


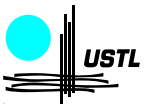

Mémoire de Master Recherche
en Informatique

Présenté par
Simon Duquennoy
le 13 juin 2007

Haute performance pour serveurs Web embarqués

Sous la direction de :
Gilles Grimaud Université de Lille 1
Jean-Jacques Vandewalle Gemalto Labs

Université des Sciences et Technologies de Lille
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Résumé

De nos jours, les systèmes embarqués sont de plus en plus présents autour de nous. Ils ont un besoin grandissant d'accessibilité.

En embarquant des serveurs Web dans ces matériels, on permet à quiconque de se connecter facilement à un routeur, une carte à puce ou un capteur. La connexion se fait à partir d'un ordinateur utilisant un simple navigateur Web. Le principal avantage d'une telle solution est qu'elle permet d'éviter l'installation de logiciels dédiés du côté des machines clientes pré-identifiées. Cependant, tout informaticien sait qu'il est très difficile d'embarquer un serveur Web généraliste dans quelques centaines d'octets de mémoire.

Dans ce mémoire, on montre qu'au lieu de l'approche classique par couches de communication, l'utilisation d'une « micro pile IP » avec une approche transversale est une bonne solution. Notre implémentation, dynaWeb, est plus performante que les autres propositions, sans sacrifier aucune fonctionnalité, et peut être utilisée dans des systèmes embarqués très contraints en mémoire.

Mots clés : systèmes embarqués, serveur Web, micro-IP, approche transversale

Abstract

Nowadays, embedded systems are more and more present around us. They have an increasing accessibility need.

Embedding a web servers in those systems allows anyone to easily connect himself to a router, a smart card or a sensor. The connection can be done from any computer, using only a Web navigator. The main interest of this solution is to avoid dedicated software installation on the pre-identified client computers. However, any computer scientist knows that embedding a general purpose Web Server in a few hundreds bytes of memory is very hard.

In this master's thesis, we show that the use of a "micro-IP stack" with a cross-layer approach is a good solution, instead of the classical communication layers design approach. Our implementation, dynaWeb, is more efficient than other proposals without sacrificing any functionality, and can work in a very memory-constrained embedded system.

Keywords: embedded systems, Web server, micro-IP, cross-layer approach

Remerciements

Je tiens à remercier en premier lieu Gilles Grimaud *et Jean-Jacques* Vandewalle *pour leur encadrement sans faille ainsi que pour les nombreuses discussions que nous avons pu avoir.*

Je remercie David Simplot-Ryl *pour m'avoir accueilli au sein de l'équipe POPS.*

Mes remerciements s'adressent à l'ensemble de l'équipe POPS, pour leur accueil chaleureux. Merci à Fadila Khadar *et Antoine* Gallais*, avec qui j'ai partagé le bureau 101, pour leur bonne humeur quotidienne. Je remercie Vincent* Bénony *pour ses précieux conseils concernant la programmation des pilotes ainsi que pour son fer à souder salvateur.*

Cette année de Master Recherche fut éprouvante. Aussi, je remercie, par ordre alphabétique, Adeline, Amaury, Cédric, Jean-Philippe, Loïc, Thomas et Yoann pour les nombreux repas et pauses café effectués, toujours dans la bonne humeur. Nous avons toujours su concilier débats acharnés et discussions autour de la tâche aveugle chez le calamar.

Table des matières

Introduction

La présence de systèmes informatiques embarqués dans notre quotidien est devenue, au fil des années, de plus en plus importante et transparente. 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 leur utilisateur 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. 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.

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 spécifique. Du coté du client, seul un navigateur Web standard est requis, ce qui, à l'heure de l'omniprésence de l'Internet, ne pose bien sûr plus aucun problème.

Il est évident qu'un serveur Web généraliste (solution du type Linux, Apache et J2EE ou Windows, IIS et .net, ...) est demandeur de trop de ressources pour être embarqué sans problème sur une cible ne disposant que de quelques centaines d'octets de mémoire. Dans ce mémoire, nous nous intéressons aux problématiques liées à la conception d'un serveur Web embarqué performant et d'applications Web hautement interactives.

Nous commençons par décrire précisément notre problématique dans le chapitre 1. Un état de l'art des technologies du Web, des solutions pour le réseau embarqué et des serveurs Web embarqués est présenté dans le chapitre 2. Le chapitre 3 réalise une analyse poussée des comportements des protocoles du Web et de deux serveurs Web existants auxquels nous nous comparerons ensuite. Enfin, le chapitre 4 propose des solutions aux biais que nous avons ainsi identifiés. Notre prototype, dynaWeb, y est décrit et comparé aux solutions existantes.

Chapitre 1

Problématique

1.1 Motivation initiale

Le besoin d'interaction des systèmes embarqués est grandissant. On parle, pour décrire les communications de ces derniers avec l'humain ou avec d'autres systèmes, de « l'Internet des choses ». Ce terme n'est encore lié à aucune technologie et n'est pas encore précisément défini.

Les exemples de systèmes susceptibles d'utiliser l'Internet des choses sont nombreux. Une carte SIM d'un téléphone cellulaire contient souvent un agenda aux fonctionnalités assez réduites. Il serait intéressant de pouvoir interagir efficacement avec cet agenda et de permettre au téléphone de se connecter à Internet pour tisser les informations locales à celles stockées sur un serveur distant. L'application locale peut ainsi être enrichie par les informations distantes, tandis que l'application distante peut être personnalisée par les informations locales. On peut aussi avoir besoin de consulter ou de configurer facilement un simple routeur, ou de consulter des données récoltées par un capteur de terrain.

L'idée d'utiliser les technologies du Web pour l'Internet des choses est séduisante. Cette solution est étudiée dans ce mémoire. Elle nécessite la conception d'un serveur Web offrant de bonnes performances en système très contraint. Voici les avantages de cette solution sur l'utilisation de logiciels dédiés aux applications :

- On évite la phase d'installation de logiciel du coté du client. Aujourd'hui, les ordinateurs disposent tous d'un navigateur Web ; ils peuvent se connecter à un serveur Web sans être préalablement identifiés.
- La plateforme de développement de l'application à embarquer se résume aux technologies du Web. Elle est indépendante du matériel sur lequel l'application s'exécute, et est déjà bien connue des développeurs Web. En cas de mise à jour de l'application, rien n'est à changer du coté du client, qui dispose toujours d'un simple navigateur.
- On permet aussi la communication entre objets. Il est envisageable qu'un objets initie une connexion vers un autre pour récupérer ou communiquer de l'information. Le Web permet même, via SSL, d'échanger des données de manière sécurisée.
- La mise à jour de l'application ou des contenus à livrer peut se faire facilement après déploiement, en utilisant simplement l'interface Web fournie.

1.2 Problèmes soulevés

Les protocoles du Web (voir section 2.1) ont été développés sur mesure lors des débuts d'Internet. Dans ce contexte, les serveurs Web (initialement simples serveurs de contenus statiques) sont souvent de puissantes machines disposant d'une grande quantité de mémoire de travail et de stockage. La principale difficulté de l'utilisation d'un serveur Web dans un système embarqué réside dans le fait que, fondamentalement, les protocoles du Web ainsi que le travail réalisé par un serveur n'ont pas été prévus pour l'embarqué, en raison de la lourdeur des échanges réalisés et des grandes quantités de données que l'on est amené à gérer.

Un serveur Web généraliste et puissant utilise de manière transparente de nombreuses couches d'abstraction logicielles. Un système de fichiers est utilisé, souvent complété d'un système de gestion de base de données. Le serveur HTTP est dissocié du processus chargé d'exécuter les applications (cgi, servlets). Enfin, un système d'exploitation muni d'une pile IP généraliste est utilisé.

Un premier raisonnement consiste à se dire que puisque les cibles ne sont pas assez puissantes pour accueillir des serveurs Web, dans quelques années, la loi de Moore changera la donne. Plutôt que d'absorber ses progrès en installant des applications toujours plus lourdes sur nos matériels, nous préférons l'utiliser pour réduire les coûts de production des matériels (en réduisant leur encombrement). C'est pourquoi nous cherchons à garder sous contrôle la consommation en ressources des serveurs Web embarqués. Voici les principales difficultés liées à l'utilisation d'un serveur Web embarqué :

Contraintes protocolaires

L'utilisation du Web standard – permettant une interopérabilité avec tout client Web – nous apporte un grand nombre de contraintes, et empêche toute optimisation au niveau des protocoles utilisés. C'est dans la manière d'utiliser ces protocoles, conformément à leurs RFC, que l'on pourra agir. Il devient alors impossible d'utiliser les nombreux travaux [6, 16] qui ont été réalisés pour développer des protocoles réseau adaptés aux systèmes embarqués.

Non contrôle du client

La raison principale du choix de l'utilisation du Web est le fait que l'on n'ait pas besoin de développer, d'installer et de mettre à jour une application sur la machine cliente. Cela implique pour nous une forte contrainte, car les optimisations apportées au serveur devront s'adapter aux différents clients existants ainsi qu'aux systèmes d'exploitation sur lesquels ils sont exécutés. Ces clients sont bien entendu optimisés pour une utilisation classique du Web (*i.e.*, à des accès à des serveurs Web classiques (puissants et performants) sur Internet).

Découpe en couches logicielles

Dans une configuration classique (non embarquée), la pile IP est prise en charge par le système d'exploitation. Elle n'est pas optimisée pour un usage en particulier. L'application (dans notre cas, un serveur Web) qui utilise ce système d'exploitation est contraint d'utiliser la pile qui lui est fournie. Dans le cadre de l'embarqué, il n'y a plus de frontière entre l'application et le noyau ; il est alors possible d'intégrer un serveur Web à ce dernier.

Fortes contraintes matérielles

La ressource la plus limitante dans notre contexte est la quantité de mémoire de travail disponible. Il va donc s'agir de minimiser le nombre et la taille des tampons utilisés pour l'implantation des protocoles ainsi que des pilotes. Une carte à puce dispose en général de 8 à 32 ko de mémoire vive et de 500 ko à 1 Mo de mémoire non volatile. Un capteur de terrain peut ne disposer que de 4 ko de mémoire vive, 4 ko d'EEPROM et 128 ko de mémoire flash adressable. La gestion des entrées-sorties est elle aussi souvent contrainte, utilisant des débits faibles. Enfin, la puissance de calcul dans les systèmes embarqués est souvent très limitée ; ils utilisent des microcontrôleurs cadencés à quelques mega-Hertz.

1.3 Utilisation d'AJAX pour l'embarqué

1.3.1 Présentation du Web 2.0

Lors des débuts du Web, les serveurs accessibles sur Internet servaient des sites composés de pages statiques. Ces pages étaient mises à jour de manière plus ou moins fréquente. La forte généralisation de l'utilisation d'Internet a provoqué le développement de nouvelles techniques de diffusion d'information. Ainsi, le « Web 1.5 »[1] fait son apparition, permettant la génération dynamique des pages distribuées sur Internet à partir de bases de données stockées du coté du serveur. Le Web 1.5 permet ainsi au concepteur du site de se concentrer sur les contenus qu'il souhaite distribuer et non sur leur forme. Cependant, ils restent proposés de manière statique par le webmestre.

Depuis quelques années, le Web 2.0 a fait son apparition. Son principal objectif est de permettre aux internautes, outre le fait d'être consommateurs de contenus, d'en être également producteurs. Avec cette nouvelle façon d'utiliser le Web sont nés de nouveaux concepts d'ergonomie, visant par exemple à réduire le nombre de clics des utilisateurs, à dissocier au mieux la mise en forme et les contenus ou à augmenter l'interactivité des contenus servis.

1.3.2 Particularités de la technique AJAX pour l'embarqué

Technologiquement, le Web 2.0 est couramment associé à la technique AJAX (décrite en section 2.1.3), permettant de réduire grandement la bande passante utilisée en distribuant plus de contenus et en factorisant les informations de mise en forme. Avec une telle technique, un certain nombre de traitements sont déportés depuis le serveur vers le navigateur Web. Ce dernier, par exécution de code javacript, est amené à envoyer des requêtes HTTP au serveur de manière asynchrone[2]. Dans notre situation où le serveur est fortement contraint, ce gain de bande passante et ce déplacement de « l'intelligence » du côté du client sont des facteurs clés, qui vont nous permettre d'optimiser le rapport entre la richesse de l'application et sa consommation (de calcul, de mémoire, et de bande passante).

Le traffic engendré par une application de type AJAX présente quelques particularités. La phase de chargement de l'application, pendant laquelle le client récupère les informations de mise en forme ainsi que du code (JavaScript) à exécuter, n'est à effectuer que lors de la connexion. Elle est caractérisée par de nombreux envois de données statiques de grande

[1] Le terme « Web 1.5 » n'a en fait été utilisé que rétrospectivement, après la généralisation du Web 2.0.
[2] C'est la création de l'objet JavaScript *XmlHttpRequest* qui a permis l'utilisation de la technique AJAX.

taille. Ensuite, le code exécuté par le client provoque de nouvelles requêtes pour récupérer des données non mises en forme. Ces données sont petites dans la plupart des cas.

La figure 1.1 permet de caractériser un traffic classique d'application AJAX. On y montre la répartition des paquets contenant une réponse HTTP en fonction de leur taille. On observe que pendant la phase d'initialisation (phase 1), les paquets volumineux sont plus fréquents. Pendant la seconde phase, la plupart des contenus échangés sont générés dynamiquement par le serveur. Ils sont le plus souvent de petite taille. Ces mesures ont été réalisées sur des applications AJAX disponibles sur internet (boite mail, calendrier en ligne, ...).

Si on souhaite utiliser une application de type AJAX sur un serveur Web embarqué, il doit être efficace dans ces deux domaines :
- service de contenus statiques de grande taille ;
- service de petits contenus générés dynamiquement.

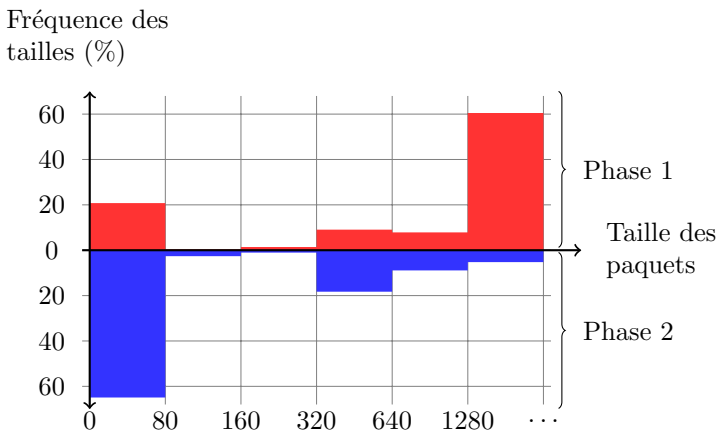

FIG. 1.1 – Répartition des tailles de paquets pour une application AJAX

1.4 Démarche

Les systèmes embarqués que nous ciblons peuvent être munis d'une interface réseau filaire (*e.g.*, routeur, carte à puce...) ou sans fil (*e.g.*, routeur, capteur de terrain...). Ces deux situations ont des contraintes très différentes, c'est pourquoi notre étude se concentrera sur le cas d'une connexion filaire, utilisant par exemple une connexion USB ou série.

Une analyse précise du Web et de ses traffics est à réaliser, permettant l'identification des difficultés liées à l'utilisation du Web en embarqué. Une étude des solutions existantes nous permettra ensuite d'établir une base de travail. Á partir de ces deux études, nous souhaitons proposer des solutions adaptées à l'exécution d'applications de type AJAX sur un système embarqué. Ces solutions doivent garder ces objectifs principaux :
- L'économie en ressources ;
- La possibilité de servir des contenus statiques comme dynamiques ;
- Une efficacité particulière sur les applications de type AJAX.

Chapitre 2

État de l'art

Ce chapitre, après une description des technologies du Web, dresse un état de l'art des réseaux pour les systèmes embarqués puis, à l'intersection de ces deux domaines, des serveurs Web embarqués.

2.1 Les protocoles du Web

Le Web est un terme général regroupant de nombreux protocoles et technologies. La figure 2.1 présente les protocoles utilisés par le Web en les plaçant dans le modèle standard OSI. Les protocoles de communication liés au Web se situent à partir de la couche 3 jusqu'à la couche 7 de ce modèle.

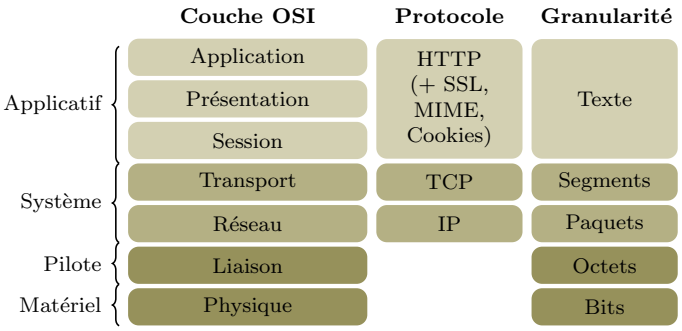

FIG. 2.1 – Les protocoles du Web dans le modèle OSI

2.1.1 Présentation des protocoles

Principe de base de IP

IP [11] est un protocole de couche réseau développé pour Internet. Les entités sont identifiées par une adresse IP unique sur le réseau, grâce à laquelle les paquets sont routés. IP ne permet pas de s'assurer du succès de l'envoi des paquets, ni de leur ordre d'arrivée. L'en-tête d'un paquet IP est (hors options) de 20 octets.

Principe de base de TCP

TCP [12], protocole de couche transport, est utilisé pour le Web comme moyen de communication fiable et sans pertes. Il fonctionne en mode connecté sur le protocole IP, et utilise un en-tête de 20 octets (hors options). En numérotant les segments envoyés et en proposant un mécanisme d'accusés de réception et de retransmissions, TCP assure une connexion fiable et sans perte entre deux hôtes. Ainsi, chaque paquet envoyé peut accuser la réception d'autres paquets. La notion de *port* permet d'identifier une connexion, et ainsi d'établir plus d'une connexion entre chaque paire de machines. La quantité de données contenues dans un segment est limitée par la MSS (Maximum Segment Size), négociée lors de l'initiation de la connexion.

Ce protocole est défini par une RFC, spécifiant un grand nombre de points, et laissant une part de liberté sur de nombreux autres. La RFC 2001 [17] propose des politiques de gestion des congestions, permettant d'adapter le débit sortant aux limites de débit du réseau ou aux congestions qui y ont lieu. L'algorithme de Karn [10] est un exemple largement utilisé de politique de gestion des retransmissions. En affinant l'estimation du temps d'aller-retour entre les deux hôtes, on peut ainsi mieux estimer le moment où retransmettre un paquet ou non.

Principe de base de HTTP

Le protocole HTTP [1, 7], utilisé au dessus de TCP, permet à un client Web de communiquer avec un serveur Web, c'est à dire de demander des pages ou d'envoyer des données. C'est un protocole sans état utilisant un simple système de requête / réponse. Toutes les informations protocolaires contenues dans les requêtes HTTP sont sous forme de texte.

2.1.2 Présentation transversale des traffics Web

Pour mieux comprendre le comportement des différents protocoles du Web, il faut en réaliser une analyse transversale. Nous présentons ici les différentes phases d'un échange Web classique, utilisant HTTP, TCP et IP, sur une couche de liaison ethernet. Les schémas explicatifs utilisés par la suite décrivent, pour chaque protocole, la taille des en-têtes utilisés. Les drapeaux de TCP sont également mentionnés.

Initiation de la connexion

Avant de pouvoir échanger des données via HTTP, une connexion TCP doit être ouverte. Cette phase est initiée par le client. Elle est illustrée en figure 2.2. On constate qu'elle consiste en une *poignée de main* en trois temps. Trois paquets IP d'une soixantaine d'octets y sont transmis. Durant cet échange, et grâce aux en-têtes TCP et IP, certaines options peuvent être négociées. Avec une couche de liaison de type ethernet, dont la MTU (Maximum Transfert Unit) est de 1500 octets, la MSS de TCP sera de 1460 octets. En règle générale, on a $MSS = MTU - IPH - TCPH$, où IPH est la taille des en-têtes IP et TCPH celle des en-têtes TCP.

Échange de données utilisant HTTP

Une fois la connexion établie, les requêtes HTTP peuvent être transmises du client vers le serveur. La figure 2.3 permet de mieux comprendre les échanges de paquets qui ont alors

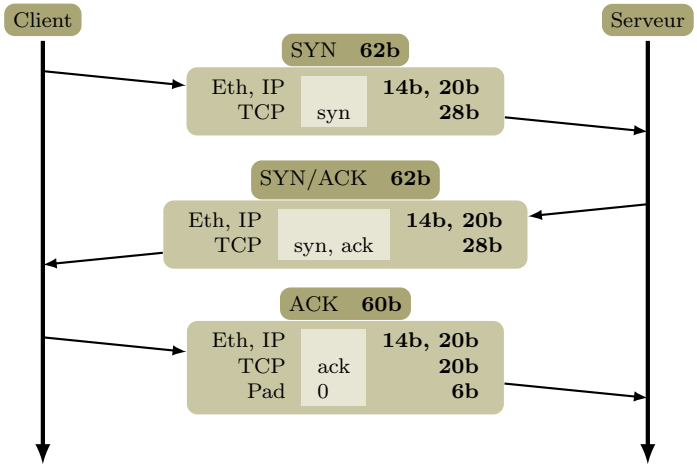

Fig. 2.2 – Initiation de la connexion

lieu, dans le cas de requêtes HTTP de type GET (on utilise ici une liaison ethernet). Le client initie alors un échange par l'envoi d'une requête contenant l'adresse de la page qu'il souhaite se voir acheminer, et dont la taille, souvent de plus de 500 octets, varie en fonction du client Web utilisé (un navigateur classique envoie de lourdes requêtes, tandis que la commande Unix `wget` se contente de moins de 200 octets).

Le serveur répond alors en envoyant un en-tête HTTP suivi du contenu demandé par le client. La taille de cette réponse est bien sûr très variable. Elle est composée d'un ou plusieurs paquets IP utilisés pour transporter les données TCP depuis le client vers le serveur.

On constate que tous les paquets qui transitent sur le réseau accusent la réception du dernier paquet reçu (drapeau *ack*). Des paquets ne contenant aucune donnée utile transitent également dans le seul but d'accuser la réception d'autres paquets.

Fermeture de la connexion

La connexion TCP peut être fermée par le client ou par le serveur, dépendamment de la persistance des connexions appliquée au niveau de HTTP (voir section 2.1.2). La figure 2.4 décrit cette phase. Celui qui initie la fermeture envoie un paquet contenant le drapeau *fin*, qui sera ensuite accusé. L'autre hôte continue éventuellement de transférer des données, puis, lorsqu'il décide d'effectivement rompre la connexion TCP, envoie à son tour un paquet muni d'un drapeau *fin*. Un dernier accusé est alors transmis, puis la connexion est fermée. Quatre paquets IP circulent pour cette poignée de mains.

Comparaison de HTTP 1.0 et HTTP 1.1

Dans la norme HTTP 1.0, à chaque fois qu'un client souhaite envoyer une requête au serveur, il doit établir une nouvelle connexion TCP avant de l'envoyer. Le serveur demandera la fermeture de cette connexion dès qu'il l'aura traitée. En plus de la requête et de la réponse, trois paquets IP circulent alors sur le réseau pour ouvrir la connexion, et quatre pour la fermer.

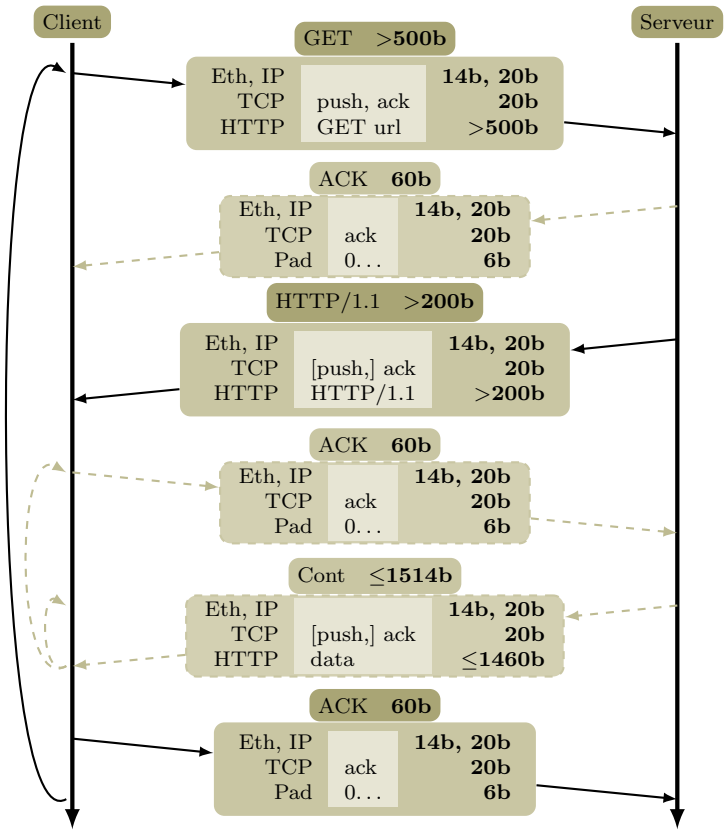

Fig. 2.3 – Suite de requêtes HTTP (liaison ethernet)

Un mise à jour de la norme, HTTP 1.1, permet de réduire le surcoût considérable engendré par ce multiples ouvertures et fermetures de connexion. Lorsque le client le demande et que le serveur le permet (option *keep-alive* dans l'en-tête HTTP), une seule connexion TCP peut être réutilisée pour de nombreuses requêtes. La plupart de clients Web ouvrent, si nécessaire, deux connexions auprès du serveur pour envoyer parallèlement plusieurs requêtes. L'option *keep-alive* n'est utilisable que lorsque le serveur est capable de fournir la taille de données qu'il envoie (champ *content-length*). Dans le cas contraire, la fin de transmission est indiquée par fermeture de la connexion TCP.

D'une manière générale, garder une connexion TCP pour plusieurs requêtes HTTP n'est possible qu'en HTTP 1.1 et lorsque le serveur peut indiquer la taille des données qu'il envoie.

2.1.3 Fonctionnement d'AJAX

Depuis quelques années, le Web 2.0 a généralisé l'utilisation de la technique *AJAX* (Asynchronous JAvascript and XML). La figure 2.5 permet de comprendre la principe d'AJAX.

Voici le descriptif des deux phases mentionnées sur le diagramme :

Phase 1 Étape de chargement initial, non spécifique à la technique AJAX. Dans un premier temps, le client demande le code HTML d'une page Web. Une fois ce code reçu, il

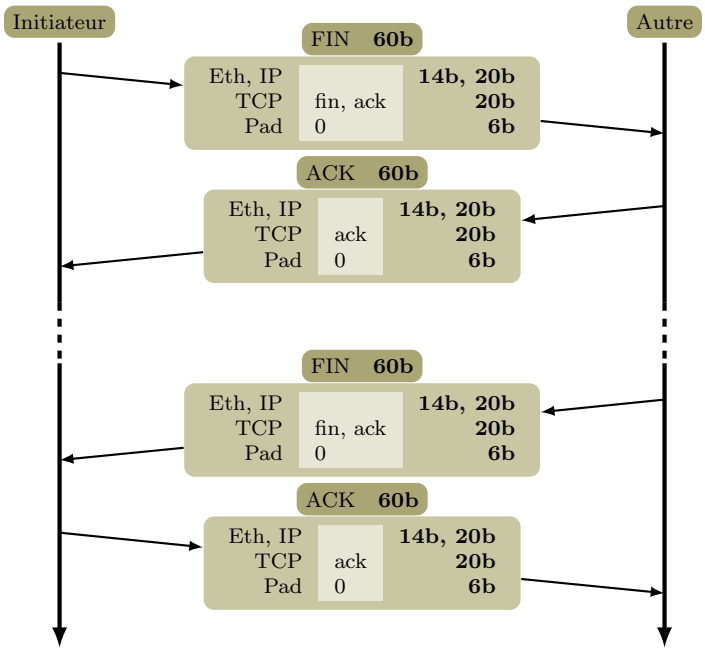

FIG. 2.4 – Fermeture de la connexion

en extrait les liens vers d'autres fichiers requis, comme des feuilles de style *CSS*, des images, ou des codes JavaScript. Les CSS permettent de factoriser les informations de mise en forme, tandis que les scripts contiennent du code qui doit être exécuté par le navigateur Web.

Phase 2 C'est ici qu'AJAX prend tout son intérêt. Le code JavaScript chargé précédemment demande au navigateur, de manière asynchrone (et déclenché de manière transparente lors d'un évènement), d'envoyer de nouvelles requêtes HTTP au serveur Web. Ces requêtes ont pour objectif de demander au serveur des contenus non mis en forme, donc très légers. Classiquement, les réponses à ces requêtes sont au format XML ; elles peuvent néanmoins prendre toute autre forme. Le code JavaScript se charge alors de placer dynamiquement dans la page les contenus reçus, après les avoir traités si nécessaire. Ce mécanisme est rendu possible par l'utilisation de l'objet JavaScript *XmlHttpRequest*.

L'intérêt de la technique AJAX est de permettre d'échanger des informations de manière très fréquente, augmentant ainsi l'interactivité, tout en ayant des débits les plus faibles possibles. Un site Web réalisé avec cette technique peut alors ressembler à une réelle application accessible en ligne via un navigateur Web. On sort ainsi de l'ancestrale vision où le client Web est très léger et où seul le serveur est actif ; ici, le client est plus actif que le serveur (pour une même session).

2.2 Les solutions réseau pour l'embarqué

Les systèmes embarqués ont un grand besoin de communications réseau. C'est pourquoi de nombreuses propositions ont été réalisées pour adapter ces communications aux particularités

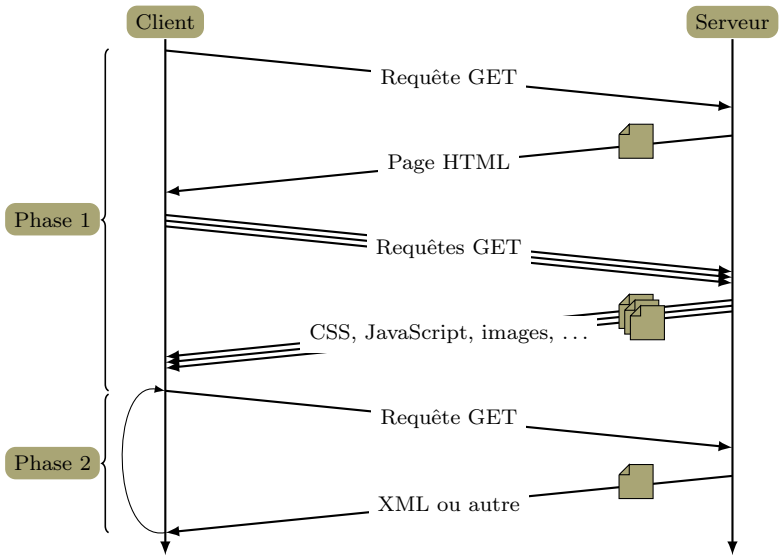

FIG. 2.5 – Fonctionnement d'AJAX

et aux contraintes de ces systèmes : contraintes de performance, interfaces réseau sans fil, besoin d'autonomie, etc.

2.2.1 Protocoles réseau taillés sur mesure

Dans le cas d'une connexion entre deux hôtes situés à plus d'un saut dans un réseau sans fil, les retransmissions TCP sont particulièrement coûteuses. En effet, à chaque saut, pendant la transmission du paquet, chaque nœud empiète sur la bande passante de ses voisins. Dans [6], une technique de cache distribué des segments TCP est proposée, permettant à chaque nœud du réseau de conserver en cache les derniers segments qu'il a reçu, et ainsi d'éviter des retransmissions de bout en bout.

Une solution classique pour l'utilisation de TCP/IP en milieu contraint est d'utiliser des méthodes de compression des en-têtes [6, 16]. En se basant sur les redondances entres les en-têtes des paquets successifs d'une même connexion, on peut réduire considérablement la taille des en-têtes, et ainsi augmenter le taux de données utiles dans chaque paquet. Les données passent pour cela dans un compresseur avant d'être envoyées sur le réseau, et, à leur arrivée, passent dans un décompresseur avant d'être traitées.

Ces méthodes, si elles ont le mérite d'être bien adaptées à l'application qu'elles visent, ont l'inconvénient, pour la plupart, de modifier les protocoles utilisés. Une des priorités de notre problématique est de s'adapter aux protocoles existants (pour permettre une connexion depuis un simple navigateur Web ; voir section 1.2), donc de ne pas les tailler aux besoins de notre application. Ce type d'optimisation de protocoles ne répond donc pas à nos besoins.

2.2.2 Adaptation de TCP aux systèmes contraints

Une autre manière de s'adapter aux contraintes de l'embarqué est d'adapter les politiques des protocoles aux contraintes que l'on se fixe. Par exemple, dans le cas de l'utilisation

de TCP/IP avec une interface réseau sans fil, les politiques standard vues en section 2.1.1 ne sont plus adaptées. En effet, sur une interface ethernet, le fait de perdre un paquet indique une congestion du réseau. Les politiques classiques engendrent alors une diminution du rythme d'émission des paquets. Avec une interface sans fil, on peut perdre un paquet de manière ponctuelle en raison, par exemple, de bruit environnant ; dans une telle situation, la diminution du débit d'émission sera inadaptée.

Sarolahti *et al.* [14] proposent une politique de gestion de retransmissions prenant en compte la possibilité de pertes locales et ponctuelles. Xylomenos propose [18] une technique d'élimination locale des erreurs. Située en dessous de TCP, elle permet d'ignorer les pertes ponctuelles dues à l'interface sans fil. Ainsi, aucune modification n'est à apporter dans l'implémentation de TCP, pour qui les pertes ponctuelles deviennent invisibles.

2.2.3 Les piles TCP/IP minimalistes

Le protocole TCP est un protocole relativement lourd, utilisant une machine à états finis comportant 11 états ainsi que de nombreux mécanismes (fenêtres de congestion, de retransmission, accusés de réception, etc.). Ainsi, une pile TCP/IP classique, écrite pour un système d'exploitation d'ordinateur de bureau, nécessite environ 10 000 lignes de code C, résultant en un code compilé de plusieurs centaines de kilo-octets, et ayant une consommation mémoire de quelques kilo-octets [15].

L'écriture d'une pile TCP/IP pour l'embarqué est donc un réel défi à relever. Les cibles matérielles sont limitées à quelques kilo-octets, voire quelques centaines d'octets de mémoire vive. À simple titre de comparaison, la MSS classique de TCP utilisé sur un lien ethernet est de 1460 octets... Des solutions sont donc à trouver pour minimiser la taille des tampons et en optimiser la gestion. Le code exécutable, quant à lui, peut être placé en mémoire morte, dont la taille, supérieure à celle de la RAM, est elle aussi limitée. Il s'agit alors de tirer profit au mieux des ressources disponibles tout en proposant un service réseau le meilleur possible.

Présentation de lwIP et µIP

Dunkels propose [3] deux piles TCP/IP spécialement conçues pour les systèmes embarqués très contraints en mémoire, lwIP et µIP.

LwIP est une pile complète mais simplifiée, incluant les implémentations de IP, ICMP, UDP et TCP, et est assez modulaire pour permettre facilement la gestion de nouveaux protocoles. LwIP est capable de supporter plusieurs interfaces réseau et propose un système de configuration flexible.

µIP, quand à elle, propose un ensemble de services strictement minimal : seule une interface réseau peut être gérée, et les seuls protocoles gérés sont IP, ICMP et TCP. L'ajout d'un nouveau protocole n'y est pas aussi aisé que dans lwIP. LwIP et µIP sont écrits en langage C, et ont été portés sur plusieurs architectures 8 ou 16 bits.

Gestion de la mémoire

Nous nous intéressons ici aux choix qui ont été réalisés pour gérer la mémoire dans les piles lwIP et µIP.

LwIP utilise une gestion dynamique de la mémoire utilisant un ensemble global de paquets de taille fixe. Ces paquets sont servis à la demande aux couches ou aux pilotes qui en ont besoin, que ce soit pour stocker des paquets ou des états de connexion.

Dans µIP, les états de connexion sont stockés dans une table globale et statique. Un unique tampon est utilisé pour l'ensemble des connexions, que ce soit pour les données entrantes ou sortantes. Lorsque l'application lit des données, la pile les lui place dans le tampon global. L'application les traite, puis, au besoin, utilise à nouveau ce tampon pour placer des données à envoyer.

Les données qui continuent d'arriver depuis le réseau ne seront pas écrites dans le tampon global tant que l'application n'en demande l'accès en lecture. Ces données sont donc placées dans des tampons au niveau du pilote ou du matériel (la plupart des contrôleurs ethernet ont un tampon de quatre fois la taille d'une MTU). Si plus aucun tampon n'est disponible, les données entrantes sont défaussées.

Détails d'implémentation

Le modèle standard TCP/IP propose une approche par couches, certes efficace pour la conception ou la compréhension des protocoles, mais non optimale en performances pour une implémentation. LwIP propose une implémentation totalement modulaire, où chaque couche est écrite dans un module. Dans le code de µIP, les différentes couches ne sont pas clairement dissociables.

Au niveau de l'API, au lieu d'utiliser le classique système de *sockets*, les auteurs ont choisi une approche orientée événements, où l'application est notifiée par la pile de l'arrivée de nouveaux paquet à traiter. L'utilisation de sockets, utilisant des primitives d'attente passive, nécessiterait l'implémentation sous-jacente d'un système de gestion multi-tâches (par exemple pour une attente de données entrantes), trop lourd en consommation mémoire (allocation de plusieurs piles d'exécution, changements de contexte, ...).

La tableau 2.2.3 récapitule l'ensemble des fonctionnalités de LwIP et µIP.

Fonctionnalité	µIP	lwIP
Ré-assemblage des fragments IP	√	√
Options IP	–	–
Interfaces multiples	–	√
UDP	–	√
Connexions TCP multiples	√	√
Options TCP	√	√
MSS de TCP variable	√	√
Estimation de RTT	√	√
Fenêtre glissante TCP	–	√
Contrôle de congestion TCP	–	√
Données TCP désordonnées	–	√
Données urgentes TCP	√	√
Stockage des données pour retrans.	–	√

Tab. 2.1 – Récapitulatif des fonctionnalités de lwIP et µIP

2.3 Les serveurs Web embarqués

Les piles IP présentées précédemment sont de bonnes solutions pour réaliser des applications réseau embarquées sur des cibles fortement contraintes. Cependant, ces piles sont généralistes, et ne tirent pas partie de la connaissance de l'application qu'elles devront supporter. Dans notre cas, seul un serveur Web devra être supporté par la pile. La connaissance de l'application à exécuter est, au niveau de la pile réseau, une réelle source d'optimisations. Dans cette section, nous présentons les travaux qui ont été réalisés sur l'élaboration de serveurs Web embarqués. La plupart du temps, la pile IP de ces derniers est taillée sur mesure n'est pas dissociable du serveur ; ils font tous deux partie du noyau.

2.3.1 Conception d'une pile IP spécialisée

La plupart des des serveurs Web embarqués exploitent les mêmes points pour spécialiser leur pile. Voici une liste non exhaustive des principaux axes d'optimisation de ces piles spécialisées :

Connexions passives Un serveur Web n'a pas besoin d'initier de connexion TCP. Il a seulement besoin d'attendre de nouvelles connexions. Cela permet de simplifier l'automate classique de TCP, dont certains états deviennent superflus.

Données urgentes Le mécanisme de données urgentes de TCP n'est pas utilisé par un serveur Web. Il est possible de négliger la gestion de cette information au niveau de la pile IP, optimisant ainsi la taille du code et le coût de traitement.

Connaissance des protocoles On sait qu'un serveur Web n'a pas besoin d'utiliser UDP, ICMP, ou d'autres protocoles. Cela permet bien sûr de simplifier grandement la pile de protocoles utilisée.

Contrôle de congestion Ce point n'est valable que dans le cas d'une connexion directe depuis un client vers le serveur Web embarqué, c'est à dire, par exemple, d'un client qui se connecte à la carte SIM de son téléphone cellulaire. Dans cette situation, aucune congestion du réseau ne peut survenir, car le réseau est de type point-à-point. On peut alors se passer du contrôle de congestion TCP (permettant d'adapter son débit aux congestions d'un nœud du réseau).

Fragmentation IP La fragmentation IP étant très peu fréquente [2], il est envisageable, dans le contexte d'une pile IP minimaliste, d'en exclure la gestion.

2.3.2 Serveurs Web embarqués existants

Serveurs Web matériels

Certains serveurs Web embarqués [8, 13] se basent sur une architecture matérielle dédiée. Ces serveurs sont alors distribués sous forme d'un code écrit avec langage de description de matériel, comme VHDL, associé, si nécessaire, à du code logiciel. Ces serveurs peuvent être utilisés sur un FPGA, ou fondus sur un circuit ASIC. Bien qu'ils offrent de bonnes performance, ces serveurs ne correspondent pas à notre étude, qui vise l'installation de serveurs Web sur des cibles matérielles classiques. De plus, une telle solution limite fortement l'évolutivité du produit.

Le serveur mIP

Shon propose, dans [15], un serveur Web embarqué appelé mIP. Ce serveur exploite les différents points décrits précédemment pour la spécialisation de pile IP. 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. Ce serveur utilise un modèle de programmation événementiel, dans lequel l'application est appelée par TCP lorsque de nouvelles données sont à traiter. Un tampon applicatif global est utilisé pour la lecture et l'écriture de données, tandis que le stockage des données entrantes s'opère au niveau des pilotes ou du matériel, à la manière de µIP (voir section 2.2.3).

Proposition de Ju *et al.*

Dans [9], Ju *et al.* proposent une méthode de précompilation des pages statiques. Les pages que l'on souhaite embarquer dans le serveur Web subissent, avant la compilation, un pré-traitement ayant pour rôle de générer un code C contenant les pages de manière structurée. Ainsi, les pages sont stockées en mémoire adressable (non nécessairement de la RAM, ce peut être de la ROM) et sont ainsi directement accessibles par le programme. Cette méthode n'est bien sûr pas utilisable si l'on souhaite stocker les pages sur une mémoire non adressable, comme, par exemple, de la Flash NAND. La figure figure 2.6 détaille les différentes étapes nécessaires à l'utilisation d'un telle technique.

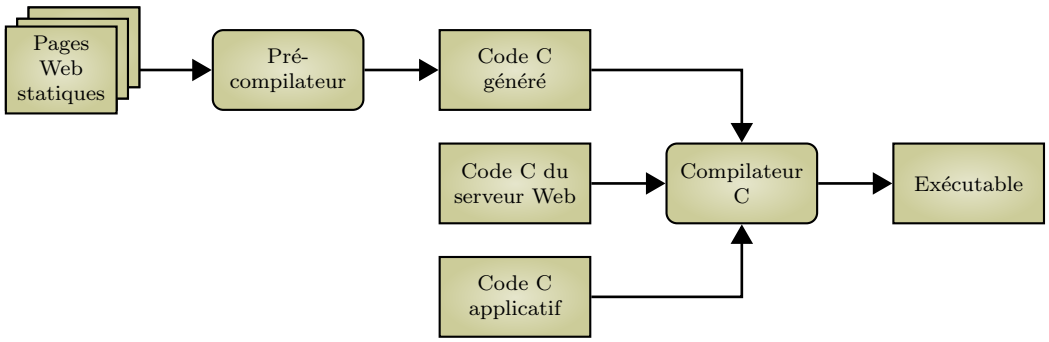

Fig. 2.6 – Précompilation HTML vers C

Le serveur miniWeb

MiniWeb, proposé par Dunkels [4], est un serveur Web embarqué 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.

Au lieu de lire le contenu de la requête GET, miniWeb identifie la page à servir en fonction du port sur lequel le client se connecte (variant par exemple de 80 à 85).

Les pages à servir sont stockées en mémoire adressable (technique de précompilation des pages), découpées en paquets IP prêts à être envoyés. Ainsi, les calcul à réaliser au moment de

l'émission des paquets sont minimaux. Les checksum IP et TCP sont partiellement calculés pour chaque paquet, et complétés au moment de l'émission en fonction des points variables (adresse IP distante, port distant, numéro d'accusé). MiniWeb ne permet pas d'envoyer des contenus générés dynamiquement..

Enfin, le serveur est limité à l'utilisation de HTTP 1.0, obligeant le client à initier une nouvelle connexion TCP avant chaque requête HTTP. Il se charge donc de fermer la connexion ouverte à la fin de l'émission de chaque fichier.

Voici la liste des principales fonctionnalités d'un serveur Web classique qui n'ont pas été implémentées dans miniWeb, pour minimiser sa consommation mémoire :

- Connexions TCP multiples ;
- Choix dynamique de la MSS TCP ;
- États de l'automate TCP ;
- Réceptions inattendues (paquets RST de TCP, etc.) ;
- Persistance de la connexion ;
- Options TCP et IP ;
- Stabilité face à des paquets « forgés ».

Ce serveur Web permet d'avoir une borne minimale pour l'occupation mémoire d'un serveur Web embarqué. Cependant, les fonctionnalités qu'il propose sont largement insuffisantes pour exécuter une application Web AJAX comme on souhaite le faire. Parmi les choix drastiques réalisés, nous pointons et expliquons ci-dessous ceux qui sont incompatibles avec le type d'application que nous avions identifié :

- La limitation du nombre de pages à servir empêche la diffusion d'une application sur le client potentiel.
- Le contenu des requêtes HTTP est ignoré, empêchant l'utilisateur de communiquer des données au serveur.
- Il est impossible de servir des contenus dynamiques, donc d'utiliser AJAX.

Chapitre 3

Étude des serveurs Web embarqués existants

Pour mieux analyser et connaître les serveurs Web embarqués qui ont déjà été proposés, nous avons décidé de les mettre à l'épreuve en les portant sur une même plateforme. Par la suite, cela nous permettra de comparer nos propositions à l'existant. Dans ce chapitre, une étude fine des performances des protocoles du Web est réalisée. Ensuite, la phase de portage des deux serveurs que nous avons choisis est décrite, suivie d'une analyse de leurs comportements.

3.1 Étude transversale des performances de HTTP, TCP et IP

Les performances d'un serveur Web embarqué sont étroitement liées à la manière d'utiliser les différents protocoles du Web. Les performances de ces protocoles en fonction de différents facteurs sont analysées dans cette section.

3.1.1 Définitions

Il est nécessaire, pour mieux comprendre le fonctionnement de TCP et IP, de définir quelques termes.

Définition 3.1.1 *On appelle accusés différés*[1] *la technique consistant à n'accuser un segment TCP que lorsque qu'une des conditions suivantes est vraie :*

1. *On a reçu un second segment ;*
2. *200 ms se sont écoulées.*

Définition 3.1.2 *On appelle nombre de paquets en vol le nombre de paquets envoyés par un hôte et dont l'accusé de réception TCP n'a pas encore été reçu.*

[1] Les accusés différés sont plus connus sous le nom de *delayed acks*.

Définition 3.1.3 *Une somme de contrôle est une somme, par groupe de 2 octets, d'un ensemble de données. L'en-tête IP et celui de TCP incluent une somme de contrôle, ou* checksum*, pour permettre au récepteur des données de détecter d'éventuelles erreurs de transmission.*

Définition 3.1.4 *La fenêtre annoncée, dans TCP, est le nombre d'octets qu'un hôte accepte de recevoir en provenance de l'autre hôte avec lequel il est connecté. Dans chaque en-tête TCP, une nouvelle fenêtre est annoncée.*

Surcoût de TCP/IP

L'utilisation de TCP sur IP permet une connexion fiable et sans pertes de données entre deux hôtes d'un réseau. Ces fonctionnalités sont permises par l'utilisation d'en-têtes et de paquets de contrôle, engendrant un surcoût on négligeable. La taille de l'en-tête d'un paquet IP sans options est de 20 octets. Celle d'un paquet TCP sans options est, elle aussi, de 20 octets. Les options TCP et IP ne sont pas nécessaires à un simple service Web. Le mécanisme d'accusés de réceptions de TCP engendre lui aussi un surcoût. Les accusés ont une taille de 40 octets, et sont susceptibles, au maximum, d'être envoyés à chaque réception d'un paquet. 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 :

$$Surcoût_{données} = \frac{Taille_{IP} + Taille_{TCP} + \alpha \times Taille_{ACK}}{MSS}$$

Avec la technique des accusés différés (implémentée par la plupart des piles IP), ont peut estimer α. En effet, après deux paquets reçus, on envoie toujours un accusé. Si on reçoit les paquets avec une période supérieure à 200 ms, on envoie un accusé par paquet reçu. On a alors $0,5 \leq \alpha \leq 1$.

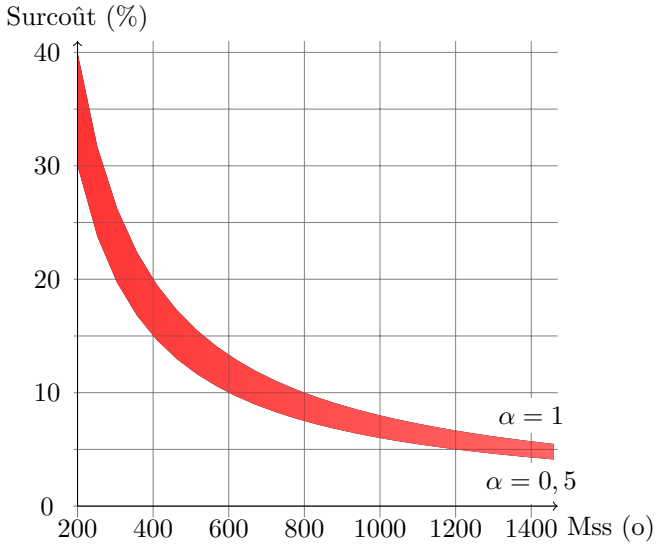

Fig. 3.1 – Surcoût en données de IP et TCP en fonction de la MSS

La figure 3.1 montre l'évolution du surcoût en volume de données transmises de IP et TCP en fonction de la MSS utilisée (pour une MSS allant de 200, le minimum autorisé, à 1460, la valeur classique utilisée pour ethernet). On constate que l'augmentation de cette MSS permet de grandement réduire le surcoût, quelle que soit la valeur de α. Il est à noter que dans la plupart des cas, la consommation en mémoire augmente avec la MSS à cause de l'utilisation de tampons, ce qui est critique pour l'embarqué.

Il est intéressant d'estimer également le débit utile maximal atteignable dans une configuration donnée. Pour l'estimation du débit maximal, le déclenchement de la politique d'accusés différés chez l'hôte distant influe grandement. Si on envoie les paquets un à un, en attendant de recevoir l'accusé d'un paquet avant d'envoyer le suivant, on limite le nombre de paquets en vol à un. On force ainsi α à valoir 1. Du coté du récepteur utilisant les accusés différés, le délai de 200 ms va être respecter avant l'envoi de chaque accusé, limitant fortement le débit utile de la connexion. On note β la fréquence de déclenchement du délai des accusés différés. On a $0 \leq \beta \leq 1$: avec un seul paquet en vol, β vaut 1, tandis qu'en envoi continu de données, β vaut 0.

Soit $Délai_{ACK}$ le temps d'attente avant l'envoi d'un accusé différé, c'est à dire classiquement, 200 ms. Le débit utile, en fonction de la MSS utilisée est :

$$Débit_{utile} = \frac{MSS}{(MTU + \alpha \times Taille_{ACK})/Débit_{physique} + \beta \times Délai_{ACK}}$$

La figure 3.2 montre l'évolution du débit utile en fonction de la MSS utilisée, avec une ligne fonctionnant à 56 Kbps. Les débits obtenus en fonction de α et β. On constate que la limitation à un seul paquet en vol ($\alpha = 1$, $\beta = 1$) réduit considérablement le débit utile. Les meilleures performances sont obtenues avec $\alpha = 0,5$ et $\beta = 0$, la situation où l'on envoie des données en continu.

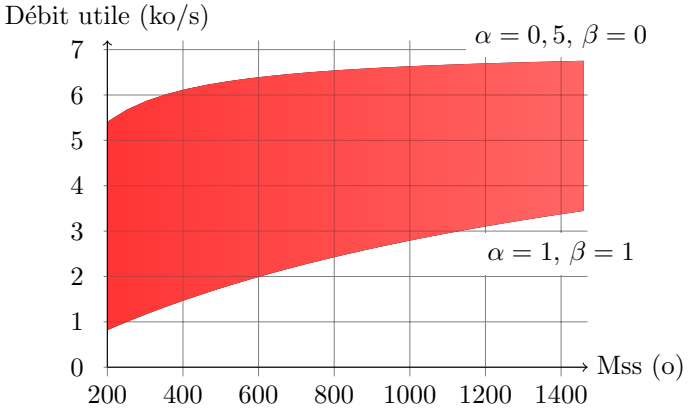

Fig. 3.2 – Débit utile de maximal fonction de la MSS

Dans un contexte général, il n'y a pas de relation directe entre α et β. Il est par exemple possible que l'hôte décide d'accuser tous les paquets reçus sans aucun délai, plaçant α à 1 et β à 0. Dans le cas où l'hôte distant utilise la technique classique des accusés différés, il en est autrement. Dans cette situation, α et β sont linéairement liés, car un accusé est envoyé pour un seul segment reçu (influant sur α) si et seulement si on a attendu le délai de 200 ms (influant sur β). On a alors $\alpha = 0,5 + \frac{\beta}{2}$.

Surcoût de HTTP

L'utilisation de HTTP, sur TCP et IP, ajoute de nouveaux surcoûts. Ce protocole est particulièrement verbeux, et toutes les informations qu'il transporte sont du texte pur. Le navigateur *Mozilla-Firefox*, lorsqu'il envoie une requête GET, utilise un en-tête d'environ 410 octets. Une réponse HTTP minimale utilise quant à elle un en-tête d'environ 40 octets. Sur un aller-retour de GET, le surcoût en quantité de données est d'environ 450 octets.

Ce surcoût s'ajoute bien sûr à ceux des couches précédentes, dépendamment des autres facteurs vus précédemment. Soit $Taille_{GET}$ la taille d'une requête GET, $Taille_{RET}$ la taille d'un en-tête de réponse et $Volume_{utile}$ la taille des données utiles que le serveur doit envoyer au client. Le nombre de paquets qui transitent sur le réseau lors d'une requête GET dépend de la MSS utilisée, il vaut :

$$Nombre_{paquets} = \left\lceil \frac{Taille_{GET}}{MSS} \right\rceil + \left\lceil \frac{Volume_{utile} + Taille_{REP}}{MSS} \right\rceil$$

On peut alors calculer le volume de données transmis lors d'une requête GET :

$$\begin{aligned} Volume_{réel} = {} & Nombre_{paquets} \times (Taille_{IP} + Taille_{TCP}) && \textit{En-têtes TCP et IP} \\ & + Volume_{utile} + Taille_{GET} + Taille_{REP} && \textit{Données HTTP} \\ & + (Nombre_{paquets} - 1) \times \alpha \times Taille_{ACK} && \textit{Volume des accusés} \\ & + \begin{cases} Volume_{tcp\_open} + Volume_{tcp\_close} & \text{si } \textit{HTTP 1.0} \\ 0 & \text{sinon} \end{cases} && \textit{Poignées de mains TCP} \end{aligned}$$

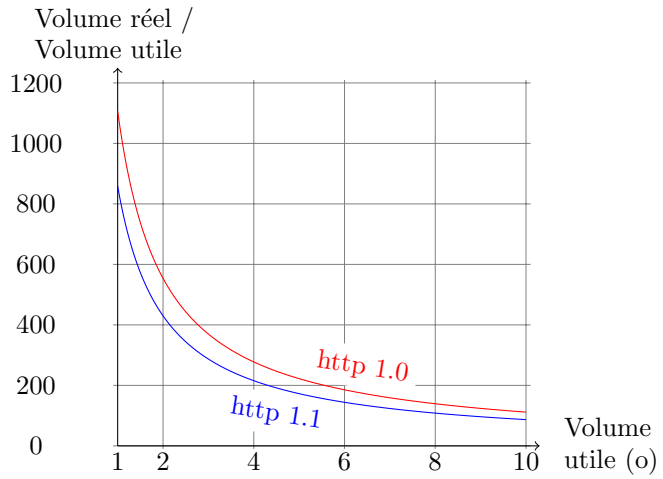

Fig. 3.3 – Surcoût engendré par HTTP

La figure 3.3 nous montre l'évolution du rapport entre le volume réel transmis et le volume utile de données à transmettre, en fonction de ce dernier. Le calcul prend en compte l'ensemble des protocoles HTTP, TCP et IP (on fixe α à 1 et MSS à 1460). On observe ici les performances de ces protocoles lors de l'envoi de données de petite taille. On constate

que le surcoût diminue très vite lorsque la taille des données augmente (amortissement d'un ensemble de coûts fixes). La différence de performances entre HTTP 1.1 et HTTP 1.0 est notable.

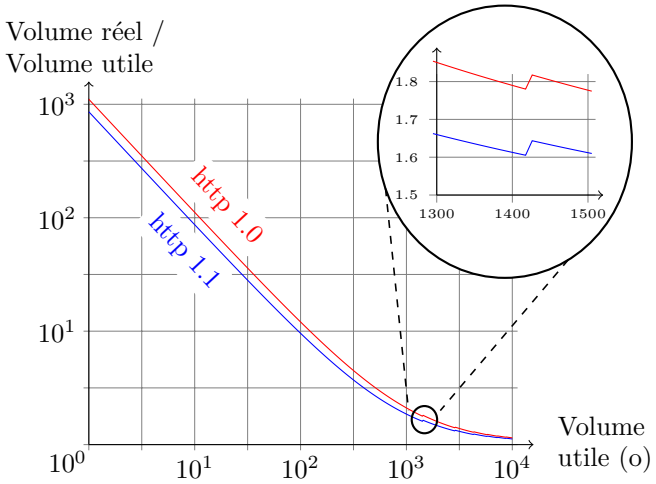

Fig. 3.4 – Surcoût engendré par HTTP

La figure 3.4 trace les mêmes fonctions que la figure 3.3 avec des échelles différentes, de manière à ne pas se limiter aux envois de petites données. Ici, les échelles utilisées en abscisse comme en ordonnée sont logarithmiques.

On remarque que la chute du surcoût est fulgurante lorsque le volume de données utiles augmente. Pour la transmission d'un seul octet en HTTP 1.0 il est de 1047. Lorsque l'on transmet 10000 octets en utilisant HTTP 1.1, le ratio s'approche de 1.12. Les pics présents sur la courbe correspondent aux cas où une légère augmentation de la taille des données utiles provoque l'utilisation d'un paquet supplémentaire, imposant de nouveaux en-têtes TCP et IP et d'éventuels accusés. En utilisant une MSS plus petite, ce phénomène serait encore plus fréquent.

3.2 Portage de serveurs Web existants

La cible que nous avons choisie pour tester les serveurs Web est une Game Boy Advance (GBA) (figure 3.5). Elle est équipée d'un microprocesseur ARM7TDMI 32 bits, basé sur une architecture RISC et fonctionnant à une fréquence de 16.78 MHz (2^{24} Hz). Ce processeur ne dispose pas de MMU (Memory Management Unit). Il embarque 32 Ko d'IWRAM (Internal Work RAM) et 96 Ko de VRAM (Video RAM). 256 Ko d'EWRAM (External Work RAM) sont également intégrés dans la GBA. Elle dispose de plus d'un port spécifique que nous allons utiliser comme un port série pour réaliser nos communications réseau.

Les performances de cette console sont comparables à celles de systèmes embarqués classiques (carte à puce, capteur), ce qui en fait un bon choix pour nos implémentations. Il est possible d'écrire dans une mémoire Flash externe un binaire exécutable, ce qui permet de programmer aisément la GBA.

Fig. 3.5 – La Game Boy Advance

Nous avons choisi de porter le serveur miniWeb (voir section 2.3.2) ainsi que la pile µIP (voir section 2.2.3). Le portage de miniWeb nous permet de nous comparer à un serveur totalement minimaliste en terme de mémoire. En portant µIP ainsi que le serveur Web fourni par ses développeurs, on peut se comparer à un serveur s'exécutant sur une pile IP généraliste et complète.

3.2.1 Étapes nécessaires à la réalisation d'un portage

Les serveurs que nous avons choisi de porter sont des serveurs embarqués minimalistes. Ils n'utilisent pas de système d'exploitation, et sont écrits pour être exécutés par le processeur embarqué en mode noyau. Les abstractions telles que les processus légers ou les primitives d'allocation dynamique de mémoire n'y sont pas fournies.

Compilation croisée

La première étape d'un portage consite à rendre le code source fourni compilable vers l'architecture cible souhaitée, *i.e.*, l'ARM7TDMI. Dans notre cas, la principale barrière lors de cette étape consistait dans l'utilisation de l'option de compilation `-fpack-struct` de `gcc`. Cette option demande d'ignorer la classique politique d'alignement des variables et des structures en mémoire. Elle était nécessaire au bon fonctionnement des deux serveurs, qui exploitaient à plusieurs reprises cet agencement particulier des variables en mémoire.

Avec cette option, `gcc` génère par exemple des accès à des mots sur des adresses non alignées à 4, ce qui, sur une architecture x86 – pour laquelle les codes sources fournis étaient destinés – ne pose aucun problème. L'ARM7TDMI, quant à lui, ne peut pas réaliser de tels accès en mémoire. Quelques modifications du code ont donc du être réalisées pour palier à ce problème (ajout d'éléments dans certains structures pour contrôler l'alignement, etc.).

Écriture des pilotes

L'étape suivante consiste à implémenter les pilotes du matériel. Les codes sources fournis utilisant des API différentes, plusieurs jeux de pilotes ont du être écrits lors de la réalisation des deux portages.

Dans notre situation, les pilotes dont nous avons besoin sont ceux permettant d'accéder à l'interface réseau, c'est à dire au port de la GBA permettant une communication série avec

l'extérieur. Le protocole de couche liaison que nous avons choisi est SLIP, un protocole très léger en surcoût et limité aux communications point-à-point, situation dans laquelle nous nous trouvons. De plus, SLIP est un protocole classique et souvent géré nativement par les systèmes d'exploitation.

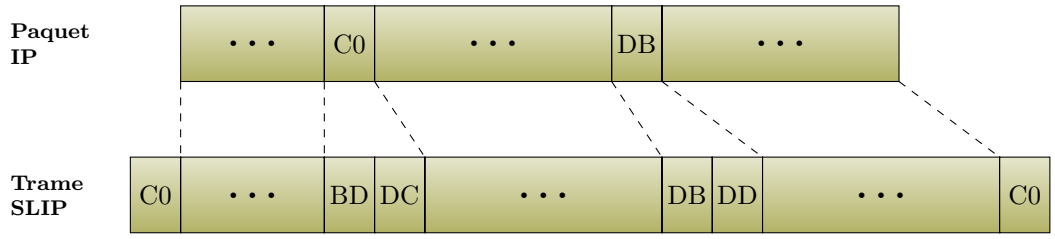

Fig. 3.6 – Fonctionnement du protocole SLIP

Le fonctionnement du protocole SLIP est illustré par la figure 3.6. SLIP ajoute systématiquement au début et à la fin de chaque trame à transmettre le caractère `0xC0`. Le caractère d'échappement `0xDB` suivi de `0xDC` permet de transmettre un `0xC0` sans qu'il soit interprété comme une fin de trame. Enfin, pour transmettre un `0xDB` sans qu'il soit interprété comme un caractère d'échappement, on envoie `0xDB` suivi de `0xDD`.

Pour l'écriture de ces pilotes, on a besoin de réaliser des lectures et des écritures sur le port de communication de la GBA. Pour écrire des données, on pilote le matériel en écrivant (et en lisant) dans des registres spécifiques. Pour la lecture de données, on écrit un gestionnaire d'interruption qui se déclenche lors de chaque réception d'un octet. On lit alors les données reçues par l'intermédiaire, toujours, de registres adressables dédiés.

3.2.2 Portage de µIP et miniWeb

µIP et miniWeb utilisent deux APIs différentes pour accéder aux pilotes.

µIP lit et écrit les données par paquets IP, via un tampon applicatif global. Il est nécessaire d'utiliser au niveau des pilotes un second tampon pour stocker les données entrantes pendant que µIP continue de s'exécuter. Ces deux tampons doivent être d'au moins la taille de la MTU utilisée.

L'API de miniWeb fonctionne par octets, il traite les données entrantes à la volée, sans les stocker. Il est possible de ne pas utiliser de tampon au niveau des pilotes. Lorsqu'on utilise un tampon (pour limiter le nombre de paquets défaussés), sa taille n'influe pas sur la MSS utilisable.

3.3 Analyse des serveurs portés

Une fois les portages des deux serveurs réalisés, il est possible d'en évaluer les performances sur plusieurs points et d'identifier leurs biais.

Dans nos tests, on utilise un lien série offrant un débit de 56 Kbps (kilo-bits par seconde). Étant connecté en point-à-point à la GBA, on peut choisir la taille de la MTU (Maximum Transfert Unit) utilisée. Nous avons réalisé des tests avec des tailles de MTU variables, allant

de 240 (impliquant la MSS minimale de TCP, 200 octets) à 1500 octets (qui est aussi la MTU utilisée pour ethernet).

3.3.1 Analyse de µIP

Remarques générales

Le serveur Web de la pile µIP permet le service de fichiers statiques comme de contenus dynamiques. L'ensemble des contenus à servir sont placés dans un même répertoire. Un outil fourni permet de générer des fichiers *c* à partir de ces contenus (technique de précompilation des fichiers, voir section 2.3.2). Plusieurs formats de fichiers sont pris en compte, et le type MIME qui y est associé (permettant au client Web de connaître le type du fichier reçu) est automatiquement déduit de l'extension du fichier. Les données contenant ces pages compilées étant constantes, elles peuvent être placées en mémoire morte, souvent disponible en plus grande quantité que la mémoire vive.

Les contenus dynamiques sont servis par interprétation des fichiers d'extension `.shtml`. Ces fichiers sont au format HTML, et contiennent des balises dans un format spécifique. Avant de servir une telle page, le serveur remplace les balises par un contenu dynamique, obtenu par simple appel d'une fonction spécifiée par la balise (à la manière des JSP ou des ASP). L'ajout de nouvelles fonctions de génération de contenus n'est pas aisé, et nécessite de modifier certains fichiers de code source.

Les concepteurs de ce serveur ont été contraints, dans un souci d'économie de mémoire et d'allègement des traitements, à utiliser la norme HTTP 1.0 (voir section 2.1.2). Cela engendre bien sûr un surcoût considérable (deux poignées de mains supplémentaires pour chaque requête), en particulier lors du service de petits fichiers (ce surcoût est mis en évidence dans la figure 3.3).

Les applications fournies avec µIP (dont le serveur Web fait partie) sont implémentées en utilisant la technique des protothreads [5], fournissant une abstraction de blocage dans le code *c* avec un très faible surcoût en mémoire.

Analyse des traffics

Comme nous l'avons évoqué en section 3.3.1, µIP limite son nombre de paquets en vol à un (pour des raisons détaillées ci-après). Ainsi, lorsqu'un paquet est envoyé, µIP attend d'en recevoir l'accusé avant de continuer à envoyer des données. Si l'hôte distant utilise la technique des accusés différés, chaque accusé ne sera envoyé que 200 ms après la réception de chaque segment, limitant fortement le débit maximal sortant. Ce problème a été soulevé par les auteurs de µIP [3]. La section 3.1.1 donne plus de précisions sur ce problème.

Cette limitation du nombre de paquets en vol a ses raisons. Elle est principalement due au fait que l'écriture d'une pile IP générique mais très économe en mémoire nécessite de faire de nombreux sacrifices. Des performances ont ici été sacrifiées pour gagner en généricité.

La gestion de retransmissions est un des points qui a conduit à cette limitation. En effet, la stratégie classique de TCP consiste à stocker en mémoire tous les segments envoyés mais non accusés. Si aucun accusé n'arrive après un laps de temps, on peut retransmettre les segments concernés. Dans un souci d'économie de mémoire, les concepteurs de µIP décident de ne pas stocker ces données en mémoire. Lorsqu'une retransmission est nécessaire, µIP demande à l'application de régénérer le paquet à envoyer. Comme le nombre de paquets en vol est

limité à un, l'application n'a qu'à réitérer son dernier envoi. Sans cette limitation, et sans connaître l'application exécutée (µIP n'est pas écrite pour une application en particulier), cette politique est impossible. Cependant, si on sait que seul un serveur Web sera utilisé, il est possible d'appliquer cette stratégie sans limiter le nombre de paquets en vol (ce que nous détaillons en section 4.1.2).

Consommation mémoire

La réduction de la consommation mémoire est la première cible de µIP. La taille du code exécutable (segment de texte) après portage de µIP et son serveur Web est de 44.5 ko.

Les variables globales, dans µIP utilisent 13,6 ko de mémoire si on positionne la MTU à 1500 octets. Avec une MTU de 240 octets, on peut réduire ce chiffre à 11,2 ko. L'espace mémoire utilisé en pile d'exécution pour les informations d'appels de fonctions et les variables locales représente 204 octets. Pour mesurer la consommation en pile, on initialise cette dernière, avant toute exécution, par une série de mots ayant une valeur bien particulière. Puis, après mise à l'épreuve du serveur, on en stoppe l'exécution pour réaliser une capture de la pile d'exécution. La zone qui a subi des modifications nous permet de connaître, à quelques octets près, la quantité de mémoire consommée par le serveur en pile d'exécution.

Biais identifiés de µIP

On évoque en section 3.2.2 le fait qu'µIP utilise un tampon applicatif global, et que les pilotes nécessitent l'utilisation d'un second tampon. Ces deux tampons doivent être d'une taille au moins égale à la MTU utilisée (ils doivent pouvoir contenir un paquet complet). On a alors le choix entre privilégier la mémoire ou les performances du serveur. La mémoire étant une contrainte dure vis à vis du matériel ciblé, on sera parfois obligé de diminuer la MTU utilisée pour réduire la taille des tampons. Dans une telle situation, la perte de performance est loin d'être négligeable, comme en témoignent la section 3.3.1 et les figures 3.1 et 3.2.

De manière générale, µIP n'offre pas une consommation mémoire aussi faible qu'on pourrait le souhaiter. Certains systèmes embarqués, tel une carte à puce, ne disposent que de quelques kilo-octets voire quelques centaines d'octets de mémoire vive.

3.3.2 Analyse de miniWeb

Remarques générales

Contrairement à µIP, miniWeb ne permet que le service de pages statiques. Ces pages sont elles aussi compilées en un code *c* utilisé par le serveur via un outil fourni. L'aspect minimaliste en mémoire de miniWeb a conduit ses concepteurs à réduire à son strict minimum ses fonctionnalités. Ainsi, les noms de fichiers ne sont pas pris en compte ; les fichiers à servir sont simplement associés à un port (par exemple allant de 80 à 84).

MiniWeb ne supporte que HTTP 1.0, plus simple et plus léger en mémoire que HTTP 1.1. Comme pour µIP, cela nuit fortement au débit sortant. De plus, miniWeb n'autorise pas l'ouverture de plusieurs connexions simultanées. Dans le cas où le client Web décide d'ouvrir plusieurs connexions (pour accéder à plusieurs fichiers de manière parallèle), ses demandes sont refusées, il est contraint d'attendre la fin de la réception du fichier courant avant de les réitérer.

Remarques sur les traffics

La manière de stocker les pages en mémoire utilisée par miniWeb est très différente de celle de µIP. Au lieu de placer en mémoire le contenu brut des fichiers, miniWeb découpe ces fichiers en paquets d'une taille définie à la compilation. À chaque paquet, on ajoute les en-têtes IP et TCP, ou du moins l'ensemble des informations qui y seront embarquées et qui sont connues à la compilation (adresse source, port source, numéro de séquence, identifiant IP, etc.).

Le fait d'utiliser une nouvelle connexion à chaque requête permet de simplifier grandement la gestion de ces paquets, dont on connaît à l'avance les numéros de séquence. La gestion des accusés de réception reçus en est elle aussi grandement simplifiée : indépendamment de tout contexte, un numéro d'accusé pointe toujours vers la même portion du fichier en cours d'émission. Même la fermeture de connexion qui suit l'envoi de la réponse est stockée dans la suite des paquets à envoyer, plutôt que de gérer le classique (et lourd) automate TCP lors de la déconnexion.

Cette manière de stocker les fichiers est indispensable pour atteindre une occupation mémoire aussi basse que celle de miniWeb, car elle permet de nombreuses simplifications au niveau du code d'envoi des fichiers. Cependant, elle présente un inconvénient majeur : comme les paquets sont préfabriqués, leur taille est déterminée à la compilation.

Si le client propose une MSS inférieure à celle choisie pour les paquets, le serveur ne peut pas lui répondre. Ce constat force miniWeb à utiliser la plus petite MSS qui puisse être acceptée par TCP, qui est de 200 octets. Dans la plupart des cas, le client propose une MSS bien supérieure à 200, et, à son détriment, la gestion statique des paquets de miniWeb bride fortement la connexion.

Toutefois, et contrairement à µIP, miniWeb est capable de gérer plusieurs paquets en vol, ce qui améliore largement le débit sortant (section 3.3.1, figures 3.1 et 3.2). En cas de perte de données, miniWeb n'a qu'à retransmettre les paquets préfabriqués stockés en mémoire adressable.

Somme de contrôle

Le calcul des sommes de contrôle IP et TCP est particulièrement coûteux. Pour accélérer ce traitement, miniWeb calcule ces sommes à la compilation des fichiers, et les inclut directement dans les paquets qui seront envoyés. Lors de l'envoi, la somme est simplement mise à jour avec les quelques données qui n'étaient pas connues à la compilation, comme l'adresse et le port sortants, le numéro d'accusé, etc.

Architecture de miniWeb

MiniWeb parvient à gérer l'ensemble de la pile IP et du serveur de manière totalement monolithique. Cela permet de n'utiliser aucun tampon en réception comme en émission. La granularité des primitives de lecture et d'écriture fournie par les pilotes est l'octet. Ainsi, lorsqu'un paquet est reçu, il est traité octet par octet. La quasi-totalité des octets lus sont traités directement puis écrasés par l'octet suivant.

Au niveau de l'écriture des pilotes, il est possible, si le serveur traite chaque octet en un temps inférieur à la période de réception d'un octet sur le réseau, de ne pas utiliser de

tampon. Nous avons proposé une implémentation de pilotes permettant cela, réduisant ainsi au strict minimum la consommation mémoire.

On peut tout de même déplorer le fait que le contenu des requêtes HTTP n'est pris en compte à aucun moment. Lorsqu'une connexion a été ouverte par le client, tout nouveau paquet entrant engendre l'émission de la page associée au port de la connexion.

L'absence de tampon en réception au niveau des pilotes provoque bien sur des pertes de paquets. C'est pourquoi nous avons également implémenté des pilotes utilisant un tampon circulaire, diminuant le nombre de pertes et donc de retransmissions du client.

Consommation mémoire

La table 3.3.2 compare la consommation mémoire de miniWeb à celle du serveur de µIP. MiniWeb tient bien son rôle de serveur à consommation mémoire minimale (moins de 400 octets de mémoire vive occupée).

Serveur Web	Taille du code exécutable	Taille des variables globales	Taille occupée sur la pile
µIP	44.5 ko	11.2 ko	204 o
miniWeb	6.4 ko	88 o	268 o

Tab. 3.1 – Comparatif des consommations mémoire des serveurs étudiés

Le prix à payer du minimalisme

Le minimalisme de miniWeb nous permet de fixer une borne inférieure pour la consommation mémoire d'un serveur Web embarqué. Cependant, ses fonctionnalités sont largement insuffisantes pour nos besoins. De plus, sa grande simplicité implique des failles, et il est possible, lorsque quelques pertes se produisent au niveau de la liaison, ou si l'on reçoit un paquet forgé, que le serveur entre dans un état incohérent voire une boucle infinie. Ce manque de stabilité est un réel problème pour un serveur Web.

L'impossibilité de lire le contenu des requêtes HTTP et de servir des contenus dynamiques sont deux problèmes majeurs, rendant totalement impossible l'utilisation de miniWeb pour une application réelle.

Le tableau 3.3.2 liste l'ensemble des fonctionnalités implémentées par µIP et miniWeb.

Fonctionnalité	µIP	miniWeb
Options IP	–	–
Ré-assembage des fragments IP	√	–
UDP	–	–
ICMP	√	–
Options TCP	√	–
Données urgentes TCP	√	–
Stockage des données pour retrans.	–	–
Estimation de RTT	√	–
Requêtes POST	–	–
Lecture de l'URL requise	√	–
MSS de TCP variable	√	–
Connexions TCP multiples	√	–
Plusieurs paquets en vol	–	√
HTTP 1.1	–	–
Contenus dynamiques de taille bornée	√	–
Contenus dynamiques de taille non bornée	√	–
Pré-calcul des checksum	–	√

TAB. 3.2 – Fonctionnalités proposées par µIP et miniWeb

Chapitre 4

Propositions et évaluations

Le portage des serveurs µIP et miniWeb nous a permis d'en identifier les principaux biais. Il reste alors à proposer des solutions pour éviter les principales faiblesses observées. Un serveur Web embarqué idéal allierait les fonctionnalités complètes de µIP et la faible consommation mémoire de miniWeb.

Dans cette section, nous proposons des solutions aux différents problèmes soulevés, avec pour objectif une consommation mémoire aussi faible que possible tout en ne sacrifiant aucune fonctionnalité dont nous avons besoin pour la réalisation d'applications Web embarquées. Une implémentation de l'ensemble de nos propositions a été réalisée, engendrant le serveur *dynaWeb*.

4.1 Présentation du serveur dynaWeb

4.1.1 Objectifs de dynaWeb

DynaWeb a pour objectif de montrer qu'un serveur Web dont l'implémentation est monolithique, traitant les différents protocoles de manière globale, offre de nombreux avantages. Intuitivement, on comprend que des économies de mémoire et de traitement sont réalisables. On montre avec dynaWeb qu'outre ces gains, des optimisations très efficaces deviennent envisageables grâce une approche transversale des protocoles.

Comme on l'a mentionné en section 1.3, on souhaite utiliser notre serveur Web embarqué comme un moyen de fournir une application hautement interactive. Ce type d'application a un fonctionnement à deux vitesses, commençant par une phase de chargement de lourds contenus statiques, suivie par de nombreuses requêtes vers de petits contenus générés dynamiquement (voir section 1.3.2). C'est pourquoi on souhaite permettre à dynaWeb d'être robuste dans ces deux situations, aux besoins en ressources totalement différents. La première phase a besoin d'un bon débit, tandis que la seconde nécessite des temps de traitement et des surcoûts protocolaires réduits à leur strict minimum.

4.1.2 Modèle de fonctionnement

DynaWeb fonctionne sans tampon de réception, et utilise une API d'accès à l'interface de communication capable de lire et d'écrire les données octet par octet. Ces derniers sont lus un

à un, analysés et traités à la volée. Seule l'URL contenue dans la requête HTTP entrante est placée dans un tampon d'une taille de quelques dizaines d'octets. Contrairement à miniWeb, dynaWeb est ainsi capable de servir un contenu à partir de son nom. Le fonctionnement de la boucle principale de dynaWeb est décrit par l'algorithme 1.

```
tant que vrai faire
    attendre_paquet()
    traiter_IP()
    connexion ← traiter_TCP()
    url ← traiter_HTTP()
    si checksum_ok() alors
        connexion.url ← url
        dynaweb_output()
    fin
fin
```

Algorithme 1 : Boucle principale de dynaWeb

L'analyse de l'en-tête TCP nous permet d'extraire la connexion en cours, ce qui est nécessaire pour la gestion de plusieurs connexions et de HTTP 1.1. La gestion de connexions multiples permet au client Web d'envoyer plusieurs requêtes de manière parallèle, ce qu'il fait classiquement pour accéder à plusieurs fichiers référencés par exemple dans un code HTML (JavaScript, CSS, etc.).

La fonction `dynaweb_output()`, envoie des paquets de contrôle ou des contenus quand cela est nécessaire (voir section 4.1.3), et ce pour toutes les connexions en cours. L'état de chaque connexion est stocké dans une structure, permettant de connaître à tout instant les actions à entreprendre. Lors d'une émission, dynaWeb envoie autant de paquets que possible, sans en attendre l'accusé (on a alors plusieurs paquets en vol). La limite du nombre de paquets en vol est fixée par la fenêtre TCP annoncée par le client.

Il est à noter que pour l'instant, dynaWeb ne traite que les requêtes de type GET. Comme pour µIP et miniWeb, toute requête POST entrante n'est pas prise en compte.

La primitive `attendre_paquet()`, implémentée par les pilotes, réalise une lecture bloquante. Lorsqu'un délai est écoulé, une fonction (`dynaweb_timer()`) est appelée, provoquant si nécessaire la retransmission des paquets non accusés. Comme on n'utilise aucun tampon en émission, la retransmission consiste à produire à nouveau les paquets non accusés. On peut aisément retrouver les parties du fichier en cours d'envoi qui sont à retransmettre à partir du dernier numéro d'accusé reçu et du numéro de séquence courant. Contrairement à µIP (voir section 3.3.1), on arrive à gérer plusieurs paquets en vol sans pour autant les conserver en mémoire pour de futures retransmissions. La retransmission de contenus dynamiques est traitée en section 4.3.2.

4.1.3 Modèle applicatif

DynaWeb n'est pas un serveur figé. Notre objectif est de permettre le développement aisé d'applications Web AJAX. Ces applications, outre des contenus statiques (voir section 4.2), contiennent des *services*, qui génèrent dynamiquement des données à retourner au client (voir section 4.3).

Dans notre modèle, chaque application est un répertoire contenant un ensemble de fichiers à servir ainsi qu'un ensemble de services (sous forme de fichier c). Un outil développé pour

dynaWeb permet de placer automatiquement ces fichiers et services dans des fichiers source *c* et des bibliothèques, qui seront compilés avec dynaWeb pour y être embarqués. Un index est également crée, il permet d'associer à chaque URL un contenu statique ou dynamique.

C'est dans la fonction `dynaweb_output()` que l'application est gérée par dynaWeb. C'est ici qu'on envoie les contenus demandés par le client, après les avoir éventuellement générés via les services de l'application.

4.2 Service de contenus statiques

4.2.1 Proposition pour le stockage des pages en mémoire

Biais à contourner

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 des économies de mémoire (moins d'informations d'état à gérer) et de traitement (calculs de checksum réalisés lors de la compilation). Cependant, comme on le montre en section 3.3.2, cela nuit aux performances du serveur, qui est obligé d'utiliser une MSS de 200 octets s'il veut pouvoir servir des requêtes quelle que soit la MSS proposée par le client.

À l'opposé de ce choix, µIP ne précalcule pas les paquets, et est capable de s'adapter à la MSS du client. Il est alors contraint à générer dynamiquement les en-têtes IP, TCP et HTTP ainsi que les sommes de contrôle, engendrant un réel surcoût.

Solution choisie pour dynaWeb

Pour réaliser des économies de ressources, on utilise dans dynaWeb une technique proche de celle de miniWeb. Lors de la précompilation des pages, l'en-tête HTTP, indépendant de la MSS finalement utilisée, est alors généré (incluant la longueur du contenu ainsi que son type). Le fichier est ensuite placé linéairement en mémoire.

Au lieu de calculer (et stocker) à l'avance les en-têtes TCP et IP pour chaque paquet à envoyer, on stocke un unique modèle d'en-tête précalculé. Il contient l'ensemble des champs qui sont statiquement connus pour l'ensemble des paquets qui seront envoyés (taille de fenêtre, protocole utilisé, adresses et port source, etc.). La somme de contrôle de ces données est réalisée hors ligne. Lors de l'envoi d'un paquet, on met à jour les en-têtes et les checksum de IP et TCP en y incluant les champs qui n'étaient pas connus à la compilation.

La somme de contrôle des données du fichier à envoyer, au lieu d'être calculée sur l'ensemble du fichier, est calculée par bloc. Les sommes ainsi calculées sont stockées les unes à la suite des autres, avant les données du fichier. Lors de l'envoi d'un paquet, dynaWeb n'a plus qu'à sommer les checksum partielles du fichier pour connaître la somme totale. Les gains engendrés par cette méthode sont analysés en section 4.2.2.

Notre proposition concilie à la fois les économies de ressources de la méthode de miniWeb et l'adaptation dynamique de la MSS que µIP réalise.

4.2.2 Proposition pour le calcul de checksum sur les fichiers

Comme mentionné en section 4.2.1, la somme de contrôle des contenus statiques est calculée lors de la compilation, par blocs de données. Ces blocs ont une taille fixe déterminée à la compilation ; nous allons en analyser l'impact.

La taille des blocs influe à la fois sur le temps de calcul de checksum et l'espace de stockage nécessaire en mémoire morte. L'espace occupé par les checksum précalculés est de :

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

.

Lors de l'envoi d'un paquet, le calcul de la somme de contrôle est une phase critique, représentant une part non négligeable de l'activité du processeur. Le tableau 4.1 montre les gains réalisés en utilisant le calcul par bloc des checksum. On mesure le temps d'activité du processeur lors de l'envoi d'un contenu statique de 1460 octets (valeur de la MSS utilisée). On constate qu'en augmentant la taille de blocs, on gagne à la fois en temps de calcul et en consommation de mémoire morte.

Taille des blocs	Temps processeur	Gain en temps	Coût en mémoire
0	138 ms	0 %	0 %
4	123 ms	11 %	50 %
8	115 ms	17 %	25 %
16	112 ms	19 %	12.5 %
32	110 ms	20 %	6.25 %
64	109 ms	21 %	3.13 %
128	109 ms	21 %	1.6 %

Tab. 4.1 – Impact des checksum précalculés

À partir d'une taille de blocs de 128 octets, le temps de calcul de checksum passe en dessous de la milliseconde. Il y a cependant un inconvénient à utiliser une trop grande taille de blocs. En effet, on est contraint à toujours envoyer une quantité de données multiple de cette taille. Avec une valeur trop élevée, on s'adapte moins finement à la MSS proposée par le client. Voici, en fonction de la MSS, la quantité de données que l'on peut envoyer par segment TCP en fonction de la taille des blocs :

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

La taille de bloc idéale est donc discutable ; elle dépend à la fois de l'application visée et des priorités que l'on se fixe.

4.3 Service de contenus dynamiques

4.3.1 Solution choisie pour dynaWeb

Le point clé des applications que l'on souhaite embarquer sur notre serveur est la phase de service des contenus dynamiques. Cette partie est inexistante dans miniWeb, nous ne pourrons nous comparer qu'à µIP. Le choix réalisé par µIP est décrit en section 3.3.1. Les

contenus dynamiques sont en fait des balises incluses dans des fichiers au format HTML. Ces balises sont interprétées, leur résultat est placé dans le tampon global, puis le contenu du tampon est envoyé.

Notre priorité, en matière de contenus dynamiques, est de permettre le service de petits contenus, comme c'est souvent le cas dans les applications AJAX. Pour l'instant, notre implémentation ne permet pas le service d'un contenu dynamique d'une taille supérieure à la MSS courante. En effet, ce cas de figure soulève plusieurs problèmes, décrits en section 4.3.2. L'unique tampon global de dynaWeb, déjà utilisé pour lire l'URL requise par le client lors de la réception d'une nouvelle requête, nous sert aussi à la génération de contenus.

Chaque « service » est en fait une fonction, écrite en *c*, prenant en arguments un pointeur vers le tampon où écrire et la taille de ce tampon. L'outil chargé du pré-traitement des contenus statiques compile également l'ensemble des services utilisés par l'application et crée une bibliothèque qui sera utilisée ensuite lors de la compilation de dynaWeb. Une table contenant un ensemble de paires (*nom*, *contenu*) permet au serveur, à partir d'une URL requise par le client, de trouver la page ou le service qu'il doit envoyer.

Une fois le service exécuté, dynaWeb a accès au contenu qu'il doit servir. L'en-tête HTTP est automatiquement ajouté par le serveur. La somme de contrôle de cet en-tête est calculée hors ligne, permettant de gagner du temps au moment du service. Le seul point que l'on ne connaît pas à l'avance dans cet en-tête est la longueur du contenu, qui est ajouté dynamiquement au checksum par le serveur. Encore une fois, cette optimisation est rendue possible par l'approche transversale des protocoles (c'est dans la gestion de TCP que l'en-tête HTTP est créé).

DynaWeb utilise HTTP 1.1, contrairement à µIP et miniWeb. Sur le service de petits contenus (comme c'est le cas ici), la persistance des connexion est un réel avantage (voir section 3.1.1), diminuant considérablement le traffic entrant et sortant, améliorant ainsi le temps de réponse. Dans le cas du service d'un contenu dynamique plus volumineux dont on ne connaît pas à la taille l'avance, on est contraint de ne pas utiliser le mode *keep-alive*.

4.3.2 Discussion sur le service de contenus dynamiques

Un point critique du service de contenus dynamiques est le fait qu'il est impossible de connaître la taille du contenu de la réponse HTTP avant la fin de l'exécution du service. Hors, il est nécessaire de renseigner cette taille dans l'en-tête HTTP si l'on souhaite conserver la connexion TCP ouverte après l'envoi. Si on ne renseigne pas cette valeur, on indique la fin de la transmission par la fermeture de la connexion.

Il est également envisageable d'utiliser une solution où le service n'écrit pas ses données dans un tampon, mais utilise une primitive d'écriture fournie par le serveur. Cette dernière se charge de placer les données en tampon. Si le contenu généré est plus petit que le tampon, on l'envoie en étant capable de spécifier la longueur de la réponse, et donc en conservant la connexion HTTP. Sinon, on envoie les données du tampon sans préciser la longueur du contenu HTTP, et on est contraint à fermer ensuite la connexion.

La génération de contenus dynamiques se complique lorsque la taille de ces contenus est susceptible de dépasser la MSS. En effet, dans ce cas, on est contraint à envoyer un paquet avant la fin de la génération de contenu, puis à continuer d'exécuter le service. En effet, dynaWeb ne fournit pas d'abstraction telle que les processus légers, donc pas de primitive de blocage. Un service qui a commencé à s'exécuter doit se terminer avant de rendre la main

au serveur. Ce comportement est problématique car il fige, pendant l'exécution du service, l'activité du serveur (qui ne peut donc plus recevoir de paquets).

Une solution à ce problème consiste à consommer les éventuels accusés entrants entre chaque émission de contenus dynamiques, pour rester attentif aux tailles de fenêtre TCP annoncées par le client, et ainsi se donner les moyens de terminer la transmission du contenu en cours de génération.

La solution choisie par µIP consiste à utiliser les protothreads [5], qui fournissent une primitive de blocage sans processus légers. Cela impose cependant plusieurs contraintes au programmeur du service, comme l'impossibilité d'utiliser des variables locales, des constructions de type *switch-case*, etc.

Enfin, chez µIP comme chez dynaWeb, si un contenu dynamique est perdu et qu'une retransmission est nécessaire, les données sont à nouveau générées, car les paquets à retransmettre ne sont pas conservés en mémoire (voir section 3.3.1). Le fait d'appeler un service plus de fois qu'il n'a été requis pas le client peut parfois poser problème (il suffit d'imaginer un service qui débite un compte...). Dans certains cas, des appels multiples n'auront aucun effet de bord et fourniront toujours les mêmes résultats (on parle dans ce cas d'idempotence de la fonction appelée). Il est aussi possible qu'une fonction non idempotente supporte très bien des appels multiples pour un même service. Dans certains cas, comme par exemple un capteur qui renseigne une donnée capturée, il est même préférable de fournir une valeur obtenue le plus tardivement possible, donc de re-générer la réponse autant de fois qu'il y aura de retransmissions.

Ce constat pousse à se poser la question de « l'API idéale », qu'il serait intéressant d'établir pour les services fournis par dynaWeb. On pourrait par exemple proposer au programmeur d'un service d'indiquer, via l'API de services de dynaWeb, s'il souhaite :

- interdire des appels multiples du service en cas de retransmission ;
- autoriser ces appels multiples si la pile est en manque de mémoire ;
- forcer un nouvel appel à chaque retransmission pour une envoyer un contenu mis à jour.

4.4 Évaluation des performances

4.4.1 Service d'un fichier statique volumineux

Test réalisé

Pour comparer les performances de dynaWeb à celles des autres serveurs, commençons par les mettre à l'épreuve sur le service d'un fichier statique relativement volumineux. Le fichier choisi est une image de 55.9 ko.

On teste les trois serveurs Web en y accédant depuis une machine sous Linux, non encore connectée au serveur (pour que l'ouverture et la fermeture de connexion soit aussi incluse dans les mesures de dynaWeb, capable de fonctionner en HTTP 1.1). Pour mieux comprendre l'impact de la MSS utilisée, les mesures ont été réalisées en faisant varier la MTU de 300 octets à 1500 octets. La figure 4.1 montre l'évolution du temps nécessaire au service du fichier pour les trois serveurs en fonction de la MTU utilisée.

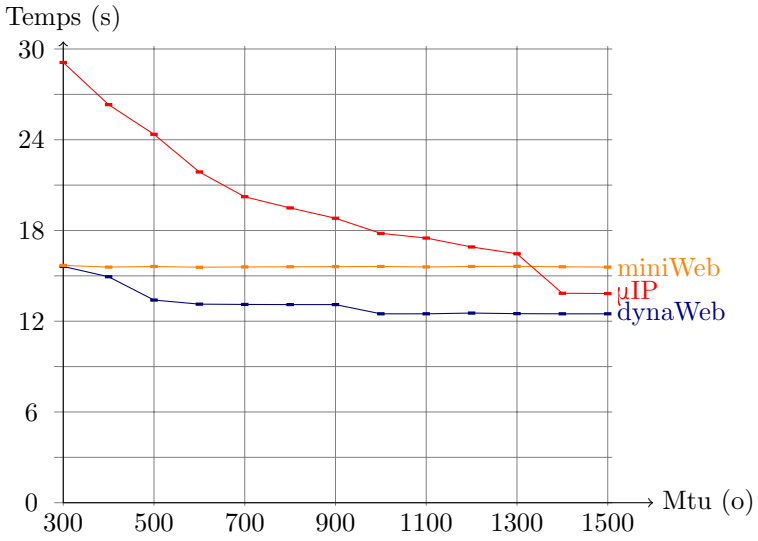

FIG. 4.1 – Service d'un fichier de 55.9 ko

Analyse des résultats

Ces mesures sont, quelle que soit la MTU utilisée, en faveur de dynaWeb. Avec une petite MTU, miniWeb obtient des performances similaires à dynaWeb. Les performances de miniWeb sont constantes car il conserve une MSS de 200 quelle que soit la valeur proposée par le client.

On constate aussi qu'avec une petite MTU, µIP est très loin derrière les deux autres serveurs. Dans cette configuration, le nombre de paquets nécessaire à la transmission du fichier est le plus élevé. Cet écart de performances s'explique par la limitation de µIP à un paquet en vol, déclenchant la politique des accusés différés chez le client. En augmentant la MSS, µIP se rapproche de dynaWeb, mais ne parvient pas à le rattraper.

La politique de gestion des accusés différés, sous Linux, est un peu particulière. Elle ne se déclenche qu'après un certain nombre de paquets échangés sur une même connexion. C'est ce qui explique le palier dans les performances de µIP. À partir d'une MTU de 1400, le nombre de paquets envoyés n'est pas suffisant pour déclencher les accusés différés. Même dans ces conditions, µIP n'atteint pas les performance de dynaWeb, mais dépasse celles de miniWeb (à la MSS limitée). Les deux raisons principales à cela sont le fait (i) que la limitation à un paquet en vol est un handicap, même sans accusés différés, (ii) que l'implémentation monolithique réalisée par dynaWeb permet d'améliorer les performances.

Il est important de rappeler que µIP, contrairement à dynaWeb, occupe d'autant plus de mémoire que la MSS utilisée est grande. Au vu de sa grande baisse de performances observée avec une petite MSS, on est tenté de sacrifier de la mémoire, qui, pourtant, est la ressource la plus contrainte de nos cibles.

4.4.2 Évaluation d'une application de type AJAX

Application testée

Il est intéressant de mesurer les performances de µIP et dynaWeb sur une application de type AJAX, conforme à notre motivation initiale. 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 la GBA. Le contenu alors généré indique l'état des boutons de la GBA. La page affiche ainsi, en temps réel, l'ensemble des boutons enfoncés de la console portable. Pour cela, un fichier HTML de 1.9 ko est utilisé. Il requiert un fichier *JavaScript* de 1.7 ko. Les informations *CSS* sont contenues dans le fichier HTML. Le contenu dynamique servi périodiquement n'est que de deux octets. La figure 4.2 donne un aperçu de la page servie par la GBA.

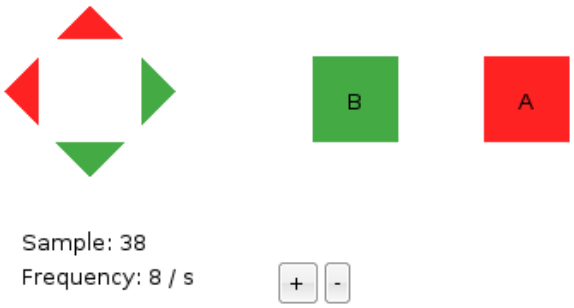

Fig. 4.2 – Page AJAX servie par la GBA

Analyse des résultats

Les résultats mesurés pour les trois serveurs sont présentés dans le tableau 4.2. Ils ont été réalisés en utilisant une MTU de 1500 octets. DynaWeb utilise un tampon en réception de 100 octets, et un tampon global (pour récupérer l'URL requise et pour générer les contenus dynamiques) de 50 octets.

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

Tab. 4.2 – Performances mesurées sur l'application AJAX

La première colonne donne le temps de chargement initial, incluant l'accès au fichier `index.html` via une première connexion TCP et l'accès au fichier *JavaScript* `funs.js` sur une seconde connexion. En effet, comme l'envoi du fichier d'index nécessite plusieurs paquets, le client prend l'initiative d'ouvrir une seconde connexion pour commencer le chargement du fichier de script avant la fin de la réception de l'index. Le temps nécessaire à la réalisation de cette phase est légèrement plus court pour dynaWeb. Ici, l'écart n'est pas énorme car les fichiers échangés sont trop petits pour déclencher l'utilisation des accusés différés par le

client. Le gain de performances de dynaWeb est dû à d'autres facteurs, tels que la réduction du coût de traitement et l'envoi de plusieurs paquets en vol.

On constate qu'au niveau du temps de service du contenu dynamique, dynaWeb devance largement µIP. La principale cause de cette avance est l'utilisation de HTTP 1.1 qui allège beaucoup le traffic. Ce gain est d'autant plus flagrant que le contenu à servir est petit. Pour exclure l'impact de HTTP 1.1, nous avons aussi mesuré, chez µIP, le délai écoulé entre l'émission de la requête GET et la réception de sa réponse (excluant ainsi les poingnées de mains superflues). La comparaison reste largement à l'avantage de dynaWeb, qui sert le contenu en 112 ms au lieu de 199 pour µIP.

Le temps de réponse impacte directement sur la fréquence d'échantillonage qui peut être supportée par le serveur. DynaWeb permet ainsi 8 échantillonnages par seconde, contre 3 pour µIP.

4.4.3 Consommation mémoire

La ressource la plus contrainte de nos cibles est la mémoire. Son optimisation est visée par la pile µIP, miniWeb, et par notre proposition, dynaWeb. Un comparatif de l'espace mémoire occupé par le code exécutable, les variables globales et la pile d'exécution pour les trois serveurs étudiés est proposé dans le tableau 4.4.3. Ces mesures ont été réalisées en réduisant la consommation mémoire au minimum. Les tampons de µIP sont réduis à leur plus petite taille, tandis que miniWeb et dynaWeb fonctionnent sans tampon.

Serveur Web	Taille du code exécutable	Taille des variables globales	Taille occupée sur la pile
µIP	44.5 ko	11.2 ko	204 o
miniWeb	6.4 ko	88 o	268 o
dynaWeb	10.4 ko	176 o	208 o

Tab. 4.3 – Comparatif des consommations mémoire des serveurs étudiés

Ces mesures montrent que µIP, une pile IP minimaliste, nous fournit, avec son serveur Web, la solution qui est de loin la plus lourde en mémoire. Sa consommation est incomparable à celle des deux autres serveurs, et augmente rapidement avec la MSS maximale autorisée. Ce point montre qu'une forte intégration et une implémentation monolithique des protocoles permet une économie de mémoire considérable. Cette économie est aussi due à l'aspect totalement généraliste de la pile µIP, par opposition aux piles IP dédiées des deux autres serveurs.

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.

4.4.4 Fonctionnalités fournies

Le tableau 4.4.4 récapitule l'ensemble des fonctionnalités implémentées par les trois serveurs. La pile µIP propose la gestion d'autres protocoles que TCP et la gestion de données urgentes inutilisées dans le cadre d'un serveur Web. Les options TCP sont également prises en compte par cette pile, ce qui n'est pas nécessaire pour un simple service Web. Les faiblesses les plus importantes du serveur Web de µIP sont la limitation à un paquet en vol et l'utilisation de HTTP 1.0, qui nuisent considérablement à ses performances.

Fonctionnalité	µIP	miniWeb	dynaWeb
Options IP	–	–	–
Ré-assembage des fragments IP	√	–	–
UDP	–	–	–
ICMP	√	–	–
Options TCP	√	–	–
Données urgentes TCP	√	–	–
Stockage des données pour retrans.	–	–	–
Estimation de RTT	√	–	–
Requêtes POST	–	–	–
Lecture de l'URL requise	√	–	√
MSS de TCP variable	√	–	√
Connexions TCP multiples	√	–	√
Plusieurs paquets en vol	–	√	√
HTTP 1.1	–	–	√
Contenus dynamiques de taille bornée	√	–	√
Contenus dynamiques de taille non bornée	√	–	–
Pré-calcul des checksum	–	√	√

Tab. 4.4 – Fonctionnalités proposées par les trois serveurs Web

Comme nous l'avons remarqué à de nombreuses reprises, les fonctionnalités offertes pas le serveur miniWeb le rendent inutilisable pour une application réelle.

Les fonctionnalités requises par nos objectifs initiaux (voir section 1.4) 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.

4.5 Bilan des apports réalisés

L'implémentation de dynaWeb a permis de valider nos propositions et d'en mesurer les impacts. On a ainsi montré qu'il était possible de fournir un grand nombre de fonctionnalités tout en étant très économe en terme de mémoire. MiniWeb et µIP n'avaient pas su concilier ces deux points.

On a 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, révélant des solutions à des des biais qui ne peuvent être résolus avec une implémentation classique par couches.

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.

DynaWeb est un serveur Web complet et adapté au comportement des clients classiquement rencontrés (utilisant par exemple les accusés différés). Ses performances sont supérieures à celles des autres serveurs testés, et sa consommation mémoire est réduite.

Conclusion et perspectives

Dans ce mémoire, nous avons tout d'abord effectué une analyse détaillée des points sensibles pour la conception de serveurs Web embarqués. Nous avons réalisé des portages et de nombreuses évaluations de performances.

Nous avons alors proposé de nouvelles solutions pour améliorer les performances et réduire les coûts de ces serveurs. L'approche classique par couches de communication engendre un ensemble de surcoûts non acceptable pour l'embarqué.

L'implémentation de nos propositions – dynaWeb – nous a permis d'en prouver la validité et d'en évaluer l'apport.

Nous avons montré que l'utilisation d'une approche transversale pour l'implémentation d'un serveur Web embarqué permet une amélioration nette des performances et une consommation réduite en ressources.

Cette approche transversale ouvre la porte à de nouvelles stratégies permettant, là encore, de gagner sur le plan des performances et de la consommation.

De nombreuses perspectives s'offrent désormais à nous. Il serait intéressant d'étendre notre approche à l'utilisation d'autres protocoles, par exemple en permettant la gestion de UDP et ICMP. Ces protocoles augmenteraient les possibilités d'interaction du système embarqué ciblé.

Il serait également intéressant de permettre à notre serveur d'établir des connexions HTTP vers des sites distants. Des applications tissant des informations locales à des informations distantes seraient ainsi envisageables. La gestion d'une connexion sécurisée utilisant SSL est aussi un point à étudier, nécessaire pour certains types d'applications.

Le service de contenus dynamiques est une piste que l'on souhaite également approfondir. L'utilisation de servlets java permettrait de rendre encore plus dynamiques les applications Web à embarquer. De nombreux axes restent à étudier, comme l'élaboration d'une API idéale pour servlets embarqués.

Enfin, nous souhaitons étendre nos travaux au cas de systèmes utilisant une interface sans fil (*e.g.*, ZigBee, ...). Dans un réseau de capteurs, une approche utilisant une passerelle entre le « monde extérieur » et le réseau permettrait de nouvelles optimisations (à l'intérieur du réseau, les normes classiques ne sont ainsi plus imposées).

Bibliographie

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. Rfc 1945 : Hypertext transfer protocol — http/1.0, May 1996.

[2] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *Communications Magazine, IEEE*, 40(5) :94–101, 2002.

[3] 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.

[4] A. Dunkels. The proof-of-concept miniweb tcp/ip stack, 2005. `http://www.sics.se/~adam/miniweb/`.

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads : simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06 : Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[6] A. Dunkels, T. Voigt, and J. Alonso. Making tcp/ip viable for wireless sensor networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, Jan. 2004.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616 : Hypertext transfer protocol — http/1.1. Internet Engineering Task Force : RFC 2616, June 1999.

[8] 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.

[9] 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.

[10] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *SIGCOMM Comput. Commun. Rev.*, 25(1) :66–74, 1995.

[11] J. Postel. Rfc 791 : Internet protocol, Sept. 1981.

[12] J. Postel. Rfc 793 : Transmission control protocol, Sept. 1981.

[13] 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.

[14] P. Sarolahti, M. Kojo, and K. Raatikainen. F-rto : an enhanced recovery algorithm for tcp retransmission timeouts. *SIGCOMM Comput. Commun. Rev.*, 33(2) :51–63, 2003.

[15] S. Shon. Protocol implementations for web based control systems. *International Journal of Control, Automation, and Systems*, 3 :122–129, March 2005.

[16] R. Sridharan, R. Sridhar, and S. Mishra. Poster : A robust header compression technique for wireless ad hoc networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(3) :23–24, 2003.

[17] W. Stevens. Rfc 2001 : Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, 1997.

[18] G. Xylomenos, G. Polyzos, P. Mahonen, and M. Saaranen. Tcp performance issues over wireless links. *IEEE Communications Magazine*, 39(4) :52–58, 2001.