le nombre maximal d'octets en vol) et n_{max} (le nombre maximal de segments en vol). Soit $t(n, s)$ le temps requis pour envoyer cs octets en n paquets. Les en-têtes de IP et de TCP sont de chacun 20 octets (lorsqu'ils sont utilisés sans option, ce qui constitue le cas le plus fréquent) :

$$t(s, n) = \frac{40 \times n + s}{BW} \tag{1}$$

Soit T le temps requis pour envoyer cs octets en n segments TCP et recevoir tous les accusés TCP. À cause de la politique des accusés différés, le nombre d'accusés vides reçus est $\lceil \frac{n}{2} \rceil$. Lorsque le nombre de segments est impair, l'accusé du dernier segment est retardé de 200 ms après la réception du segment.

$$T(s, n) = \begin{cases} t(s, n) + t\left(0, \lceil \frac{n}{2} \rceil\right) + RTT & \text{if } n \text{ is even} \\ t(s, n) + t\left(0, \lceil \frac{n}{2} \rceil\right) + RTT + 0.2 & \text{otherwise} \end{cases} \tag{2}$$

Lorsqu'un hôte envoie un contenu de cs octets, il envoie tout d'abord d fois wnd octets. wnd est le nombre maximal d'octets envoyés mais non encore accusés (égal à la somme des tailles des segments en vol). La valeur de d est :

$$d = \left\lfloor \frac{cs}{wnd} \right\rfloor \tag{3}$$

Ces $d \times wnd$ octets sont envoyés en n paquets. Supposons qu'afin d'éviter d'avoir un nombre impair de paquets en vol (pour mieux interagir avec les accusés différés), la pile TCP choisit une valeur paire pour n lorsque cela est possible. Afin de simplifier nos formules, nous définissons une fonction $toEven$:

$$toEven(a) = \begin{cases} a & \text{if } n \text{ is even} \\ a + 1 & \text{otherwise} \end{cases} \tag{4}$$

Soit $n1$ le nombre de segments TCP composant chaque fenêtre pleine :

$$n_1 = min\left(n_{max}, toEven\left(\left\lceil \frac{wnd}{MSS} \right\rceil\right)\right) \tag{5}$$

Lorsque $d \times wnd$ octets ont été envoyés, il reste r octets à envoyer :

$$r = cs \mod wnd \tag{6}$$

Ces r octets restants sont envoyés en n_2 segments TCP :

$$n_2 = min\left(n_{max}, toEven\left(\left\lceil \frac{r}{MSS} \right\rceil\right)\right) \tag{7}$$

Finalement, la durée d'envoi ST peut être calculée comme :

$$ST(cs, wnd, n_{max}) = d \times T(wnd, n_1) + T(r, n_2) \tag{8}$$

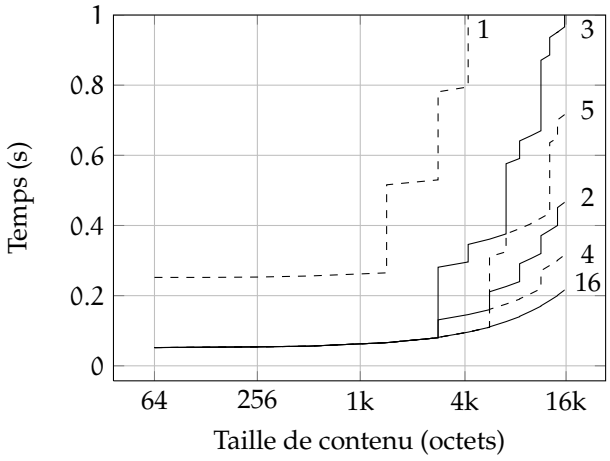

FIG. 1 – Temps d'envoi en fonction de la taille du contenu. Chaque courbe représente un nombre de segments en vol maximal différent.

La Figure 1 synthétise notre modèle en montrant le temps requis pour envoyer un contenu en fonction sa taille. Chaque courbe représente une valeur différente de n_{max}. La MSS est positionnée à 1460 octets[6]. Le débit est de 100 ko/s, et le RTT est de 50 ms.

Le temps d'envoi évolue par étapes, à cause de la discrétisation introduite par le découpage des données en paquets. Les courbes montrent à quel point les performances sont dégradées avec $n_{max} = 1$. Les autres valeurs impaires pour n_{max} sont également contre-performantes, à cause des accusés différés. Pour des valeurs paires, l'envoi est d'autant plus rapide que n_{max} est grand.

3.3. Résumé et analyse

L'analyse de trafics que nous avons réalisée a permis d'identifier trois points particulièrement importants pour la conception de pile TCP/IP pour EWS :

Support de multiples segments en vol Ce point est coûteux car les segments en vol doivent être stockés en mémoire, mais nous avons montré qu'il avait un impact fort sur les performances.

Support des connexions persistantes HTTP Cela permet d'éviter de nombreuses ouvertures et fermetures de connexions, mais nécessite de gérer plusieurs connexions simultanément, donc de les stocker en mémoire.

Support de grandes MSS Une grande MSS améliore le débit utile, mais nécessite de gérer de plus grands paquets, ce qui est difficile dans le contexte de systèmes très contraints en mémoire.

Comme la plupart des piles TCP embarquées, uIP est limité à un seul paquet en vol. Miniweb n'envoie que des séries de paquets IP calculés hors-ligne, il n'a donc pas besoin de conserver ses paquets en vol en mémoire vive, et supporte ainsi plusieurs paquets en vol. Par contre, cela lui impose de fixer la taille des segments à la compilation. Cette taille doit être aussi petite que TCP l'autorise (200 octets), car lors de l'ouverture d'une connexion, la MSS est choisie comme la valeur minimale proposée par les deux hôtes. Enfin, le serveur Web de uIP ainsi que Miniweb ne gèrent pas les connexions persistantes, afin de limiter le nombre de connexions simultanément présentes en mémoire.

4. Un framework dédié aux applications Web embarquées

Dans cette section, nous présentons une taxinomie des contenus Web, puis nous proposons un ensemble de stratégies en adéquation avec cette taxinomie, afin de permettre l'élaboration d'un framework dédié aux applications Web embarquées.

[6] La MSS la plus couramment utilisée dans TCP est de 1460 octets.

4.1. Une taxinomie des contenus Web

Nous proposons ici une taxinomie des différents types de contenus d'applications Web. Nous avons identifié deux critères de classification. Le premier (noté A) est basé sur la taille du contenu :

A1 - Contenu court contenus plus petits que la fenêtre d'envoi TCP ;

A2 - Contenu long contenus plus grands que la fenêtre d'envoi TCP.

Le second critère (noté B) concerne la persistance des données :

B1 - Contenu statique fixé hors-ligne Il s'agit des fichiers connus hors-ligne embarqués avec le serveur.

B2 - Contenu statique fixé à l'exécution Ces contenus sont fixés lors de l'exécution puis servis plusieurs fois. Exemple : carte SIM fournissant un une liste de contacts.

B3 - Contenu généré persistent Ces contenus sont générés par le serveur indépendamment de l'état matériel et/ou logiciel de l'équipement. Un second service produira un résultat différent ou aura des effets de bord. Exemple : configuration, ou modification de session, etc.

B4 - Contenu généré volatile Ces contenus doivent être générés aussi tard que possible avant la réception du client. Exemple : fonction qui retourne un échantillon mesuré par un capteur.

B5 - Contenu généré idempotent Une fonction idempotente est une fonction déterministe et sans effet de bord (inspiré des Untrusted Deterministic Functions [15]). Dans un contexte donné, elle peut être appelée autant de fois que nécessaire, en produisant toujours le même résultat et sans modifier le contexte. Exemple : chiffrement ou calcul sur des données.

4.2. Une métrique dédiée au service de contenus embarqués

Dans le contexte de systèmes contraints, le service d'un contenu ne doit pas seulement être rapide, mais aussi économe en mémoire. Ici, nous prenons pour exemple uIP[6], une des piles TCP/IP embarquées les plus populaires. uIP impose une limite à un seul segment TCP en vol en permanence.

Dans le serveur Web de uIP, lors du service d'un contenu statique, les segments envoyés ne sont pas stockés en mémoire, car il est aisé de les re-générer en cas de retransmission. Le temps requis pour l'envoi d'un contenu de cs octets est :

$$T_s = ST(cs, MSS, 1) \tag{9}$$

Lors du service d'un contenu dynamique, chaque segment en vol est stocké en RAM, afin de pouvoir faire face à une éventuelle retransmission. Dans une première phase (a), les segments envoyés sont de la taille de la MSS. Dans une seconde phase (b), la pile envoie la fin des données (de taille $cs \mod MSS$ octets). La durée totale de service d'un contenu de cs octets est donc de :

$$T_{d_a} = ST(cs - cs \mod MSS, MSS, 1) \tag{10}$$

$$T_{d_b} = ST(cs \mod MSS, MSS, 1) \tag{11}$$

$$T_d = T_{d_a} + T_{d_b} = ST(cs, MSS, 1) \tag{12}$$

Ces deux exemples illustrent le fait qu'à lui seul, le temps de transmission n'est pas un critère suffisant pour évaluer une stratégie d'envoi de données sur un canal TCP. La quantité de mémoire utilisée pour cet envoi est aussi un critère important. Nous proposons d'introduire la notion de *charge mémoire* afin de synthétiser ces deux critères. La charge mémoire est l'intégration de la quantité de mémoire utilisée pendant l'envoi. Elle s'exprime en $octet.seconde$ (o.s). En réduisant la charge mémoire, un serveur Web peut être embarqué dans des matériels plus contraints (donc moins chers), et il passe mieux à l'échelle (un EWS sert pas des milliers de clients, mais des problèmes de passage à l'échelle peuvent survenir avec seulement quelques connexions).

Avec le serveur Web de uIP, lors de l'envoi d'un contenu statique, la charge mémoire est nulle, car les paquets en vol ne sont pas stockés en mémoire. Lors de l'envoi d'un contenu dynamique, la charge mémoire est :

$$C_d = T_{d_a} \times MSS + T_{d_b} \times (cs \mod MSS) \tag{13}$$

	B1	B2	B3	B4	B5
A1	S0	S0	S1	S2	S1,S2
A2	S0	S0	S3	S4	S3,S4

TAB. 1 – Stratégies appliquées pour chaque type de contenus de notre taxinomie

4.3. Stratégies pour le service de contenus Web

Dans cette section, nous proposons des stratégies pour servir efficacement (en termes de performances et de charge mémoire) les contenus Web en fonction de leur classification dans notre taxinomie. Nous avons identifié cinq stratégies différentes, synthétisées dans la Table 1.

Stratégie S0 : "*contenus pré-traités*"

Cette stratégie est utilisée pour les fichiers statiques (B1, B2). Leurs *checksums* TCP sont calculés hors-ligne, et sont placés aux cotés des données des fichiers, dans une mémoire persistante adressable. L'adresse des données à envoyer est calculée à partir du numéro de séquence TCP courant, il n'est donc pas nécessaire de stocker les segments envoyés en mémoire pour d'éventuelles retransmissions. La charge mémoire est nulle, et il n'y a aucune limitation sur le nombre de segments en vol.
A l'aide de notre modèle de trafic (dans notre configuration réseau de référence : débit de 10000 o/s, RTT de 50 ms, MSS de 1460 octets), on calcule que le service d'un fichier de 8 ko (soit la moyenne des contenus statiques observés en Section 3.1.3) est 11.2 fois plus rapide avec S0 qu'avec la stratégie de uIP.

Stratégie S1 : "*tampon d'attente d'accusés*"

Cette stratégie s'applique aux contenus dynamiques persistants de petite taille (A1/B3). Le générateur de contenu, écrit par le développeur d'application Web, réalise des appels à une routine *outc* pour envoyer des données. Cette routine stocke les données dans un tampon et calcule le *checksum* TCP à la volée. À la fin de l'exécution du générateur, la réponse HTTP est envoyée. Le segment est conservé en mémoire tant qu'il n'est pas accusé, afin de permettre des retransmissions. Un tampon partagé permet de servir simultanément plusieurs contenus A1/B3. Lorsque ce tampon est plein, les requêtes sont mises en attente ; elles s'exécuteront lorsque la réception d'accusés aura permis de libérer de la mémoire.
Selon notre modèle, et dans notre configuration réseau de référence, l'envoi d'un contenu de 600 octets (soit la moyenne des contenus dynamiques observés en Section 3.1.3) est 4.5 fois plus rapide avec S1 qu'avec la stratégie de uIP. Le ratio de charge mémoire entre les deux stratégies est aussi de 4.5.

Stratégie S2 : "*tampon volatile*"

Cette stratégie est utilisée pour les contenus dynamiques volatiles de petite taille (A1/B4). Comme pour S1, la fonction génératrice utilise *outc* pour envoyer des données. Dès que la réponse HTTP est envoyée, elle est défaussée de la mémoire. Lorsqu'une retransmission est nécessaire, le générateur est appelé une nouvelle fois pour produire une nouvelle réponse HTTP. Cette stratégie est aussi rapide que S1, et a une charge mémoire nulle.

Stratégie S3 : "*générateur persistent*"

Cette stratégie s'applique aux contenus dynamiques persistants de grande taille (A2/B3). La fonction génératrice, basée sur une routine *outc* bloquante, ne doit être exécutée qu'une fois par requête HTTP. Lorsque le tampon est plein (ou rempli de plus d'une MSS), la routine bloquante rend la main à la pile TCP, qui envoie un segment (ou deux, si nécessaire pour éviter un nombre impair de paquets en vol) puis continue de traiter l'activité du réseau. La pile rend la main au générateur aussi tôt que possible afin qu'il termine son exécution.
Soit M la taille du tampon utilisé par la routine *outc*. Avec $M = MSS$, selon notre modèle, dans la configuration réseau de référence, l'envoi d'un contenu dynamique de 8 ko est 4.1 fois plus rapide avec S3 qu'avec la stratégie de uIP. Une analyse du modèle montre que le temps d'envoi avec S3 n'est pas proportionnel à M : il augmente plus lentement que ce dernier. En augmentant M, le service est plus rapide, mais la charge mémoire est plus élevée. En utilisant S3 avec $M = 300$ octets, l'envoi a la même

durée qu'avec uIP (qui, elle, utilise toujours un tampon de 1460 octets), impliquant une charge mémoire bien moindre.

Stratégie S4 : *"générateur volatile"*

Cette stratégie est utilisée pour les contenus dynamiques volatiles de grande taille (A2/B4). La fonction génératrice prend en paramètre le numéro de séquence relatif actuel ainsi que la quantité maximale de données à générer. Elle est capable de générer des fragments de la réponse HTTP en fonction de ces arguments. Il n'y a aucune limitation sur le nombre de paquets en vol, car chaque segment peut être généré ou re-généré à tout instant. Cette stratégie est aussi rapide que S0 et a une charge mémoire nulle. Il est à noter qu'en utilisant la stratégie S2 ou S4, les segments retransmis peuvent différer des segments originaux. Ce comportement n'est pas en accord avec la sémantique originale de TCP, mais cela n'engendre aucun effet indésirable en pratique.

Générateurs idempotents

Les contenus idempotents (B5) laissent la pile TCP libre de choisir la stratégie la plus efficace. En cas de retransmission, les données peuvent être re-générées (S4) ou récupérées depuis un tampon (S3). Dans la plupart des cas, S4 est plus efficace et économe en mémoire. Cependant, dans certains cas particuliers où le générateur réalise de lourd calculs (*e.g.*, chiffrement), S3 peut être plus intéressant en terme de temps, de charge mémoire et par conséquent, d'énergie. Le choix de la meilleure stratégie dans ce cas pourrait se baser sur une mesure du temps d'exécution du générateur, et reste un problème ouvert

4.4. Une architecture dédiée aux serveurs Web embarqués

Afin de fournir un support pour l'ensemble des stratégies décrites en Section 4.3, nous proposons une architecture conçue de manière transversale, où le framework applicatif, HTTP et la pile TCP/IP sont étroitement liés et indissociables.

4.4.1. Description du moteur d'exécution

Notre architecture est événementielle avec la granularité des paquets IP. Les paquets entrants sont traités séquentiellement, engendrant des événements au niveau TCP, HTTP ou applicatif. Couplée à nos stratégies, cette architecture n'utilise que peu de tampons. Le tampon de réception peut être plus petit que la MSS, car les données entrantes sont traitées octets par octet, dès leur réception. La structure utilisée pour représenter l'état des connexions est très différente des *sockets* usuels, et n'inclut aucun tampon. Cela permet de supporter les points critiques identifiés en Section 3.3 :

Support de plusieurs paquets en vol Dans la plupart des cas (fichiers statiques, contenus volatiles ou idempotents), il n'y a pas de limitation sur ce point. Dans les autres situations, le nombre de paquets en vol est limité par la mémoire disponible.

Connexions persistantes HTTP Chaque connexion occupe entre 30 et 36 octets en mémoire, ce qui rend aisé le support de multiples connexions persistantes simultanées.

Gestion de grandes MSS En entrée comme en sortie, notre architecture gère de grands paquets même avec de petits tampons.

4.4.2. Une API pour les contenus dynamiques

Comme décrit en Section 4.3, les générateurs de contenus envoient des données via des appels à *outc*, sans se préoccuper de l'état courant de la pile TCP. Dépendamment de la stratégie utilisée, les appels à *outc* ont des comportements différents.

Implémentation de générateurs persistants

Dans le cas de S1 et S3, la routine *outc* est bloquante, car le code des générateurs ne doit s'exécuter qu'une seule fois. Nous proposons d'utiliser les protothreads [8], qui fournissent une abstraction bloquante avec un faible coût

```
Data : i a static integer
input : samples a table to send,
        nb_samples its size
PT_BEGIN
for i=0 to nb_samples do
    preprocess_sample(samples[i])
    PT_OUTC(samples[i])
end
flush_samples(samples, 0, nb_samples)
PT_END
```

Algorithme 1 : Générateur persistent implémenté avec des protothreads

```
Data : start, end, i local integers
input : samples a table to send,
        nb_samples its size,
        seq_index the sequence index,
        seq_length the sequence size
start ← seq_index
end ← min(start + seq_length, nb_samples)
for i=start to end do
|  outc(samples[i])
end
```

Algorithme 2 : Générateur volatile

```
Data : start, end, i local integers
input : samples a table to send,
        nb_samples its size,
        state the current state
PT_BEGIN
for state.i=0 to nb_samples do
|  PT_OUTC_CTX(samples[state.i], state)
end
PT_END
```

Algorithme 3 : Générateur volatile implémenté avec des protothreads étendus

en mémoire. L'algorithme 1 est un exemple de générateur
persistent pour un capteur de terrain. Il traite puis envoie un ensemble d'échantillons. *PT_OUTC* réalise un appel à *outc* et bloque jusqu'à ce que les données soient effectivement envoyées et accusées.

Implémentation de générateurs volatiles

S2 et S4 se basent sur une version non bloquante de *outc*. Le générateur s'exécute dans un contexte donné (index et taille des données à produire), fourni par la pile TCP. L'algorithme 2 est un exemple de générateur volatile qui envoie un ensemble d'échantillons, sans effet de bord. Cette fonction est sans état et peut être appelée autant de fois que nécessaire avec n'importe quels paramètres.
Dans le cas général, il n'est pas aisé d'écrire une telle fonction. Nous proposons une extension des protothreads afin de faciliter l'écriture de générateurs volatiles. Comme mentionné par leurs auteurs [8], un état peur être associé aux protothreads pour le stockage de variables locales. En stockant de multiples *continuations* et états pour chaque générateur, il est possible de ré-exécuter n'importe quelle part du code du générateur autant de fois que nécessaire. En fait, une paire continuation/état est conservée pour chaque segment en vol, permettant sa reconstruction au besoin, avec une consommation mémoire très basse. D'une manière plus générale, cette approche de multiples paires continuation/états par protothread constitue une manière intéressante de supporter des points de re-jeu.
L'algorithme 3 a le même comportement que l'algorithme 2 mais est écrit avec nos protothreads étendus. Le code est plus lisible et facile à écrire, car il ne contient plus de calcul d'index explicite. Ici, l'état ne contient que l'itérateur de la boucle.

5. Expérimentations

Afin de mettre à l'épreuve nos propositions et stratégies, nous utilisons notre propre prototype, nommé *Smews*. Il fournit une API pour concevoir des applications Web embarqués en adéquation avec la taxinomie décrite en Section 4.1. Nous comparons notre prototype avec les deux EWS que nous avons sélectionnés comme références (le serveur de uIP ainsi que Miniweb). Afin de rendre nos expérimentations reproductibles, nous diffusons le code source de Smews publiquement[7].

5.1. Description de l'application Web de référence

Nous prenons pour exemple l'application décrite en Section 1. Le serveur Web est embarqué dans une carte SIM. Il exécute une application de gestion du carnet d'adresses.
L'application Web est constituée de quatre fichiers statiques et trois contenus dynamiques :

index.html, style.css, logo.png trois fichiers constituant la page principale, de respectivement 752, 826 et 5805 octets ;

script.js les scripts client (de 3613 octets), incluant des interactions AJAX ;

cb_extract générateur idempotent utilisé pour servir la liste des contacts ;

cb_get générateur idempotent retournant les données d'un contact identifié par des arguments d'URL ;

[7] Smews est disponible à : `http://www2.lifl.fr/~duquenno/Research/Smews`

cb_add générateur persistent qui ajoute un contact, et retourne un code de statut.

L'implémentation de cette application est entièrement possible dans Smews, car il supporte les contenus statiques et dynamiques, ainsi que les arguments d'URL. La nature des contenus dynamiques (volatiles, idempotents et persistants) est spécifiée par le concepteur de l'application Web, puis prise en compte par Smews à lors de l'exécution. Dans uIP, nous avons pu implémenter une version simplifiée de l'application, sans le typage avancé des contenus dynamiques. Quant à Miniweb, il est seulement capable de servir les fichiers statiques.

5.2. Configuration des expérimentations

Du coté du client, nous avons choisi un ordinateur utilisant Windows XP et Internet Explorer 6 (ce qui constitue la configuration la plus utilisée par les clients sur Internet[8]). Il est important de noter que la pile TCP/IP de Windows implémente la politique TCP des accusés différés.

Nous avons porté uIP, Miniweb et Smews sur une même carte à puce (une Funcard7). Elle utilise un microcontrôleur AVR 8 bits cadencé à 8 MHz avec 8 ko de RAM et 16 ko d'EEPROM. La Funcard communique via une liaison série, dont le débit est de 10 ko/s. Sa latence engendre un RTT de 5 ms.

Nous présentons dans [9] une analyse des empreintes mémoire minimales de trois serveurs sur la Funcard. uIP et son serveur Web requièrent environ 3 ko de RAM et 12 ko d'EEPROM. Smews n'utilise qu'environ 200 octets de RAM et 8 ko d'EEPROM. Miniweb n'a besoni que de 100 octets de RAM et 4 ko d'EEPROM pour s'exécuter (ses fonctionnalités sont très limitées : il est incapable de servir des contenus dynamiques, de gérer de multiples connexions ou des MSS variables, et il ne décode pas les requêtes HTTP).

5.3. Résultats expérimentaux

Notre procédure de test se décompose en deux étapes. La première étape est la connexion initiale entre le navigateur Web et la carte à puce. Les quatre fichiers statiques sont servis, et la liste des contacts est récupérée via une requête à `cb_extract`. Durant la seconde phase, nous mettons à l'épreuve les générateurs de petits contenus dynamiques : des appels à `cb_label` et `cb_add` sont alors effectués.

5.3.1. Performance

Ressource	uIP			Miniweb			Smews			Facteur de vitesse	
	modèle	mesure	diff.	modèle	mesure	diff.	modèle	mesure	diff.	$\frac{uIP}{Smews}$	$\frac{Mweb}{Smews}$
index.html 752 o	0.77	0.70	10 %	0.12	0.14	-11 %	0.14	0.16	-11 %	× 4.5	× 0.9
style.css 826 o	0.78	0.70	11 %	0.14	0.14	-4 %	0.15	0.16	-9 %	× 4.4	× 0.9
script.js 3613 o	1.48	1.36	9 %	0.47	0.50	-5 %	0.43	0.44	-1 %	× 3.1	× 1.1
logo.png 5805 o	1.91	1.76	9 %	0.73	0.78	-6 %	0.66	0.66	0 %	× 2.7	× 1.2
cb_extract 1915 o	1.10	1.01	4 %	–	–	–	0.26	0.29	-11 %	× 3.5	–
cb_get 28 o	0.70	0.68	2 %	–	–	–	0.06	0.06	0 %	× 10.6	–
cb_add 2 o	0.70	0.67	4 %	–	–	–	0.06	0.06	0 %	× 10.8	–
page entière	–	8.2	–	–	2.6	–	–	1.8	–	× 4.6	× 1.4

TAB. 2 – Performances des trois serveurs mesurées et comparées au modèle (en secondes). Pour chaque ressource, l'erreur commise par le modèle est indiquée comme un pourcentage par rapport aux mesures.

La Table 2 montre le temps passé par chacun des trois serveurs pour servir chaque ressource constituant l'application Web. Les mesures sont réalisées avec *tcpdump*, et comparées aux valeurs théoriques calculées par notre modèle.

Le serveur Web de uIP ainsi que Miniweb ne supportent pas les connexions persistantes. Pour rendre les résultats comparables, les temps d'ouverture et de fermeture de connexion ne sont pas inclus dans les résultats que nous présentons, excepté dans le temps de service global de la page. C'est pourquoi, pour uIP et Miniweb, ce temps global est largement supérieur à la somme des temps individuels.

Tout d'abord, nos mesures soulignent l'écart de performances entre uIP et Miniweb/Smews (8.2 s pour la page entière, comparé à 2.6 ou 1.8 s avec Miniweb/Smews). C'est principalement dû au fait que

[8] Statistiques des clients WWW : `http://www.w3schools.com/browsers/`

uIP est limité à un seul paquet en vol. Miniweb est plus rapide (de 10 %) que Smews pour seulement deux fichiers statiques. C'est parce que (i) ses réponses HTTP contiennent un en-tête minimal et (ii) il commence à envoyer sa réponse avant d'avoir reçu la requête entière[9]. Dans les autres situations, Smews est plus rapide que Miniweb, et supporte toutes les ressources de l'application.
Ensuite, ces mesures montrent que le modèle que nous avons proposé est assez satisfaisant (erreur maximale de 11 %). Pour Miniweb et Smews, le modèle sous-estime toujours la valeur mesurée. C'est dû aux surcoûts non prédictibles du coté du client et du réseau. Pour uIP, le modèle surestime les mesures. C'est dû à l'implémentation des accusés différés TCP dans Windows, qui diffère légèrement des standards : il ne déclenche pas le délai de 200 ms lorsqu'un paquet est entièrement reçu, mais dès la réception du premier octet.

5.3.2. Charge mémoire

A l'aide de capture de trafics et de l'accès au code source des serveurs, nous avons pu estimer la charge mémoire provoquée par chaque requête sur des contenus dynamiques. La Table 3 présente ces mesures et les compare aux valeurs calculées par notre modèle.

Ressource		uIP		Smews	
		modèle	mesure	modèle	mesure
cb_extract	1915 B	666	603	0	0
cb_get	28 B	29	30	0	0
cb_add	2 B	25	24	1.3	1.5

TAB. 3 – Charges mémoire mesurées comparées au modèle (en $octet.seconde$)

L'erreur maximale commise par le modèle est de 11 %. Ici, Smews sert les contenus idempotents (`cb_extract` et `cb_get`) comme des contenus volatiles (sans les conserver en mémoire), permettant d'avoir une charge mémoire nulle dans ce cas. `cb_add` retourne une réponse de 2 octets avec une charge de 1.5 o.s pour Smews contre 24 o.s pour uIP. Cela montre l'efficacité des stratégies introduites en Section 4.3. La charge mémoire de uIP est élevée lors du service de longs contenus (603 o.s pour 1.9 ko de données), tandis qu'elle reste nulle chez Smews. Miniweb n'apparaît pas dans ces comparaisons car il n'est pas capable de servir de contenus dynamiques.
Ces expérimentations montrent la faisabilité et l'efficacité de nos stratégies et de l'architecture que nous proposons. Elles soulignent aussi la bonne précision du modèle présenté en Section 3.

6. Conclusions

La plupart des systèmes informatiques embarqués sont des systèmes fermés communiquant avec des terminaux dédiés. Ces solutions *ad hoc* permettent d'atteindre les meilleurs compromis entre coûts et performances, mais rendent difficile l'interaction généralisée entre usagers, équipements de tous types, et réseaux tels que l'Internet. En nous intéressant aux solutions basées sur des serveurs Web embarqués, nous avons montré qu'il était possible de transposer un modèle applicatif déjà répandu au monde de l'embarqué, tout en respectant de très fortes contraintes sur les coûts de fabrication.
Nous avons établi et validé un modèle de trafic pour les petits systèmes envoyant des données au dessus de TCP. Les stratégies que nous avons établies à partir de ce modèle permettent de servir efficacement tous types de contenus Web avec une très faible charge mémoire. En étant soucieuses à la fois de la couche de transport et des contenus applicatifs, elles rendent possible l'utilisation d'un modèle applicatif de haut niveau (en l'occurrence, celui du Web) dans les matériels les plus contraints. Notre prototype, Smews, confirme l'efficacité de nos stratégies dans ce contexte face aux solutions de l'état de l'art.
Dans nos travaux futurs, nous souhaitons étudier l'usage de serveurs Web pour réseaux de capteurs. Dans ce contexte, les communications multi-sauts et les pertes de paquets fréquentes invalident une partie des hypothèses initiales de notre modèle, et introduisent de nouveaux défis. Nous souhaitons également étudier l'impact du support de connexions sécurisées sur nos stratégies de service.

[9] Miniweb ne décode pas les requêtes HTTP, il se contente d'affecter un port à chaque page à servir.

Bibliographie

1. I. Agranat. Engineering web technologies for embedded applications. *Internet Computing, IEEE*, 2(3) :40–45, May-June 1998.
2. E. Altman, K. Avrachenkov, and C. Barakat. A stochastic model of tcp/ip with stationary random losses. *IEEE/ACM Trans. Netw.*, 13(2) :356–369, 2005.
3. R. Braden. Rfc 1122 : Requirements for internet hosts - communication layers, 1989.
4. G. H. Cooper. Tinytcp, 2002. `http://www.csonline.net/bpaddock/tinytcp/`.
5. M. Domingues. A simple architecture for embedded web servers. *ICCA'03*, 2003.
6. A. Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys '03 : Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM Press.
7. A. Dunkels. The proof-of-concept miniweb tcp/ip stack, 2005. `http://www.sics.se/~adam/miniweb/`.
8. A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads : simplifying event-driven programming of memory-constrained embedded systems. In *Proc. of SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM Press.
9. S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The web of things : interconnecting devices with high usability and performance. In *6th International Conference on Embedded Software and Systems (ICESS'09)*, HangZhou, Zhejiang, China, May 2009.
10. J. J. Garrett. Ajax : A new approach to web applications. Adaptivepath, 2005.
11. D. Guinard and V. Trifa. Towards the web of things : Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, Apr. 2009.
12. D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards physical mashups in the web of things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, June 2009.
13. G.-j. Han, H. Zhao, J.-d. Wang, T. Lin, and J.-y. Wang. Webit : a minimum and efficient internet server for non-pc devices. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 5, pages 2928–2931 vol.5, 2003.
14. H.-T. Ju, M.-J. Choi, and J. W. Hong. An efficient and lightweight embedded web server for web-based network element management. *Int. J. Netw. Manag.*, 10(5) :261–275, 2000.
15. M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP '97*, pages 52–65, New York, NY, USA, 1997. ACM Press.
16. T. V. Lakshman and U. Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3) :336–350, 1997.
17. T. Lin, H. Zhao, J. Wang, G. Han, and J. Wang. An embedded web server for equipments. *ispan*, 00 :345, 2004.
18. X. Liu, Y. Hui, W. Sun, and H. Liang. Towards service composition based on mashup. volume 0, pages 332–339, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
19. M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3) :67–82, 1997.
20. OMA. Smartcard-web-server, 2008.
21. J. Padhye, V. Firoiu, and D. Towsley. A stochastic model of tcp reno congestion avoidence and control. Technical report, Amherst, MA, USA, 1999.
22. J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling tcp reno performance : a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2) :133–145, 2000.
23. J. Riihijarvi, P. Mahonen, M. Saaranen, J. Roivainen, and J.-P. Soininen. Providing network connectivity for small appliances : a functionally minimized embedded web server. *Communications Magazine, IEEE*, 39(10) :74–79, Oct. 2001.
24. S. Shon. Protocol implementations for web based control systems. *International Journal of Control, Automation, and Systems*, 3 :122–129, March 2005.
25. H. Shrikumar. Ipic - a match head sized webserver., 2002.