Haute performance pour serveurs Web embarqués

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

[1] IRCICA/LIFL, CNRS UMR 8022, Univ. Lille 1, INRIA Futurs, équipe POPS
{Simon.Duquennoy,Gilles.Grimaud}@lifl.fr
[2] Gemalto Labs, France
jean-jacques.vandewalle@gemalto.com

Résumé Les systèmes informatiques qui nous entourent ont un besoin grandissant d'accessibilité. Nous nous intéressons ici à la solution consistant à embarquer un serveur Web dans ces matériels, ensuite accédés via un simple navigateur Web. Les technologies imposées par l'utilisation d'un serveur Web sont cependant peu adaptées à l'embarqué.

Après une analyse poussée des solutions existantes, nous proposons et testons de nouvelles techniques pour rendre possible le service d'applications Web interactives par des matériels ne disposant que de quelques kilo-octets de mémoire vive. Notre implémentation, *dynaWeb*, est plus rapide que les solutions existantes, pèse moins d'un kilo-octet de mémoire volatile, et permet le service d'applications AJAX interactives.

Mots clés : systèmes embarqués, serveur Web, micro-IP, AJAX

1 Introduction

1.1 Contexte

Un nombre sans cesse croissant de systèmes informatiques envahit notre quotidien dans la plus grande discrétion, de façon quasiment imperceptible. Nous avons besoin d'interagir efficacement avec ces systèmes pour les exploiter au mieux. Un routeur, une carte à puce ou un capteur de terrain ont besoin d'échanger des informations avec le monde qui les entoure, que ce soit pour informer leurs usagés ou pour en recevoir des instructions.

Ce besoin grandissant d'interaction rend de plus en plus inadaptée la solution classique du développement d'applications clientes et serveurs utilisant des protocoles dédiés, tels que les modèles prônés jusqu'alors pour les cartes à puces ou les systèmes de domotique. Ces applications taillées sur mesures, si elles ont le mérite d'être efficaces, nécessitent un temps et un coût de développement trop importants. Leur évolution est difficile car elle peut nécessiter la mise à jour des clients. De plus, une telle approche nécessite une lourde phase d'installation du coté de la machine cliente (forcément pré-identifiée) souhaitant interagir avec le système embarqué (qu'il s'agisse d'un guichet de retrait bancaire ou d'un PDA utilisé pour configurer le système d'alarme d'une maison).

Une solution à ce problème consiste à installer un serveur Web sur le système embarqué. L'application serveur devient alors une simple application Web, plus facile et rapide à développer qu'une application serveur dédiée. Voici les principaux avantages de cette solution :

1. On évite la phase d'installation de logiciel chez le client. À l'heure de l'omniprésence de l'Internet, tout client potentiel possède un navigateur Web.
2. Le support applicatif du Web est uniforme et très répandu. Il assure le bon fonctionnement des applications Web en garantissant leur portabilité sur des clients très variés (ordinateur de bureau, PDA, téléphone).
3. Le déploiement d'applications est aisé, par simple consultation de pages Web.
4. La mise à jour de l'application peut se faire facilement après déploiement. Le client n'a jamais à subir de modifications. Une nouvelle version se caractérise par un nouveau contenu distribué par le serveur Web.

1.2 Problèmes soulevés

Les capteurs de terrain ou les cartes à puces sont matériellement très contraints (souvent un processeur cadencé à quelques MHz, quelques kilo-octets de mémoire vive et quelques centaines de kilo-octets de mémoire statique). L'utilisation d'un serveur Web généraliste (*e.g.*, Linux, Apache et J2EE ou Windows, IIS et .net, ...) sur ce type de cible pose cependant plusieurs problèmes, liés à la lourdeur des protocoles IP, TCP et HTTP utilisés pour le Web. Le Web a été initialement conçu pour des serveurs lourds, disposant de beaucoup de ressources, utilisant des threads.

Dans notre cas de figure, le serveur Web embarqué n'a pas besoin de faire face à des milliers de connexions simultanées. C'est un des points qui va rendre envisageable son implémentation. De plus, l'évolution des technologies du Web a permis l'émergence d'un framework applicatif appelé AJAX [1]. Il peut permettre de déporter une partie de la charge du serveur vers le client. Le fonctionnement d'une application Web utilisant AJAX comporte deux phases :

1. La phase de chargement pendant laquelle le client collecte un certain nombre de fichiers. Ils contiennent de la mise en forme (css), du contenu (html) et du code applicatif (javascript).
2. La phase d'exécution, pendant laquelle le code applicatif interagit avec le serveur en lui envoyant des requêtes. Les contenus retournés par le serveur sont souvent de petite taille, et sont interprétés et mis en forme par le client.

La figure 1 synthétise les trafics observés lors des 2 phases pour des applications AJAX disponibles sur Internet (*e.g.*, gmail, calendar, etc.). Elle montre que de nombreux contenus de grande taille sont diffusés (transfert de fichiers), suivis de nombreux petits paquets (contenus dynamiques). Pour diffuser une application AJAX interactive, un serveur Web embarqué se doit donc d'être particulièrement efficace dans ces deux situations. Ce critère constitue un de nos principaux objectifs.

Nous dressons un état de l'art des techniques réseau pour l'embarqué en section 2. En section 3, nous présentons une étude fine des serveurs Web embarqués et de leurs performances. Enfin, en section 4, nous proposons et évaluons (via notre implémentation de référence, *dynaWeb*) de nouvelles techniques pour le service d'applications AJAX hautement interactives.

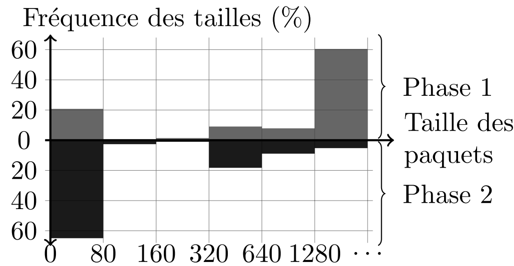

Fig. 1. Répartition des tailles de paquets pour une application AJAX

2 État de l'art

2.1 Adaptations de TCP/IP à l'embarqué

La communication via TCP/IP dans les systèmes embarqués est un problème qui a déjà été largement traité. On propose par exemple d'alléger le coût des retransmissions avec des caches TCP distribués [2]. Des techniques de compression des en-têtes TCP et IP ont également fait leurs preuves [2,3]. Ces méthodes, si elles ont le mérite d'être bien adaptées à leur application, nécessitent des modifications protocolaires. Cela les rend inutilisables au regard des bénéfices identifiés en section 1.1.

2.2 Les piles TCP/IP pour l'embarqué

Le protocole TCP est un protocole relativement lourd, utilisant un automate à 11 états et des mécanismes complexes (*e.g.*, des fenêtres de congestion, de retransmission, des accusés de réception, ...). Ainsi, une pile TCP/IP classique représente un code compilé de plusieurs centaines de kilo-octets et nécessite plusieurs centaines de kilo-octets de mémoire vive [4].

Dunkels propose [5] deux piles TCP/IP spécialement conçues pour les systèmes embarqués très contraints en mémoire, lwIP et µIP.

LwIP est une pile complète mais simplifiée. Elle est capable de supporter plusieurs interfaces réseau et propose un système de configuration flexible. L'implémentation des protocoles est réalisée par couches.

µIP, quant à elle, propose un ensemble de services réduit : une seule interface réseau peut être gérée, et les seuls protocoles intégrés sont IP, ICMP et TCP. Les implémentations des protocoles de µIP sont liées entre elles.

2.3 Serveurs Web embarqués existants

Serveurs Web matériels [6,7] Ils utilisent une architecture matérielle dédiée permettant l'obtention de bonnes performances. Cependant, cette approche limite fortement l'évolutivité et ne permet pas d'exploiter les cibles matérielles déjà produites et présentes en masse sur le marché.

Le serveur mIP est proposé par Shon [4]. La principale contribution de ce travail réside dans la démonstration qu'il est possible de réaliser un serveur Web embarqué en simplifiant un grand nombre de fonctionnalités des protocoles de la pile TCP/IP standard.

Le serveur miniWeb Proposé par Dunkels [8], c'est un serveur Web embarqué monolithique (*i.e.*, incluant sa propre pile TCP/IP dédiée) ayant pour priorité l'économie de mémoire. Il a été conçu par les auteurs de lwIP et µIP. Les fonctionnalités de ce serveur ont été réduites à leur strict minimum, de manière à réaliser le serveur Web le plus léger qui soit. MiniWeb sera détaillé en section 3.2.

3 Étude des serveurs Web embarqués

3.1 Étude transversale des performances de HTTP, TCP et IP

Nous étudions ici le surcoût engendré par l'utilisation de TCP/IP sur une connexion sans latence (un serveur Web embarqué peut être accédé directement en point à point, avec une latence très faible) et half-duplex (comme c'est souvent le cas en embarqué). La couche de liaison n'est pas prise en compte dans cette étude car TCP/IP peuvent être utilisées sur des couches de liaison très variées, engendrant des surcoûts très différents.

Il est important de comprendre que lorsuq'il est utilisé pour HTTP, TCP a un comportement très simple et prédictible. Le client HTTP envoie une requête, puis le serveur retourne une réponse. Lors de ces deux phases, le receveur n'emet rien d'autre que des accusés de réception ne contenant aucune donnée. Dans un contexte général, il est important de comprendre que TCP a un comportement bien moins régulier (données échangées dans les deux sens, accusés contenant des données, etc.).

TCP et IP protocoles utilisent des en-têtes d'une taille totale d'au moins 40 octets. De plus, des accusés de réception sont régulièrement échangés sur le réseau. Soit α la fréquence de transmission d'un accusé par paquet reçu. Le surcoût de TCP/IP en quantité de données transitant sur le réseau est tel que :

$$Surco\hat{u}t_{donn\acute{e}es} = \frac{Taille_{IP} + Taille_{TCP} + \alpha \times Taille_{ACK}}{MSS}$$

La technique des accusés différés est utilisée dans la plupart des piles IP. Elle consiste à n'accuser un segment TCP que lorsque qu'une des conditions suivantes est vraie : (i) On a reçu un second segment ou (ii) 200 ms se sont écoulées.

Soit β la fréquence de déclenchement du délai des accusés différés. On a $0 \leq \beta \leq 1$: avec un seul paquet en vol (*i.e.*, envoyé mais non encore accusé), β vaut 1, tandis qu'en envoi continu de données, β vaut 0. Si les accusés différés sont utilisés, on a toujours $\alpha = 0,5 + \frac{\beta}{2}$. On note $D\acute{e}lai_{ACK}$ le temps d'attente avant l'envoi d'un accusé différé (classiquement, 200 ms). Le débit utile, en fonction de la MSS utilisée est :

$$D\acute{e}bit_{utile} = \frac{MSS}{(MTU + \alpha \times Taille_{ACK})/D\acute{e}bit_{physique} + \beta \times D\acute{e}lai_{ACK}}$$

La figure 2 montre l'évolution du débit utile en fonction de α, β et de la MSS, avec une ligne fonctionnant à 56 Kbps. Les meilleures performances sont obtenues avec $\alpha = 0,5$ et $\beta = 0$, situation ou l'on tolère plus d'un paquet en vol. On est dans ce cas obligé de stocker tous ces paquets pour pouvoir les retransmettre en cas de perte, ce qui conduit à un surcoût en mémoire.

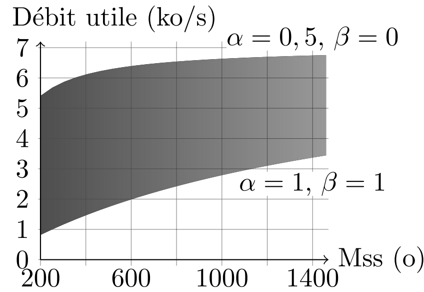

Fig. 2. Débit utile de maximal fonction α, β et de la MSS

3.2 Analyse de serveurs Web embarqués existants

Nous avons réalisé une analyse poussée de deux serveurs Web embarqués complémentaires : le serveur Web de µIP (généraliste et riche en fonctionnalités) et le se serveur miniWeb (extrêmement léger en mémoire). Pour en évaluer les performances et en identifier les biais, nous les avons tous deux portés vers une même cible matérielle dont les caractéristiques sont décrites dans le tableau 1.

Tab. 1. Caractéristiques de notre cible

Microprocesseur	IWRAM	SRAM	EEPROM
ARM7TDMI 32 bits à 16.78 MHz	32 ko	256 ko	8 Mo

Analyse du serveur Web de la pile µIP Il permet le service de contenus statiques comme dynamiques. Les fichiers à servir sont précompilés puis embarqués avec le serveur Web lors de la compilation. Les contenus dynamiques sont générés à partir de balises présentes dans des fichiers au format HTML.

La pile µIP réalise plusieurs sacrifices afin de réaliser des économies de mémoire. Par exemple, dès paquet est envoyé, il est défaussé de la mémoire. Si une retransmission est nécessaire, µIP demande à l'application (*i.e.*, le serveur Web) de lui fournir à nouveau le dernier paquet envoyé. Pour permettre ce mécanisme, le nombre de paquets TCP en vols est limité à un. Cela engendre un déclenchement systématique du délai d'accusés différés chez l'hôte distant (*i.e.*, le client Web), ralentissant considérablement la connexion (voir section 3.1).

Sur la cible que nous utilisons (*c.f.* tableau 1), la taille en mémoire persistante (code et données en lecture seule) après portage de µIP et son serveur Web est de 16.6 ko. Les variables globales utilisent 3.6 ko de mémoire si on positionne la MTU à 1500 octets. Avec une MTU de 240 octets, on peut réduire ce chiffre à 1 ko, mais une faible MTU a un impact très négatif sur les performances (voir section 3.1). L'espace mémoire utilisé en pile d'exécution pour les appels de fonctions et les variables locales représente 204 octets.

Analyse de miniWeb Contrairement à µIP, miniWeb ne permet que le service de pages statiques. Il ne lit pas le contenu des requêtes HTTP, se contenant de servir une page en fonction du port sur lequel il a reçu la demande. De plus, il ne supporte qu'une connexion TCP (non persistante) à la fois.

Les fichiers sont stockés en mémoire sous forme de paquets ordonnés prêts à être envoyés, incluant des en-têtes IP, TCP et HTTP pré-remplis. Les sommes de contrôle sont elles aussi pré-calculées sur l'ensemble des données connues à l'avance. La séquence de communication au niveau des paquets est ainsi préparée hors-ligne, simplifiant considérablement les traitements.

Cette méthode permet un gain non négligeable de ressources mais elle présente un inconvénient majeur : elle impose de fixer à la compilation la taille de la MSS TCP à utiliser. Lors de l'initiation dune connexion TCP, la MSS est choisie comme étant la plus petite MSS supportée par les deux hôtes. Pour faire face à toutes les situations, on est obligé de choisir la plus petite MSS permise par TCP, qui est de 200 octets.

Toutefois, et contrairement à µIP, miniWeb est capable de gérer plusieurs paquets en vol. En cas de perte de données, miniWeb n'a qu'à retransmettre les paquets préfabriqués stockés en mémoire adressable. MiniWeb est capable de traiter puis défausser les paquets qu'il reçoit octet par octet, au fur et à mesure de leur arrivée. Nous avons ainsi pu réaliser des pilotes n'utilisant aucun tampon en réception, ou un tampon de taille indépendante de la MSS utilisée.

MiniWeb, après portage, tient bien son rôle de serveur à consommation mémoire minimale. Il requiert 7.1 ko de mémoire persistante et utilise 88 octets de variables globales et 268 octets utilisés sur la pile (sur la cible décrite en tableau 1). Cependant, ce serveur est trop minimal au regard des usages que nous souhaitons en faire (*c.f.* section 1.2). Son impossibilité de lire le contenu des requêtes HTTP et de servir des contenus dynamiques n'est pas acceptable. De plus, miniWeb n'est pas stable : la réception d'un paquet forgé peut en effet le placer dans un état incohérent. Le déni de service est ainsi très facile à provoquer.

4 Propositions et évaluations

Notre serveur Web idéal doit être efficace lors du service de lourds fichiers statiques et de petits contenus dynamiques. Dans cette section, nous proposons des techniques pour solutionner les biais identifiées de µIP et miniWeb, tout en assurant un maximum de fonctionnalités (*e.g.*, gestion de plusieurs paquets en vol, de plusieurs connexions TCP persistantes, etc.). Nous avons créé un serveur Web monolithique (*dynaWeb*) permettant de tester ces propositions.

4.1 Service de contenus statiques

Proposition L'idée de la précompilation des pages sous forme de paquets prêts à être envoyés, telle qu'elle est pratiquée par miniWeb, permet de réaliser d'importantes économies de ressources. Cependant, cela nuit au trafic, puisque le serveur est obligé d'utiliser une MSS fixe de petite taille.

Nous proposons une approche similaire tout en évitant de figer la MSS. Comme miniWeb, on génère à l'avance les en-têtes HTTP pour chaque fichier à envoyer, mais on ne découpe pas à l'avance les fichiers en paquets. Des en-têtes IP et TCP génériques sont créés hors-ligne, contenant l'ensemble des champs connus à l'avance (taille de fenêtre, protocole utilisé, adresses et port source, checksum partiel).

Les checksum des données du fichier sont calculés, quant à eux, sur des blocs du fichier, d'une taille choisie à l'avance. Lors de l'envoi d'un paquet, dynaWeb n'a plus qu'à sommer les checksum partiels du fichier pour connaître la somme totale. L'espace occupé par les checksum précalculés est de :

$$Taille_{checksums} = 2 \times \left\lceil \frac{Taille_{fichier}}{Taille_{blocs}} \right\rceil$$

La figure 3 montre les gains réalisés de notre méthode. On mesure le temps d'activité du processeur lors de l'envoi d'un contenu statique de 1460 octets (valeur de la MSS utilisée). En augmentant la taille des blocs, on gagne à la fois en temps de calcul et en consommation de mémoire morte.

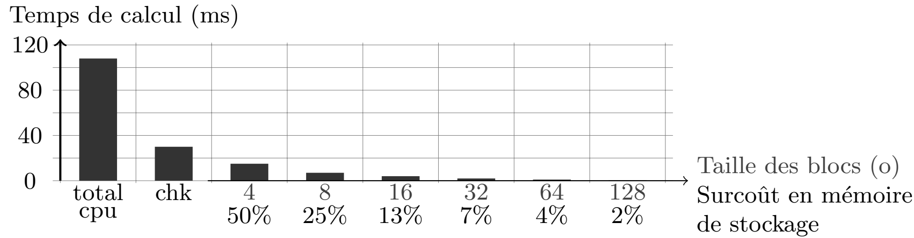

Fig. 3. Impact des checksum précalculés

L'utilisation d'une grande taille de bloc permet de gagner du temps et de l'espace en mémoire. Cependant, la taille maximale des données que l'on peut envoyer doit être multiple de cette taille. Son choix doit dépendre à la fois de l'application visée et des priorités que l'on se fixe. Elle est de :

$$Taille_{envois} = MSS - (MSS \mod Taille_{blocs})$$

Évaluation des performances DynaWeb est capable de gérer plusieurs paquets en vol. De plus, il est capable d'utiliser une MSS non connue à l'avance. La figure 4 montre que ces deux points permettent un meilleur débit maximal utile que µIP et miniWeb.

Nous avons confronté les 3 serveurs Web au service d'un fichier statique volumineux (55.9 ko) sur une ligne à 56 kb/s. Nous avons fait varier la MTU de 300 à 1500 octets (donc la MSS de 260 à 1460) pour observer le comportement des serveurs dans différentes situations. La figure 5 montre le temps nécessaire à la transmission de notre fichier selon le serveur et la MTU utilisée.

Avec une petite MSS, miniWeb obtient des performances similaires à dynaWeb. C'est en s'adaptant à la MSS que dynaWeb obtient les meilleures performances pour une MSS plus élevée.

Avec une petite MSS, µIP est très loin derrière les deux autres serveurs. Cet écart de performances s'explique par la limitation de µIP à un paquet en vol, déclenchant le délai des accusés différés chez le client. Avec une grande MSS, c'est l'implémentation monolithique de dynaWeb qui lui permet de rester plus rapide que µIP. Il est important de rappeler que µIP, contrairement à dynaWeb, occupe d'autant plus de mémoire que la MSS utilisée est grande.

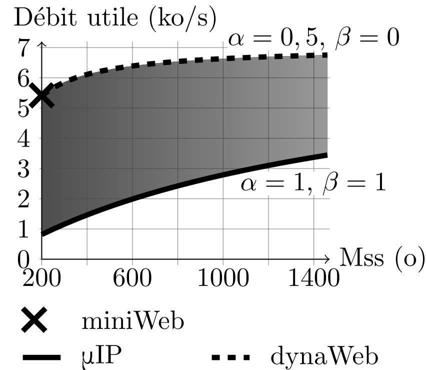

Fig. 4. Débit utile de maximal fonction de la MSS

4.2 Service de contenus dynamiques

Solution choisie pour dynaWeb Le point clé des applications AJAX est la phase de service des contenus dynamiques. Comme nous l'avons mentionné en section 1.2, une de nos priorités est l'optimisation du service de petits contenus générés.

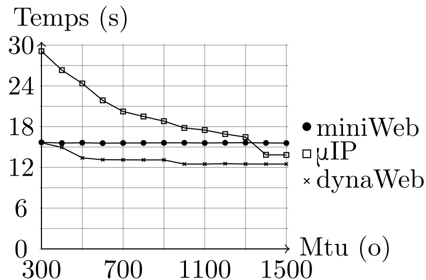

Fig. 5. Service d'un fichier de 55.9 ko à 56 kb/s

Chaque « service » est en fait chargé d'écrire les données à envoyer dans un tampon. L'en-tête HTTP (ainsi que sa somme de contôle TCP) est précalculé puis ajouté automatiquement par dynaWeb. Contrairement à µIP et miniWeb, dynaWeb est capable de gérer des connexions persistantes, améliorant considérablement le temps de réponse lors du service de petits contenus.

Pour l'instant, notre implémentation ne permet pas le service d'un contenu dynamique d'une taille supérieure à la MSS courante. Ce cas de figure soulève en fait un problème : pendant la génération, si la taille du contenu dépasse la MSS, on doit se bloquer pour réaliser l'envoi d'un paquet et continuer à gérer TCP. N'étant pas au dessus d'un système d'exploitation, nous ne disposons pas de primitive de blocage telle que des threads pourraient nous fournir. La solution choisie par µIP consiste à utiliser les protothreads [9], mais cette méthode est trop restrictive (impossibilité d'utiliser des variables locales, ou de *switch-case*, ...). La recherche d'une meilleure solution fera partie de nos travaux futurs.

Évaluation des performances Nous avons mesuré les performances de µIP et dynaWeb sur une application AJAX. MiniWeb n'apparaît pas dans ces tests, car il est incapable de servir des contenus dynamiques. L'application consiste en une page Web envoyant périodiquement une requête asynchrone au serveur installé sur une console portable aux caractéristiques décrites dans le tableau 1.

Le contenu alors généré indique l'état des boutons de la console. La page affiche ainsi, en temps réel, l'ensemble des boutons enfoncés de la console portable. Un fichier HTML de 1.9 ko est utilisé ainsi qu'un fichier *JavaScript* de 1.7 ko. L'état des boutons est servi périodiquement sur 2 octets. Les résultats mesurés pour les deux serveurs sont présentés dans le tableau 2.

Tab. 2. Performances mesurées sur l'application AJAX (MTU : 1500 octets)

Serveur utilisé	Temps de chargement initial (ms)	Temps de réponse (ms)	Aller-retour GET (ms)
µIP	1251	279	199
dynaWeb	1034	112	112
accélération	20%	149%	77%

Le chargement initial des fichiers statiques est légèrement en faveur de dynaWeb. Le service du contenu dynamique est quant à lui beaucoup plus rapide pour dynaWeb que pour µIP. Il y a deux raisons à cela : (i) dynaWeb utilise une connexion persistante (HTTP 1.1), évitant de nombreux échanges TCP pour chaque donnée à envoyer (ii) le service pur de contenu (aller-retour GET sans compter les poignées de mains TCP) est accéléré par le calcul à l'avance des en-têtes, des checksum, et par l'implémentation monolithique. Ainsi, dynaWeb supporte 8 échantillonnages par seconde, contre 3 pour le serveur de µIP.

4.3 Point sur dynaWeb

Comme µIP et miniWeb, dynaWeb vise en priorité l'économie de mémoire. Le tableau 3 donne l'occupation en mémoire de ces 3 serveurs. Ces mesures sont les bornes minimales d'occupation (*i.e.*, tampons de µIP réduis à leur plus petite taille, utilisation de miniWeb et dynaWeb sans tampons en réception).

Tab. 3. Comparatif des consommations mémoire des serveurs étudiés

Serveur Web	Mémoire volatile		Mémoire persistante
	Globales	Pile	
µIP	1 ko	204 o	16.6 ko
miniWeb	88 o	268 o	7.1 ko
dynaWeb	152 o	92 o	9.2 ko

Ces mesures montrent que µIP avec son serveur Web est de loin la solution la plus lourde en mémoire. De plus sa consommation augmente avec la MSS utilisée. Ce point montre qu'une forte intégration et une implémentation monolithique des protocoles permet une économie de mémoire considérable. MiniWeb est le serveur le plus léger en mémoire, mais sacrifie un nombre de fonctionnalités beaucoup trop important pour les applications que l'on vise.

DynaWeb offre une consommation mémoire proche de celle miniWeb mais est plus souple, efficace, et fournit beaucoup plus de services que ce dernier. Les fonctionnalités requises par nos objectifs initiaux (voir section 1.2) sont presque toutes fournies par dynaWeb. Notre serveur reste pour l'instant incapable de

servir des contenus dynamiques de taille non bornée, ce qui constitue une de nos principales perspectives.

Le serveur de la pile µIP illustre bien les surcoûts liés à l'approche par couches et à l'utilisation d'une pile IP générique. La plupart des limitations de ce serveur (nombre de paquets en vol, utilisation de HTTP 1.0, etc.) sont des concessions qui ont été réalisées pour limiter la consommation de mémoire. Malgré ces limitations, la consommation en ressources de ce serveur est largement supérieure à celle de dynaWeb.

5 Conclusion et perspectives

Nous avons montré que la forte intégration des protocoles permet de réaliser de nombreuses optimisations, que ce soit en consommation mémoire ou en temps de calcul. De plus, une telle implémentation permet une approche transversale. Cette approche permet, lors de la gestion de la pile TCP/IP, de mettre en œuvre des solutions aux principaux biais qui ont été identifiés et caractérisés.

Nous souhaitons étendre ces travaux pour traiter le cas des réseaux sans fil, pour intégrer plus de protocoles, et pour permettre au serveur d'établir des connexions (éventuellement sécurisées) vers l'extérieur. Il serait alors possible pour les applications AJAX embarquées d'utiliser des techniques de *mashup* (fusion d'information en provenance de sources différentes). Par ailleurs, comme nous l'avons mentionné en section 4.2, notre approche sur le service de contenus dynamiques reste à étendre. Nous envisageons aussi de travailler sur l'élaboration d'une « API idéale » pour servlets embarqués.

Références

1. Garrett, J.J. : Ajax : A new approach to web applications (2005)
2. Dunkels, A. , *et al.* : Making tcp/ip viable for wireless sensor networks. In : EWSN 2004, Berlin, Germany (January 2004)
3. Sridharan, R. *et al.* : Poster : A robust header compression technique for wireless ad hoc networks. SIGMOBILE Mob. Comput. Commun. Rev. **7**(3) (2003) 23–24
4. Shon, S. : Protocol implementations for web based control systems. International Journal of Control, Automation, and Systems **3** (March 2005) 122–129
5. Dunkels, A. : Full tcp/ip for 8-bit architectures. In : MobiSys '03, New York, NY, USA, ACM Press (2003) 85–98
6. Han, G.j. *et al.* : Webit : a minimum and efficient internet server for non-pc devices. In : GLOBECOM '03. IEEE. Volume 5. (2003) 2928–2931 vol.5
7. Riihijarvi, J. *et al.* : Providing network connectivity for small appliances : a functionally minimized embedded web server. Communications Magazine, IEEE **39**(10) (Oct. 2001) 74–79
8. Dunkels, A. : The proof-of-concept miniweb tcp/ip stack (2005) `http://www.sics.se/~adam/miniweb/`.
9. Dunkels, A. *et al.* : Protothreads : simplifying event-driven programming of memory-constrained embedded systems. In : SenSys '06, New York, NY, USA, ACM Press (2006) 29–42