N° d'ordre : 40306

Thèse

présentée le 19 juillet 2010
pour l'obtention du titre de
Docteur de l'Université Lille 1 – Sciences et Technologies
(spécialité informatique)

par
Simon Duquennoy

Smews : un système d'exploitation dédié au support d'applications Web en environnement contraint

Composition du jury :

Présidente :	Christine Morin	IRISA, Université de Rennes 1
Rapporteurs :	Didier Donsez	LIG, Université Joseph Fourier – Grenoble 1
	Pierre Sens	LIP6, Université Pierre et Marie Curie – Paris 6
Examinateurs :	Laurent Réveillère	LaBRI, Institut Polytechnique de Bordeaux
	Jean-Jacques Vandewalle	CNFPT, École Nationale Supérieure des Sciences de l'Information et des Bibliothèques
Directeur :	Gilles Grimaud	LIFL, Université Lille 1 – Sciences et Technologies

« L'art est toujours le résultat d'une contrainte. Croire qu'il s'élève d'autant plus haut qu'il est plus libre, c'est croire que ce qui retient le cerf-volant de monter, c'est sa corde. »

André Gide

Remerciements

Je tiens en premier lieu à remercier Hélène pour son soutien au quotidien durant les trois années de vie que représentent cette thèse. Sa patience, sa compréhension et sa confiance ont été plus que précieuses face à mes nuits de rédaction et mes périodes de doute.

J'ai eu la chance de réaliser cette thèse sous la direction de Gilles GRIMAUD. Je le remercie pour sa disponibilité, ses conseils avisés, son ouverture d'esprit et ses qualités humaines. Sa capacité à remettre en question ce qui semble acquis pour beaucoup est sans doute la clé de sa créativité ; elle aura été à l'origine de nombreux débats autour d'un verre (ou deux ?) aux sujets aussi variés que passionnants.

Merci à Jean-Jacques VANDEWALLE pour son aide et sa rigueur scientifique. Son éclairage en tant que ≪ l'industriel de service ≫ m'a souvent permis de voir ce qui n'apparaît pas sans recul ni expérience.

Je remercie les membres de mon jury pour avoir accepté d'y prendre part ainsi que pour l'attention qu'ils ont prêtée à mon travail. Merci donc à Didier DONSEZ et Pierre SENS d'avoir rapporté mon manuscrit. Merci Laurent RÉVEILLÈRE d'avoir examiné ma thèse et à Christine MORIN d'en avoir présidé le jury.

Merci également à Isabelle SIMPLOT-RYL d'avoir dirigé ma thèse durant les deux premières années de celle-ci. Merci à David SIMPLOT-RYL de m'avoir accueilli au sein de l'équipe POPS.

La majeure partie des résultats expérimentaux présentés dans cette thèse ont été obtenus avec le projet Smews. Je remercie les étudiants, stagiaires et doctorants qui y ont contribué.

Je remercie Amaury ≪ Copain ≫ pour sa bonne humeur et son amitié, Thomas pour les discussions que nous avons eues sur la viabilité du pigeon voyageur en tant que couche liaison du modèle OSI, Geoffroy pour sa confiance et son regard critique ainsi que Arnaud ou le guide du thésard en phase terminale. Merci également à ceux qui avec qui j'ai partagé repas, sorties, ou discussions autour d'un café : Alex, Christophe, Dorina, Fadila, Grégory, Jean-François, Jean-Philippe, Jovan, Julien, Kévin, Loïc, Marie-Émilie, Michaël, Milan, Nadia, Nicolas, Raphaël, Roudy, Tahiry, Yoann. J'ai une pensée particulière pour ceux d'entre eux qui sont en train de rédiger... courage !

Je remercie chaleureusement mes parents Paulette et Jacques ainsi que Julien et Adeline, pour leur soutien, leur présence et leur écoute. Enfin, merci à mes amis, ma famille et à ceux qui m'ont entouré pendant ces trois années.

Tack så mycket.

Table des matières

Remerciements **5**

1 Introduction **11**
1.1 Contexte 11
1.2 Thèse 11
1.3 Structure du document 12

2 État de l'art **15**
2.1 Systèmes d'exploitation embarqués 15
2.1.1 Définitions 15
2.1.2 Des techniques diverses pour des besoins variés 17
2.1.3 Structures des systèmes d'exploitation 19
2.2 Communication dans les matériels contraints 22
2.2.1 Adaptation des standards IP 22
2.2.2 Piles IP embarquées 24
2.2.3 Serveurs Web embarqués 26
2.3 Performance des serveurs d'application 29
2.3.1 Architectures de serveurs Web 29
2.3.2 Notification en contexte Web 31
2.3.3 Ordonnancement des requêtes 33
2.4 Synthèse 36

3 Vers un macro-noyau dédié au Web des objets **37**
3.1 Contexte – le Web des objets 37
3.1.1 L'Internet des objets et les réseaux personnels 37
3.1.2 Vers un Web des objets 39
3.1.3 Du Web mondial au Web des Objets 40
3.2 Problématique – un défi de taille 41
3.2.1 Des équipements aux ressources limitées 42
3.2.2 Des applications réseau exigeantes 43
3.2.3 Allier compacité, fonctionnalités et performances 46
3.3 Approche – vers un système à macro-noyau dédié 46
3.3.1 Systèmes dédiés à une famille d'applications 46

3.3.2 Vers un système à macro-noyau dédié 48
3.3.3 Concevoir un système à macro-noyau dédié au Web des objets 49

4 Analyse du support applicatif dédié 51

4.1 Analyse du trafic 51
4.1.1 Contexte de l'analyse 51
4.1.2 Modèle de trafic 52
4.1.3 Identification des facteurs impactants 54
4.1.4 La notion de charge mémoire 55
4.2 Taxinomie des ressources Web 57
4.2.1 Critère A – Taille des données 58
4.2.2 Critère B – Persistance des données 59
4.2.3 Critère C – Schéma d'interaction 60
4.3 Stratégies adaptées aux propriétés applicatives 61
4.3.1 Stratégies d'émission des données 61
4.3.2 Gestion des schémas d'interaction 66
4.4 Conclusion 68

5 Conception du macro-noyau 69

5.1 Un macro-noyau évènementiel 69
5.1.1 Description conceptuelle : l'omniprésence des évènements 69
5.1.2 Description opérationnelle : l'architecture du noyau 71
5.1.3 Optimisations permises par la construction transversale 73
5.1.4 Mesure des bénéfices 76
5.2 Continuations légères et ré-invocables 77
5.2.1 Fils d'exécution et mémoire consommée 78
5.2.2 Caractérisation des tailles des contextes 79
5.2.3 Coroutines à pile partagée 80
5.3 Ordonnancement des requêtes 83
5.3.1 Identification des métriques appropriées 84
5.3.2 Un compromis entre rapidité et partage des ressources 86
5.3.3 Considération affinée de la charge mémoire 90
5.4 Conclusion 92

6 Mise à l'épreuve expérimentale 93

6.1 Smews : un macro-noyau pour un micro-serveur 93
6.1.1 Présentation de Smews 93
6.1.2 Interface applicative 95
6.2 Consommation mémoire et performances 95
6.2.1 Empreinte Mémoire 97
6.2.2 Résultats à l'exécution 99
6.3 Schémas d'interaction et passage à l'échelle 104
6.3.1 Description des expérimentations 104

6.3.2 Résultats . . . 105
6.3.3 Synthèse . . . 109
6.4 Conclusion . . . 111

7 Conclusion et perspectives 113
7.1 Synthèse . . . 113
7.2 Résumé des contributions et publications . . . 114
7.3 Perspectives . . . 114

Bibliographie personnelle 117

Bibliographie 119

Liste des sigles et acronymes 127

Liste des figures 129

Liste des tableaux 131

A Compléments sur le protocole TCP 133

B Interface applicative de Smews 135

C Résultats expérimentaux sur la notification d'évènements 137

Premier chapitre

Introduction

Nous introduisons ici le contexte de ce travail, nous présentons la thèse défendue et décrivons la structure de ce mémoire, afin d'en guider la lecture.

1.1 Contexte

Les progrès technologiques permanents ont conduit ces dernières années à une généralisation de l'informatique ubiquitaire. Constituée de petits équipements autonomes, son ampleur n'a cessé d'augmenter. Les téléphones portables, assistants personnels, GPS, cartes à puces, capteurs ou systèmes de domotique n'en sont que quelques exemples. À l'heure où Internet est passé dans les mœurs, on vise une interconnexion entre ces objets, un *Internet des objets*. Une solution fournissant aux utilisateurs un accès naturel à cette informatique enfouie consiste à y transposer les standards du Web. On parle alors du *Web des objets*. Puisque ces technologies ont été la source du succès d'Internet, pourquoi ne pas les appliquer aux équipements qui nous entourent ? Il suffit pour cela d'y déployer une pile de communication standard supportant le service d'applications Web. On apporte alors à ces matériels l'interopérabilité dont ils ont besoin, au niveau réseau comme applicatif.

Cependant, les standards du Web ont été conçus pour Internet. Ils sont adaptés aux besoins de ce réseau mondial, aux capacités de ses serveurs et de ses infrastructures. Le déploiement de serveurs d'applications Web au sein d'équipements aux fortes contraintes de coût tels qu'une carte à puce ou un capteur forme alors un véritable verrou technologique. Les solutions qui permettent le développement d'un tel logiciel sont fondées soit sur une architecture intégrée conçue comme un tout, soit sur un système d'exploitation généraliste. Aucune de ces deux solutions n'est satisfaisante. Un système intégré est potentiellement optimal mais est à usage unique, il est fermé au support d'applications et souffre en pratique de limitations fonctionnelles pour des raisons évidentes de génie logiciel. Un système d'exploitation généraliste est simplement trop générique pour atteindre les objectifs de compacité et de performances requis.

1.2 Thèse

La thèse défendue dans ce mémoire propose une alternative aux deux approches opposées que sont les systèmes intégrés et les systèmes d'exploitation généralistes. Afin d'allier performance, compacité et richesse fonctionnelle, nous présentons une solution dans laquelle le système

d'exploitation est dédié à une famille d'applications de haut niveau. Nous proposons aux applications une interface correspondant précisément à leurs besoins, supportée par un *macro-noyau* optimisé depuis la gestion du matériel jusqu'aux couches hautes du logiciel.

Dans le cas du *Web des objets*, il s'agit de présenter un support dédié aux applications Web et d'intégrer au noyau la pile IP, le serveur et le conteneur d'applications. À la manière d'un langage spécifique à un domaine, l'objectif de notre approche est de guider la conception de logiciel par la contrainte et d'en assurer un support efficace puisque spécialisé et conçu par des experts du domaine d'application.

Cette thèse connait *a priori* deux faiblesses largement soulignées par les travaux scientifiques antérieurs :

Extension verticale du noyau L'intégration des couches hautes du logiciel au sein du noyau conduit à une extension de ce dernier. Pour des raisons de criticité et de maintenabilité, on cherche habituellement à en minimiser la taille ;

Spécialisation du développement Il est nécessaire de concevoir un système dédié pour chaque famille d'applications visée, nécessitant à chaque fois un travail nouveau.

Nous pensons que ces problèmes peuvent être soit minimisés, soit compensés par les qualités du système résultant. Nous attendons les bénéfices suivants :

Efficacité du noyau En concevant le noyau du système dédié à partir d'une connaissance précise du domaine d'application et en supprimant toute interface applicative générique, nous espérons obtenir des performances élevées tout en minimisant la consommation de ressources ;

Richesse du support applicatif En incluant dans le noyau le conteneur d'applications, nous comptons fournir de nombreuses fonctionnalités de manière optimale et présenter aux applications une interface dédiée, guidant la construction d'applications riches et efficaces, factorisant les traitements critiques.

Dans ce mémoire, nous appliquons cette approche au cas du *Web des objets*. Nous détaillons les étapes de la construction et évaluons les bénéfices d'un système d'exploitation dédié au support d'applications Web en environnement contraint.

1.3 Structure du document

Cette thèse se décompose en sept chapitres, incluant la présente introduction.

Le **chapitre 2** présente un état de l'art des systèmes d'exploitation embarqués, des équipements mobiles communicants et des serveurs d'application. Il en analyse l'architecture et en discute l'applicabilité dans le contexte de serveurs d'applications Web embarqués.

Le **chapitre 3** présente le contexte de ces travaux puis, en relation avec l'état de l'art, extrait la problématique adressée dans ce mémoire. Enfin, il détaille la notion de système d'exploitation à macro-noyau dédié. Il en discute les avantages et inconvénients attendus, en comparaison aux architectures classiques.

Le **chapitre 4** définit l'interface entre noyau et applications. Pour cela, il introduit un modèle de trafic et une taxinomie des ressources constituant les applications Web. Il présente une nouvelle métrique, la *charge mémoire*, dont le but est de synthétiser la performance et la consommation de mémoire du système. En partant de cette analyse, il décrit des stratégies internes à la pile de communication permettant l'émission efficace de données adaptée aux propriétés des applications.

Le **chapitre 5** se focalise sur les mécanismes internes du macro-noyau. Nous en décrivons l'architecture, mais aussi l'intégration des mécanismes de haut niveau en son sein. Le problème du support à bas coût de multiples tâches applicatives est ensuite discuté. Il est notamment nécessaire, pour le support des stratégies que nous avons définies, de gérer des continuations ré-invocables. Nous nous intéressons enfin à l'ordonnancement des requêtes, dans l'optique de maîtriser performances, charge mémoire et équité du service offert aux clients.

Le **chapitre 6** décrit l'application réelle et les bénéfices de notre approche et de nos propositions, par l'intermédiaire de notre prototype, Smews. La taille et la composition du noyau, la consommation de ressources, les performances et la charge mémoire sont alors mesurés. Nous discutons ensuite du problème de la notification d'évènements, particulièrement exigeante en termes de passage à l'échelle. L'objectif est ici d'évaluer si les bénéfices offerts par ce système d'exploitation dédié sont à la hauteur des sacrifices qu'il exige.

La **chapitre 7** résume les contributions de ces travaux, en discute les limites puis en ouvre les perspectives.

Second chapitre

État de l'art

Les systèmes embarqués, de par leurs contraintes particulières, constituent un domaine de l'informatique à part entière pour lequel les techniques classiques sont souvent inadaptées et remplacées par des solutions novatrices. Dans ce chapitre, nous présentons et analysons tout d'abord les approches existantes pour la construction de systèmes d'exploitation embarqués. Ensuite, nous nous intéressons aux systèmes embarqués mis en réseau et aux solutions qui existent pour y transposer les protocoles réseau de l'Internet. Nous présentons alors les travaux relatifs à la performance des serveurs d'application et nous concluons en en discutant l'applicabilité dans le contexte de serveurs d'applications Web embarqués.

2.1 Systèmes d'exploitation embarqués

Le système d'exploitation joue un rôle majeur dans la manière dont un système informatique est programmé. En contexte embarqué, il existe une très grande variété de systèmes d'exploitation présentant des constructions et des propriétés particulières, répondant à des objectifs très variés.

2.1.1 Définitions

Nous définissons ici deux notions nécessaires à la compréhension de cet état de l'art. Tout d'abord, nous nous intéressons aux *systèmes embarqués* et en particulier à la famille des POPS (Petits Objets Portables et Sécurisés). Ensuite, nous définissons la notion de *système d'exploitation* en contexte embarqué.

Systèmes embarqués

Les systèmes embarqués, ou systèmes enfouis, sont des systèmes informatiques aux propriétés particulières. Des systèmes de contrôle d'un avion aux bornes automatiques, en passant par les systèmes de guidage de missiles, les équipements médicaux ou l'électroménager, leurs domaines d'application sont nombreux et leurs objectifs divers. Ils disposent de matériels aux caractéristiques hétérogènes. Les techniques y permettant le développement et le déploiement de logiciel sont très différentes de celles utilisées dans l'informatique personnelle. S'ils n'ont pas de définition largement acceptée, on peut tout de même lister un ensemble de propriétés qui leur sont couramment associées :

Spécialisation à un ensemble de tâches Contrairement aux ordinateurs personnels, les systèmes embarqués sont le plus souvent dédiés à un ensemble figé de tâches identifiées dès la conception ;

Contraintes matérielles Les contraintes de coûts de production, de compacité et d'autonomie se répercutent sur le matériel constituant les systèmes embarqués. Il est courant que la matériel et le logiciel soient conçus conjointement ;

Criticité des applications Les systèmes embarqués sont au service de l'équipement dans lequel ils sont intégrés. Les tâches qu'ils exécutent sont souvent critiques et leur dysfonctionnement peut avoir des conséquences sur l'environnement dans lequel ils sont déployés ;

Accessibilité limitée Les systèmes embarqués sont difficiles voir impossibles d'accès car enfouis dans l'équipement qu'ils contrôlent. La maintenance de ces derniers est ainsi rendue difficile, coûteuse, et nécessite une expertise particulière.

Dans ce mémoire, nous nous intéressons à un ensemble particulier de systèmes embarqués, les POPS. Ces derniers visent ainsi à constituer une informatique ambiante massivement déployée, dans laquelle chaque individu est connecté à un grand nombre de systèmes informatiques. Cette vision succède à celle de l'informatique personnelle, qui elle même a remplacé celle des serveurs centralisés partagés : le nombre de machines au service d'un individu n'a ainsi cessé d'augmenter.

Les POPS sont destinés à une production de masse à très bas coût, c'est pourquoi leur matériel est aussi contraint que ne le permet le logiciel qu'ils embarquent. Dans ce contexte, les conséquences de la loi de Moore (qui prédit une augmentation exponentielle de la densité des transistors sur une plaquette de silicium) ne sont pas utilisées au profit de la puissance des matériels, mais au profit de leur nombre.

Ainsi, l'informatique ambiante constituée par les POPS représente déjà un volume de production impressionnant, devançant largement celui de l'informatique personnelle. Ainsi, en 2008, le nombre de cartes à puces vendues s'élève à 5 milliards [1], à comparer à 0,15 milliards ordinateurs portables [2]. En 2002, [Utr02] les microprocesseurs ont été diffusés au coût moyen de 6$ et étaient destinés à l'embarqué pour 98 % d'entre eux ; les processeurs 16 et 32 bits représentant moins de 20 % du volume total.

Systèmes d'exploitation en contexte embarqué

Habituellement, les notions de système d'exploitation, de noyau et d'application sont définies en s'appuyant sur la nature de l'édition des liens entre logiciels et/ou les différents niveaux de privilèges du microprocesseur. Les pratiques dans le domaine des systèmes embarqués sont très variées et diffèrent sensiblement de celles des ordinateurs personnels. Par exemple, dans des systèmes d'exploitation destinés à l'embarqué comme TinyOS [LMP+05], Contiki [DGV04] ou Mantis OS [BCD+05], l'édition des liens entre applications et noyau est principalement réalisée de manière statique. Notons également qu'une part importante des microprocesseurs embarqués ne disposent pas de différents niveaux de privilèges. Dans les systèmes destinés à ce type de processeurs, l'isolation des processus est soit inexistante, soit assurée par une mécanisme alternatif (comme dans t-kernel [GS06]).

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

Afin d'éviter toute ambiguïté due à la notion de système d'exploitation dans un contexte embarqué, nous proposons de se fonder sur les définitions suivantes :

Système d'exploitation Logiciel en charge de supporter les fonctionnalités de base des applications d'un système informatique. Ses rôles incluent par exemple la gestion du matériel, le partage du microprocesseur, l'isolation de processus, la prise en charge de protocoles réseaux, *etc.* ;

Noyau Base logicielle d'un système d'exploitation fournissant un sous-ensemble ou la totalité des services du système d'exploitation ;

Application Logiciel situé au-dessus du système d'exploitation, s'appuyant sur les routines fournies par ce dernier pour accomplir diverses tâches.

Ces définitions laissent beaucoup de liberté en ce qui concerne les fonctionnalités fournies par noyau, système d'exploitation et applications. Dans le domaine de l'informatique embarquée, chaque champ d'application connait des solutions très variées à ses problèmes spécifiques.

2.1.2 Des techniques diverses pour des besoins variés

La grande variété des besoins des systèmes embarqués a permis l'émergence de techniques pour la construction de systèmes d'exploitation répondant à des besoins très précis. Nous en détaillons ici les exemples les plus pertinents dans le cadre de ce mémoire.

Systèmes temps réel

Les systèmes temps réel sont des systèmes critiques, souvent en charge de contrôler un procédé physique, exécutant des tâches sous contraintes temporelles dites strictes ou souples. Ils sont par exemple utilisés dans les industries de production, l'aéronautique ou la robotique. Ils peuvent être programmés sans système d'exploitation, à l'aide d'outils dédiés permettant de maîtriser les temps d'exécutions au pire cas. Les systèmes d'exploitation temps réel (ou RTOS [3]) répondent aux besoins des systèmes temps réel en permettant aux applications de spécifier leurs contraintes temporelles et en utilisant des techniques d'ordonnancement dédiées.

Parmi les systèmes d'exploitation temps réel les plus connus et couramment utilisés, on citera VxWorks [Riv87], QNX [SD95], Linux ou Windows CE. D'autres RTOS sont spécialement conçus pour les systèmes embarqués aux ressources très limitées, tels que les POPS. Ainsi, μC/OS-II [Lab98] ou [Bar09] fournissent un jeu de fonctionnalités minimal et sont principalement constitués de leur ordonnanceur. Ils ne fournissent pas nécessairement une gestion de tout le matériel sur lequel ils sont déployés, si bien que certaines applications pilotent directement certaines parties du matériel (interfaces de communication, éventuels périphériques). Ces systèmes peuvent être utilisés dans des systèmes ne disposant que de quelques dizaines de kilo-octets de mémoire persistante.

Systèmes pour cartes à puce

Les cartes à puce sont des petits systèmes informatiques disposant le plus souvent d'un microprocesseur et en charge de gérer des données personnelles et/ou sensibles. Ces systèmes bénéficient de techniques de sécurisation à la fois matérielles et logicielles, empêchant l'accès aux données critiques, la duplication ou la falsification. Les caractéristiques techniques des

3. RTOS : *Real-Time Operating System*

cartes à puces sont calquées sur les besoins du logiciel qu'elles embarquent. Afin d'en réduire le coût de production, la seule manière de les programmer a longtemps été d'utiliser des systèmes entièrement intégrés, capables de s'exécuter en environnement matériel très contraint. Aujourd'hui, les cartes bancaires sont encore souvent conçues de cette manière.

La technologie Java Card [Che00], apparue en 1996, propose une alternative pour la programmation des cartes à puce, aujourd'hui largement utilisée dans les cartes SIM [4] des téléphones portables. Les cartes de ce type embarquent une petite machine virtuelle exécutant du *bytecode* Java Card, une variante de java dédiée à l'embarqué. Ainsi, les applications Java Card (ou *applets*) sont portables et diffusables après la phase de déploiement. Cette spécialisation à l'exécution d'*applets* est typique de nombreux systèmes embarqués, spécialisés à un type de tâches.

Rendue publique en 2008, la nouvelle version Java Card 3.0 *connected* est tournée vers les technologies du Web. En plus de fournir une machine virtuelle, la carte puce exécute un serveur Web et gère une pile de communication TCP/IP. Les applications ne sont plus des *applets* mais des *servlets*, ce qui en améliore considérablement l'interopérabilité. Ces nouvelles cartes entrent dans le cadre du Web des objets, qui constitue le contexte de ce mémoire.

Systèmes pour capteurs de terrain

Les réseaux de capteurs sans fil sont des réseaux *ad hoc* faits de nœuds autonomes au matériel très contraint et au fort besoin d'autonomie. Ils ont conduit à de nouvelles recherches pour l'élaboration de systèmes d'exploitation poids plume à fortes exigences. TinyOS [LMP+05] est un des premiers systèmes ciblant particulièrement ces nœuds. Il permet la construction de systèmes par assemblage statique de composants. Les applications et les modules du système sont écrits en nesC [GLvB+03], extension du langage C avec orientation composants pour systèmes contraints. Les objectifs de TinyOS incluent le support de matériels très limités, l'exécution réactive de multiples tâches, la flexibilité à la construction et une faible consommation énergétique. Le système SOS [HKS+05], également fondé sur une approche à composants, permet quant à lui le chargement et déchargement dynamique de modules lors de l'exécution, à l'aide de code indépendant de toute position.

Dans les systèmes embarqués tels que les capteurs, disposant de seulement quelques kilooctets de mémoire, la pile d'exécution des tâches consomme une part importante de la mémoire disponible. Le plus souvent, une unique tâche est utilisée, dans laquelle une boucle principale attend et gère des évènements en appelant diverses routines ou applications. Ce type de système, dit évènementiel, est difficile à écrire et à maintenir car son code source ne reflète pas directement son flot d'exécution. On l'oppose aux systèmes *thread*és, dans lesquels chaque tâche dispose de sa propre pile d'exécution et peut être suspendu. On citera par exemple Mantis OS [BCD+05], dédié aux capteurs, qui est contraint d'utiliser une taille de pile statique pour chaque *thread*. Le choix *a priori* de cette taille est problématique, imposant un compromis entre limitation de la profondeur maximale d'appels et sur-consommation de mémoire.

Partant de ce constat, Dunkels *et al.* ont introduit en 2006 un nouvel outil, les *protothreads* [DSVA06], dont l'objectif est de permettre l'écriture de systèmes évènementiels légers avec un code source linéaire, reflétant le flot d'exécution du programme. Implémentés comme un ensemble de macros en C, ils fournissent des routines d'attente passive non bloquantes. Lorsqu'un *protothread* entre en attente, il quitte son contexte d'exécution (aucun état n'est

4. SIM : *Subscriber Identity Module*

stocké dans sa pile) et rend la main à l'ordonnanceur (la boucle principale) auquel il retourne simplement une continuation (pointeur sur la position courante dans le code). À chaque instant, le système n'a donc besoin que d'une seule pile d'exécution ; la mémoire consommée par chaque *protothread* est seulement celle de sa continuation, qui est de 2 à 3 octets. Le système d'exploitation Contiki [DGV04], destiné aux capteurs, est entièrement fondé sur les *protothreads*.

Les systèmes informatiques que nous avons cités (systèmes temps réel, cartes à puces et capteurs de terrain) nécessitent des techniques de conception radicalement différentes. Leurs systèmes d'exploitation reposent sur des structures variées, avec chacune leurs particularités en termes de performances, de souplesse ou de robustesse.

2.1.3 Structures des systèmes d'exploitation

Les techniques mises au point pour les ordinateurs personnels sont souvent transposées au monde de l'embarqué, impliquant un effort d'adaptation plus ou moins important. Ainsi, les innovations qui ont mené des premiers logiciels intégrés aux exo-noyaux ont été appliquées à l'informatique enfouie. De par la grande variété des systèmes embarqués (en termes de contraintes et d'objectifs), chaque innovation vient enrichir l'ensemble des techniques existantes plutôt que de remplacer ces dernières ; c'est ainsi que co-existent aujourd'hui des modèles de systèmes d'exploitation très variés.

Systèmes intégrés

Comme les premiers ordinateurs, les systèmes embarqués ont longtemps été programmés sans distinguer le système d'exploitation des applications. Dans de tels systèmes, dits *intégrés*, la prise en charge du matériel et les traitements applicatifs sont indissociables. Il n'y a pas d'applications à proprement parler et les différents modes de privilège du microprocesseur (s'ils existent) ne sont pas utilisés.

Aujourd'hui, on utilise encore des systèmes intégrés pour programmer les équipements aux ressources les plus limitées, dont on souhaite maîtriser finement le coût de production. Comme nous l'avons mentionné précédemment, c'est le cas de certaines cartes à puces ou des systèmes temps réel ne se basant pas sur un RTOS. L'avantage des systèmes intégrés est qu'ils constituent un logiciel entièrement dédié donc particulièrement efficace, compact et peu consommateur de ressources. Leur inconvénient est qu'ils ne peuvent être développés que par des experts, ayant à la fois une bonne connaissance du matériel sur lequel le système sera déployé et du domaine applicatif visé. Le développement et la maintenance de tels systèmes sont difficiles à cause de l'absence de toute isolation entre les différentes parties du logiciel.

Afin d'automatiser la construction de systèmes intégrés, des techniques de spécialisation peuvent être utilisées. En partant d'un logiciel complet et générique, elles sont capables d'extraire, par raffinement, une version spécialisée embarquable. On allie ainsi généricité et maintenabilité du code source avec efficacité et faible empreinte de l'image compilée. En contrepartie, le logiciel produit n'est pas extensible, car l'analyse permettant la spécialisation est réalisée statiquement. Cette approche a été appliquée, entre autre pour des systèmes mis en réseaux [Bha06] et pour des machines virtuelles [CGV10].

Systèmes à noyau monolithique

Les premiers systèmes d'exploitation à proprement parler (*e.g.* Multics, Unix), apparus dans les années 1960, étaient construits autour d'un noyau dit *monolithique*, dans le but de fournir une base de code sur laquelle on exécuterait de multiples applications. Un noyau monolithique gère en une unique entité l'ensemble des fonctionnalités du système d'exploitation. Dans ce modèle, les interfaces applicatives sont génériques, se fondant par exemple sur la norme POSIX [5]. Toute application gère ses accès au matériel ou aux services du système *via* cette interface.

La majorité des systèmes d'exploitation d'ordinateurs utilisent aujourd'hui un noyau monolithique (*e.g.* Windows, Linux). Cette architecture de système est également adoptée par de nombreux systèmes d'exploitation dédiés aux systèmes embarqués, tel que le système temps réel VxWork ou le système pour capteurs Contiki.

Au fil des années, les évolutions technologiques, les nouveaux protocoles et types de matériels engendrent un accroissement de la taille des noyaux monolithiques. Le code d'un tel noyau devient alors difficile à étendre et à maintenir. Afin d'en faciliter le développement, la plupart des noyaux monolithiques sont désormais construits de manière modulaire. C'est par exemple le cas de Linux, ou, dans un contexte embarqué, du système TinyOS, qui est construit par assemblage de composants. Cependant, malgré la modularité de son code, le noyau constitue une base de confiance (ou TCB [6]) de grande taille, devenant rapidement source de failles de sécurité ou de dysfonctionnement. La réduction de l'étendue de la TCB est ce qui a motivé la création des systèmes à micro-noyaux.

Systèmes à micro-noyau ou à exo-noyau

L'idée sur laquelle reposent les micro-noyaux est de sortir du noyau toute partie de code pouvant l'être. Le noyau se contente alors de gérer et de sécuriser les accès au matériel. Les autres services du système d'exploitation (*e.g.* gestion d'une pile de communication ou d'un système de fichiers) sont des services externes au noyau, ce qui rend leur exécution moins critique. En effet, la défaillance d'un service n'entraine pas de lourdes conséquences sur le système ; elle peut être récupérée par un redémarrage du service. En réduisant la taille de leur noyau, ces systèmes offrent une meilleure maintenabilité, portabilité et fiabilité que ceux à noyau monolithique.

Le premier représentant historique des micro-noyaux est Mach [RBF+89], dont dérive l'actuel MacOS. On citera également la famille de noyaux L4 [Lie95], encore active, et dont des versions embarquées existent, ainsi que les systèmes temps réel QNX et ChorusOS. Le système pour capteurs SOS peut également être considéré comme un micro-noyau, dont les services sont des modules qui peuvent être chargés et déchargés dynamiquement.

Les exo-noyaux [EKJ95, KEG+97] poussent à l'extrême l'idée de diminuer la taille de la TCB, en n'incluant que les parties basses des pilotes du matériel. Le reste des fonctionnalités du système d'exploitation est assuré par des bibliothèques externes (les LibOS dans le cas du noyau exOS). Ainsi, le noyau se contente de multiplexer de manière sécurisée les accès au matériel. Les applications peuvent, au besoin, utiliser leur propre bibliothèque afin d'implémenter à leur guise les routines bas niveau du système. Ce modèle de système d'exploitation a été transposé pour cartes à puce avec le projet Camille [Gri00].

5. POSIX : *Portable Operating System Interface*
6. TCB : *Trusted Computing Base*

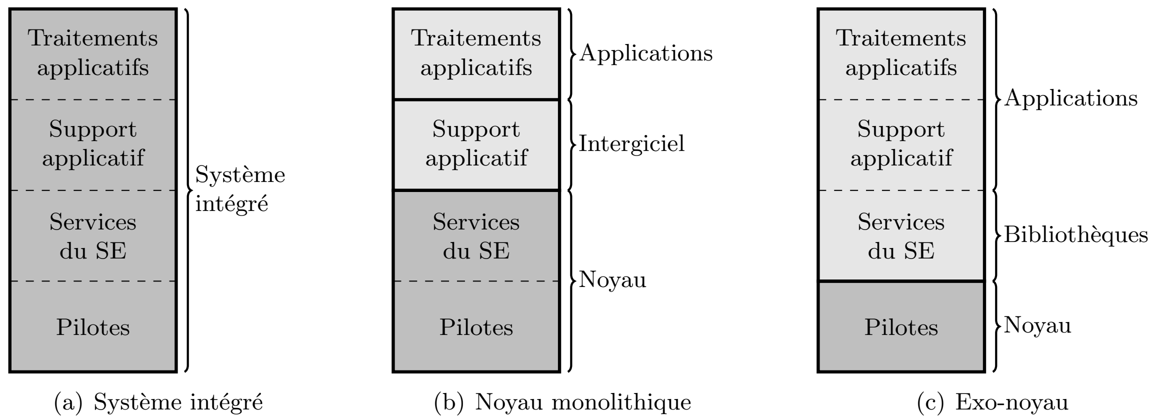


FIGURE 2.1 – Structure des différents modèles de système d'exploitation

L'importance du découpage entre noyau et applications

Le découpage entre noyau et applications joue un rôle majeur dans les performances et les possibilités offertes par un système. On considère ici un système logiciel classique, composé de pilotes (gérant le matériel), des services de base du système (accès au réseau, communication entre applications, *etc.*) et des applications. On détaille le cas où l'on distingue un conteneur d'applications, fournissant à ces dernières des routines adaptées à leurs besoins spécifiques, par opposition aux routines génériques du système. Ce conteneur d'applications peut être par exemple une machine virtuelle Java ou un serveur d'applications Web.

Les systèmes intégrés, dont le noyau englobe les applications (voir figure 2.1(a)), n'ont pas à définir d'interface applicative. Cette absence d'interface permet d'implémenter l'ensemble du logiciel de manière transversale. La vue d'ensemble du système permet de bénéficier de nombreuses optimisations (grâce à la bonne maîtrise des structures de données, des conventions d'appel et une bonne connaissance des traitements globalement réalisés). C'est pourquoi les systèmes intégrés ont le potentiel d'optimiser les performances et de minimiser la consommation de ressources du logiciel qui peut alors être déployé au sein d'équipements peu coûteux.

Dans un noyau monolithique, les accès au matériel ou communications entre applications se font *via* une interface imposée, découlant du découpage entre noyau et applications (voir figure 2.1(b)). Lorsqu'un système fournit une interface, cette dernière a impact sur les performances et la consommation de ressources de ce dernier, pour deux raisons :

Franchissement de l'interface Le franchissement d'une interface a un coût variable, allant de l'appel système classique (qui nécessite un changement du mode du processeur et de l'espace d'adressage) à un simple appel de fonction voire une injection dans le code de la couche supérieure réalisée *a priori* par une macro. Lors de la manipulation de tampons, il est courant que de lourdes copies de données soient également nécessaires ;

Adaptation à l'interface Le logiciel utilisant une interface est écrit en suivant un ensemble de contraintes imposées par cette dernière. Il doit manipuler des structures de données imposées, parfois exécuter des actions redondantes (impliquées par des franchissements consécutifs), re-calculer des valeurs qui auraient pu être partagées ;

Dans un système à noyau monolithique, les conteneurs d'application sont typiquement implémentés comme une application particulière – un *intergiciel* –, à l'interface entre le

système d'exploitation et les applications qu'il supporte. Un intergiciel a pour but de fournir une interface adaptée aux besoins d'un certain type d'applications, comme par exemple des applications Web. Les applications sont alors construites de manière efficace, car suivant une interface adaptée à leurs besoins. Elle sont développées de manière portable, car ne dépendant pas directement du système d'exploitation. Enfin, un intergiciel est souvent utilisé pour supporter des communications performantes, inter-opérables et transparentes entre applications de systèmes distribués. En revanche, l'usage d'un intergiciel ajoute une interface supplémentaire au système complet, impliquant les surcoûts évoqués ci-avant.

La philosophie des micro-noyaux repose sur la manière de concevoir le système d'exploitation, dont on réduit la taille de la TCB en extrayant certains services du noyau. L'interface qu'ils proposent est comparable à celle des noyaux monolithiques, par exemple de type POSIX. On y retrouve typiquement la notion d'Intergiciel.

Les exo-noyaux, quant à eux, offrent aux applications la possibilité d'interagir avec le matériel au plus bas niveau, en fournissant leurs propres bibliothèques dédiées (voir figure 2.1(c)). Elles sont quasiment libres de toute interface, ce qui leur permet d'atteindre de très bonnes performances [KEG+97]. La philosophie des exo-noyaux pousse à concevoir tout logiciel au-dessus du noyau comme un tout, totalement intégré et sans interfaces, permettant une implémentation aussi efficace que possible.

Bien que suivant des approches en apparence opposées, les systèmes intégrés et les exo-noyaux présentent les mêmes avantages en termes de performances ; l'absence d'interfaces sur la construction de la quasi-totalité du logiciel étant une source importante d'optimisations.

Cette analyse des systèmes d'exploitation comme un ensemble de fonctionnalités liées soit par une interface, soit par une implémentation intégrée, guide la suite de cet état de l'art et de ce mémoire. En effet, la mise en réseau des systèmes enfouis nécessite de s'interroger sur la manière d'en concevoir le logiciel embarqué et son système d'exploitation. L'intégration des POPS dans les réseaux actuels, *via* les standards de l'Internet, constitue ainsi un défi à part entière.

2.2 Communication dans les matériels contraints

La suite protocolaire IP constitue aujourd'hui un standard largement diffusé et adopté de par son usage dans l'Internet. Dans le cadre de l'Internet des Objets, l'adoption de ces standards pour les systèmes embarqués permet d'imaginer une fusion entre l'actuel Internet et celui des Objets. Partant du constat que les contraintes des systèmes embarqués rendaient difficile cette transposition, un premier axe de recherche consiste à décliner les protocoles standard pour les adapter aux contraintes atypiques de ces nouveaux réseaux.

2.2.1 Adaptation des standards IP

De par leurs contraintes matérielles, la plupart des petits objets communicants utilisent des protocoles *ad hoc*, répondant précisément à leurs besoins. Ainsi, il est possible de contrôler finement leur consommation de ressources (énergie, mémoire, calcul, débit) en fonction de leurs besoins fonctionnels de communication.

Les réseaux de capteurs sont un excellent exemple de matériels très contraints à fortes exigences de communication. Dans ce domaine, le routage efficace en énergie et en temps est un problème à part entière, encore très actif en termes de recherche et conduisant à des solutions

complètement nouvelles et spécifiques [AKK04].

Il a été montré et il est aujourd'hui communément admis [Fal03, DAV+04, HAN05] que la transposition des protocoles de l'Internet dans les systèmes contraints et/ou mobiles pose de nombreux problèmes. Conçus à la fin des années 80 [Bra89], ces protocoles sont adaptés à l'Internet tel qu'il était envisagé à cette époque, constitué de machines sans fortes contraintes de mémoire, de réseau, de calcul ou d'autonomie.

Adaptation de la suite IP

La suite IP est l'ensemble des protocoles de base utilisés sur Internet, incluant IP, ICMP, IGMP, UDP et TCP. Ces protocoles, trop consommateurs en ressource pour les réseaux aux contraintes atypiques, se voient parfois déclinés en des versions allégées afin de correspondre aux propriétés du réseau visé.

Une technique couramment employée pour alléger le trafic IP consiste à compresser les en-têtes des protocoles afin d'augmenter la quantité de données utiles présentes dans chaque paquet. Les solutions existantes [Jac90, DNP99], en se fondant sur les redondances entre les en-têtes des paquets successifs, réduisent les en-têtes UDP/IP à seulement 4 octets (au lieu de 28 octets). Elles ne sont cependant utilisables que sur un réseau local coopérant et réduisent les plages d'adresses et de ports utilisables.

Dans le cas de communications multi-saut en mode connecté, la mise en ordre des segments rend plus complexe les techniques de compression d'en-têtes. Sridharan *et al.* [SSM03] proposent une solution dans laquelle tous les nœuds stockent des informations d'état sur toutes les connexions dont ils sont en charge du routage. Ainsi, la mise en ordre de segments se fait efficacement, sans compromettre la compression des en-têtes.

Dans le cas des réseaux de capteurs sans-fil, il a également été proposé d'adresser les nœuds de manière géographique afin d'améliorer le routage, la gestion des sous-espaces d'adressage et la compression des en-têtes IP [DVA04].

Afin de permettre l'usage d'IPv6 dans des réseaux de capteurs, le groupe 6LoWPAN [7] [HC08] s'intéresse au support de ce protocole sur une liaison de type IEEE 802.15.4. La taille maximal des paquets, de 127 octets, est *a priori* incompatible avec IPv6, imposant des paquets d'au moins 1280 octets. La compression d'en-têtes (rappelons que les adresses IPv6 sont de 16 octets), l'adaptation de la taille des paquets et la gestion de la consommation d'énergie constituent les principaux défis adressés par 6LoWPAN.

Adaptations relatives à TCP

Fall [Fal03] explique l'inadaptation de TCP [8] aux réseaux fortement contraints en identifiant trois propriétés nécessaires au bon fonctionnement de TCP, non vérifiées dans le cas de ces réseaux :

Persistance de chemins de bout en bout pendant un échange En cas d'absence momentanée d'un chemin entre une source et une destination, TCP provoque des retransmissions inutiles ;

Stabilité des temps d'aller-retour La variation des temps de trajet biaise le calcul des délais de retransmission réalisé par TCP à partir de l'estimation des temps d'aller-retour ;

7. 6LoWPAN : *IPv6 over LoW Power wireless Area Networks*

8. Une description du fonctionnement de TCP est donnée en annexe A

Faible probabilité de pertes de paquets sur chaque chemin TCP interprète les pertes de segment comme un signe de congestion du réseau et réagit en diminuant le débit sortant. Dans les réseaux sans fil, des pertes indépendantes de la congestion peuvent survenir.

Dans [DVA04], une technique de cache distribué des segments TCP est proposée, dans laquelle chaque nœud conserve en mémoire les segments qu'il route tant qu'ils n'ont pas été acquittés. En cas de perte d'un segment, la retransmission de ce dernier ne se produit pas de bout en bout, mais à partir du dernier saut ayant relayé le segment.

Dans [Fal03], une architecture de réseau tolérant aux délais est proposée, fondée sur des passerelles en charge d'estomper les fluctuations du réseau. Les passerelles agissent au niveau d'une sur-couche ajoutée à TCP, en charge de véhiculer un ensemble de méta-données en plus des messages originaux de TCP. Cette solution réduit significativement l'interopérabilité du réseau ; motivation première à l'usage de la suite IP pour l'informatique ambiante.

Sarolahti *et al.* [SKR03] proposent une politique de gestion de retransmissions pour TCP prenant en compte la possibilité de pertes locales et ponctuelles. Xylomenos propose [XPMS01] 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 lequel les pertes ponctuelles deviennent invisibles.

Ces adaptations, en agissant au niveau des protocoles, rendent plus efficace la suite IP dans les réseaux fortement contraints. Mais au delà des propriétés atypiques du réseau, le fait de déployer une pile IP dans des nœuds ne disposant que de quelques kilo-octets de mémoire soulève de nombreuses questions et nécessite des solutions adaptées.

2.2.2 Piles IP embarquées

Le protocole TCP, en charge de gérer des connexions fiables entre deux hôtes d'un réseau, est relativement complexe. Ses mécanismes incluent la gestion de la congestion du réseau, des retransmissions des segments perdus et de remise en ordre des données. L'implémentation de tous ces mécanismes est coûteuse en mémoire car elle requiert l'usage de nombreux tampons. Le code d'une pile IP telle que celle d'un système BSD ou Linux nécessite environ 10 000 lignes de code C, résultant en un code compilé d'environ cent kilo-octets et nécessitant plusieurs centaines de kilo-octets de mémoire vive [Sho05].

Les implémentations de TCP/IP destinées à l'embarqué sont très nombreuses, ciblent des matériels divers et supportent des fonctionnalités variées. Les plus complètes d'entre elles, comme μC/TCP-IP [Mic09a], openTCP [Ope03] ou les dérivées de l'implémentation de FreeBSD [MBKQ96], nécessitent plusieurs dizaines de kilo-octets de mémoire persistante et s'exécutent pour la plupart au-dessus d'un système d'exploitation, temps réel ou non.

La réalisation d'une pile IP destinée à être embarquée dans un équipement disposant de quelques kilo-octets de mémoire constitue un défi. Afin d'y répondre, certaines piles de communication sont conçues comme totalement autonomes ; elles ne reposent sur aucun système d'exploitation sous-jacent et accèdent au matériel directement. On peut les considérer comme un fragment autonome d'un système dont les primitives fournies aux applications se limitent à la gestion du réseau.

Nom	Réf	Nature/origine	Cibles	Code min.
OpenTCP	[Ope03]	libre	portable	64 ko
μC/TCP-IP	[Mic09a]	Micriμm	RTOSμC	42 ko
lwIP	[Dun03]	libre	portable	21 ko
Microchip TCP/IP	[Mic09b]	libre	portable	20 ko
NicheLite TCP/IP	[Tec06]	Interniche	portable	12 ko
CMX MicroNet	[Sys02]	CMX Systems	portable	5 ko
uIP	[Dun03]	libre	portable	5 ko
TI MSP430 TCP/IP	[Ins01]	Texas instrument	CPU MSP430	4 ko

Table 2.1 – Caractéristiques d'une sélection de piles IP embarquées

Piles IP sans système sous-jacent

Les piles IP autonomes sont des implémentations très légères qui souvent ne supportent que quelques-uns des protocoles de la suite IP et n'en n'implémentent qu'un sous-ensemble des fonctionnalités. La plupart d'entre elles ne tolèrent qu'un unique segment non acquitté à chaque instant, ne gèrent pas la congestion du réseau, la remise en ordre des données entrantes, la fragmentation IP et les données urgentes de TCP.

Par exemple, TinyTCP [Coo02] ne supporte qu'une seule connexion à chaque instant. Une telle simplification permet de réduire significativement la quantité de mémoire vive consommée ainsi que la taille du code dédié à TCP, mais impose de fortes limites en termes d'utilisabilité. L'implémentation de CMX Systems, la pile MicroNet [Sys02], est quant à elle limitée statiquement à 16 connexions. La solution de Texas Instrument pour MSP430 [Ins01] atteint une empreinte mémoire de seulement 4 kilo-octets. En plus de la limitation à une unique connexion, cette implémentation ne réalise pas de calcul de sommes de contrôle sur les données entrantes ; une opération qui est particulièrement demandeuse de ressources.

Dunkels *et al.* proposent deux implémentations [Dun03], lwIP et uIP. LwIP supporte un grand nombre de protocoles (*e.g.* IP, ICMP, UDP, TCP) et en implémente la quasi-totalité des fonctionnalités, avec une empreinte mémoire de 21 kilo-octets. uIP ne supporte que TCP/IP, ne gère qu'un seul segment non acquitté et a un nombre maximal de connexions fixé statiquement, mais atteint une empreinte mémoire de seulement 5 kilo-octets. Ces deux implémentations, largement utilisées dans l'industrie comme dans la recherche, proposent d'excellents compromis entre compacité et fonctionnalités.

La table 2.1 propose une synthèse des piles IP embarquées les plus reconnues et largement utilisées. La taille minimale de leur code compilé donnée est celle communiquée par les auteurs des différentes implémentations. Elle doit donc être interprétée comme un ordre de grandeur, car la cible matérielle utilisée pour la mesure n'est pas toujours connue.

Les *sockets* et la gestion de la mémoire

Le point d'entrée au réseau fourni aux applications par un système d'exploitation est communément fondé sur la notion de *socket* réseau. Aujourd'hui, les *sockets* de Berkeley se sont imposés comme l'API[9] *de facto* pour la manipulation de *sockets*. Dans le cas de communications TCP, les primitives fournies par cette API présentent chaque connexion

9. API : *Application Programming Interface*

comme un flux bidirectionnel, dans et à partir duquel on peut écrire et lire des données de manière ordonnée.

En observant la construction des piles TCP/IP embarquées, on constate que la notion de *sockets* est omniprésente et que l'API qui y est associée est systématiquement conçue pour être aussi proche possible des *sockets* de Berkeley. Même uIP, dont l'implémentation est intégralement événementielle et réalisée avec des *protothreads* (détaillés en sous-section 2.1.2), propose des routines bloquantes pour envoyer et recevoir des données sur les *protosockets*, des *sockets* à base de *protothreads*.

Lors de l'envoi de données sur un canal TCP avec des *sockets*, le système stocke les données à envoyer dans un tampon puis les envoie. Il ne libérera la mémoire associée au tampon que lorsque les données auront été acquittées. Usuellement, l'appel système demandant une transmission est bloquant ; il ne retourne que lorsque assez de mémoire est disponible pour stocker les données à envoyer. Dans un système ne disposant que de quelques kilo-octets de mémoire vive, le nombre et/ou la taille des segments gérés à chaque instant est très limité. Les piles IP les plus compactes telles que uIP ou celle de Texas Instrument ne gèrent qu'un unique segment à la fois. En plus de limiter la consommation de mémoire vive (tout segment envoyé doit être stocké en attente de la réception de son accusé), ce choix simplifie grandement l'implémentation et en réduit la taille (le segment concerné par tout traitement est toujours accessible au même endroit). Cependant, chaque envoi de segment bloque jusqu'à réception de son accusé, ce qui réduit considérablement le débit maximal atteignable [Dun03].

Une technique particulièrement intéressante est proposée et utilisée dans uIP, qui supprime de la mémoire les segments dès qu'ils sont envoyés. En cas de retransmission, le système demande au *protosocket* de reproduire les données du segment perdu, ce qui est rendu possible par la constante limitation à un unique segment envoyé non acquitté (l'application est bloquée jusqu'à réception de l'accusé). Une autre approche consiste à s'abstraire de toute interface générique, ce qui est rendu possible dans le cas d'une pile IP dédiée conçue et implémentée conjointement à l'application qu'elle supporte. Cette technique est par exemple applicable au cas des serveurs Web embarqués.

2.2.3 Serveurs Web embarqués

Les piles IP présentées précédemment sont génériques. Elles sont une bonne solution pour réaliser des applications réseau sur des matériels fortement contraints. On s'intéresse ici aux solutions permettant de supporter, en plus des protocoles IP, un serveur Web, au sein de matériels tout aussi voire encore plus contraints.

Des systèmes intégrés

Plusieurs travaux ont montré qu'il était possible d'embarquer un serveur Web et sa pile IP dédiée dans des matériels très contraints [Agr98, JCH00, Shr02, HZW$^+$03, Dom03, LZW$^+$04]. Il s'agit de systèmes intégrés, conçus conjointement avec leur pile de communication, indépendants de tout système d'exploitation.

Ce type de serveur est typiquement construit et lié aux contenus Web lors d'une phase de compilation. La procédure de construction d'un serveur Web intégré prêt au déploiement est décrite dans [Agr98] et synthétisée par la figure 2.2. Des pages Web statiques (simples fichiers) sont tout d'abord pré-compilées à l'aide d'un outil dédié. Le résultat de la cette étape est ensuite compilé avec le code de la pile HTTP/TCP/IP afin de produire une image exécutable

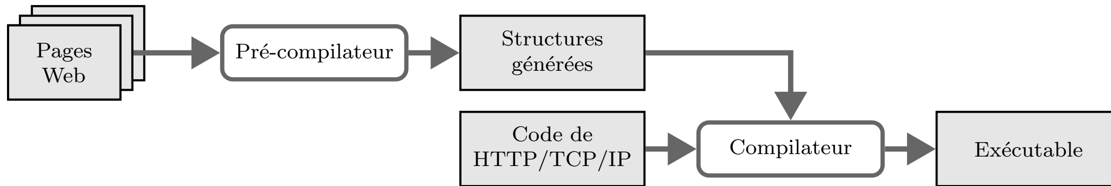


FIGURE 2.2 – Chaîne de compilation typique d'un serveur Web intégré

par le matériel visé. Les solutions les plus évoluées permettent l'usage de balises dédiées au sein des pages, permettant d'y injecter du contenu généré dynamiquement lors de l'exécution. Le code associé à chacune de ces balises est lui aussi compilé et inclus dans l'image finale.

La conception intégrée de la pile de communication HTTP/TCP/IP permet d'envisager des optimisations transversales dites fonctionnelles ou d'implémentation. Cumulées, ces optimisations permettent de cibler des matériels très contraints, donc très peu coûteux.

Optimisations fonctionnelles

Shon [Sho05] a proposé mIP, une pile IP dédiée au service HTTP qui n'implémente qu'un sous ensemble très restreint des fonctionnalités habituelles des protocoles HTTP, TCP et IP. Il en résulte un serveur Web autonome très petit (moins de 4 ko), capable de communiquer avec un hôte, mais soufrant de nombreuses limitations. Voilà une synthèse des simplifications réalisées par mIP, ainsi que par de nombreux autres piles IP dédiées :

Ensemble restreint de protocoles Seuls les protocoles HTTP, TCP et IP sont nécessaires au fonctionnement d'un serveur Web ;

Simplification de l'automate TCP En simplifiant la machine à état interne à TCP, on en simplifie grandement l'implémentation. Par exemple, on peut considérer qu'un serveur Web n'a pas besoin d'initier de nouvelles connexions ;

Absence de contrôle de congestion Le plus souvent, le contrôle de congestion n'est pas supporté par les serveurs Web embarqués. Ce mécanisme est jugé inutile au vu des faibles trafics générés par de tels serveurs ;

Absence d'autres mécanismes Des mécanismes tels que la fragmentation, le TTL, les options d'en-tête, le calcul de temps d'aller-retour ou les données urgentes TCP sont exclues des serveurs Web visant les plus petits matériels. Certains d'entre eux vont jusqu'à ne pas supporter les retransmissions TCP et se limitent à la gestion d'une unique connexion.

Complétant ces optimisations fonctionnelles, des optimisations transversales d'implémentation sont également rendues possibles par la construction intégrée de ces serveurs Web.

Optimisations d'implémentation

Les piles IP dédiées au service HTTP ne sont pas contraintes d'utiliser une abstraction générique du réseau telle que les *sockets*. C'est ce qui permet d'envisager des optimisations transversales, à tout les niveaux du logiciel, des pilotes au serveur HTTP. Certaines solutions vont jusqu'à concevoir le matériel conjointement avec le logiciel [RMS+01, HZW+03].

Le serveur MiniWeb [Dun05] est une preuve de concept dont l'objectif est de fournir un serveur HTTP/TCP/IP portable avec une empreinte mémoire minimale. Son architecture est

Nom	Réf	Nature/origine	Cibles	Code min.
Barracuda	[Log05]	Real Time Logic	Windows, Unix	>100 ko
POS-EWS	[CJ00]	Académique	divers RTOS	30 ko
TinyWS	[PKGZ08]	Académique	uIP	10,6 ko
RomPager	[All00]	AllegroSoft	autonome	10 ko
PicoWeb	[FHL99]	Lightner Engineering	AT90S8515	8 ko
mIP	[Sho05]	Académique	autonome	3,7 ko
Webit	[HZW+03]	Académique	divers RTOS	3,5 ko
MiniWeb	[Dun05]	Libre	autonome	2,8 ko
webACE	[Whi99]	Autre	ACE1101MT8	1 ko
iPic	[Shr02]	Libre	PIC 16F84	384 octets

Table 2.2 – Caractéristiques d'une sélection de serveurs Web embarqués

entièrement évènementielle, ayant la granularité des paquets IP. Lorsque MiniWeb traite un paquet entrant, les données sont lues, traitées et défaussées au fur et à mesure, ne nécessitant aucun tampon. Les données sortantes, qui sont nécessairement statiques et situées en mémoire adressable non volatile, sont elles aussi envoyées sans tampon. En cas de retransmission, il est tout à fait possible de ré-accéder aux données précédemment envoyées, en calculant l'adresse de départ à partir du numéro de séquence du segment perdu.

Dans MiniWeb, lors de la pré-compilation des pages (*c.f.* figure 2.2), les données à envoyer sont pré-découpées en paquets IP incluant en-têtes TCP et IP, simplifiant grandement la tâche de la pile IP lors de l'exécution. De plus, les sommes de contrôle TCP sont calculées pour chaque segment, ce qui accélère sensiblement le traitement des paquets sortants. Ce point impose toutefois de fixer la taille des segments TCP lors de la compilation et non lors de l'exécution. Afin de pouvoir accepter des connexions sans connaissance préalable du client, cette taille est fixée à 200 octets, une valeur sensiblement inférieure à celle de 1460 octets qui est la plus fréquemment rencontrée.

Tour d'horizon

La table 2.2 propose une synthèse des serveurs Web embarqués les plus connus et couramment utilisés. Comme pour la table 2.1, les tailles minimales de code sont à interpréter comme des ordres de grandeur. Les trois premiers serveurs, donnés à titre d'exemple, ne suivent pas une construction intégrée. Barracuda [Log05] est destiné à s'exécuter comme une application Windows ou Unix. À son empreinte mémoire de plus de 100 kilo-octets, il faut donc ajouter celle du système d'exploitation. POS-EWS [CJ00], quant à lui, nécessite un système temps réel (RTOS) pour s'exécuter. Enfin, TinyWS [PKGZ08] est un serveur Web s'appuyant sur la pile IP uIP.

Les exemples suivants intègrent tous leur propre pile IP dédiée. Comme nous l'avons mentionné en sous-section 2.1.3, l'absence de toute interface entre système et applications permet de minimiser la consommation de ressources de ces systèmes. Les deux implémentations les plus légères, webACE [Whi99] et iPic [Shr02], atteignent une empreinte mémoire d'un kilo-octet ou moins. Leurs restrictions fonctionnelles sont bien sûr très importantes : ils ne servent que des données statiques ne dépassant pas un segment et ne s'adaptent à aucune fluctuation du trafic. Ils sont écrits pour un matériel spécifique, leur code n'est par conséquent pas portable. Capables de servir des pages Web complètes, MiniWeb [Dun05], Webit [HZW+03]

et mIP [Sho05] ont tous un empreinte mémoire de mois de 4 kilo-octets.

Ces serveurs Web et piles IP ont le plus souvent pour objectif de minimiser leur empreinte mémoire. Dunkels, dans ses travaux sur les systèmes embarqués communicants, part du postulat que ces matériels ne produisent pas assez de données pour nécessiter que l'on se soucie de leurs performances [Dun07]. Les solutions qu'il propose, pour la plupart, dégradent les performances au profit de la compacité. Nous pensons au contraire que dans le cadre d'un Web des objets, ces systèmes doivent être conçus dès le départ en alliant performances, compacité et capacité à passer à l'échelle. C'est une condition *sine qua non* pour en espérer un usage massif aux possibilités aussi importantes que possible. Dans cette optique, nous nous intéressons aux travaux qui ont permis aux serveurs d'application d'atteindre les performances dont ils ont aujourd'hui besoin sur un Internet toujours plus exigeant.

2.3 Performance des serveurs d'application

Les serveurs d'application sur Internet, dont l'exemple le plus courant est celui des serveurs Web, ont de fortes exigences de performances, de stabilité et de passage à l'échelle sous de très fortes charges. De nombreux travaux ont été conduits, visant à en repousser les limites. En moins de 20 ans, les architectures logicielles des serveurs Web ont ainsi considérablement évolué.

2.3.1 Architectures de serveurs Web

Dès 1999, Kegel (*The C10K problem* [Keg99]) a pointé le fait que les serveurs Web sous-exploitaient largement le matériel dont ils disposaient ; malgré leurs plusieurs giga-octets de mémoire vive, ils étaient incapables de gérer plus de 10 000 clients simultanément. Afin de faire face à ce problème de passage à l'échelle en pleine période d'explosion du trafic Internet, Kegel explique qu'il sera nécessaire de faire évoluer l'architecture des serveurs et les abstractions fournies par les systèmes d'exploitation.

Historique de l'évolution des architectures

En adéquation avec le modèle Unix traditionnel, les premiers serveurs HTTP créaient un processus pour chaque nouvelle requête à traiter. Les performances de ces serveurs ont rapidement été limitées par le coût élevé des créations de processus. Des serveurs tels que le NCSA httpd [NCS93] ont alors commencé à pré-allouer statiquement un ensemble de processus, auxquels un processus maître distribuait les requêtes qu'il recevait. Des serveurs plus récents tels que Apache [Apa95] n'utilisent plus de processus maître : chaque processus (ou *thread*) se charge de l'acceptation des connexions, en acquérant ainsi la charge.

Ces premières implémentations gèrent la concurrence des requêtes en répartissant ces dernières sur différents processus ou *threads*, on dit qu'elles suivent une approche *multi-thread*ée. Afin de réduire les coûts intrinsèquement liés à la présence de nombreux processus (*e.g.* changements de contexte, mémoire allouée aux piles d'exécution), des serveurs plus récents tels Zeus [Zeu99], Flash [PDZ99] ou Lighttpd [Lig03] se fondent sur une approche dite évènementielle. Un unique processus (ou plus exactement un par processeur), à l'aide de la routine `select()`, est à l'écoute de tout évènement à traiter sur l'ensemble des connexions gérées par le serveur. Tous les évènements sont traités consécutivement, sans préemption.

En 2001, Welsh *et al.* ont introduit SEDA, une architecture évènementielle par étages pour le passage à l'échelle de serveurs Internet [WCB01]. Il s'agit d'une solution à la fois évènementielle et *thread*ée, dont le but est d'allier les hautes performances à la stabilité et l'équité des temps de réponse. Une architecture SEDA est faite de composants, chacun en charge d'une tâche précise. Les connexions entre les composants indiquent les dépendances des tâches. Chaque composant gère un ensemble de *threads*, lui permettant d'exécuter de manière concurrente plusieurs instances de la tâche à laquelle il est dédié.

L'adaptation de l'interface entre système et applications

La perpétuelle course aux performances qui mène à l'élaboration de nouvelles solutions logicielles se voit parfois contrainte par les primitives des systèmes d'exploitation. Il arrive alors que des adaptations des primitives offertes par les systèmes soient proposées, uniquement pour répondre à ces besoins spécifiques. Par exemple, la routine `sendile()` a été créée spécialement pour accélérer les transferts d'un descripteur de fichier à un autre, évitant de lourdes copies de données entre les espaces mémoire du noyau et de l'application (cette routine est aujourd'hui couramment utilisée par les serveurs de fichiers). On citera également l'interface POSIX `aio`, créée pour permettre l'accès à un périphérique de stockage (*e.g.* disque dur, SSD) de manière asynchrone ; les routines habituellement bloquantes empêchant l'implémentation d'un serveur de fichiers entièrement évènementiel.

La routine `select()`, qui permet à une application de se mettre en écoute d'un ensemble d'évènements, s'exécute en temps croissant linéairement avec le nombre d'évènements gérés. De nouvelles interfaces permettant d'en améliorer le passage à l'échelle ont été proposées [BM98]. Une gestion des priorités sur les évènements est proposée par Benga [BDM98]. Dans le même article, une notion de container de ressource est présentée, représentant l'ensemble des ressources accédées par chaque application. Le système d'exploitation, à l'aide de ces informations, est capable de gérer plus finement l'allocation des ressources aux processus et l'exécution des interruptions matérielles.

Cette évolution continuelle des routines système est une conséquence des surcoûts inhérents aux interfaces génériques, présentés en sous-section 2.1.3. Par ailleurs, d'autres travaux ont évalué l'impact des abstractions du système sur les performances de serveurs d'applications. Un des premiers résultats de Engler sur les performances des exo-noyaux a été obtenu en étudiant les serveurs Web [KEGW96, KEG+97]. Son serveur, utilisant une implémentation dédiée de TCP/IP *via* une libOS, est alors mesuré comme étant entre 3 et 24 fois plus rapides que NCSA httpd. Une étude plus récente (donc dans des conditions matérielles différentes) mesure un facteur d'accélération de 2 à 6 obtenu par intégration de serveurs Web dans le noyau Linux [JKN+01].

Évènements ou *threads* : un débat sans fin ?

Apparu en même temps que la notion de processus, le débat opposant les architectures *thread*ées aux évènementielles est encore non clos. En 1979, Lauer et Needham ont tenté d'y mettre fin [LN79] en montrant que les systèmes à base de passage de messages (*i.e.* systèmes évènementiels) ou à base de processus étaient en fait équivalents ; aucun d'entre eux n'étant préférable à l'autre en termes de performances, et ce, quelle que soit l'application visée.

Malgré ces résultats, de nombreux travaux en faveur des systèmes évènementiels ont vu le jour, incluant l'emblématique *Why Threads Are A Bad Idea (for most purposes)* [Ous96]. Les

avantages associés au paradigme événementiel sont les suivants :

- synchronisation obtenue par construction ;
- contrôle fin de l'ordonnancement et localité des données ;
- légèreté des états.

En réponse à ces travaux, on citera le non moins emblématique *Why Events Are a Bad Idea (for High-Concurrency Servers)* [BCB03] dont le propos est de montrer que la supériorité des évènements n'est due qu'à l'inexistence des outils permettant la compilation et la gestion efficace des *threads*. Les auteurs mettent alors en avant les qualités des solutions à base de *threads* en termes d'expressivité du code qui correspond directement au flot de contrôle des applications.

C'est ainsi que le choix entre ces deux paradigmes est aujourd'hui encore au cœur de la conception des serveurs Web. Il existe cependant des applications où cet éternel débat est moins ouvert et pour lesquelles tous s'accordent à utiliser soit *threads* soit évènements ; c'est le cas de la notification en contexte Web.

2.3.2 Notification en contexte Web

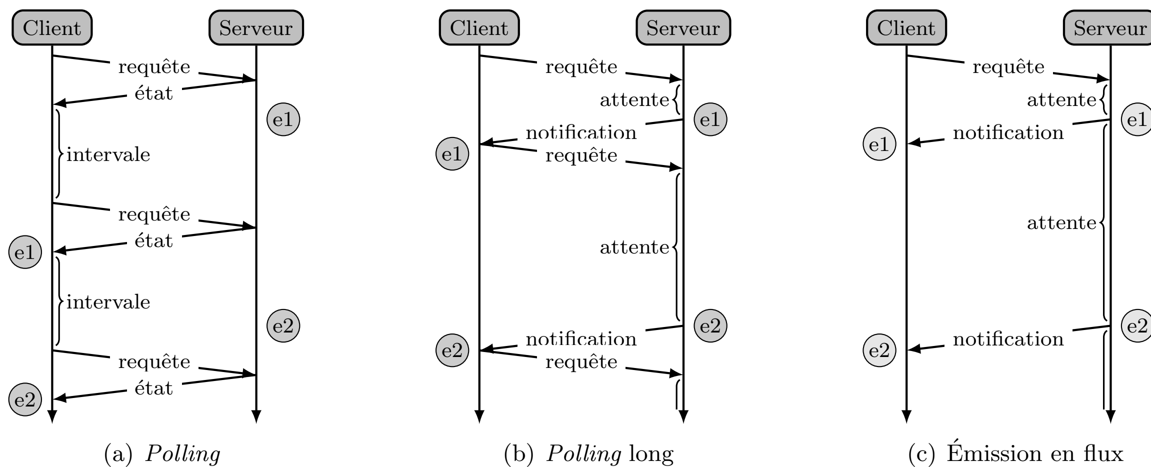
Le Web est en perpétuelle évolution. Créé comme un ensemble de standards permettant le partage de fichiers entre sites distants, il est devenu un support d'applications partagées et par la même un média incontournable. Les technologies initiales ont très peu évolué et présentent leurs limites lorsqu'il s'agit de les appliquer à certains nouveaux usages.

Problèmes liés à la notification

Le protocole HTTP, à la base du Web, suit un simple schéma requête-réponse. Chaque requête identifie la ressource à laquelle le client souhaite accéder et engendre l'envoi d'une réponse par le serveur. Aujourd'hui, les serveurs gèrent des applications à part entière. Il arrive que l'application s'exécutant sur le navigateur du client ait besoin d'être notifiée de changement d'états qui ont eu lieu sur le serveur. On citera les application de suivi de bourse, d'enchères, de discussion ou d'informations. Les notification d'évènements sont difficiles à réaliser en suivant le mode de fonctionnement classique du Web, dans lequel seul le client est initiateur de requêtes, car seul le serveur est publiquement accessible.

Dans cette situation, la solution naturelle serait de permettre au serveur d'initier des connexions vers le client. Même en supposant que chaque client dispose de sa propre adresse IP (par exemple avec IPv6), une telle solution reste non envisageable. Elle imposerait à tout client de jouer également le rôle de serveur, c'est-à-dire d'être à l'écoute des connexions entrantes. Cela serait une source importante d'insécurité, permettant par exemple des attaques de type déni de service. Dans le schéma actuel, le client est toujours initiateur des connexions qu'il a avec le serveur ; il a le contrôle des ressources qu'il consacre à chaque serveur.

Partant de ce constat, l'approche la plus courante, le *polling*, consiste à questionner le serveur à intervalles réguliers. Le choix de l'intervalle à utiliser est alors difficile, imposant un compromis entre la consistance des données du client et la charge imposée au réseau et au serveur. Une solution plus élaborée consiste à adapter dynamiquement cet intervalle afin de tendre vers la période moyenne de publication des évènements [SLR98, DKP$^+$01]. Cette solution n'est efficace que lorsque cette période est peu variable.



(a) *Polling* (b) *Polling* long (c) Émission en flux

FIGURE 2.3 – Illustration des techniques de notification d'évènements pour le Web

Présentation de Comet

En 1995, Netscape a introduit une technique permettant au serveur d'envoyer des données au navigateur [Net95]. Le client envoyait une requête HTTP à laquelle le serveur ne répondait pas immédiatement. La connexion TCP était alors maintenue ouverte et utilisée par le serveur pour envoyer des données lorsqu'il souhaitait notifier un évènement au client. Le navigateur attendait continuellement la fin de cette réponse HTTP potentiellement infinie. Il ne recevait que des parties de réponse, contenant les données notifiées par le serveur.

Ce n'est qu'en 2006 [Rus06] que le nom *Comet* a été attribué aux techniques de notification Web, face au nombre grandissant d'application en nécessitant l'usage. On distingue aujourd'hui deux techniques de notification : le *polling* long et l'émission en flux. Avec le *polling* long, chaque requête est suspendue en attendant une unique réponse du serveur. Avec l'émission en flux, le serveur fragmente les réponses HTTP si bien que chaque requête est suivie d'une série de notifications. La figure 2.3 illustre les techniques du *polling*, du *polling* long et de l'émission en flux.

Support de Comet

Les conteneurs d'applications Web associent à certaines URLs du code permettant au serveur de générer les réponses HTTP. Lorsqu'une requête cible une URL correspondant à ce code, celui-ci est exécuté dans un *thread*, il génère des données puis retourne. Dans le cas de Comet, ce code doit entrer dans une attente passive pour une durée indéterminée, jusqu'à ce qu'un évènement se produise. Ainsi, pour chaque client en attente, une connexion TCP et un *thread* sont alloués sur le serveur, engendrant une très forte consommation de ressources. On est ici dans un cas typique où un intergiciel (le conteneur d'applications Web) fournit un support inadapté à certains besoins applicatifs (la notification d'évènements). Il est alors nécessaire d'en proposer une déclinaison à l'écoute de ces besoins particuliers.

De nouveaux conteneurs d'application ont ainsi été conçus en incluant un support natif de Comet. Ils s'exécutent sur des serveurs évènementiels, afin de ne pas allouer un *thread* à chaque requête en cours. Le code associé à la génération des données est lui aussi construit de manière évènementielle. Ainsi, au lieu d'être appelé puis suspendu pour chaque nouvelle requête, il

n'est appelé qu'au moment de la notification du client. Plusieurs standards de Comet ont déjà été proposés, incluant Bayeux [Doj08b], BOSH [PSSAM09] ou les Web *Sockets* [W3C09]. Les conteneurs supportant Comet sont multiples ; on citera Cometd [Doj08a], Grizzly [Gla06] et Google DWR [DWR07]. Ils s'exécutent tous sur des serveurs évènementiels tels que Glassfish [Gla06] ou Jetty [Mor07].

Bozdag *et al.* ont étudié les différentes solutions de notifications pour applications Web [BMD07, BMD09]. Ils ont conduit un grand nombre d'expériences afin de comparer différentes approches (*polling* et *polling* long) et implémentations (Cometd et DWR). Leurs conclusions montrent que Comet fournit une bonne réactivité, offre un bon niveau de cohérence des données chez le client et utilise peu de ressources du réseau. Cependant, le classique *polling* passe mieux à l'échelle en termes d'usage CPU. D'autres travaux [DKP+01, BD08] proposent des solutions adaptatives où le choix entre *polling* et Comet est réalisé lors de l'exécution, dépendamment de la congestion du serveur et des exigences du client.

On a vu que la manière de gérer la notification d'évènements dans une application Web a un impact important sur les performances obtenues et sur l'expérience du client comme utilisateur. En cas de forte charge, la manière dont le serveur ordonnance le service des différentes requêtes joue également un rôle important sur les performances perçues par les utilisateurs.

2.3.3 Ordonnancement des requêtes

La théorie de l'ordonnancement est un domaine de recherche qui trouve son application dans des domaines très variés. On s'intéresse ici à l'impact sur les performances et l'équité de l'ordonnancement des requêtes au sein d'un serveur Web.

Définitions

Dans le contexte de serveurs Web, les tâches à ordonnancer sont les requêtes en attente, la ressource à partager est la liaison réseau et la durée d'une requête est la taille de la réponse associée. L'ordonnancement consiste alors à élire la requête qui doit être servie à chaque instant.

Selon la notation en 3 champs définie par Graham *et al.* [GLLRK79], on s'intéresse par exemple au problème $1|r_i, pmtn|\sum C_i$, qui doit être interprété comme :

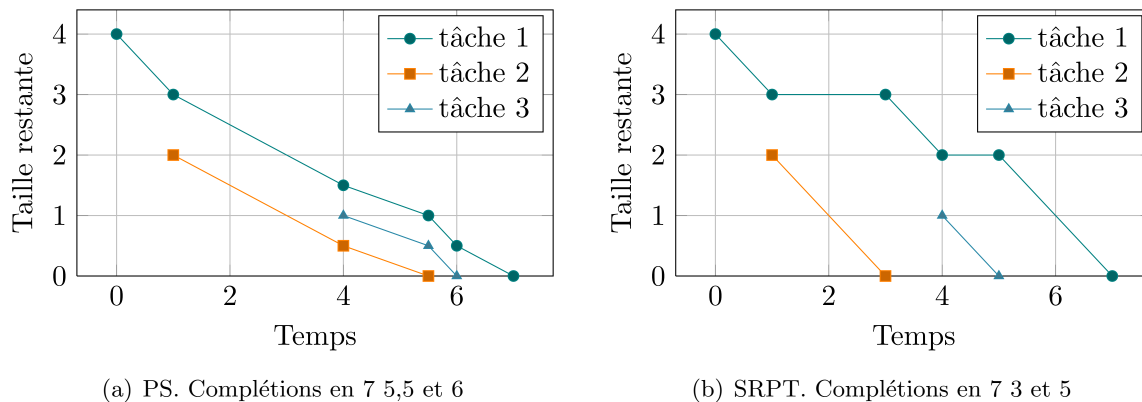
Premier champ L'ordonnancement se fait sur une unique machine ;

Second champ Chaque tâche i a une date d'arrivée r_i dans le système et l'ordonnancement est préemptif (*pmtn*) ;

Troisième champ L'objectif est de minimiser $\sum C_i$, la somme des dates de complétion C_i des tâches.

Les serveurs Web classiques, s'exécutant au-dessus d'un système d'exploitation tel que Linux ou Windows, sont contraints d'utiliser la politique du tourniquet (ou *round-robin*), car l'interface offerte par les *sockets* de Berkeley ne permet pas de contrôler l'ordonnancement des connexions. Le tourniquet est souvent considéré comme équitable par construction. Dans le cadre de la théorie de l'ordonnancement, on remplace le plus souvent cette technique par la politique PS [10] qui alloue à chaque instant une part équitable des ressources disponibles à toutes les tâches. En considérant que les tâches sont indéfiniment divisibles, PS simplifie grandement les analyses d'algorithmes d'ordonnancement [PG93].

10. PS : *Processor Sharing*



(a) PS. Complétions en 7 5,5 et 6 (b) SRPT. Complétions en 7 3 et 5

FIGURE 2.4 – Ordonnancement dans lequel SRPT est plus efficace que PS pour chaque tâche

Suivant des analyses empiriques [PF94, CB97], il est communément accepté qu'on peut modéliser un trafic Web selon une file d'attente M/G/1 à longue traîne. Le processus d'arrivée est donc poissonnien, la répartition des tailles des tâches suit couramment une loi de Pareto bornée de paramètre légèrement supérieur à 1.

Optimiser les performances

Dans le cadre d'un serveur Web, l'optimisation des performances consiste à minimiser le temps de réponse moyen des requêtes traitées par le serveur. Ce temps de réponse est la différence entre la date de complétion C_i et la date d'arrivée r_i. La minimisation du temps de réponse moyen des requêtes est équivalent au problème $1|r_i, pmtn|\sum C_i$.

La politique SRPT [11] consiste à élire à chaque instant la tâche dont la durée restante est la plus courte. Dans un serveur Web, il s'agit de la requête dont la quantité de données restantes à envoyer est la moindre. L'optimalité de SRPT, pour toute entrée, a été prouvé en 1968 [Sch68]. On peut intuitivement penser que cette optimalité est atteinte à cause de la favorisation des tâches courtes sur les tâches plus longues. Les détracteurs de SRPT pointent le fait que, sous une très forte charge, la plus longue tâche du système n'est jamais élue par l'ordonnanceur.

Il a cependant été montré (prouvé [BHB01], simulé [GW03] et expérimenté [HBSBA03, SHB06]) qu'avec un trafic réaliste (file d'attente M/G/1), SRPT ne pénalise que très peu les tâches longues en comparaison à PS. La pénalisation des longues tâches n'apparait que sous des charges approchant la capacité maximale du réseau. Pour certaines entrées, aucune tâche n'est pénalisée par SRPT, qui améliore alors le temps de réponse de toutes les tâches par rapport à PS. Ce phénomène est illustré par la figure 2.4, qui en montre un exemple trivial.

Dans certaines situations, la taille des données à envoyer n'est pas connue à l'avance. C'est par exemple le cas des contenus Web générés dynamiquement. Il est alors possible de travailler avec des estimations de la taille de contenus, calculée par exemple en moyennant les exécutions précédentes. Une analyse des performances de SRPT en fonction de la précision des estimations de la taille des tâches a été réalisée par Lu *et al.* [LSD04].

11. SRPT : *Shortest Remaining Processing Time*

Le cas pondéré

Dans de nombreuses situations, les tâches en attente dans le système sont consommatrices de ressources. Par exemple, dans un serveur Web, chaque requête en attente occupe une certaine quantité de mémoire. Ainsi, de nombreux travaux s'intéressent à l'optimisation des temps de réponse pondérés. Pour chaque tâche i, on associe alors un poids w_i. L'algorithme WSRPT [12] [Smi56] optimise la version simplifiée de ce problème, $1||\sum w_iC_i$, dans laquelle toutes les tâches sont disponibles au temps $t = 0$. WSRPT consiste à élire à chaque instant la tâche minimisant le rapport entre durée restante et poids.

Le problème considérant des dates d'arrivée dans le système, $1|r_i, pmtn|\sum w_iC_i$, est quant à lui NP-dur [LLLRK84, LY90]. Plusieurs variantes de WSRPT ont été proposées afin d'atteindre les meilleures performances possibles dans ce contexte [CKZ01, BD07].

Prise en compte de l'équité

La notion d'équité d'un ordonnanceur, contrairement à celle de sa performance, est difficile à définir. Certains travaux se sont intéressés à classifier les algorithmes d'ordonnancement en fonction de leur équité. Dans [FH03], un algorithme est dit équitable si et seulement si il fournit un temps de réponse inférieur ou égal à PS pour toute tâche et pour toute entrée. Selon cette définition, SRPT est montré non-équitable (l'exemple de la figure 2.4 n'est qu'un cas particulier). Un nouveau protocole nommé FSP [13] est alors présenté, approchant les performances de SRPT tout en garantissant l'équité. FSP, comme SRPT, prioritise les tâches au temps restant les plus courts, tout en évitant la stagnation des plus longues tâches.

Un autre travail [WHB03] propose une classification des algorithmes comme étant soit toujours équitable, parfois équitable ou jamais équitable. Dans [GW06], deux types d'iniquité sont identifiés. L'iniquité endogène pénalise certaines tâches à cause de leur durée, alors que l'iniquité exogène pénalise les tâches en fonction de la charge du serveur au moment de leur entrée dans le système. De nombreux travaux considèrent qu'un algorithme est équitable si sa vitesse de traitement n'est pas influencée par sa taille (absence d'iniquité endogène). En s'appuyant sur cette définition, Gong *et al.* présentent des algorithmes intermédiaires entre SRPT et PS [GW04].

Une nouvelle métrique de mesure de l'équité nommée RAQFM [14] est proposée dans [RLAI04]. Fondée sur la notion de discrimination individuelle des tâches, elle a pour objectif de considérer à la fois la durée des tâches et leur ancienneté dans le système. Cette nouvelle notion d'équité se démarque des précédentes en ne se fondant pas uniquement sur les dates de complétion des tâches.

De la théorie à la pratique

Les travaux cités précédemment permettent de maitriser performances et équité, éventuellement avec pondération. La clé de la performance repose, selon toutes ces études, sur la prise en compte de la taille restante des tâches à ordonnancer. Des solutions permettant aux serveurs Web tels que Apache d'utiliser une stratégie de type SRPT ou FSP ont été proposées [HBSBA03], nécessitant des modifications au sein même du système d'exploitation, afin d'en adapter la gestion des *sockets*. Pourtant, les serveurs Web utilisés aujourd'hui refusent tous

12. WSRPT : *Weighted Shortest Remaining Processing Time*
13. FSP : *Fair Sojourn Protocol*
14. RAQFM : *Resource-Allocation Queueing Fairness Measure*

d'adopter ces stratégies, continuant de leur préférer le classique tourniquet. Un tel écart entre théorie et pratique est plutôt surprenant ; il remet en question les métriques utilisées dans les travaux que nous avons présentés.

2.4 Synthèse

L'étude des serveurs d'applications que nous avons présentée est riche d'enseignements. Initialement programmés ≪ naïvement ≫, suivant le rôle initial des abstractions fournies par le système d'exploitation, ces serveurs n'atteignaient pas des performances suffisantes. Leurs concepteurs ont donc du envisager des changements d'architecture : du flot de contrôle linéaire des *threads* aux évènements gérés de manière *ad hoc*, ou des lectures/écritures de descripteurs de fichiers aux routines spécialisées. Les interfaces des routines système ainsi que leur implémentation ont alors été corrigées, adaptées aux besoins de ces serveurs aux fortes exigences. On a observé une évolution de même nature au niveau des intergiciels, lorsque les applications les plus exigeantes (exemple de la notification de clients) se sont trouvées bridées par leurs conteneurs.

L'état de l'art de la transposition des protocoles de l'Internet aux systèmes embarqués nous montre qu'il est difficile de s'éloigner des constructions logicielles traditionnelles. Ainsi, les piles IP pour l'embarqué proposent une abstraction du réseau tout à fait similaire à celle rencontrée dans les systèmes d'exploitation classiques où chaque canal est représenté par un *socket*, dans lequel on peut lire ou écrire des données de manière bloquante. Contrairement à ces solutions génériques, les applications embarquées sont la plupart du temps dédiées à une tâche précise. C'est par exemple le cas des serveurs Web embarqués, dont les implémentations les plus simples s'abstraient de toute interface, au profit d'une conception intégrée, leur permettant d'envisager un déploiement dans des matériels incroyablement contraints. Il est cependant accepté que la compacité de ces systèmes s'accompagne d'un sacrifice de leurs performances [Dun07].

La question du juste découpage entre noyau et applications a été largement étudiée et discutée par le passé, aboutissant aux différents modèles de système d'exploitation que l'on connait aujourd'hui, du système intégré à l'exo-noyau. Autour de ces modèles sont venues se greffer des solutions aux besoins spécifiques des systèmes embarqués, apportant des réponses concrètes aux questions de la gestion de multiples tâches, de l'économie d'énergie ou du chargement dynamique d'applications.

Dans le contexte de serveurs d'applications embarqués, la tâche du système semble trop dédiée pour qu'il soit pertinent d'utiliser un système fournissant une interface générique. Il est alors naturel de se tourner vers les systèmes intégrés ; démarche classiquement suivie pour les logiciels dédiés. Afin de repousser les limites atteintes par les solutions existantes (performances et fonctionnalités sacrifiées pour la compacité) et en s'inspirant de l'évolution des serveurs d'applications (dont les interfaces se sont adaptées aux comportements applicatifs de haut niveau), nous pensons qu'il est nécessaire de dépasser le simple système intégré, pour se tourner vers une système d'exploitation dédié alliant implémentation transverse et interface applicative de très haut niveau. Dans le chapitre suivant, nous décrivons les objectifs, les défis et l'architecture d'un tel système.

Troisième chapitre

Vers un macro-noyau dédié au Web des objets

*Beaucoup s'accordent à penser que l'Internet, dans quelques années, sera étendu par inclusion des objets qui nous entourent. On parle alors d'*Internet des objets*. Une solution séduisante – le* Web des objets *– consiste à construire un tel réseau autour des technologies du Web qui ont fait le succès de l'Internet. Dans ce chapitre, nous présentons tout d'abord le Web des objets, constituant le contexte de ce mémoire. De ce contexte nous extrayons la problématique de cette thèse. Enfin, nous proposons une approche nouvelle pour y apporter des solutions qui sera développée dans la suite du document.*

3.1 Contexte – le Web des objets

L'Internet des objets est en pleine construction, si bien que ses standards ne sont pas encore complètement définis. Avant d'en présenter une solution spécifique – le Web des objets – nous en décrivons les propriétés et les besoins.

3.1.1 L'Internet des objets et les réseaux personnels

L'Internet des objets représente l'extension de l'Internet aux objets qui nous entourent. Il s'agit d'un concept – et non d'un ensemble de technologies – qui est le plus souvent associé aux technologies RFID ou au Web sémantique. On retiendra la définition suivante de l'Internet des objets, proposée en 2008 par Bassi *et al.* [Bas08] :

> ≪ *Things having identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environmental, and user contexts.* ≫

L'Internet des objets va de simples objets identifiés par étiquettes électroniques à des objets embarquant un système informatique actif communiquant. C'est à cette classe d'objets, capables de prendre part activement aux interactions, que nous portons notre attention. L'ensemble des objets avec lesquels une personne est capable d'interagir constitue un réseau personnel, ou PAN [1].

1. PAN : *Personal Area Network*

Besoins des réseaux personnels

Les réseaux personnels sont volatils, constitués des objets entourant leurs usagers. Cinq besoins principaux ont été identifiés pour le bon fonctionnement d'un réseau personnel [For08, Sti08] :

Découverte La découverte permet de tenir à jour la liste des objets et services constituant le réseau personnel ;

Adressage L'adressage est la manière dont les objets et les services qu'ils proposent sont rendus accessibles ;

Contrôle Le contrôle est ce qui permet de piloter les objets du PAN ;

Notification d'évènements La notification d'évènements permet aux objets du PAN de notifier un usager de l'occurrence d'un évènement ;

Présentation La présentation permet aux objets de fournir une interface aux usagers afin de piloter les interactions.

Étant donné que le réseau personnel est construit autour d'un usager, il n'a d'existence qu'en présence de ce dernier. Les objets qui le composent ne deviennent accessibles que lorsqu'ils ont été inclus dans un PAN. Il est également envisageable qu'un objet soit partagé par plusieurs réseaux personnels, donc accessible par plusieurs usagers. Les protocoles de liaison couramment associés aux PANs sont nombreux, incluant Bluetooth, IrDA ou ZigBee. Les nœuds peuvent communiquer directement entre eux si leurs interfaces le permettent, ou doivent être routés par l'usager du réseau, qui sert aussi de passerelle dans le cas d'un accès à un réseau externe, tel que l'Internet. Les couches applicatives, quant à elles, sont le plus souvent gérées par une architecture soit orientée service, soit de type REST [2].

Architectures orientées service

Les Services Web constituent un ensemble de protocoles permettant la communication entre clients et serveurs utilisant XML et se fondant sur les standards SOAP et WSDL. Ces standards utilisent le Web comme média, interprétant HTTP comme un protocole de transport. Les services Web sont utilisés principalement pour la construction de systèmes distribués.

L'utilisation de ces protocoles dans les réseaux personnels a été standardisée avec UPnP [3] [For08]. UPnP se fonde sur les technologies des Services Web afin de construire une architecture orientée service. Son successeur annoncé, DPWS [4], est fondé sur une construction similaire. Il vise entre autre à améliorer la sécurisation des échanges avec les Services Web.

Architectures REST

On oppose classiquement aux architectures orientées service les architectures de style REST [Fie00]. Dans une architecture REST, chaque fonctionnalité des applications est associée à une URL. Les clients interagissent avec les applications en respectant la sémantique originale des commandes HTTP *get*, *post*, *put*, *delete*. Les échanges se font sans état et de manière indépendante. Tout état applicatif est donc stocké uniquement du coté des clients et les requêtes contiennent les informations du client dont le serveur a besoin pour générer la réponse. Elles

2. REST : *REpresentational State Transfer*
3. UPnP : *Universal Plug and Play*
4. DPWS : *Devices Profile for Web Services*

peuvent être traitées sans ordre et par des entités autonomes. En conséquence, les serveurs n'ont pas besoin de maintenir d'informations sur l'état applicatif des clients.

Partant du constat que les Services Web constituaient une très lourde suite protocolaire et imposaient un couplage fort entre les entités l'utilisant, Stirbu propose en 2008 [Sti08] de transposer le style REST original du Web aux réseaux personnels, constituant ainsi un *Web des objets*.

3.1.2 Vers un Web des objets

Le Web des objets [Sti08, GT09] est une approche centrée sur l'usager, dans laquelle les objets d'un réseau personnel sont accessibles *via* les protocoles du Web, suivant une architecture REST. Cette architecture diminue grandement le coût du ticket d'entrée dans un tel réseau en y simplifiant sensiblement les interactions. Dans le Web des objets, l'usager n'a besoin que d'un simple navigateur Web, aujourd'hui disponible sur tout ordinateur, assistant personnel ou *smartphone*. Les objets, quand à eux, exécutent un serveur d'applications Web auquel se connectent les clients. Le Web des objets apporte des solutions à tous les besoins des réseaux personnels décrits dans la sous-section précédente :

Découverte La découverte des objets peut être réalisée à l'aide d'un dépot centralisé, par exemple géré par la machine de l'usager [Sti08] ;

Adressage Au niveau réseau, les objets sont accédés par leur adresse IP, fournie par exemple avec le protocole DHCP. Au niveau applicatif, les objets et leurs ressources sont adressés à l'aide de leurs URLs ;

Contrôle Le contrôle des objets se fait en envoyant des commandes HTTP aux objets (*get* pour obtenir des informations, *post* pour piloter l'objet, *etc.*). C'est l'application Web embarquée qui exécute l'action spécifiée par la requête ;

Notification d'évènements La notification peut être réalisée en contexte Web en utilisant le paradigme Comet [Rus06] ;

Présentation Les objets présentent leurs données et leur services sous la forme d'applications Web. C'est le navigateur du client qui est en charge du rendu et qui permet à l'usager d'interagir avec l'objet.

En plus de son adaptation aux besoins des réseaux personnels, le Web des objets présente d'excellentes propriétés d'interopérabilité au niveau réseau comme applicatif.

Interopérabilité des communications

Le Web des objets se fonde sur les technologies de l'Internet, il se repose donc sur la suite protocolaire IP. Les objets du réseau peuvent ainsi accéder (ou être accédés) directement aux (ou par des) nœuds des autres réseaux existants, tels que l'Internet. L'extension du réseau Internet avec le Web des Objets se fait donc naturellement, sans passerelle, sans nouveau protocole. De plus, le protocole IP supporte déjà le déploiement au-dessus d'un très grand nombre de protocoles de liaison, ce qui résout le problème de la variété des interfaces de communication des divers équipements du réseau personnel.

Notons également que les réseaux personnels sont susceptibles de manipuler des données sensibles (*e.g.* données bancaires ou relatives à la santé). Comme en démontre l'évolution de UPnP vers DPWS, la confidentialité constitue une préoccupation majeure des PAN. Dans le Web des objets, la sécurité est naturellement supportée par HTTPS. Ce protocole est

aujourd'hui largement utilisé sur Internet, se fondant sur des protocoles de chiffrement éprouvés. De plus HTTPS permet l'utilisation de techniques de chiffrement variées. Il est par exemple possible d'utiliser un chiffrement fondé sur les courbes elliptiques, particulièrement adapté aux d'équipements limités en ressources [Woo01]. Le groupe 6LoWPAN [HC08], en charge de permettre la transposition d'IPv6 aux réseaux utilisant une liaison de type IEEE 802.15.4, propose quant à lui de sécuriser le réseau au niveau IP, avec le protocole IPSeC. Ces solutions de sécurité, en plus d'être éprouvées, sont déjà utilisées dans les réseaux existants ; c'est ce qui permet aux nœuds du Web des objets d'interagir de manière sécurisée et transparente avec Internet.

La grande facilité que l'on constate pour lier le Web des objets aux réseaux existants ouvre des portes à de nouvelles interactions entre applications.

Interopérabilité des applications

Les applications Web, constituant le cœur du Web des objets, présentent de bonnes propriétés d'interopérabilité. Elles sont en effet exécutables depuis tout navigateur, système d'exploitation et équipement matériel. Les communications entre applications Web sont également aisées, permettant la construction d'applications par fusion de contenus provenant de diverses sources. Avec le Web des objets, ce type de fusion, appelé *mashup*, est naturellement possible entre objets du réseau personnel [GTPL09, GT09] ou entre un objet et des applications d'Internet.

Des applications très novatrices sont permises par l'utilisation du *mashup*. On citera l'exemple d'un site d'une banque qui, recoupant ses données avec celles stockées sur la carte bancaire, propose à l'usager une liste d'opérations parfaitement à jour. On peut également imaginer un dispositif personnel stockant un carnet d'adresses, qui, lors de la navigation sur Internet, se voit fusionné avec les diverses pages rencontrées (injection de liens relatifs à un contact, *etc.*). Ces *mashups* offrent des possibilités infinies.

Une autre force du Web des objets réside dans le fait que le développement d'applications Web est aujourd'hui une activité largement maitrisée. De nombreux outils, bibliothèques ou méthodologies sont disponibles, connus et massivement utilisés. Le développement d'application pour le Web des objets en bénéficie directement, rendant aisée la programmation des équipements environnants. L'écriture d'une application Web est de toute évidence bien plus accessible, rapide et sûre, que la programmation d'une application typique pour système embarqué. La portabilité des applications Web est également particulièrement intéressante lorsqu'il s'agit de déploiement dans des matériels hétérogènes.

3.1.3 Du Web mondial au Web des Objets

Après quelques années de recul, on observe que le succès de l'Internet est dû à l'adoption de quelques applications simples : principalement le Web et les courrier électroniques. Aujourd'hui, les technologies du Web sont incroyablement répandues. L'accès constant à l'information depuis tout lieu et équipement est maintenant entré dans les mœurs. C'est de cet état de fait que nait le grand potentiel du Web des objets. Il serait toutefois déraisonnable de penser que tout ce qui s'est appliqué au *World Wide Web* s'appliquera de la même manière au Web des objets. Ces deux réseaux présentent en effet quelques différences importantes, en termes d'architecture de réseaux et de nature des serveurs qui en composent le cœur.

Architecture des réseaux

Le *World Wide Web* et d'une manière plus générale, le réseau Internet, sont fondés sur une architecture centrée sur les serveurs. Les applications Web sont publiquement accessibles *via* leur nom de domaine et sont en charge d'un ensemble d'utilisateurs. Le Web des objets, quand à lui, suit une architecture centrée sur les clients. Un réseau personnel n'existe en effet que par la présence de son usager, qui en constitue le cœur. Les objets ne sont *a priori* pas accessibles publiquement, même s'ils peuvent être partagés par plusieurs réseaux personnels, voire rendus visibles sur Internet.

Notons également que la structure du *World Wide Web* est statique, alors que celle du Web des objets est dynamique ; les objets pouvant se connecter et se déconnecter à tout instant. L'étape de découverte des services disponibles est particulièrement importante dans ce contexte et ne peut être réalisée de la même manière que sur Internet, c'est-à-dire par l'usage de moteurs de recherche.

Nature des serveurs

Dans le *World Wide Web*, les serveurs sont basés sur des machines dédiées puissantes, disposant de grandes quantités de mémoire et de liaisons réseau rapides. Ils sont capables de gérer plusieurs centaines de milliers de requêtes simultanément. Les serveurs les plus exposés sont quand à eux composés d'une ferme de machines auxquelles ils répartissent la charge des requêtes à traiter.

Dans le Web des objets, les serveurs s'exécutent sur un système informatique embarqué. Ces systèmes, aux caractéristiques très variées, sont décris plus précisément en sous-section 3.2.1 ; on retiendra dans un premier temps qu'ils sont plus contraints que les équipements utilisés par les clients pour se connecter. Cette inversion de puissance entre client et serveur a des conséquences sur la manière de penser les applications Web. Alors que dans le *World Wide Web* les applications à clients légers ou à clients lourds coexistent, avec chacune leurs avantages et inconvénients, dans le Web des objets, il est indiscutable que le client doit être aussi lourd que possible : afin d'alléger le serveur, il est en effet préférable de déporter sur le client tout traitement qui peut l'être.

Les propriétés si particulières du Web des objets en termes de réseau et de ressources matérielles, associées aux fortes exigences fonctionnelles des applications Web, sont à l'origine de la problématique de ce document.

3.2 Problématique – un défi de taille

À l'heure où la tendance va au déploiement de navigateurs Web dans tous les équipements personnels munis d'un écran, le Web des objets propose d'embarquer des serveurs Web dans les systèmes environnements qui, eux, sont si contraints qu'ils ne disposent pas d'écran. Cette extension du Web des objets jusqu'à l'informatique enfouie constitue un grand défi. Il s'agit en effet d'embarquer des serveurs d'applications Web exigeants (en performances et fonctionnalités) dans des équipements aux fortes contraintes matérielles.

Équipement	Processeur	Mémoire	Interface réseau
Funcard 7 Carte à puce	Atmel AT90S8515 AVR 8 bits, 4 MHz	512 octets RAM 512 EEPROM 8 ko Flash interne 64 ko Flash externe	Liaison série, 115,2 Kbps
Platinium Carte à puce	Atmel AT90SC6464C AVR 8 bits, 4 MHz	3 ko RAM 64 ko EEPROM 64 ko Flash interne	Liaison série, 115,2 Kbps
MICAz Capteur	Atmel ATmega128L AVR 8 bits, 8 MHz	4 ko RAM 4 ko EEPROM 128 ko Flash interne 512 ko Flash externe	Liaison série, 115,2 Kbps 802.15.4, 250 Kbps
WSN430 Capteur	TI MSP430-1611 16 bits, 8 MHz	10 ko RAM 48 ko EEPROM 1 Mo Flash externe	Liaison série, 115,2 Kbps 802.15.4, 250 Kbps

Table 3.1 – Caractéristiques matérielles d'une sélection d'équipements

3.2.1 Des équipements aux ressources limitées

Le déploiement à grande échelle du Web des objets, touchant l'informatique enfouie, nécessite de minimiser le coût unitaire des équipements ciblés. On espère tirer parti des progrès technologiques exponentiels non pas pour augmenter les ressources unitaires des équipements, mais pour en permettre la production de masse. Les systèmes informatiques que nous visons ont donc de très fortes contraintes matérielles, en termes de microprocesseur, de mémoire et d'interfaces réseau. La table 3.1 détaille les caractéristiques de quelques équipements typiques.

Microprocesseur

Les microprocesseurs utilisés dans les systèmes embarqués ont des caractéristiques variées, offrant différents compromis entre performances, consommation énergétique et coût de production. Les processeurs auxquels on s'intéresse fonctionnent à une cadence de quelques mégahertz et sont principalement basés sur des registres de 8 ou 16 bits. Le plus souvent, ils ne fournissant pas de gestion de multiples niveaux de privilèges, ni d'unité de gestion de la mémoire (MMU [5]). Pour répondre à des besoins spécifiques, par exemple pour la confidentialité des données échangées par les cartes à puces, un coprocesseur cryptographique peut être présent sur la même puce que le processeur principal.

Mémoire

Les systèmes embarqués disposent de plusieurs mémoires aux propriétés hétérogènes, en termes de latence, de débit, de nature des accès ou de densité (nombre de transistors requis par bit). Alors que certaines mémoires sont situées sur la même puce que le microprocesseur, d'autres sont externes. La RAM [6], ou mémoire de travail, est à la fois rapide, adressable et accessible en lecture comme en écriture. Elle ne conserve les données que lorsqu'elle est

5. MMU : *Memory Management Unit*
6. RAM : *Random Access Memory*

alimentée : on dit qu'elle est volatile. Cependant, elle est très coûteuse (en termes de transistors par bit) et n'est utilisée qu'en quantités limitées (typiquement de l'ordre de quelques kilo-octets dans le cas des cibles que nous visons).

La ROM [7], quant à elle, n'est accessible qu'en lecture (elle est écrite une unique fois avant déploiement), mais est la moins coûteuse des mémoires et peut donc être présente en grandes quantités. L'EEPROM [8] et la mémoire Flash des mémoire non volatiles (ou persistantes) qui permettent la lecture et l'écriture. Elles sont moins rapides que la RAM (principalement en écriture), mais sont aussi plus denses, donc présentes en de plus grandes quantités. L'EEPROM est adressable à l'octet, alors que la Flash n'est accessible que par pages. Ces mémoires peuvent être internes ou externes à la puce du microprocesseur. La Flash peut être disponible en très grandes quantités (de l'ordre du méga voire du giga-octet) et ainsi servir de mémoire de stockage de masse.

Interfaces réseau

Dans le cadre du Web des objets, les systèmes auxquels nous nous intéressons disposent tous d'interfaces de communication. Par exemple, les cartes à microprocesseur communiquent avec un lecteur de carte *via* une liaison série ou, pour les plus récentes, un port USB. D'autres équipements, comme les capteurs de terrain, utilisent la norme IEEE 802.15.4, un protocole de communication sans fil à bas débit assurant une faible consommation énergétique (souvent utilisé avec la pile de communication ZigBee). De nombreux autres protocoles comme Ethernet, IrDA ou Bluetooth sont par exemple utilisés par les systèmes de domotique. Ces interfaces offrent des débits de l'ordre de la centaine de kilo-bit ou du méga-bit).

Le déploiement de serveurs d'application au sein d'équipements aussi contraints que ce que nous avons décrits constitue un premier défi. De plus, les applications Web dont le Web des objets a besoin présentent de très fortes exigences fonctionnelles, rendant cette tâche encore plus difficile.

3.2.2 Des applications réseau exigeantes

Les applications Web se sont développées dans un contexte où les accès à Internet étaient de plus en plus rapides et les machines de plus en plus puissantes. Les technologies qui y sont associées sont de plus en plus demandeuses de ressources, du coté des serveur comme de celui des clients.

Répartition des charges entre client et serveur

Les applications Web naissent de l'association de contenus définis statiquement et de contenus générés dynamiquement. Les contenus dynamiques sont générés par du code qui s'exécute soit sur le serveur, soit sur le client. Les contenus générés par le serveur ont typiquement accès à des données de compte associées aux clients ou à des données privées stockées par exemple en base de données. Les contenus générés par le client permettent de dynamiser l'application au sein du navigateur, sans aller-retours entre clients et serveurs.

La technique AJAX [9] [Gar05] permet d'obtenir des applications hautement interactives grâce à une fusion efficace de données statiques et dynamiques générées sur le serveur et le

7. ROM : *Read-Only Memory*
8. EEPROM : *Electrically Erasable Programmable Read-Only Memory*
9. AJAX : *Asynchronous Javascript And XML*

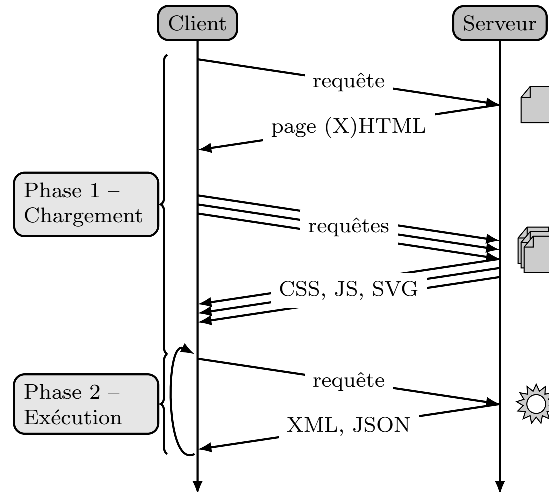


FIGURE 3.1 – Schéma de fonctionnement d'AJAX en deux phases

client. Elle est aujourd'hui cœur du concept d'application Web et sera aussi nécessaire au Web des objets qu'elle ne l'est au *World Wide Web*. On peut décrire le fonctionnement d'une application AJAX en y distinguant deux phases, résumées par la figure 3.1 :

Phase 1 – chargement Le client commence par collecter les fichiers statiques constituant l'application, incluant des informations de structure ((X)HTML [10]), de mise en forme (CSS), du code applicatif (JavaScript) et éventuellement d'autres données multimédia ;

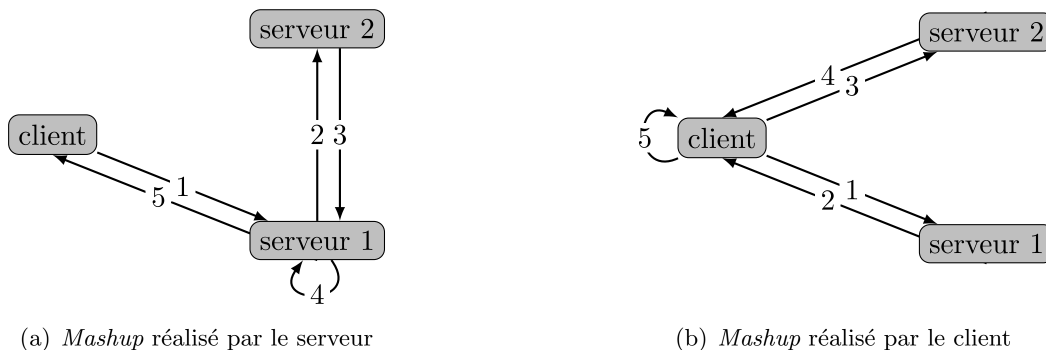
Phase 2 – exécution Le client exécute le code applicatif téléchargé lors de la première phase, puis interagit avec le serveur en envoyant des requêtes de manière asynchrone. Les réponses sont le plus souvent des contenus bruts (au format XML ou JSON) générés par le serveur, mis en forme et intégrés à l'application lors de leur interprétation par le client.

La technique AJAX présente de bonnes propriétés dans le contexte du Web des objets, où les clients disposent de plus de ressources que les serveurs. Les données dynamiques générées par le serveur sont envoyées sous forme brute, non mise en forme, ce qui allège le trafic et la charge du serveur. Un certain nombre de traitements sont simplement déportés du serveur vers le client.

C'est également à l'aide d'AJAX qu'il est possible d'effectuer du *mashup* applicatif efficace, déchargeant au maximum le serveur. La figure 3.2(a) illustre la réalisation de *mashup* sans AJAX, réalisée par le serveur. Lorsqu'une fusion de données est nécessaire, le serveur 1 est en charge d'interroger le serveur 2, de fusionner les contenus, puis d'envoyer le résultat au client. Avec AJAX, le *mashup* peut se réaliser du coté du client, comme l'illustre la figure 3.2(b). C'est alors un script qui, en s'exécutant dans le navigateur du client, interroge le serveur 2 puis réalise la fusion des données. La charge du serveur 1 est ainsi diminuée et la réactivité du client est augmentée (la page s'affiche avant la fin de la fusion de données). Notons qu'en pratique, afin de dépasser les politiques de sécurité des navigateurs, il est nécessaire de passer par une technique de script croisé entre sites, ou XSS.

Le support d'applications AJAX constitue ainsi le premier objectif en termes de fonctionnalités pour les applications du Web des objets, car il permet une répartition efficace des

10. (X)HTML : *(eXtensible) Hypertext Markup Language*



(a) *Mashup* réalisé par le serveur

(b) *Mashup* réalisé par le client

FIGURE 3.2 – Illustration du *mashup* centré sur le serveur ou le client, en 5 étapes. Le client exécute une application du serveur 1, intégrant des informations en provenance du serveur 2.

charges entre client et serveur. Cet allégement des tâches du serveur permet à ce dernier de mieux passer à l'échelle, servant un plus grand nombre de requêtes et de clients.

Passage à l'échelle et performances

Les serveurs du Web des objets, malgré leurs ressources limitées et l'aspect local des réseaux dans lesquels ils évoluent, devront supporter le passage à l'échelle et gérer plusieurs centaines de connexions. En effet, les applications Web nécessitent aujourd'hui que chaque client ouvre plusieurs connexions vers chaque serveur auquel il est connecté, afin de gérer plusieurs requêtes asynchrones simultanées. Un navigateur Web tel que Mozilla Firefox ouvre ainsi jusqu'à 15 connexions par serveur. De plus, chaque serveur doit être capable de gérer plusieurs clients, car il peut être inclus dans plusieurs réseaux personnels, voire être accessible depuis Internet. Les applications réalisant du *mashup* sont elles aussi responsables d'une augmentation du nombre de clients potentiellement connectés à chaque serveur.

Afin de répondre à un des besoins fondamentaux des réseaux personnels (voir sous-section 3.1.1), les applications du Web des objets devront supporter la notification d'évènements (utile pour capteur souhaitant lever une alerte ou pour une application qui a besoin d'informer ses clients de la mise à jour de divers états du serveur). Dans le contexte d'applications Web, c'est le paradigme Comet (présenté en sous-section 2.3.2) qui permet ce type de notification. Cependant, le support de Comet en environnement contraint constitue un défi important ; il est même encore considéré comme problématique dans le contexte de serveurs d'applications classiques. Comet impose en effet au serveur de maintenir ouvertes autant de connexions qu'il y a de clients en attente de notification, engendrant classiquement une consommation de mémoire importante du coté du serveur.

Notons également que l'adoption d'un Web des objets dépend directement du confort d'usage qu'il apporte. Un accès universel à tous les systèmes informatiques environnants n'est attractif que s'il permet des interactions suffisamment réactives. Les serveurs applicatifs du Web des objets devront dont présenter des performances aussi élevées que possible, tout en gérant plusieurs dizaines voire plusieurs centaines de connexions simultanées. Il s'agit alors de concilier ce besoin de performances et de passage à l'échelle aux fortes contraintes matérielles des équipements visés.

3.2.3 Allier compacité, fonctionnalités et performances

L'extension du Web des objets aux équipements de l'informatique enfouie nécessite de supporter de nombreuses fonctionnalités de manière performante au sein de matériels très contraints. Dans ce type d'équipements, une approche utilisant un serveur Web au-dessus d'un système d'exploitation, même temps réel, n'est pas appropriée, car elle ne réduit pas au minimum la consommation de ressources du système. Il est alors nécessaire de s'orienter vers des solutions où le serveur Web et le système d'exploitation sont conçus conjointement, comme celles que nous avons présentées dans la sous-section 2.2.3.

Cependant, aucune solution existante n'a tenté d'allier compacité, fonctionnalités et performances au sein d'un même système. Les serveurs Web les plus compacts sont par exemple limités à servir des pages statiques de moins d'un segment. Ceux qui supportent la génération de contenus dynamiques sont contraints de le faire de manière très limitée, c'est-à-dire en imposant une taille maximale et sans gérer de requêtes concurrentes (*c.f.* sous-section 2.2.3, page 26). Aucune de ces solutions ne supporte Comet ni n'intègre des considérations de passage à l'échelle, en supportant de nombreuses connexions et en se souciant de leur ordonnancement.

Le plus souvent, c'est la mémoire volatile qui constitue le goulot d'étranglement des performances des piles IP et serveurs Web embarqués. Sur des équipements ne disposant que de quelques kilo-octets de mémoire vive, le support de plusieurs connexions et la gestion de contenus de grande taille semblent difficiles (dans TCP, la taille maximale des segments est habituellement de plus d'un kilo-octet). Dans ses travaux sur la programmation de systèmes embarqués communicants [Dun07], Dunkels considère que les petits systèmes produisent si peu de données que la dégradation de performances observée avec leur pile de communication n'est pas un problème important. En conclusion de ses travaux sur uIP et lwIP, il suggère que la réduction de l'empreinte mémoire se fait nécessairement au détriment des performances du système.

Ce compromis entre mémoire occupée et performances ne satisfait pas les fortes exigences du Web des objets. Afin de satisfaire ces dernières, nous proposons de repenser l'architecture même du système constituant le serveur d'applications embarqué.

3.3 Approche – vers un système à macro-noyau dédié

Notre approche se fonde sur une nouvelle architecture de système d'exploitation – dite à *macro-noyau* – pour les systèmes informatiques dédiés à une famille d'applications, comme c'est le cas des équipements du Web des objets.

3.3.1 Systèmes dédiés à une famille d'applications

L'application du Web des objets nécessite d'embarquer des serveurs d'applications Web dans un grand nombre d'équipements informatiques. On considère ces serveurs comme un exemple de système dédié à une famille d'applications. Une famille d'applications constitue un ensemble d'applications ayant un même modèle d'exécution, nécessitant une base commune pour se déployer et s'exécuter. Les applications Web sont un exemple de famille d'applications. On citera également les applications SOAP ou les *applets* d'une carte à puce Java Card.

Dans les systèmes classiques, les applications d'une même famille sont supportées par un intergiciel, à l'interface entre le système générique et les applications spécifiques. Suivant l'exemple des applications Web, un serveur d'applications constitue l'intergiciel. Il présente

aux applications Web une API leur permettant entre autre d'associer des ressources à des URLs, d'effectuer certains traitements sur réception de requêtes ou de générer des réponses HTTP. Chaque famille d'applications a des besoins spécifiques ; les intergiciels les supportant doivent proposer une API en adéquation avec ces besoins.

La réalisation d'un système dédié à une famille d'applications est possible suivant différentes structures de système d'exploitation (une présentation de ces structures est proposée en sous-section 2.1.3). Nous présentons ici les avantages et inconvénients des structures de système existantes, dans le cas particulier du support d'applications Web.

Systèmes intégrés

Les serveurs Web visant une empreinte mémoire minimale sont construits suivant une approche totalement intégrée, dans laquelle les ressources Web, le serveur Web, la pile IP et les pilotes sont tous construits conjointement (*c.f.* figure 3.3(a)). L'absence de frontières entre différentes couches du système ouvre des portes à de nombreuses optimisations, mais empêche la distinction de toute notion d'applications et d'API. En conséquence, les serveurs Web suivant cette approche sont limités au service de quelques contenus, le plus souvent sous fortes contraintes (*e.g.* limitation aux contenus statiques ou dynamiques de taille limitée à un segment, avec une unique application à chaque instant). Il est impossible d'y déployer des ressources Web non connues lors de la conception du système.

Systèmes à noyau monolithique

Les solutions reposant sur un système à noyau monolithique distinguent clairement les applications Web de leur conteneur (ou serveur), qui s'exécute directement au-dessus du noyau (parfois uniquement constitué d'une pile IP). Le conteneur d'applications est alors un intergiciel faisant le lien entre la pile IP et les applications Web (*c.f.* figure 3.3(b)). Un tel découpage présente des avantages évidents en termes de modularité, mais il impose aux applications Web de franchir deux interfaces avant d'atteindre le matériel. Chaque interface introduit nécessairement un surcoût, car elle empêche des optimisations d'implémentation et contraint la manière dont les accès d'une couche à l'autre sont réalisés. L'évolution des architectures de serveurs Web décrite en sous-section 2.3.1 montre que ces interfaces sont souvent le goulot d'étranglement des performances du système.

Systèmes à exo-noyau

Ce type de système présente de bonnes propriétés pour les systèmes dédiés, car il permet aux applications de gérer par elles-mêmes les accès au matériel et l'implémentation des couches basses du système. Au-dessus des pilotes, tout le logiciel est construit de manière intégrée (*c.f.* figure 3.3(c)). La pile IP, le serveur et les applications Web sont alors conçus conjointement et peuvent ainsi bénéficier de nombreuses optimisations. Cependant, dans la situation qui nous intéresse, où le système est dédié à une unique famille d'applications, la capacité des exo-noyaux à multiplexer le matériel est inutilisée ; on obtient une construction similaire à celle d'un système purement intégré.

Nous ne détaillons pas ici le cas des micro-noyaux, qui, dans notre contexte, constitue un intermédiaire entre noyau monolithique et exo-noyau. Partant du constat qu'aucune des approches classiques (systèmes intégrés, à noyau monolithique, à micro-noyau ou a exo-noyau)

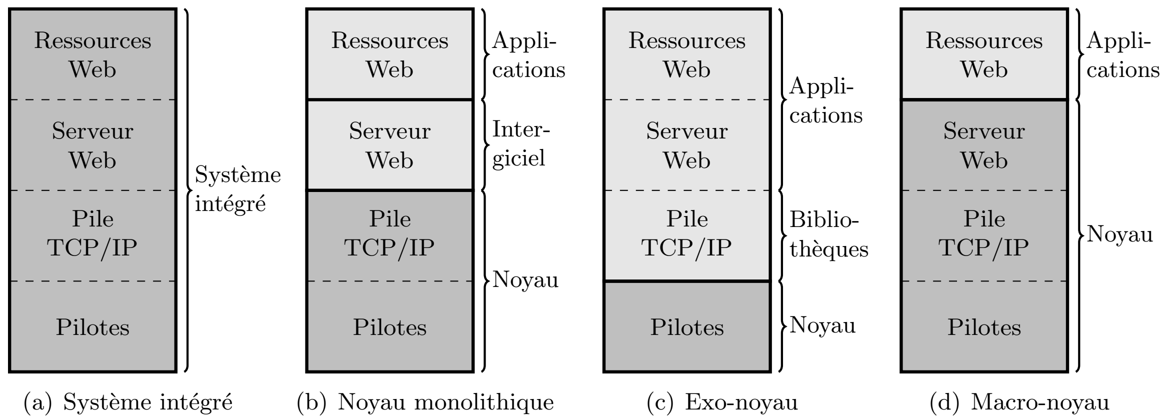


FIGURE 3.3 – Application des modèles de systèmes d'exploitation au cas du serveur Web

n'est parfaitement adaptée au support d'une famille d'applications, nous introduisons une architecture de système d'exploitation dédié fondée sur la notion de macro-noyau.

3.3.2 Vers un système à macro-noyau dédié

Le choix d'intégrer ou de séparer deux couches logicielles conduit respectivement à une amélioration des performances ou de la modularité du système. L'idée du macro-noyau est d'intégrer l'implémentation de toutes les couches basses du système et d'offrir une interface applicative de haut niveau, dédiée à une unique famille d'applications. Dans le cas du serveur d'applications Web, un macro-noyau inclut la gestion du matériel, la pile IP et un conteneur d'applications. Il offre aux applications Web une interface dédiée et adaptée à leurs besoins. Les appels système qu'il fournit sont moins généralistes et plus contraints, ce qui permet de guider le développement des applications et d'en assurer un support aussi efficace que possible. L'architecture d'un système à macro-noyau dédié au service d'application Web est illustrée par la figure 3.3(d). On souhaite maîtriser la taille de la TCB en compensant l'expansion verticale du noyau par la conception transversale et dédiée de ce dernier. En effet, la conception transversale permet d'alléger l'implémentation tandis que la conception dédiée n'impose pas de supporter des fonctionnalités aussi vastes que celles d'un système généralise.

Un système à macro-noyau peut être comparé à un système intégré, car il se fonde sur un très gros bloc logiciel conçu de manière transversale, donc potentiellement très efficace. Cependant, alors qu'un système intégré n'offre aucune interface applicative, un macro-noyau est parfaitement ouvert au déploiement d'applications inconnues *a priori*. Un tel système peut être fourni comme un tout, constituant une base de code sur laquelle des applications tierces peuvent s'appuyer.

D'une certaine manière, un système à macro-noyau s'approche également d'un système à noyau monolithique utilisant un conteneur d'applications (c'est-à-dire un intergiciel). Ces deux solutions offrent aux applications une interface de haut niveau, dédiée à une famille d'applications. Le support de cette interface dédiée est toutefois très différente dans ces deux systèmes. Un intergiciel est habituellement conçu comme une sur-couche du système d'exploitation et se voit ainsi contraint par les routines offertes par le système. Il est capable de présenter intelligemment les fonctionnalités du système aux applications, mais ne peut s'abstraire de l'interface imposée par le système. Du point de vue des fonctionnalités offertes,

un intergiciel offre finalement aux applications un sous-ensemble outillé des routines du système d'exploitation. En revanche, l'implémentation d'un macro-noyau est réalisée de manière intégrée, sans système sous-jacent ; il repose directement sur le matériel. La définition de l'interface qu'il propose peut donc être réalisée avec plus de libertés.

Le fait qu'un macro-noyau soit dédié à une unique famille impose de concevoir et de développer un système d'exploitation pour chaque domaine d'utilisation, comme par exemple celui de serveurs d'applications pour le Web des objets. On s'oppose ici aux approches spécialisantes, qui visent à automatiser la construction de logiciel au regard de diverses contraintes à partir d'une solution générique. On souhaite ici bénéficier de changements radicaux d'architecture, par exemple en termes d'interface applicative et de fonctionnement interne du noyau.

3.3.3 Concevoir un système à macro-noyau dédié au Web des objets

La conception d'un système à macro-noyau dédié est constituée de deux étapes principales : (i) l'identification des besoins de la famille d'applications visée et (ii) la construction du macro-noyau en lui-même. Ces étapes représentent un travail conséquent, qui n'est réalisé que pour une famille d'applications donnée. C'est là l'inconvénient majeur de cette approche, qui, nous l'espérons, sera largement compensé par les bénéfices de son application.

La phase d'identification des besoins applicatifs, qui doit commencer avant celle de conception du noyau, impose de réaliser une analyse poussée de la famille d'applications visée. On a ainsi une connaissance précise des besoins fonctionnels et des clés en permettant le support efficace. C'est à partir de cette analyse que l'on peut définir l'interface dédiée qu'offrira le macro-noyau. Cette interface doit satisfaire l'intégralité des besoins applicatifs, tout en permettant une interaction optimale entre applications et noyau. Son franchissement doit être aussi léger que possible. Dans le cadre du Web des objets, l'analyse se porte sur les applications Web. On espère ainsi en offrir une gestion optimale et ***supporter toutes les fonctionnalités requises*** par le Web des objets. Cette analyse et la définition de l'interface qui en découle est l'objet du chapitre 4.

La conception du noyau se fait au regard de l'interface qu'il doit supporter. Contrairement aux applications traditionnelles qui se contraignent à utiliser les routines fournies par le système, ici, c'est le système qui s'adapte aux besoins applicatifs préalablement identifiés. Il s'agit ensuite d'identifier les optimisations transversales qui permettront de construire un noyau aussi efficace que possible. Dans le contexte du Web des objets, il s'agit de construire conjointement le conteneur d'applications Web, la pile de communication et les pilotes afin de supporter l'interface dédiée ***tout en minimisant l'empreinte mémoire du système et en en maximisant les performances***. La conception d'un tel macro-noyau est étudiée dans le chapitre 5. Son implémentation et la mesure de ses bénéfices sont le sujet du chapitre 6.

Quatrième chapitre

Analyse du support applicatif dédié

La première étape dans la conception d'un système d'exploitation dédié consiste à étudier la famille d'application visée afin d'en cerner précisément les besoins. Dans notre contexte, il s'agit de fournir un support pour des applications Web répondant aux exigences et aux spécificités du Web des objets. Les contributions de ce chapitre sont doubles. Tout d'abord, nous analysons le trafic auquel un serveur d'application est confronté, puis nous présentons une taxinomie des applications Web. Partant de cette étude, nous proposons un ensemble de stratégies en vue d'adapter le comportement interne de la pile de communication aux propriétés des applications. L'objectif est ici de tirer parti des informations applicatives de haut niveau afin d'optimiser le comportement du système en termes de performances et de mémoire consommée.

4.1 Analyse du trafic

Dans un premier temps, nous nous intéressons à la modélisation du trafic produit par les applications du Web des objets. Nous introduisons ensuite une nouvelle métrique, la *charge mémoire*, dont le but est de synthétiser les performances et la consommation mémoire du système. Cette analyse constitue une étape préalable pour la définition des stratégies permettant l'adaptation de la pile de communication aux propriétés applicatives.

4.1.1 Contexte de l'analyse

L'analyse que nous proposons vise à estimer l'impact des propriétés du réseau et des paramètres du protocole TCP dans le cas d'un serveur du Web des objets.

Propriétés du réseau

Les équipements auxquels nous nous intéressons disposent d'interfaces de communication peu coûteuses telles qu'une liaison série, un émetteur-récepteur radio faible consommation (détaillé en sous-section 3.2.1, page 42). Ces interfaces permettent des débits allant de la centaine de kilo-bits par seconde à un méga-bit par seconde.

On s'intéresse également à la latence des communications entre client et serveur, qui dépend principalement de la localisation des deux machines sur le réseau. Par exemple, les liaisons point à point ont une latence de quelques milli-secondes voire quelques dizaines de

milli-secondes dans le cas sans fil. Un échange entre deux machines distantes *via* Internet a une latence entre une dizaine et quelques centaines de milli-secondes. On atteint des latences de l'ordre de la seconde lors de communications multi-sauts dans un réseau de capteurs [YD09].

Paramètres de TCP

TCP est un protocole de transport, assurant une connexion fiable entre deux hôtes sur un réseau [Pos81]. Les détails concernant son fonctionnement sont donnés en annexe A. Les mécanismes de TCP auxquels nous nous référons dans ce chapitre y sont expliqués et illustrés. Afin de fiabiliser les échanges, TCP gère un mécanisme d'accusés de réception. Comme nous l'avons mentionné en sous-section 2.2.2, cela impose à l'émetteur de conserver en mémoire les données dites *en vol*, c'est-à-dire non encore acquittées. Dans le contexte du Web des objets, la principale limite à la quantité de données en vol est liée à la mémoire disponible sur le serveur [1]. Un des objectifs de notre analyse est d'évaluer l'impact de la quantité de mémoire allouée à la gestion des données en vol sur les performances du système.

Un autre paramètre important dont nous souhaitons évaluer l'impact est la taille des segments. Elle est bornée par la MSS [2], taille maximale des segments TCP négociée lors de l'ouverture d'une nouvelle connexion. La gestion de segments de grande taille est demandeuse de mémoire vive, mais permet de maximiser le taux de données utiles dans chaque segment.

Comportement type d'un échange HTTP

Le protocole HTTP suit un modèle requête-réponse. Il a été conçu pour prendre place au-dessus de TCP/IP. Originalement, chaque requête nécessitait son propre canal TCP. Elle était encadrée d'une ouverture et d'une fermeture de connexion. Ces deux opérations nécessitent chacune une négociation constituée d'au moins trois segments TCP, particulièrement coûteuse dans le cas de liaisons à latence élevée. Afin d'offrir une meilleure interaction avec TCP, la version 1.1 de HTTP pousse à l'utilisation de connexions persistantes, sur lesquelles plusieurs requêtes peuvent se succéder.

Lors d'une session HTTP, le comportement au niveau TCP est très simple. Le client envoie une requête, puis attend une réponse du serveur. À chaque instant, un seul des deux hôtes envoie des données applicatives, pendant que l'autre ne fait qu'émettre des accusés de réception [3]. C'est en partant de ce constat que nous proposons un modèle de trafic se focalisant sur l'envoi de données depuis un serveur vers un client ne faisant qu'en accuser la réception.

4.1.2 Modèle de trafic

L'objectif du modèle que nous proposons est d'estimer précisément les performances obtenues lors du service de données applicatives en fonction des paramètres du réseau (débit, latence) et de TCP (taille des segments, quantité maximale de données en vol). Plutôt que de nous intéresser aux facteurs stochastiques de l'environnement tels que les pertes de paquets (modélisées dans [LM97, AAB05]) ou les congestions réseau (étudiées dans [MSM97, PFTK00]), nous nous focalisons sur la taille précise des paquets et sur les allers-retours nécessaires à la

1. Le serveur est également limité par la fenêtre annoncée (TCP *advertised window*) par le client, en pratique supérieure (initialisée à 8 ko dans le cas de Windows) à la quantité de mémoire disponible sur le serveur.

2. MSS : *Maximum Segment Size*

3. Si le *pipelining* HTTP est utilisée, les envois peuvent être bidirectionnels. Cependant, les navigateurs Web actuels n'utilisent pas cette fonctionnalité, préférant répartir les requêtes simultanées sur plusieurs connexions.

réception de leurs accusés. Contrairement aux modèles existants qui estiment le débit obtenu lors de l'émission de données de taille infinie, notre modèle mesure le temps requis par un hôte pour envoyer des données de taille bornée, tel qu'une réponse HTTP.

Définition du modèle

Soient D et L respectivement le débit et la latence du chemin reliant le client au serveur. La taille maximale des segments TCP est notée m alors que le nombre maximal de segments en vol imposé par le serveur est notée n. Dans un premier temps, nous considérons que tout segment est acquitté dès qu'il a été reçu. En pratique, la politique des accusés différés (dont on détaille l'impact dans les paragraphes qui suivent) change légèrement ce point.

La durée entre le début de l'émission d'un segment de taille m et la fin de la réception de son accusé est notée $t_{per}(m)$. Elle est calculée comme la durée d'émission du segment, de son en-tête TCP/IP (de 40 octets) et de l'accusé de réception (également de 40 octets) à laquelle on ajoute le temps d'aller-retour des données sur le réseau :

$$t_{per}(m) = \frac{m + 40 + 40}{D} + 2 \times L$$

Pendant la période $t_{per}(m)$, la quantité de données applicatives envoyées avec un maximum de n segments en vol de taille m (notée $d_{per}(m,n)$) est bornée par $n \times m$:

$$d_{per}(m,n) = \min\left(n \times m, t_{per}(m) \times D \times \frac{m}{m+40}\right)$$

L'envoi d'un contenu applicatif de taille d se fait en deux phases. La première est faite de multiples périodes de durée $t_{per}(m)$, où chaque émission d'une quantité $d_{per}(m,n)$ de données commence dès la réception du premier accusé de la période précédente. Le nombre de périodes de la première phase est noté $n_{per}(d,m,n)$:

$$n_{per}(d,m,n) = \left\lceil \frac{d}{d_{per}(m,n)} \right\rceil - 1$$

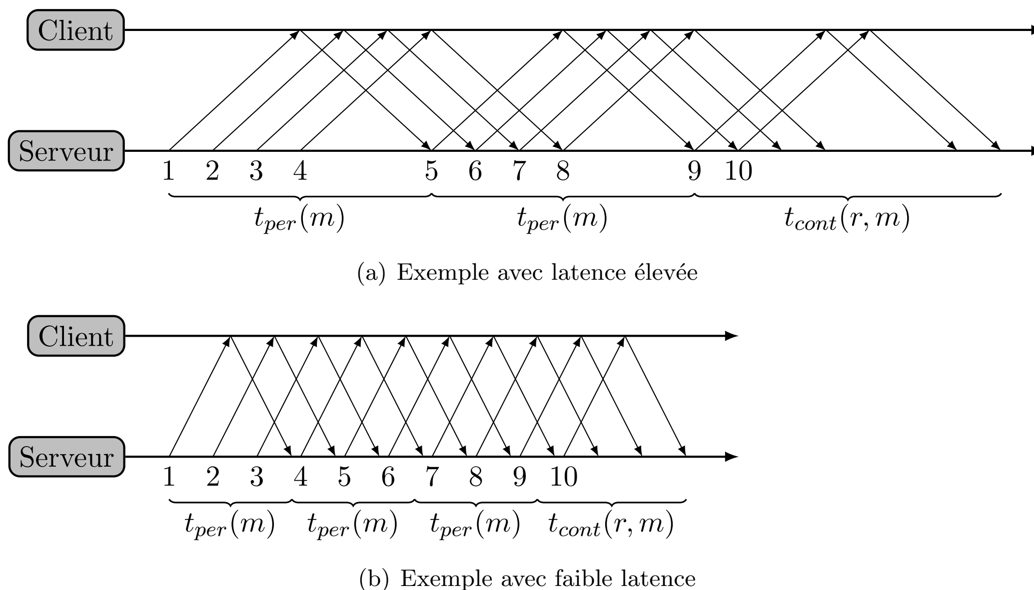
La seconde phase consiste à envoyer les données restantes (de taille $r \leq d_{per}(m,n)$) et à recevoir l'intégralité de leurs accusés. La durée de cette phase est notée $t_{cont}(r,m)$. Elle englobe la durée d'émission de $\lceil \frac{r}{sm} \rceil$ segments et de leurs en-têtes TCP/IP, le temps d'aller-retour et la durée d'émission du dernier accusé :

$$t_{cont}(r,m) = \frac{\lceil \frac{r}{m} \rceil \times 40 + r + 40}{D} + 2 \times L$$

La figure 4.1 illustre $t_{per}(m)$ et $t_{cont}(r,m)$ dans le cas d'un serveur envoyant des données à un client. Les données envoyées nécessitent 10 segments, alors que le serveur est limité à 4 segments en vol. Dans la figure 4.1(a), la fin de chaque période $t_{per}(m)$ rythme l'envoi de 4 nouveaux segments. Dans la figure 4.1(b), où la latence est moins importante, l'envoi de données se fait en continu, si bien que la liaison est utilisée au maximum de sa capacité.

On note $tt(d,m,n)$ le temps total nécessaire à l'envoi d'un contenu de d octets, avec un maximum de n segments en vol de taille m. On a alors :

$$tt(d,m,n) = n_{per}(d,m,n) \times t_{per}(m) + t_{cont}(d - d_{per}(m,n) \times n_{per}(d,m,n), m, n)$$



(a) Exemple avec latence élevée

(b) Exemple avec faible latence

FIGURE 4.1 – Illustration du modèle de trafic (limitation à 4 segments en vol).

Le cas des accusés différés

Dans TCP, lorsqu'un hôte reçoit des données sans en émettre (comme c'est majoritairement le cas lors d'une session HTTP), les accusés de réception qu'il envoie sont des segments vides, uniquement composés des en-têtes TCP et IP, d'une taille totale de 40 octets. Afin d'alléger le trafic causé par ces segments vides, la politique (facultative) des accusés différés a été introduite en 1989 [Bra89]. Un hôte TCP qui implémente les accusés différés n'accuse un segment que (i) 200 ms après sa réception ou (ii) dès qu'il reçoit un second segment. En pratique, la pile IP des systèmes d'exploitation tels que Windows, MacOS ou Linux implémentent cette politique [4].

Notre modèle ne prend pas en compte cette politique car son implémentation n'est pas systématique et son impact est le plus souvent négligeable. Cependant, dans le cas particulier où le serveur est limité à un unique segment en vol, les accusés différés du client pénalisent grandement les performances d'émission du serveur. Chaque segment n'est alors acquitté par le client que 200 ms après sa réception. Le serveur, presque constamment en attente d'accusés, est ainsi limité à 5 segments par seconde. Il est aisé de considérer ce cas particulier dans notre modèle, en augmentant artificiellement la latence de 100 ms, et ce sans perte de généricité.

4.1.3 Identification des facteurs impactants

Afin de guider la conception des stratégies internes de la pile de communication embarquée, nous identifions ici les facteurs ayant un impact significatif sur les performances de TCP.

Impact des paramètres du réseau

Les performances obtenues lors de l'émission de données sont en premier lieu conditionnées par les caractéristiques physiques du réseau et des interfaces de communication, imposant des

4. Linux désactive cette politique durant les premiers échanges d'une connexion TCP (la politique n'est activée que lorsque la quantité de données reçue a dépassé la moitié de la fenêtre TCP annoncée).

limites de latence et de débit. En calculant le produit de la latence par le débit d'un canal de communication, on connait la quantité de données qu'un émetteur peut envoyer avant le début de la réception des données par la machine distante. Les réseaux ayant un produit latence-débit élevé sont difficiles à gérer, entre autre car ils imposent l'usage de tampons de grande taille [LM97]. Le calcul de ce produit permet de connaître n_{max}, le nombre maximal de segments en vol d'une machine émettant des données. On calcule n_{max} en cumulant la durée d'émission d'un premier segment et le temps nécessaire à la réception de son accusé :

$$nev_{max} = \left\lceil \frac{2 \times D \times L + MSS + 40 + 40}{MSS + 40} \right\rceil$$

La table 4.1 instancie n_{max} pour différentes liaisons adaptées au Web des objets. La latence est fixée à 5, 25 ou 125 ms et la MSS est de 512 octets. La valeur de n_{max} est au minimum de 2. Selon les paramètres du réseau, il est parfois possible d'avoir plus de 10 segments de 512 octets en vol, ce qui représente une part importante de la mémoire des équipements que l'on vise.

		Nombre maximal de segments en vol		
Type de liaison	Débit	Latence : 5 ms	Latence : 25 ms	Latence : 125 ms
Liaison série	115,2 kbps	2	3	8
802.15.4	250 kbps	2	4	16
Bluetooth	1 mbps	4	13	58

Table 4.1 – Nombre maximal de segments en vol en fonction de la latence et du débit de la liaison (taille des segments : 512 octets)

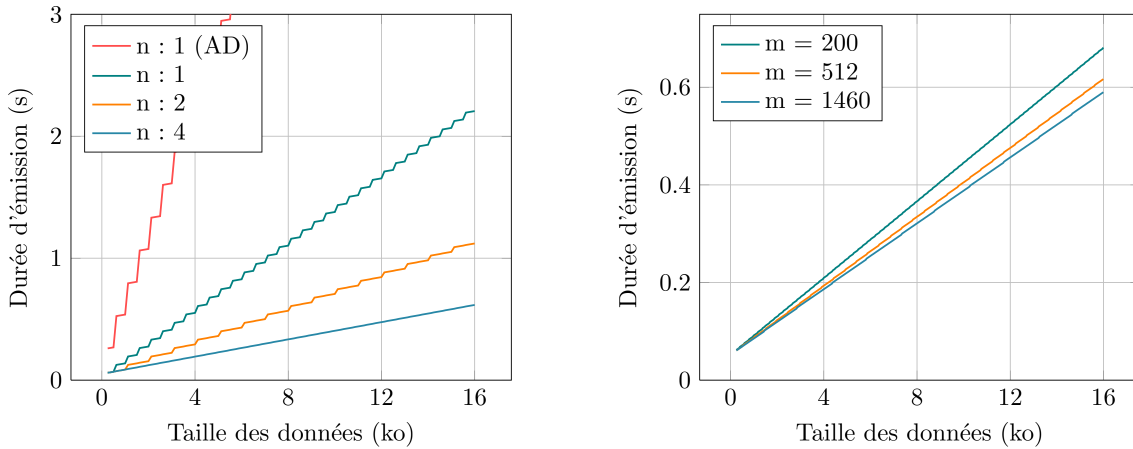
Impact des paramètres TCP

La figure 4.2(a) montre l'impact du nombre n de segments en vol sur les performances, pour des segments de 512 octets. L'accélération observée lorsque n croît est significative. Ainsi, un contenu de 16 ko est envoyé 3,6 fois plus vite avec quatre segments en vol qu'avec un seul. Dans une situation où la politique des accusés différés est utilisée, ce facteur d'accélération est de 14. Comme nous l'avons mentionné dans l'état de l'art (en section 2.2), de nombreuses piles IP embarquées sont constamment limitées à un unique segment en vol, ce qui permet une implémentation plus simple et légère. Nous montrons ici que cette limitation a un lourd impact sur les performances du système.

La figure 4.2(b) évalue l'impact de la taille des segments émis, sans contrainte sur le nombre maximal de segments en vol. La réduction de la taille des segments gérés (de 1460 à 200 octets) conduit à un allongement de la durée d'émission. Ici, cette augmentation est de 15,5 % pour des données de 16 ko. Cette réduction des performances est subie par de nombreuses piles IP embarquées, qui négocient une MSS minimale afin de réduire l'espace mémoire occupé par les segments en entrée et en sortie.

4.1.4 La notion de charge mémoire

Nous avons constaté dans l'état de l'art (voir section 2.2, page 22) que les implémentations de piles IP embarquées utilisaient diverses stratégies afin de limiter leur consommation de ressources. Par exemple, uIP [Dun03], TinyTCP [Coo02], webAce [Whi99] ou iPic [Shr02] se limitent à un seul segment en vol. Les solutions intégrées telles que MiniWeb [Dun05],



(a) Impact du nombre de segments en vol (n)
Segments de 512 octets. AD : accusés différés

(b) Impact de la taille des segments émis (m)
Sans limite sur le nombre de segments en vol

FIGURE 4.2 – Durée d'émission en fonction des paramètres TCP. $D = 250$ kbps, $L = 25$ ms.

webACE ou iPic limitent quant à elles la taille des segments gérés. Nous nous intéressons ici au compromis qu'il est possible de réaliser entre performances et consommation de mémoire sur le serveur. Dans tous les exemples suivants, on instancie les formules présentées dans le cas de l'envoi d'un contenu de 16 ko avec une taille de segments $m = 512$ octets, un débit de 250 kbps et une latence de 25 ms.

On considère tout d'abord une pile IP générique classique, conservant en mémoire tous les segments en vol. Le temps nécessaire à l'envoi d'un contenu de d octets est :

$$T_{classique}(d) = tt(d, m, n)$$

Dans le cas d'une limitation à 4 segments en vol ($n = 4$), on a :

$$T_{classique}(16\ ko) = 0,62\ s$$

On s'intéresse maintenant à la pile uIP. Afin de réduire sa consommation de mémoire, uIP ne gère qu'un unique segment en vol, non conservé en mémoire après émission. C'est l'application génératrice des données qui doit reconstruire le segment à envoyer en cas de retransmission. L'application est donc bloquée à l'aide d'un *protosocket* tant que l'accusé du segment n'a pas été reçu. Dans le cas général, c'est alors au niveau de l'application que les données émises sont conservées dans un tampon tant qu'elles n'ont pas été reçues et acquittées. Le comportement de uIP est équivalent à la stratégie classique dans le cas où $n = 1$:

$$T_{uIP}(d) = tt(d, m, 1)$$
$$T_{uIP}(16\ ko) = 2,21\ s$$

Le serveur MiniWeb est uniquement capable d'envoyer des fichiers statiques. Les segments à émettre sont tous construits hors-ligne et stockés dans une mémoire non-volatile, leur taille est donc fixée à 200 octets. Cependant, MiniWeb n'a aucune limite sur le nombre de segments en vol, car les segments envoyés n'ont pas besoin d'être stockés en mémoire vive pour d'éventuelles

retransmissions. La durée d'émission d'un contenu de taille d avec MiniWeb est :

$$\begin{aligned} T_{MiniWeb}(d) &= tt(d, 200, \infty) \\ T_{MiniWeb}(16\ ko) &= 0,68\ s \end{aligned}$$

Ces trois exemples imposent des limitations fonctionnelles différentes (pas de limite dans le premier cas, blocage entre chaque segment dans le deuxième, limitation aux données statiques dans le troisième). Ils obtiennent des performances variées et émettent les données en consommant des quantités différentes de mémoire. Nous proposons de synthétiser la performance et la consommation mémoire d'une stratégie à l'aide d'une nouvelle métrique, la *charge mémoire*.

La charge mémoire est l'intégration de la quantité de mémoire utilisée pendant l'émission des données. Elle s'exprime en octet-secondes ($o \cdot s$). De la même manière que les watt-heures intègrent une puissance (watt), les octet-secondes intègrent une consommation instantanée de mémoire (octets). En réduisant la charge mémoire associée à l'émission d'un contenu applicatif, une pile IP peut être embarquée dans des matériels plus contraints et passe mieux à l'échelle.

Dans le cas de la pile IP classique, tous les segments en vol sont conservés en mémoire pendant la durée d'émission, on obtient ainsi une charge mémoire de :

$$\begin{aligned} C_{classique}(d) &= m \times n \times tt(d, m, n) \\ C_{classique}(16\ ko) &= 1263\ o \cdot s \end{aligned}$$

Dans le cas de uIP, l'application conserve en permanence le segment en vol dans un tampon :

$$\begin{aligned} C_{uIP}(d) &= m \times tt(d, m, 1) \\ C_{uIP}(16\ ko) &= 1130\ o \cdot s \end{aligned}$$

Dans le cas de MiniWeb, aucune mémoire vive n'est consommée pour les segments en vol :

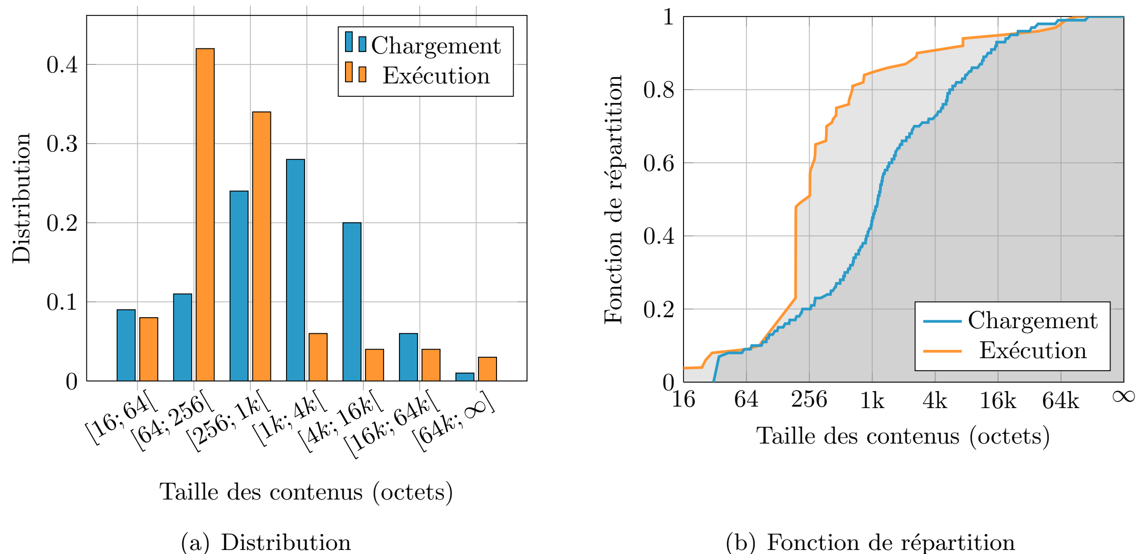
$$C_{MiniWeb}(d) = C_{MiniWeb}(16\ ko) = 0\ o \cdot s$$

Ces exemples illustrent que le temps de transmission n'est pas un critère suffisant pour évaluer une stratégie d'envoi de données sur un canal TCP. À l'aide de la charge mémoire, on évalue également le niveau de disponibilité d'un serveur en fonction des données qu'il traite. Par exemple, en se limitant à l'émission de fichiers statiques, MiniWeb atteint une charge mémoire nulle. La comparaison de la stratégie classique à celle de uIP est également riche d'enseignements. En effet, le gain de performances permis par la stratégie classique ne compense pas sa plus forte consommation instantanée de mémoire, si bien qu'elle obtient une charge mémoire légèrement supérieure à celle de uIP.

Cette notion de charge mémoire constitue un fil rouge dans la suite de ce mémoire ; elle motive la conduite de certains travaux présentés et permet l'évaluation des résultats en découlant. En imposant des contraintes fonctionnelles aux applications, les stratégies détaillées ici conduisent à des performances et des consommations mémoire variées. C'est en partant de ce constat que nous nous intéressons à la nature et aux besoins des applications Web.

4.2 Taxinomie des ressources Web

Dans l'optique d'adapter le comportement de la pile de communication aux propriétés des applications, nous nous intéressons ici à la classification des ressources Web. Pour satisfaire les



(a) Distribution

(b) Fonction de répartition

FIGURE 4.3 – Caractérisation des tailles de contenus d'applications AJAX, durant la phase de chargement et la phase d'exécution

exigences définies précédemment, ces applications sont développées en suivant le paradigme AJAX (voir sous-section 3.2.2, page 43). Elles sont construites de manière modulaire, séparent le fond de la forme et factorisent au maximum les échanges réseau et les traitements applicatifs. La taxinomie des ressources Web que nous proposons suit trois critères : (A) la taille des données, (B) la persistance des données et (C) le schéma d'interaction.

4.2.1 Critère A – Taille des données

Nous nous intéressons en premier lieu à la taille des données associées aux ressources Web. Rappelons que les applications AJAX distinguent une phase de chargement initiale et une phase d'exécution. Nous avons caractérisé le trafic produit par chacune de ces deux phases en enregistrant et analysant la trace d'une heure de navigation sur diverses applications en ligne (gestionnaire d'e-mails, recherches, réseaux sociaux, ...). Pour déterminer l'appartenance d'une requête à une phase ou une autre, nous avons outillé le navigateur Web (Mozilla Firefox 3.6 exécuté sur une station de travail classique) et ajouté des informations de provenance aux en-têtes HTTP. Toute requête initiée par un script est classée comme appartenant à la phase d'exécution de l'application. La figure 4.3(a) présente la distribution des tailles des contenus envoyés par les serveurs pendant chacune des deux phases, comptabilisant plus de 400 requêtes. La figure 4.3(b), plus fine, donne la fonction de répartition des tailles observées.

On observe que pendant la phase de chargement (durant laquelle les données échangées sont principalement statiques) la taille des contenus est significativement plus importante que pendant la phase d'exécution (principalement constituée de données générées dynamiquement). Par exemple, 78 % des données échangées durant l'exécution sont de moins de 512 octets contre seulement 28 % durant le chargement.

La taille des données constitue ainsi le premier critère de notre taxinomie. Elle a en pratique un impact important sur la manière de gérer l'émission des données. Par exemple, on constate que les serveurs Web embarqués les plus petits sont limités au service de contenus ne dépassant

pas un segment de données. Les contenus de taille plus importante nécessitent de maintenir des informations sur le numéro de séquence actuel et rendent plus complexe la gestion des retransmissions. On distingue ainsi deux ensembles :

A1 – Contenus courts contenus plus petits que la taille du tampon d'émission TCP ;

A2 – Contenus longs contenus plus grands que la taille du tampon d'émission TCP.

Dépendamment de la taille du tampon d'émission utilisée, la répartition des contenus dans la catégorie A1 et A2 est donc variable. Dans le cas général, il est impossible de déterminer par analyse statique avec certitude la taille des données générées lors de l'exécution. L'identification de l'appartenance à la catégorie A1 ou A2 se fait donc en-ligne, par le moteur d'exécution.

La connaissance plus précise de la taille des ressources pourrait également être bénéfique pour les performances du système. Ainsi, si on s'intéresse à l'ordonnancement des requêtes, on sait que les algorithmes d'ordonnancement SRPT ou FSP (précédemment présentés en sous-section 2.3.3, page 33) permettent d'atteindre de bonnes performances en se fondant sur la connaissance *a priori* des tailles des réponses aux requêtes.

4.2.2 Critère B – Persistance des données

Le second critère de notre taxinomie concerne la nature même des données constituant les ressources Web. Classiquement, on distingue simplement les contenus statiques de ceux générés dynamiquement par le serveur en réponse à une requête. Plus précisément, les contenus statiques peuvent être connus hors-lignes lors du déploiement du serveur ou générés lors de l'exécution d'une application. Les contenus dynamiques sont construits quant à eux lors de l'exécution d'un code, partie d'une application Web. Nous nous focalisons sur les propriétés de ce code : production d'effets de bord, déterminisme et sémantique des données générées. Nous affinons la distinction triviale entre contenus statiques et dynamiques avec notre second critère – la persistance des données – qui distingue cinq catégories :

B1 – Données statiques fixées hors-ligne La phase de chargement des applications AJAX consiste principalement à servir des fichiers de nature variée (*e.g.* structure (X)HTML, mise en forme (CSS), scripts (JS), données multimédia, *etc.*). Ces contenus sont connus hors-ligne et embarqués avec le serveur ;

B2 – Données statiques fixées à l'exécution Il arrive qu'un capteur enregistre en mémoire persistante un ensemble d'échantillons, ou qu'un serveur maintienne un fichier de *log* et donne accès à ces données *via* une ressource Web. Ces données sont statiques mais fixées lors de l'exécution ;

B3 – Données générées persistantes Certaines ressources sont associées à un code en charge de générer le contenu d'une réponse HTTP. Les tâches associées sont diverses, par exemple une mise à jour de données utilisateur, le pilotage d'un équipement ou un autre traitement applicatif. Lorsque ce code a un effet de bord ou dépend de l'état matériel ou logiciel, on dit qu'il est persistant. En d'autres termes, il n'est pas déterministe et/ou produit des effets de bord ;

B4 – Données générées volatiles Dans le Web des objets, il arrive typiquement qu'un capteur souhaite retourner un échantillon mesuré ou la valeur d'une variable applicative partagée. Cet échantillon est périssable ; il doit être capturé et envoyé au client aussi tard que possible avant son émission au client. Les données de ce type sont classées comme étant volatiles. Sa génération dépend de l'état du matériel ou du logiciel ; elle est donc non déterministe.

B5 – Données générées idempotentes Certaines ressources Web sont en charge de chiffrer ou de signer des données, par exemple dans le cas d'une carte à puce gérant des informations personnelles critiques. Plus généralement, lorsque la génération des données est réalisée par une fonction déterministe et sans effet de bord, on dit qu'elle est idempotente. Il s'agit de toute application au sens mathématique du terme.

À partir de la connaissance du critère B d'une ressource Web, on espère pouvoir en améliorer la prise en charge au sein du noyau, en particulier au niveau de la gestion de TCP. Les stratégies permettant d'optimiser cette prise en charge en termes de performances et de charge mémoire sont détaillées dans la prochaine section.

En termes d'interface applicative, on considère que l'application fournit la classification de la persistance des données pour chacune des ressources qui la constituent. Par exemple, le fait que des données soient volatiles n'est pas détectable, car cette propriété est par définition subjective (notion de données périssables). Il est cependant envisageable de détecter automatiquement l'idempotence d'une fonction, avec toutefois des faux négatifs. Lors de la prise en charge de la persistance dans le noyau, un faux positif n'est pas problématique, car les données persistantes constituent un cas général. En effet, la catégorie B3 est la moins contraignante pour le développeur d'applications, mais c'est elle qui donne le moins de liberté à la pile de communication pour l'émission des données générées. Lors de l'exécution, toute ressource peut être considérée comme B3 sans remettre en cause le bon fonctionnement du système et de l'application.

4.2.3 Critère C – Schéma d'interaction

Le troisième critère de notre taxinomie décrit les différents schémas d'interaction possibles. Depuis que les technologies du Web servent à véhiculer des applications plus que de simples informations, les schémas d'interaction entre clients et serveurs se sont diversifiés. Au delà du classique modèle requête-réponse prôné par les architecture REST [Fie00] (décrites en sous-section 3.1.1, page 37), on observe des échanges de type Comet avec lesquels le serveur pousse des données vers les clients [Rus06] (technique du *polling* long ou de l'émission en flux présentées en sous-section 2.3.2, page 31). Le critère C distingue trois catégories :

C1 – Interrogation Il s'agit du schéma originel du Web dans lequel un client interroge un serveur pour accéder à une ressource. Il est en concordance avec les architectures de type REST et se fonde sur les commandes HTTP classiques (*get*, *post*, *put*, *delete*...). Le serveur répond au client dès que possible, après avoir interprété la requête, effectué les traitements qu'elle requiert éventuellement, puis produit un contenu HTTP. Ce schéma est utilisé pour la plupart des ressources des applications Web ;

C2 – Alerte Dans certaines applications, on souhaite qu'un client puisse être alerté de l'occurrence d'un évènement. Le client émet alors une demande d'enregistrement auprès du serveur puis entre en attente de notification. Lorsque l'évènement ciblé se produit, le serveur lève une alerte auprès du ou des clients enregistrés en attente. Cela permet par exemple de notifier des clients qu'un traitement applicatif vient de se terminer, qu'une mesure environnementale effectuée par un capteur a dépassé un seuil. L'alerte est supportée à l'aide de Comet avec la technique du *polling* long ;

C3 – Suivi Il est parfois nécessaire de permettre aux clients de suivre l'évolution d'une information. Comme pour l'alerte, le client doit s'enregistrer auprès du serveur. Ensuite, lorsque l'information concernée est mise à jour, sa nouvelle valeur est envoyée à tous

les clients en attente. Un client peut se désabonner lorsqu'il le souhaite, afin de ne plus recevoir les mises à jour. Ce schéma d'interaction permet par exemple le contrôle en temps réel d'un état logiciel ou environnemental. Il est supporté par Comet avec la technique de l'émission en flux.

Le schéma d'interaction est renseigné explicitement pour chaque ressource qui compose une application. Ainsi, une application offrant des services d'alerte ou de suivi peut être composée de ressources aux modes de déclenchements variées, dont les contenus sont agrégés *via* les requêtes asynchrones permises par la technique AJAX.

Cette taxinomie des applications Web vient compléter notre analyse de trafic. Il est alors possible de s'intéresser à la définition d'un support efficace de tout type d'application au regard de notre modèle de trafic et de notre notion de charge mémoire.

4.3 Stratégies adaptées aux propriétés applicatives

Nous proposons ici un ensemble de stratégies permettant d'adapter le comportement de la pile de communication aux propriétés des applications supportées (selon les critères de la taxinomie). On espère ainsi dépasser sensiblement les résultats des piles IP classiques, gérant les données applicatives sans connaissance de leurs propriétés, donc de manière générique. Les bénéfices attendus en performances et en charge mémoire sont formalisés à l'aide de notre modèle de trafic.

4.3.1 Stratégies d'émission des données

Les stratégies présentées ci-après s'adaptent aux critères A (taille) et B (persistance) de notre taxinomie (le support du critère C est discuté plus loin). Nous avons ainsi identifié cinq stratégies (notées S1 à S5) différentes, décrites dans les paragraphes suivants. Leur affectation aux données selon les critères A et B est décrite par la table 4.2.

	B1/B2 – statique	**B3** – persistant	**B4** – volatil	**B5** – idempotent
A1 – court	S1	S2	S3	S2,S3
A2 – long	S1	S4	S5	S4,S5

TABLE 4.2 – Affectation des stratégies S1 à S5 aux catégories A et B de la taxinomie

Nous évaluons les bénéfices de chaque stratégie en termes de performances et de charge mémoire, selon notre modèle de trafic. On se compare à la stratégie classique, conservant tous les segments en vol en mémoire vive. On positionne la taille des segments à $m = 512$ octets et le nombre maximal de segments en vol à $n = 1$ (comme le font les piles IP embarquées uIP, TinyTCP, webAce ou iPic). On instancie le modèle avec un débit de 250 kpbs et une latence de 25 ms (caractéristiques typiques d'une liaison sans-fil de type IEEE 802.15.4). La taille des contenus est fixée soit à 256 octets (la médiane des tailles mesurées lors de la phase d'exécution, *c.f.* sous-section 4.2.1) soit à 16 ko (taille que l'on peut considérer comme importante car dépassée par moins de 10 % des contenus de la phase de chargement). Pour rappel, les performances obtenues avec la stratégie classique sont alors :

$$\begin{aligned} T_{classique}(d) &= tt(d, m, n) \\ T_{classique}(256\ o) &= 0,06\ s \\ T_{classique}(16\ ko) &= 2,21\ s \end{aligned}$$

Toujours avec la stratégie classique, la charge mémoire est :

$$\begin{aligned} C_{classique}(d) &= m \times n \times tt(d, m, n) \\ C_{classique}(256\ o) &= 15,6\ o \cdot s \\ C_{classique}(16\ ko) &= 1130\ o \cdot s \end{aligned}$$

Stratégie S1 : données pré-traitées

Cette stratégie cible les contenus statiques, fixés hors-ligne ou lors de l'exécution, sans contrainte de taille (catégories B1 et B2). Ces contenus peuvent être stockés dans une mémoire non-volatile, typiquement moins coûteuse et disponible en plus grandes quantités que la mémoire vive. Les contenus connus hors-ligne (B1) peuvent être localisés en ROM, alors que ceux qui sont créés lors de l'exécution (B2) doivent être placés dans une mémoire inscriptible, par exemple une EEPROM. Le fait que ces données soient statiquement connues permet de calculer à l'avance leurs sommes de contrôle TCP ; une des opérations les plus coûteuses en performances lors de l'émission de données [CJRS02].

La gestion des segments en vol peut être grandement améliorée par la connaissance statique des données, stockées dans une mémoire adressable. De la même manière que les serveurs embarqués limités à l'émission de données statiques (tel que MiniWeb), il est possible d'émettre un nombre illimité de segments sans tampon en mémoire vive. Lorsqu'une retransmission TCP est requise, il est possible de reconstruire le segment perdu à partir des données stockées en mémoire adressable (et non-volatile). La partie des données à ré-émettre est directement calculée à partir du numéro de séquence du segment perdu. Les performances obtenues avec S1 sont donc :

$$\begin{aligned} T_{S1}(d) &= tt(d, m, \infty) \\ T_{S1}(256\ o) &= 0,06\ s \\ T_{S1}(16\ ko) &= 0,62\ s \end{aligned}$$

La charge mémoire obtenue est quant à elle systématiquement nulle puisque les segments en vol ne sont pas stockés en mémoire vive :

$$C_{classique}(d) = 0\ o \cdot s$$

Stratégie S2 : tampon d'attente

Cette stratégie s'applique aux données générées persistantes d'une taille inférieure ou égale au tampon d'émission TCP (catégorie A1/B3). Le code applicatif en charge de générer la réponse HTTP interagit avec la pile de communication à l'aide d'une routine `output()` permettant l'émission de données. Cette routine stocke les données dans le tampon d'émission, calculant la somme de contrôle TCP à la volée. À la fin de la génération des données, la réponse HTTP est envoyée. Le segment est conservé en mémoire tant qu'il n'a pas été acquitté, afin de permettre d'éventuelles retransmissions. Dans le cas où le serveur n'a plus assez de mémoire pour stocker un segment, les autres requêtes présentes dans le système sont mises en attente. Elles ne s'exécuteront que lorsque la réception d'accusés aura permis de libérer une quantité suffisante de mémoire. Cette stratégie correspond à la stratégie classique dans le cas d'émission

de contenus court. On a donc :

$$T_{S2}(d) = T_{classique}(d), d \leq m$$
$$C_{S2}(d) = C_{classique}(d), d \leq m$$

Stratégie S3 : tampon éphémère

Dans le cas de l'émission de données générées volatiles de petite taille (A1/B4), il est possible de réduire la charge mémoire du serveur. Comme pour S2, le code applicatif génère la réponse HTTP en utilisant la routine système `output()`. En revanche, dès que la réponse HTTP est envoyée au client, les données sont libérées de la mémoire. Lorsqu'une retransmission est nécessaire, le code en charge de la génération des données est exécuté une nouvelle fois. Il produit alors de nouvelles données, pour produire une nouvelle réponse à envoyer au client.

La propriété de volatilité du code générateur assure que la fonction n'a pas d'effet de bord. Lors de sa ré-exécution, elle peut éventuellement produire des données différentes de celles obtenues lors de l'exécution initiale (l'exécution n'est pas déterministe). Dans le cas d'un capteur envoyant une mesure de température, la réponse retournée est ainsi rafraichie avant la ré-émission. Ce comportement n'est pas exactement en adéquation avec la sémantique originale de TCP. Cependant, cela ne cause aucun dysfonctionnement en pratique et permet d'améliorer la pertinence des données reçues par le client ainsi que l'usage de la mémoire du coté du serveur.

En termes de trafic, S3 est équivalente à S2 et à la stratégie classique. En revanche, elle obtient une charge mémoire nulle, augmentant significativement la disponibilité du serveur en cas de forte charge :

$$T_{S3}(d) = T_{classique}(d), d \leq m$$
$$C_{S3}(d) = 0 \; o \cdot s$$

Notons que la durée de génération des données, pendant laquelle un tampon est utilisé, est ici ignorée. Sur les matériels visés, cette durée est le plus souvent négligeable, car l'interface de communication est particulièrement lente en comparaison au processeur (*c.f.* sous-section 3.2.1, page 42).

Stratégie S4 : générateur persistant

Cette stratégie s'applique aux contenus générés persistants de grande taille (A2/B3). La routine `output()` fournie au code applicatif est cette fois bloquante. Lorsque l'application a rempli le tampon de données (de taille bornée par la MSS), la pile de communication reprend la main. Elle envoie alors le segment au client, réalise éventuellement d'autres traitements (gestion des autres connexions et des données entrantes). Si la mémoire disponible est suffisante, elle rend la main à l'application, qui reprend son exécution là où elle avait été interrompue. Si elle est insuffisante, l'exécution ne reprendra que lorsque suffisamment de mémoire aura été libérée, par exemple en conséquence de la réception d'accusés.

S4 se comporte comme une pile IP généraliste classique, qui n'a aucune connaissance des propriétés des applications qu'elle exécute. Les données persistantes sont en conséquence celles qui offrent le moins de libertés au serveur, qui n'a d'autre choix que de conserver en mémoire toutes les données tant qu'elles n'ont pas été acquittées. La S4 est équivalente à la stratégie

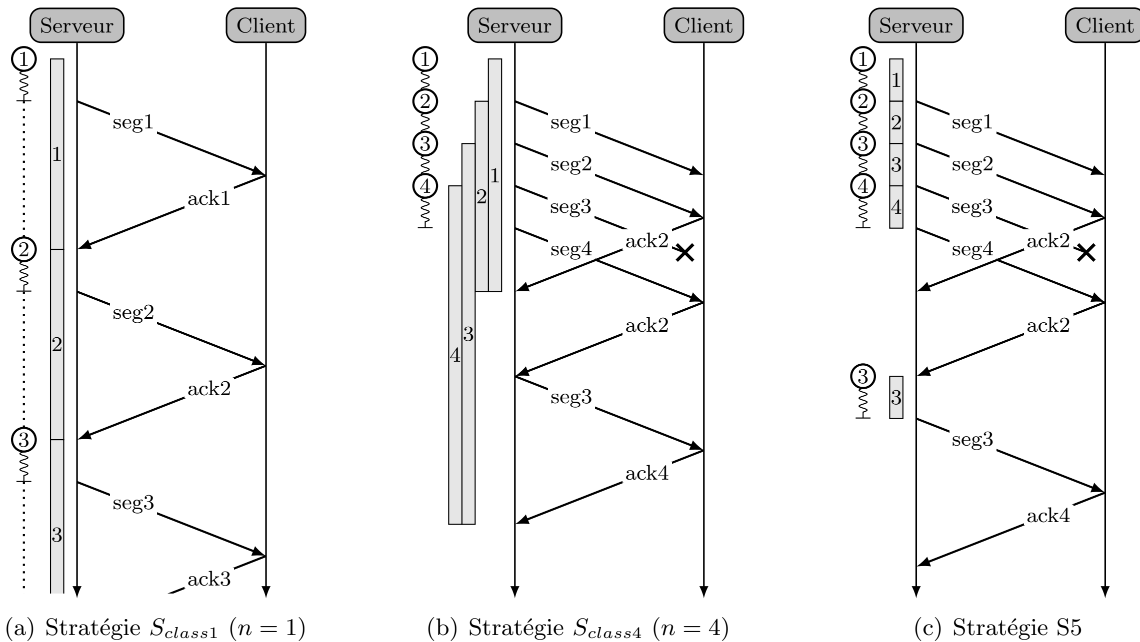


Figure 4.4 – État des fils d'exécution, de la mémoire occupée et du réseau durant l'envoi de 4 segments de données selon 3 stratégies différentes

classique, car les données persistantes représentent le cas général, dans lequel aucune propriété applicative intéressante n'a pu être extraite :

$$T_{S4}(d) = T_{classique}(d)$$
$$C_{S4}(d) = C_{classique}(d)$$

Il est intéressant d'étudier les performances et la charge mémoire de S4 en fonction du nombre n de segments en vol supportés. Une analyse de la fonction $tt()$ montre que les performances s'améliorent lorsque n augmente. En contrepartie, cette accélération s'accompagne d'une augmentation de la charge mémoire (ce cas a été rencontré lors de l'instanciation de la stratégie classique avec $n = 1$ et $n = 4$, en sous-section 4.1.4). Le positionnement de la valeur de n permet ainsi d'établir un compromis entre performances pures et charge mémoire. Ainsi, avec S4, si on souhaite maintenir un bon degré de disponibilité du serveur, il est pertinent de conserver de la mémoire à disposition en gérant un nombre raisonnable de segments en vol.

Stratégie S5 : générateur volatil

Cette stratégie s'applique aux contenus générés volatils de taille non bornée (A2/B4). La figure 4.4 détaille le comportement obtenu par la stratégie classique avec $n = 1$ (notée S_{class1}) et la stratégie classique avec $n = 4$ (notée S_{class4}) sur les contenus de grande taille. S_{class1} conduit à des performances moins bonnes que S_{class4}, mais a une consommation instantanée de mémoire moins importante (1 segment en mémoire contre 4).

L'objectif de S5 est d'allier les avantages de S_{class1} et S_{class4}. Étant donnée la volatilité des données, il n'est pas nécessaire de conserver en mémoire les données en vol. Comme la stratégie S3, S5 défausse les segments de la mémoire dès qu'ils ont été émis. Lorsqu'un segment est

perdu (cas du segment 3 dans la figure 4.4(c)), il est alors nécessaire de re-générer le segment. Pour permettre cela, on conserve pour chaque segment en vol une *continuation*, contenant toutes les données nécessaires à la ré-exécution du code applicatif ayant produit le segment. Dans le cas le plus simple, une continuation peut être simplement constituée du numéro de séquence des données à générer. Comme pour la stratégie S3, il est possible que le segment re-généré diffère du segment original, contenant des données plus récentes. La cohérence du contenu global est dans ce cas sous la responsabilité du développeur de l'application Web ; qui est lui même en charge d'identifier les générateurs de données volatiles lors de la conception de l'application Web.

Ainsi, la stratégie S5 n'impose aucune limite sur le nombre de segments en vol, permettant l'obtention de performances optimales (mêmes performances que S1, réservé aux contenus statiques), tout en permettant une charge mémoire nulle :

$$\begin{aligned} T_{S5}(d) &= T_{S1}(d) \\ C_{S5}(d) &= 0\ o \cdot s \end{aligned}$$

La prise en charge des *continuations* ainsi que la consommation mémoire qui en découle (ici négligée) sera l'objet de la seconde section du prochain chapitre.

Le cas des générateurs idempotents

Les contenus idempotents (catégorie B5) peuvent être traités comme des contenus soit persistants (stratégies S2/S4), soit volatils (stratégies S3/S5). Lorsqu'une retransmission est nécessaire, le moteur d'exécution peut décider de stocker les données en tampon tant qu'elles n'ont pas été acquittées ou de ne conserver qu'une continuation permettant la re-exécution du code. Du point de vue du client, ces deux comportements sont strictement équivalents, car la fonction est déterministe et sans effets de bord. Ainsi, contrairement au cas des contenus volatils avec la stratégie S5, il n'y a aucun problème de cohérence du contenu global ou de segments re-générés qui diffèrent potentiellement de l'original.

Le choix de la stratégie à adopter peut se faire selon plusieurs critères. On peut par exemple stocker les données en vol tant qu'assez de mémoire est disponible. Le segment peut être défaussé dès que le système a besoin de libérer de la mémoire, par exemple pour la génération de données persistantes. Le choix des données à défausser peut se faire en fonction du temps de calcul qui a été nécessaire à leur production. Par exemple, le résultat d'une signature numérique coûteuse en calcul et en énergie devra être conservé en mémoire de manière plus prioritaire que d'autres données dont la production est moins consommatrice de ressources.

Bénéfices

En guise de synthèse, la table 4.3 montre les résultats obtenus pour chacune de nos stratégies en comparaison à la stratégie classique ($S_{classique}$), qui n'exploite aucune meta-donnée applicative.

Seules les stratégies manipulant des données persistantes n'améliorent pas la stratégie classique. Dans le cas de contenus statiques, volatils ou idempotents, la charge mémoire peut être réduite à 0 et les performances accélérées d'un facteur de 3,6 (dans la configuration utilisée comme référence, sans la politique des accusés différés). Dans une telle situation, on est en mesure de gérer un grand nombre de segments en vol en disposant d'un espace mémoire très

	Durée d'émission			Charge mémoire		
Stratégie	modèle	exemple (s)	ratio	modèle	exemple ($o \cdot s$)	ratio
			Contenu court (exemple : 256 octets)			
$S_{classique}$	$t_{cont}(d,m)$	0,06	—	$n \times m \times tt(d,m,n)$	15,6	—
S_1	$t_{cont}(d,m)$	0,06	1	0	0	∞
S_2	$t_{cont}(d,m)$	0,06	1	$d \times t_{cont}(d,m)$	15,6	1
S_3	$t_{cont}(d,m)$	0,06	1	0	0	∞
			Contenu long (exemple : 16 ko)			
$S_{classique}$	$tt(d,m,n)$	2,21	—	$n \times m \times tt(d,m,n)$	1130	—
S_1	$tt(d,m,\infty)$	0,62	**3,6**	0	0	∞
S_4	$tt(d,m,n)$	2,21	1	$n \times m \times tt(d,m,n)$	1130	1
S_5	$tt(d,m,\infty)$	0,62	**3,6**	0	0	∞

Table 4.3 – Synthèse des performances et charges mémoire des différentes stratégies. Débit : 250 kbps, latence : 25 ms, accusés différés non actifs. Taille des segments : $m = 512$ octets. Nombre maximal de segments en mémoire pour les stratégies $S_{classique}$ et S_4 : $n = 1$.

limité. Le gain en performances est alors d'autant plus élevé que le produit latence-débit est important.

4.3.2 Gestion des schémas d'interaction

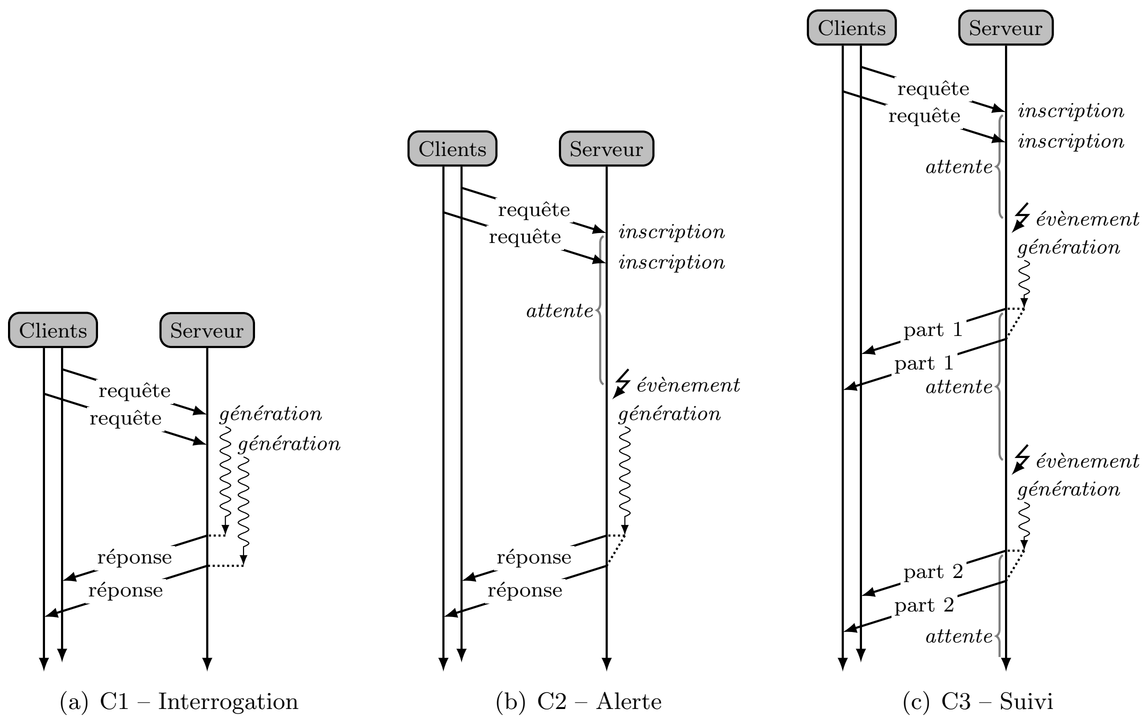
Nous nous intéressons ici à la prise en charge des ressources en fonction du schéma d'interaction qu'elles utilisent (critère C). Nous détaillons la manière dont les traitements applicatifs sont gérés par le noyau, selon que l'interaction soit de type interrogation, alerte ou suivi.

C1 – Interrogation

Lorsque le serveur reçoit une requête, il la décode et identifie la ressource ciblée. S'il s'agit d'un contenu statique (B1 ou B2), il accède au système de fichiers puis envoie une réponse HTTP. Lorsqu'il s'agit d'un contenu dynamique (B3, B4 ou B5), il génère les données à l'aide du code applicatif associé à la ressource concernée. En plus de produire des données, ce code peut effectuer tout type de traitements (gestion de l'application). Comme nous l'avons décrit précédemment, la génération des données se fait *via* une routine `output()` dont le comportement s'adapte automatiquement aux critères A et B de la taxinomie. La figure 4.5(a) décrit une suite d'interrogations. Elle illustre notamment le fait qu'il soit possible d'exécuter plusieurs applications simultanément.

C2 – Alerte

Dans le cas de l'alerte (supporté par Comet avec *polling* long), la réception de la requête HTTP engendre un enregistrement du client auprès du serveur. Éventuellement, plusieurs clients peuvent s'enregistrer comme étant à l'écoute d'un même évènement. Lorsque l'évènement est effectivement déclenché, le moteur d'exécution demande à l'application de produire la réponse HTTP qui constitue la notification.



(a) C1 – Interrogation (b) C2 – Alerte (c) C3 – Suivi

FIGURE 4.5 – Illustration des routines impliquées dans les différents schémas d'interaction

Nous proposons ici de partager un tampon d'émission entre tous les clients enregistrés auprès de la même ressource C2. Ce comportement est tout à fait atypique : dans un système classique, il est impossible de partager le même tampon entre plusieurs *sockets*. On évite ainsi de générer plusieurs fois les mêmes données ou de copier les données générées pour chaque client en attente. On réduit alors considérablement la charge mémoire du système. Dans le contexte de l'alerte, où le nombre de clients en attente est potentiellement élevé, cela permet d'espérer une meilleure capacité de passage à l'échelle. La figure 4.5(b) illustre un cas où deux clients s'enregistrent. La génération de données n'est faite qu'une seule fois puis les données sont envoyées aux deux clients.

C3 – Suivi

Le suivi d'une information (supporté par Comet avec émission en flux) commence également par l'enregistrement d'un client auprès du serveur. Chaque fois qu'un évènement se produit, l'application génère de nouvelles données, envoyées comme un fragment de réponse HTTP. C'est ainsi qu'une réponse de taille potentiellement infinie est générée par parties, avec des intervalles de temps arbitrairement grands. Malgré cela, l'application n'est pas bloquée. Son exécution se termine après la génération de chaque fragment, si bien qu'il n'est pas nécessaire de gérer un *thread* inactif, impliquant le stockage d'une pile d'exécution. On limite ainsi l'augmentation de l'usage mémoire en fonction du nombre de canaux actifs. La figure 4.5(c) illustre ce comportement. De la même manière que pour l'alerte, les données sont générées une unique fois avant d'être servies à tous les clients en attente, à partir d'un tampon partagé.

4.4 Conclusion

L'analyse de trafic que nous avons proposée a permis d'identifier les facteurs impactant sur la performance de l'émission de données dans un canal TCP. Il est important de supporter une grande quantité de données en vol et des segments d'une taille aussi élevée que possible. La notion de charge mémoire a été introduite afin de synthétiser l'usage de la mémoire vive du serveur pendant l'émission de données. Cette notion est bien adaptée aux exigence du Web des objets, où la compacité des matériels doit s'accompagner de performances élevées. Elle sera utilisée tout au long de ce mémoire pour motiver ou pour évaluer certaines propositions. Nous avons étendu notre analyse par une taxinomie des ressources constituant les applications Web, selon la taille des données, leur persistance et le schéma d'interaction entre client et serveur. Nous avons ensuite proposé un ensemble de stratégies afin d'adapter le comportement de la pile de communication aux propriétés des applications. Cette solution est permise par le fait de dédier le système d'exploitation aux besoins des applications supportées. Elle permet de dépasser les limitations classiquement imposées par les logiciels intégrés ou les systèmes d'exploitation généralistes embarqués, en termes de fonctionnalités applicatives et/ou de performances [DGV-Emsoft09].

Nos propositions soulèvent de nouveaux défis. Ainsi, le noyau devra gérer de multiples fils d'exécution simultanés pour supporter la génération de contenus dynamiques, permettant le blocage, la gestion des continuations et supportant la ré-invocation. Ces points sont habituellement très consommateurs de mémoire et leur support n'est le plus souvent pas envisagé dans des systèmes ne disposant que de quelques kilo-octets de mémoire. Ce noyau devra également être capable de passer à l'échelle, en manipulant un grand nombre de connexions et de requêtes. Les schémas d'interaction permettant l'alerte et le suivi d'information sont en effet potentiellement générateurs d'un grand nombre de connexion inactives simultanées. Le problème de l'ordonnancement des requêtes au sein de ce système devra également être abordé. Dans le prochain chapitre, nous nous intéressons à la conception même du macro-noyau dédié, dans l'optique d'apporter des réponses à chacune de ces questions.

Cinquième chapitre

Conception du macro-noyau

Après avoir cerné la nature et les besoins des applications du Web des objets, nous nous intéressons à la conception du macro-noyau dédié au support de ces dernières. Il s'agit de construire un système aussi efficace que possible avec pour objectif le support ouvert d'applications Web. Nous présentons tout d'abord l'architecture générale du noyau et évaluons un ensemble d'optimisations permises par sa construction intégrée. Nous détaillons ensuite la gestion de multiples fils d'exécution applicatifs en concordance avec les stratégies définies dans le chapitre précédent, c'est-à-dire ré-invocables et consommant peu de mémoire. Enfin, étudions l'ordonnancement des tâches au sein du noyau, dans l'optique de garantir à la fois performances, faible charge mémoire et équité du service fourni aux clients.

5.1 Un macro-noyau évènementiel

Une des forces attendues du système à macro-noyau réside dans l'intégration, au sein de la partie critique du système, de la majeure part du logiciel déployé : allant de la gestion du matériel au support applicatif de haut niveau. Nous décrivons ici l'architecture évènementielle de notre macro-noyau, tout d'abord d'un point de vue conceptuel, puis opérationnel. Nous nous attachons ensuite aux optimisations rendues possibles par l'intégration des couches hautes dans le noyau.

5.1.1 Description conceptuelle : l'omniprésence des évènements

Nous expliquons ici comment notre macro-noyau peut bénéficier d'une implémentation évènementielle à tous les niveaux du logiciel, du matériel aux applications en passant par la pile de communication.

Des pilotes de matériel naturellement évènementiels

Les microprocesseurs et leurs périphériques fonctionnent de manière évènementielle. La base logicielle les pilotant est constituée de fonctions de rappel, invoquées par des interruptions matérielles lorsqu'un évènement se produit. Ainsi, au plus bas niveau du logiciel, la notion d'évènement est naturellement présente. Classiquement, lorsqu'une interruption matérielle est levée, le logiciel est capable de différer des traitements en enregistrant des évènements logiciels, invoqués non pas par le matériel mais par le gestionnaire d'évènements du système

d'exploitation. C'est ainsi qu'on assure la remontée des évènements jusqu'à la couche applicative supérieure (par exemple la pile de communication), tout en conservant une granularité des évènements aussi fine que possible.

Support événementiel de la pile TCP/IP

Dans la plupart des systèmes d'exploitation, la pile IP constitue le point d'entrée aux applications pour la gestion des communications réseau. Elle est classiquement implémentée de manière évènementielle et fournit une interface où les données entrantes et sortantes sont présentées aux applications sous la forme de flux, suivant le paradigme des *sockets*. On retrouve cette notion de *socket* même dans les systèmes atypiques destinés aux équipements les plus contraints comme TinyOS ou Contiki. Dans notre macro-noyau, nous proposons d'étendre l'implémentation évènementielle de la pile de communication aux couches hautes du logiciel, incluant HTTP et les applications Web. L'objectif est ici de minimiser la consommation de ressources du noyau.

HTTP et le schéma requête-réponse

Le protocole HTTP est par construction évènementiel, car il suit un schéma requête-réponse. C'est ce qui explique en partie le fait que certains travaux se soient intéressés aux bénéfices des architectures évènementielles pour les serveurs applicatifs, comme nous l'avons discuté dans l'état de l'art, en sous-section 2.3.1. Nous proposons de gérer HTTP avec le moteur d'évènements du noyau, utilisé par les pilotes, IP et TCP. Il est également envisageable d'intégrer directement certains traitements relatifs à HTTP au sein même des routines en charge d'émettre ou de recevoir des segments TCP. Cette possibilité ouvre les portes à de nombreuses optimisations transversales, dont la description et l'évaluation sont l'objet de la suite de cette section.

Une interface applicative à base de fonctions de rappel

En concordance avec les stratégies de services décrites dans le chapitre précédent, nous proposons de gérer les applications à l'aide d'un ensemble de fonctions de rappel. Ce modèle applicatif est classique des conteneurs d'application Web, par exemple dédiés à la gestion de *Servlets* java. Ce fonctionnement est naturellement évènementiel, puisqu'il gère les applications comme un ensemble de routines associées à des évènements (par exemple la réception d'une nouvelle requête ou la possibilité de générer de nouvelles données). Les applications, lorsqu'elles n'ont aucun traitement à réaliser, n'utilisent aucune ressource du système car elles ne sont pas suspendues dans un *thread*, attendant la réception d'une requête.

Comet : les événements au plus haut niveau

Dans le cas d'applications Comet, c'est-à-dire de notification d'évènements applicatifs, on pousse l'implémentation évènementielle au plus haut niveau. Nous prenons ici l'exemple d'un capteur déclenchant une alerte lorsque la température ambiante dépasse un seuil. L'application utilise une interruption matérielle périodique afin de contrôler la température. Lorsqu'une alerte est nécessaire, le gestionnaire d'évènements en est informé. Il exécutera ensuite la routine applicative associée. On a alors un flux évènementiel direct entre les couches basses du noyau

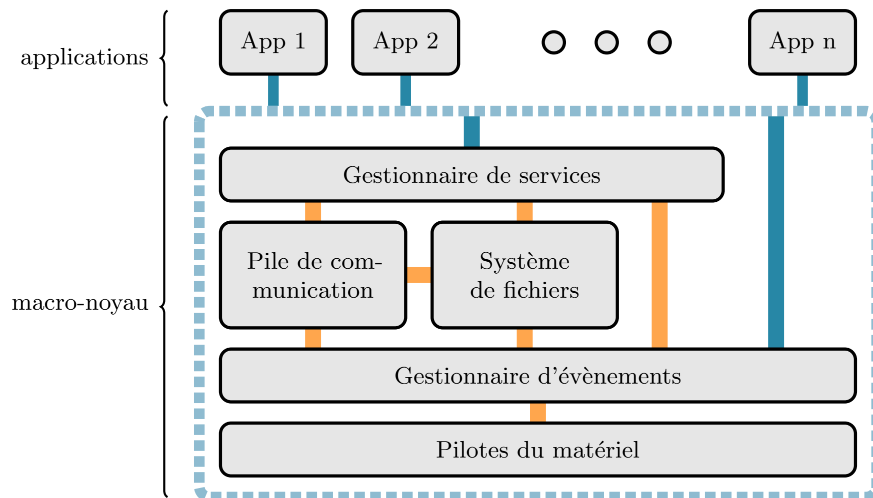


FIGURE 5.1 – Architecture du système d'exploitation à macro-noyau

et les ressources d'une application Web. On espère ainsi obtenir un système particulièrement peu consommateur de ressources et rapide à l'exécution.

5.1.2 Description opérationnelle : l'architecture du noyau

D'un point de vue opérationnel, notre macro-noyau se découpe en plusieurs entités logiques, décrites par la figure 5.1. Le gestionnaire d'évènements, première entité au-dessus des pilotes du matériel, constitue la colonne vertébrale du noyau. Il est le point d'entrée vers les autres modules du noyau. Les évènements qu'il manipule sont soit enregistrés directement par le matériel, soit par un des autres modules ou une application. La pile de communication gère les protocoles IP, TCP et HTTP. Le gestionnaire de services se charge de la construction des réponses aux requêtes des clients. Pour cela, il accède aux données statiques par le système de fichiers ou manipule les applications *via* les fonctions de rappel qu'elles fournissent. L'assemblage de ces modules constitue notre macro-noyau, au-dessus duquel viennent se greffer les applications Web.

Ce schéma d'architecture sera mentionné dans la suite de ce chapitre pour situer les différents travaux présentés. Il est complété par la figure 5.2 qui décrit le fonctionnement de notre macro-noyau sous la forme d'un diagramme de séquence lors du traitement de quelques requêtes. Afin d'illustrer l'interaction entre les différents modules du noyau, nous décrivons ci-après les traitements provoqués par les six évènements principaux de ce diagramme :

1 – Arrivée et service de `index.html` L'arrivée d'un paquet déclenche une interruption matérielle, qui notifie le gestionnaire d'évènements. Ce dernier passe alors la main à la pile de communication, qui fait l'interprétation des données reçues. La connexion TCP concernée (■) est mise à jour, la requête HTTP est décodée et notifiée au gestionnaire d'évènements. Ce dernier demande au gestionnaire de service de générer la réponse. Il accède au système de fichiers, qui réalise l'émission des données de la ressource `index.html` ;

2 – Arrivée de la requête ≪ `get alerte` ≫ Une nouvelle requête entrante cible une ressource dont le schéma d'interaction est l'alerte, correspondant à une notification

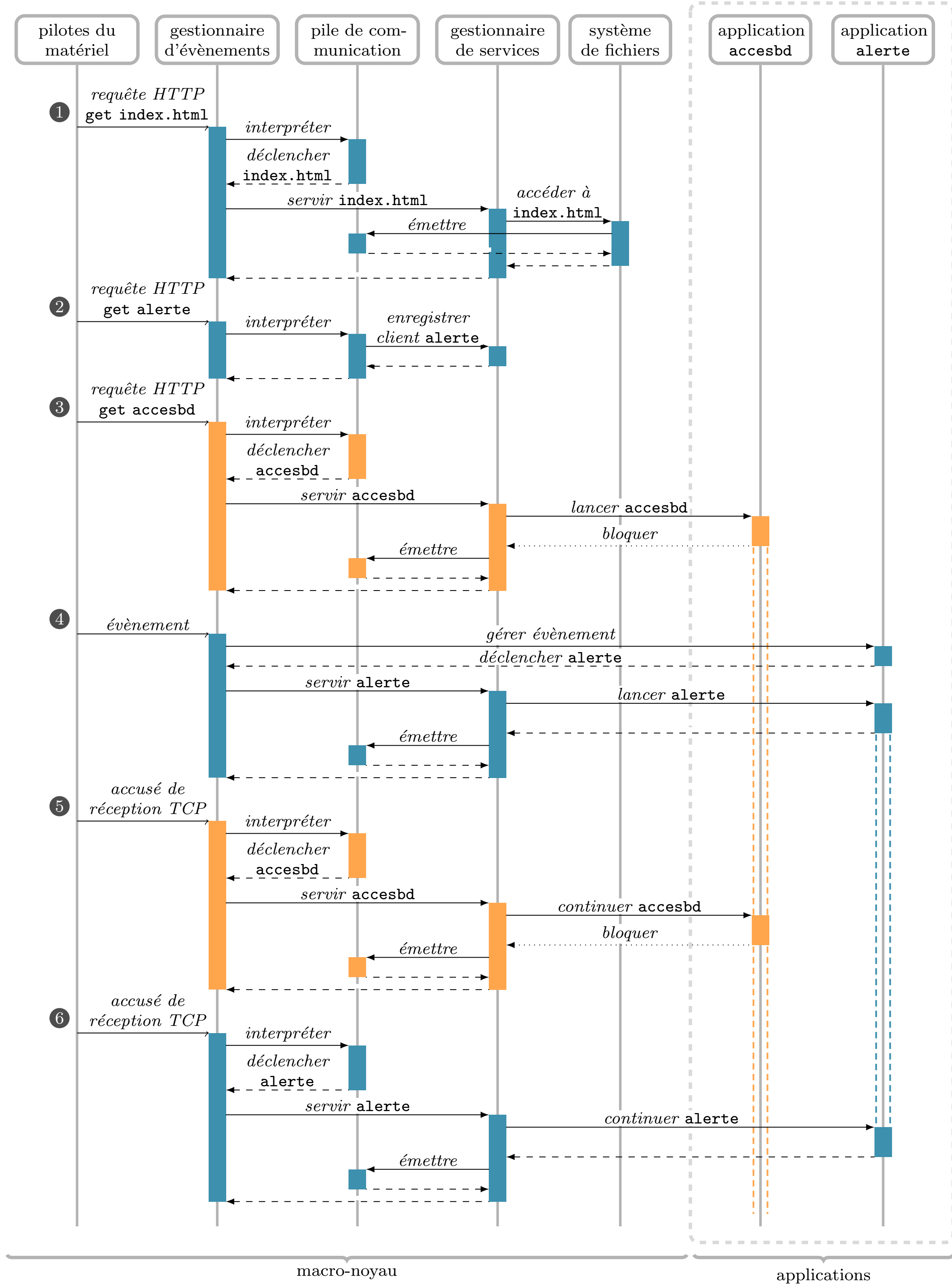


FIGURE 5.2 – Diagramme de séquence illustrant le comportement du système évènementiel lors du service des ressources `index.html`, `accesbd` et `alerte` sur deux connexions distinctes ■ et ■

Comet (voir sous-section 4.3.2, page 66). La connexion (■) est enregistrée auprès du gestionnaire de service comme étant en attente de notification. La connexion TCP sera inactive jusqu'à ce que l'alerte soit levée ;

3 – Arrivée et service de `accesbd` Le client émet une requête sur une seconde connexion (■), ciblant un contenu dynamique, dont les données seront générées par une application. Le gestionnaire de service est appelé. Il lance l'application `accesbd`, qui génère le premier segment de la réponse HTTP, puis bloque. Le blocage de l'application et les données générées sont gérées selon les stratégies de service définies dans le chapitre précédent ;

4 – Service de la requête `alerte` Un évènement déclenche une interruption matérielle, à laquelle un traitement applicatif a été associé. Le gestionnaire d'évènements appelle l'application concernée, qui demande la notification de la ressource `alerte`. Le gestionnaire de service est sollicité, il lance l'exécution de la fonction de rappel associée, qui génère un segment puis bloque ;

5 – Suite de la requête `accesbd` Pour des raisons didactiques, on suppose ici que les contraintes de mémoire interdisent d'émettre plus d'un segment en vol par connexion. Lors de la réception d'un accusé de réception, la connexion concernée (■) est débloquée. La routine applicative reprend son exécution, génère le second segment et bloque à nouveau ;

6 – Suite et fin de la requête `alerte` Un accusé arrive sur l'autre connexion (■). L'exécution de la fonction de rappel continue et retourne, puis la fin de la réponse HTTP est transmise.

5.1.3 Optimisations permises par la construction transversale

Au delà de l'implémentation entièrement évènementielle du système, la construction intégrée du noyau permet d'envisager des optimisations transversales dont l'objectif est d'améliorer les performances du logiciel embarqué et d'en réduire la consommation de ressources.

Identification des tâches les plus coûteuses

Afin d'évaluer les bénéfices de diverses optimisations transversales, nous caractérisons tout d'abord le coût d'exécution des différentes phases de la gestion d'une requête HTTP par un serveur embarqué. Nos mesures ont été réalisées sur la pile de communication uIP, qui propose une interface applicative générique à base de *protosockets*, des *sockets* construits à base de *protothreads*. Un serveur Web est fourni avec uIP à titre d'exemple d'application. On se place dans le cas simplifié d'un échange HTTP où la requête et la réponse nécessitent chacune exactement un segment TCP. La requête est de 396 octets et cible un fichier de 1460 octets. On ne s'intéresse ici qu'au temps processeur requis et non à la durée nécessaire à l'émission des données (assurée par une liaison série). Les tests ont été réalisés en déployant uIP sur un capteur WSN430, basé sur un micro-processeur 16 bits msp430. La figure 5.3 montre le temps processeur requis par les divers traitements de la pile de communication lors de la réception et l'émission d'un paquet.

Le calcul des sommes de contrôle TCP est le point le plus coûteux en traitements. Il nécessite près de 1,5 ms pour l'émission et 0,5 ms pour la réception (cet écart est simplement dû à la différence de taille entre le segment de la requête et celui de la réponse, de taille maximale). L'émission du fichier en tant que réponse HTTP impose à l'application (le serveur Web) de copier les données du fichier depuis une EEPROM vers le tampon de la pile IP, situé

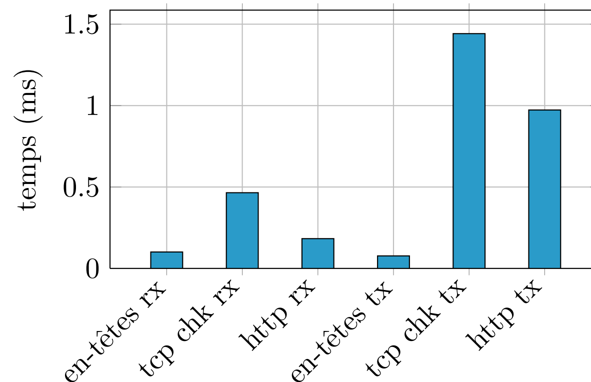


Figure 5.3 – Temps processeur (sur msp430) requis par uIP et son serveur Web lors de la réception d'une requête (396 octets) et l'émission d'un segment de réponse (1460 octets). *rx* : réception, *tx* : transmission

en mémoire vive. Cette copie de données est également assez coûteuse, nécessitant près de 1 ms. Ensuite, le décodage de la requête, qui consiste en l'occurrence à rechercher le fichier dont le nom correspond à l'URL, nécessite environ 0,2 ms. Le traitement des en-têtes IP et TCP ne nécessite quant à lui qu'envion 0.1 ms en réception comme en émission.

Des connexions dédiées

Contrairement aux piles IP gérant des connexions généralistes, la pile de communication de notre noyau peut utiliser des connexions dédiées à la gestion de requêtes HTTP et au service d'applications Web. Il est ainsi possible de gérer les données entrantes sans tampon. Lorsqu'une requête est reçue, la pile de communication interprète cette dernière à la volée et ne conserve en mémoire que les informations nécessaires à son service. Étant donnée la grande verbosité des requêtes HTTP, ce traitement réduit significativement la mémoire utilisée pour représenter une requête en attente. Dans le serveur Web de uIP, chaque connexion utilise un tampon de réception d'une taille fixée à 50 octets. Si on compare cette taille aux centaines d'octets constituant une requête HTTP classique, on constate que l'interprétation d'une requête entière nécessite alors plusieurs allers-retours (appels système bloquants) entre l'application et la pile IP ; impliquant un surcoût en terme de temps de traitement.

Lors de l'émission des données, nos connexion dédiées, couplées au gestionnaire de service, sont capables de faire une gestion efficace de la mémoire. Par exemple, dans le cas de données statiques, aucun tampon n'est nécessaire pour conserver les segments émis. La pile de communication est capable d'émettre les données directement depuis la mémoire non-volatile, évitant la lourde copie de données imposée par des connexions utilisant une abstraction de type *socket*. Dans le cas de Comet, où les mêmes données peuvent être envoyées à plusieurs clients, plusieurs connexions partagent un même tampon d'émission.

Le fait de dédier la pile de communication à HTTP permet également de supprimer un grand nombre de fonctionnalités d'une pile IP classique. L'interprétation et la génération des en-têtes IP et TCP s'en voient simplifiées, car une partie de leur contenu est figé par l'utilisation de HTTP. Nous avons implémenté un prototype afin de mettre à l'épreuve notre architecture et de mesurer les bénéfices des optimisations qu'elle permet. La structure utilisée

pour y représenter une connexion TCP et les données HTTP qui y sont associées ne nécessite que 30 à 40 octets de mémoire, à comparer aux 120 octets par connexion requis par le serveur Web de uIP. On multiplie alors par 3 à 4 le nombre de connexions qu'il est possible de gérer simultanément pour une quantité de mémoire donnée.

Traiter hors-ligne pour alléger l'exécution

Afin d'alléger la charge du serveur lors de l'exécution, nous proposons d'effectuer un grand nombre de traitements hors-ligne sur les applications Web. Ces pré-traitements sont réalisés lors de la phase de compilation des applications, avant leur liaison avec le noyau. Ils sont envisageables pour des applications déployées avec le serveur Web aussi bien qu'*a posteriori*.

Cette opération permet notamment d'alléger la phase de décodage des requêtes entrantes, réalisée par la pile de communication. En ayant la connaissance globale des applications déployées, on connaît statiquement l'ensemble des URLs atteignables et l'ensemble des arguments d'URLs possibles pour chaque ressource (on permet à chaque application de décrire le nom et le type des arguments qu'elle attend). On propose alors de construire un automate d'interprétation des URLs pour chaque chaîne à décoder. Un automate global est utilisé pour identifier la ressource ciblée et un automate dédié à chaque ressource permet d'interpréter et de décoder les arguments pour les passer à l'application concernée.

L'interprétation des chaînes se fait alors en temps constant pour chaque caractère et non en temps dépendant du nombre de ressources Web atteignables dans le système. Elle est donc très rapide et peut être réalisée sans aucun tampon. Si une requête est découpée en plusieurs paquets, il est aisé de reprendre le décodage là où il avait été interrompu, simplement à partir de l'état courant dans l'automate, synthétisant l'ensemble des caractères déjà interprétés. Le décodage des valeurs des arguments se fait également à la volée, de manière à stocker ces derniers directement dans leur forme compacte et non sous la forme de chaînes de caractères. Ainsi, toute la phase de décodage d'URL HTTP est assurée par la pile de communication et non par l'application. On permet aux applications de se concentrer sur leur traitement fonctionnel, tout en assurant un traitement rapide et consommant un minimum de mémoire.

Le calcul des sommes de contrôle TCP et IP, traitement particulièrement coûteux de la pile de communication, peut également être accéléré par la connaissance hors-ligne des contenus statiques. Dans certains serveurs Web entièrement intégrés comme iPic, webACE ou MiniWeb, tous les segments à envoyer sont entièrement construits hors-ligne, incluant les sommes de contrôle TCP. Cela impose cependant de fixer la taille des segments avant même déploiement du serveur Web. Pour des raisons de compatibilité lors de la négociation de la MSS, cette taille est typiquement fixée à 200 octets, ce qui bride les performances obtenues lors de l'émission de données (pénalité jusque 15.5 % dans le cas d'un débit de 250 kpbs et une latence de 25 ms selon notre modèle de trafic, *c.f.* sous-section 4.1.3, page 54).

Nous proposons de calculer les sommes de contrôle des fichiers statiques hors-ligne sur des blocs de données de taille fixe, en exploitant le fait que ce calcul soit associatif. Dans le système de fichier, ces sommes partielles sont stockées comme des méta-données pour chaque ressource statique. Il est alors possible d'allier calcul hors-ligne et taille dynamique des segments. Lors de l'émission des données, la somme de contrôle d'un segment se calcule non pas sur les données à émettre, mais sur les sommes partielles pré-calculées. Une contrainte est tout de même imposée sur la taille des segments, qui doit être multiple de la taille des blocs utilisés. La table 5.1 montre les bénéfices de cette technique en fonction de la taille de blocs choisie.

Avec des blocs de grande taille, le calcul des sommes de contrôle est rapide mais la taille

Taille des blocs	Mémoire requise	Taille des segments	Temps processeur
-	0 o	1460 o	1,442 ms
4 o	704 o	1460 o	0,721 ms
8 o	352 o	1456 o	0,360 ms
16 o	176 o	1456 o	0,180 ms
32 o	88 o	1440 o	0,089 ms
64 o	44 o	1408 o	0,044 ms
128 o	22 o	1408 o	0,022 ms

TABLE 5.1 – Impact de la taille des blocs des sommes de contrôle TCP calculées hors-ligne, dans le cas d'un client proposant une MSS de 1460 octets (processeur msp430)

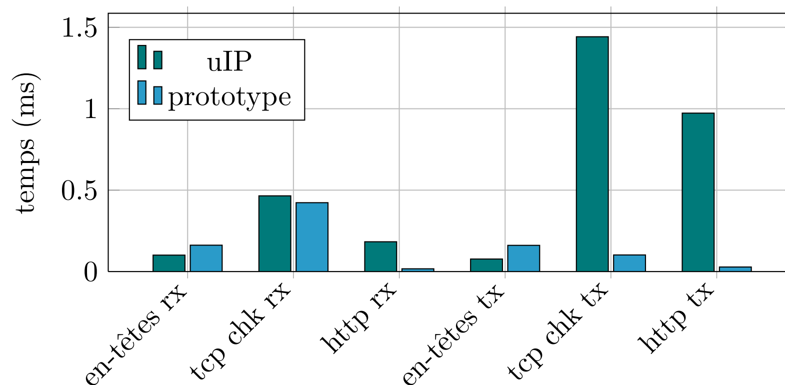


FIGURE 5.4 – Comparaison des temps processeur requis pour la gestion d'une requête pour uIP et pour notre prototype. Les conditions expérimentales sont les mêmes que pour la figure 5.3. *rx* : réception, *tx* : transmission

des segments est plus contrainte. La mémoire (non volatile) requise pour stocker les sommes partielles d'un segment décroît naturellement lorsque la taille des blocs augmente. Notons enfin qu'au moment d'embarquer un fichier statique, on est en mesure d'y adjoindre l'en-tête HTTP. Le calcul hors-ligne des sommes de contrôle est également réalisé sur cet en-tête.

5.1.4 Mesure des bénéfices

Afin de mesurer les bénéfices de la conception transversale sur les performances d'un serveur Web, nous avons déployé notre prototype sur un capteur WSN430 et mesuré le temps processeur nécessaire au service d'une requête. Notons qu'il s'agit de mesures de performances locales, évaluant ponctuellement l'impact de diverses optimisations. Une mise à l'épreuve du système complet, sur des cibles matérielles variées, considérant des ressources de diverses natures ainsi que le système d'exploitation dédié dans son ensemble, sera proposée dans le chapitre suivant. Ainsi, la figure 5.4 compare les performances de uIP à celles de notre prototype, pour les différents traitements de la pile de communication, dans les mêmes configurations que celles utilisées précédemment dans la figure 5.3.

On constate tout d'abord que les deux opérations les plus coûteuses dans uIP sont considérablement allégées dans la pile de communication de notre prototype. Le calcul de la somme de contrôle sur le segment envoyé est accéléré par la technique du calcul hors-ligne

par blocs. La copie des données du fichier depuis l'EEPROM vers le tampon de la pile IP n'est plus requise, puisque cette dernière est capable d'envoyer des données directement depuis l'EEPROM. Le calcul de somme de contrôle sur les données entrantes n'est quant à lui que légèrement accéléré. L'interprétation de la requête HTTP, nécessitant 0,2 ms dans le cas de uIP, est réalisé en seulement 0,02 ms dans notre cas grâce aux automates de décodage générés hors-ligne. Cependant, la gestion des en-têtes IP et TCP est légèrement plus lente dans notre prototype que dans uIP. Cela s'explique par le fait que notre implémentation, pour consommer aussi peu de mémoire que possible, traite les données entrantes et sortantes en flux, sans aucun tampon, interagissant directement avec le périphérique matériel. En contrepartie, les accès aux champs d'en-tête sont légèrement plus lents que lorsqu'ils sont réalisés par adressage direct d'un tampon de données, comme dans uIP. Le temps processeur total présenté dans la figure 5.4 est de 3,2 ms pour uIP contre 0,9 ms pour notre prototype.

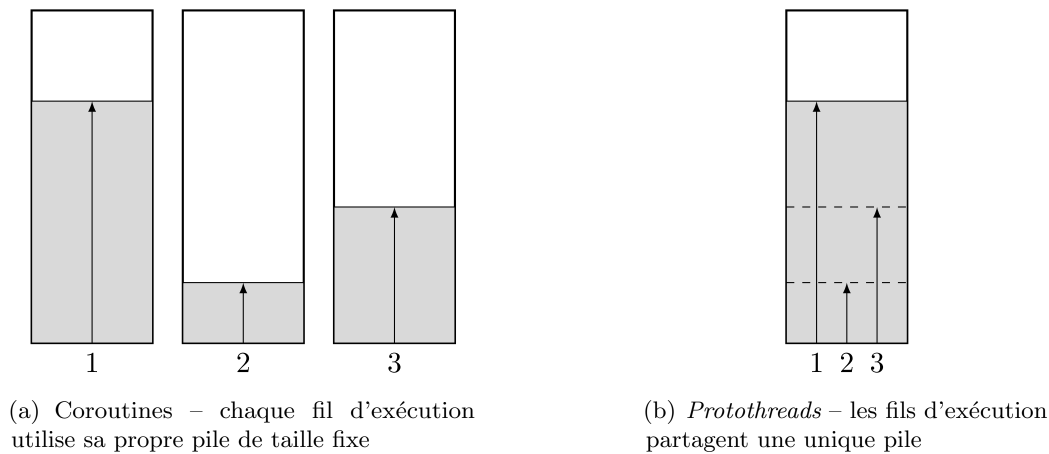
Nous n'avons pas proposé d'optimisation du décodage d'autres données entrantes que les URLs. Il serait cependant possible d'étendre cette approche au cas des requêtes *post* ou *put*. Ce type de requête est avant tout utilisé pour transmettre des données du client vers le serveur, contrairement à la classique commande *get*. Dans le cas de la réception par le serveur de données volumineuses, on peut imaginer de stocker ces dernières directement dans une mémoire non volatile, selon des modalités décrites par l'application. Cela est particulièrement pertinent pour une commande *put*, permettant à un client de stocker un fichier sur le serveur, où pour certains champs particuliers (destinés à un stockage persistant) de requêtes *post*.

Les optimisations permettant l'obtention des résultats présentés sont rendues possibles par le fait que seul HTTP est utilisé au-dessus de TCP et par la connaissance précise des applications Web embarquées avec le système complet. Dans un système classique, l'interface générique entre la pile IP et les couches supérieure interdit ce type de traitements. Dans un système entièrement intégré, de telles optimisations sont possibles, mais en pratique non réalisées, comme en témoignent les serveurs Web iPic, WebACE ou MiniWeb, fonctionellement très limités. Ces systèmes sont conçus de manière totalement fermée, pour une application précise et ne distinguent pas la notion d'application de celle de serveur. L'implémentation d'optimisations telles que le calcul d'automates de décodage ou de sommes de contrôle par blocs est envisageable lorsqu'on conçoit un système ouvert, dans lequel l'effort d'ingénierie réalisé lors de la construction du noyau est ensuite amorti par le déploiement d'applications tierces. Ces dernières, implémentées en ne se préoccupant que de traitements purement fonctionnels, sont moins sujettes aux erreurs de programmation ; elles bénéficient en conséquence d'une production plus rapide.

Nous avons montré ici que la construction transversale de notre macro-noyau permettait d'accélérer sensiblement la pile de communication tout en réduisant la mémoire qu'elle consomme. Il est ensuite nécessaire de s'intéresser à la couche supérieure, le gestionnaire de services, en charge de gérer les applications suivant les stratégies de service définies dans le chapitre précédent.

5.2 Continuations légères et ré-invocables

Le gestionnaire de services de notre macro-noyau est en charge de lancer, bloquer et restaurer de multiples routines applicatives, comme l'illustre la figure 5.2. Rappelons que parmi les stratégies présentées en section 4.3, page 61, S5 se fonde sur la notion de continuation qui représente un état dans un fil d'exécution. Cette stratégie conserve en mémoire une



(a) Coroutines – chaque fil d'exécution utilise sa propre pile de taille fixe

(b) *Protothreads* – les fils d'exécution partagent une unique pile

FIGURE 5.5 – Illustration de la réduction de mémoire consommée avec les *protothreads*

continuation pour chaque segment envoyé. Le gestionnaire de services doit donc gérer un nombre potentiellement important de continuations, c'est pourquoi il est nécessaire d'en minimiser la taille.

5.2.1 Fils d'exécution et mémoire consommée

La solution habituelle pour gérer de multiples fils d'exécution applicatifs consiste à utiliser des *threads*. Dans le cas qui nous intéresse, où les fonctions sont non préemptibles et coopératives, on parle de coroutines. Les coroutines classiques disposent chacune d'une pile d'exécution, contenant les variables locales et les appels imbriqués. Le blocage d'une coroutine consiste simplement à changer les pointeurs de pile et de programme du microprocesseur, afin de viser une nouvelle coroutine, qui se voit ainsi réactivée. Une continuation est alors constituée de la pile d'exécution et du pointeur de programme. L'avantage des coroutines est qu'elles peuvent supporter un code applicatif classique implémenté sans aucun outillage, où la notion de continuation n'apparait pas explicitement. L'inconvénient de cette solution dans le cas de processeur embarqué sans MMU est que chaque pile d'exécution doit être allouée statiquement, avec une taille choisie empiriquement. Le choix de cette taille implique un compromis entre possibilités applicatives et consommation mémoire des routines inactives. Par exemple, si on considère le système d'exploitation (pour capteurs) Mantis OS [BCD+05], la taille par défaut de la pile d'exécution des *threads* est fixée à 128 ocets.

Dans les systèmes embarqués disposant de quantités limitées de mémoire, on préfère souvent utiliser le paradigme évènementiel, moins consommateur de ressources. Les *protothreads* [DSVA06], présentés dans l'état de l'art (en sous-section 2.1.2, page 17), outillent l'écriture de code évènementiel afin que le fil d'exécution y soit lisible de manière linéaire. L'exécution des *protothreads* se fait sur la pile d'exécution de l'appelant. Contrairement aux coroutines, où chaque fil d'exécution dispose de sa propre pile d'exécution, les *protothreads* permettent de partager une unique pile, comme l'illustre la figure 5.5.

L'état d'un *protothread* bloqué est simplement constitué de l'adresse de la prochaine instruction à exécuter, représentant typiquement 2 octets sur les cibles 8 ou 16 bits qui sont visées. Lorsque plusieurs *protothreads* sont imbriqués, la continuation représente l'ensemble des points d'entrée de chaque sous-fonction. La taille de la continuation est donc proportionnelle à la

profondeur d'appel. Lors de l'activation d'un *protothread*, toute la pile d'appel est reconstruite à l'aide de la continuation par des appels de fonctions imbriquées, puis l'exécution reprend là où elle s'était arrêtée.

Cependant, les *protothreads* ont initialement été conçus pour l'implémentation de systèmes et non pour le développement d'applications. Ainsi, ils imposent plusieurs limitations lors de l'écriture du code. Par exemple, un *protothread* ne peut utiliser de variables locales et doit déclarer statiquement les sous-routines qu'il manipule. Il doit être écrit sans utiliser de bloc `switch` et doit explicitement manipuler son état par l'intermédiaire de macros. Il est contraint de n'appeler que des fonctions non bloquantes, ce qui limite l'usage de bibliothèques externes. Il peut faire appel à un sous-*protothread*, mais a la charge de gérer explicitement l'état de ce dernier. Il est possible d'enrichir le mécanisme des *protothreads* en permettant la gestion de variables locales, mais cela impose de déclarer ces dernières statiquement et de les passer explicitement en paramètre lors de chaque appel. Pour toutes ces raisons, les *protothreads* ne sont pas appropriés au développement d'applications de haut niveau telles que des applications Web.

5.2.2 Caractérisation des tailles des contextes

Le gestionnaire de services de notre noyau nécessite à la fois la légèreté des *protothreads* et l'absence d'outillage du code permise par les coroutines classiques. Afin de proposer une solution adaptée aux besoins d'applications embarquées, nous caractérisons la taille des contextes des applications du système d'exploitation pour capteurs Contiki. Nous avons instrumenté Contiki afin d'enregistrer la position du pointeur de pile lors de l'appel d'une application et lorsque cette dernière réalise un appel bloquant. L'écart entre ces deux positions indique le fragment de pile d'exécution représentant le contexte de l'application. Puisque le système Contiki et ses applications sont entièrement implémentés à l'aide de *protothreads*, leurs données locales ne sont pas conservées en pile d'exécution, mais en mémoire globale. Lors de la réalisation des mesures, nous avons donc explicitement pris en considération la taille de ces données.

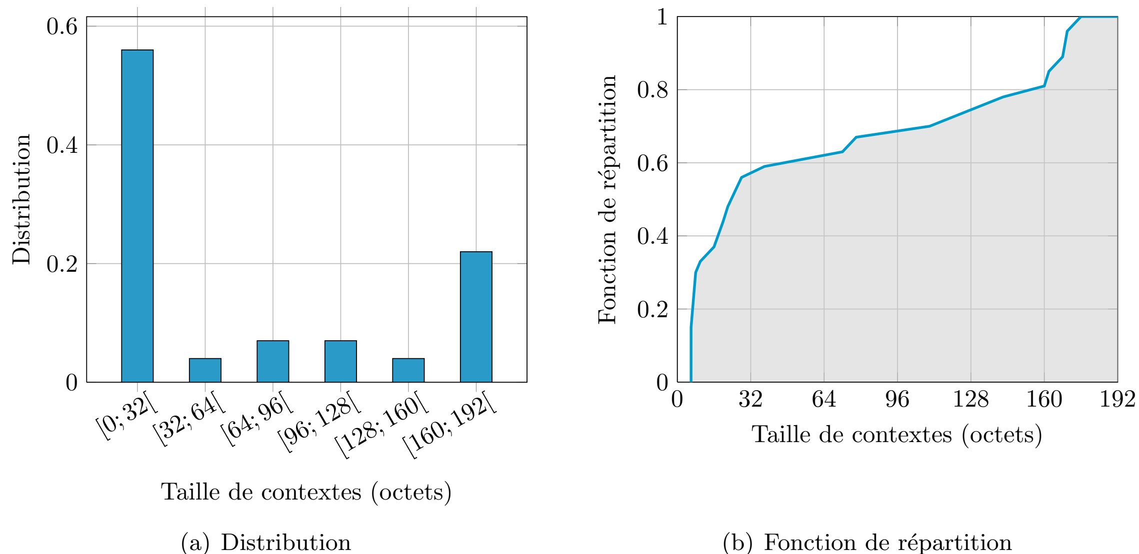
Les applications testées sont les celles fournies avec le système d'exploitation Contiki, exécuté sur le processeur 16 bits msp430 d'un capteur WSN430. Sur ce microprocesseur, la quantité de données empilées par la convention d'appel de fonction est comprise entre 2 et 14 octets. Elle comprend l'adresse de retour ainsi qu'un ensemble de registres de travail. Les applications testées incluent entre autre un serveur de ligne de commande utilisant soit UDP, soit TCP, des applications retournant le résultat d'échantillonnages du capteur, un décodeur base64 ou un serveur Web. La figure 5.6(a) montre la distribution des tailles de contexte applicatifs mesurés. Pour plus de précision, la fonction de répartition des tailles est donnée par la figure 5.6(b).

Ce travail de caractérisation nous conduit aux conclusions suivantes :

Les contextes sont de taille très variable Les contextes que nous avons mesurés ont une taille d'un minimum de 6 octets et d'un maximum de 176 octets. Cet écart important rend difficile le choix d'une taille de pile statique pour tous les fils d'exécution ;

La majorité des contextes sont de petite taille La taille médiane des contextes mesurés est de 26 octets. Un tiers d'entre eux ne dépasse pas 10 octets ;

Les fonctions utilitaires non bloquantes n'impactent pas Cette remarque ne découle pas directement de nos mesures, mais d'une observation réalisée pendant l'outillage du



(a) Distribution

(b) Fonction de répartition

FIGURE 5.6 – Caractérisation des tailles de contextes observées sur les applications de Contiki, sur un processeur msp430

système. Les fonctions appelées par une application n'ont un impact sur la taille des fragments applicatifs que si elles sont susceptibles de bloquer.

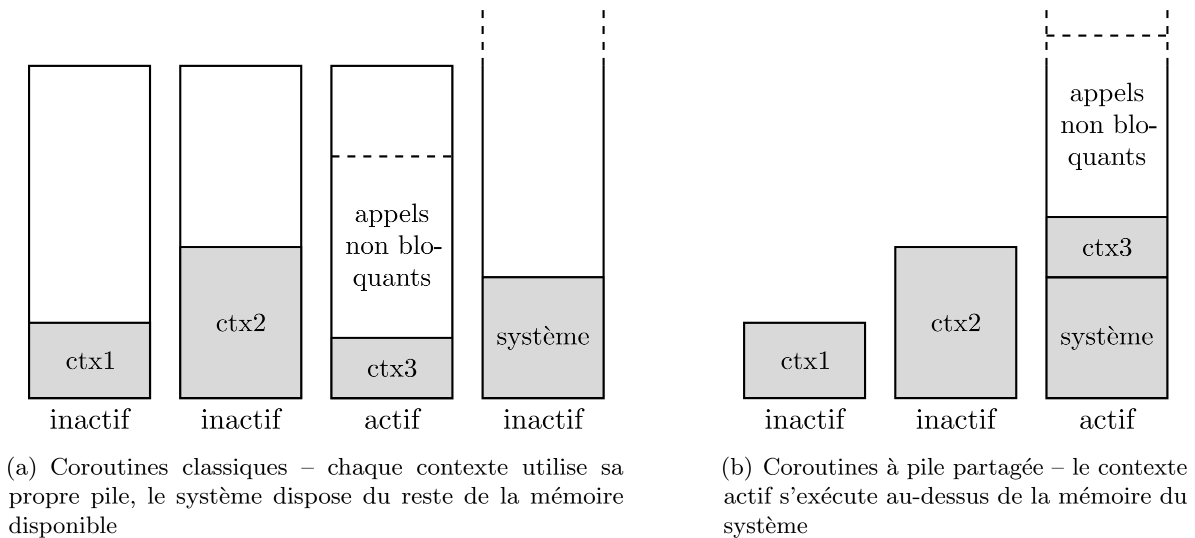
5.2.3 Coroutines à pile partagée

La caractérisation des contextes menée ci-avant suggère qu'avec une gestion plus fine de la taille des piles d'exécutions, il est possible de gérer un grand nombre de continuations en consommant peu de mémoire.

Principe de fonctionnement

Nous proposons un mécanisme à base de coroutines utilisant, comme les *protothreads*, une pile d'exécution partagée, mais étant implémentées avec du code natif classique non outillé. Nous évitons ainsi d'allouer statiquement la pile d'exécution des routines. Lors de la sauvegarde d'une continuation, nous proposons de n'enregistrer que le fragment de pile d'exécution qui est utilisée par la fonction. Nous identifions cette partie de la pile à l'aide de l'adresse de base initialement fournie et du pointeur de pile du micro-processeur. Ainsi, chaque coroutine inactive (ou continuation) a une consommation mémoire équivalente à son usage précis de la pile d'exécution au moment de son blocage. Ce point permet de minimiser la charge mémoire du système, surtout au vu de l'hétérogénéité de la répartition des tailles de contexte mesurés précédemment. Lors de la réactivation d'une coroutine, on recopie son contexte dans la pile principale puis on restaure les pointeurs de pile et de programme. La figure 5.7 illustre l'occupation mémoire qui découle de l'utilisation de coroutines classiques ou à pile partagée.

En termes d'usage de la mémoire, les *protothreads* suivent un schéma comparable à celui des coroutines à pile partagée. La seule différence se situe sur l'espace occupé par les continuations des fils d'exécution inactifs. Dans notre cas, on stocke le fragment de pile utilisé au moment du



(a) Coroutines classiques – chaque contexte utilise sa propre pile, le système dispose du reste de la mémoire disponible

(b) Coroutines à pile partagée – le contexte actif s'exécute au-dessus de la mémoire du système

FIGURE 5.7 – Illustration de l'occupation mémoire selon le type de coroutine utilisée

blocage. Dans le cas des *protothreads*, toutes les fonctions retournent avant le blocage. Seuls les points d'entrée dans chaque sous-routine sont conservés. Lors de la réactivation, le contexte du *protothread* est intégralement reconstruit.

Avantages fonctionnels

D'un point de vue fonctionnel, les coroutines à pile partagée présentent d'excellentes propriétés. Tout d'abord, en comparaison aux *protothreads*, elles autorisent l'écriture de code natif standard, non outillé et sans contrainte. Il est alors possible d'appeler du code provenant d'une quelconque bibliothèque. Enfin, l'imbrication de sous-routines bloquantes est transparente alors que dans le cas des *protothreads*, il est nécessaire de déclarer statiquement toute sous-fonction bloquante.

Contrairement aux coroutines classiques, notre solution n'impose pas une limite arbitraire sur la profondeur des appels lorsqu'une application s'exécute. La seule limite est celle de la mémoire globalement disponible. En ne sauvegardant que le fragment de pile utilisé lors d'un blocage, on n'inclut pas la mémoire consommée par les fonctions intermédiaires non bloquantes. Il est ainsi possible de réaliser des traitements nécessitant un espace important sur la pile d'exécution, comme l'interprétation ou l'encodage de données dans un format particulier (*e.g.* XML ou JSON). Les appels système bloquants s'exécutent eux aussi sur la pile partagée, après dépilement et sauvegarde du contexte de la coroutine.

Caractérisation des surcoûts

La contrepartie des coroutines à pile partagée est qu'elles conduisent à un coût élevé des changements de contexte. En effet, l'activation d'une coroutine nécessite systématiquement la sauvegarde d'un fragment de la pile courante et la restauration de la nouvelle pile. Nous mesurons ici l'importance de ce surcoût afin d'évaluer l'utilisabilité des coroutines à pile partagée dans le noyau de notre système d'exploitation dédié.

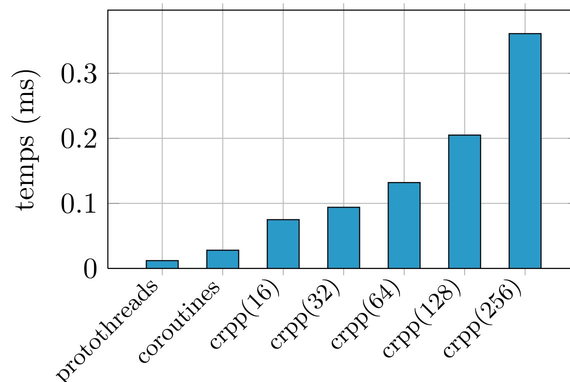


FIGURE 5.8 – Temps processeur (sur msp430) nécessaire à un changement de contexte selon la technique utilisée (crpp(n) : coroutines à pile partagée pour un contexte de n octets)

Nous avons implémenté un gestionnaire de coroutines à pile partagée pour microprocesseur 16 bits msp430, afin de comparer la durée nécessaire aux changements de contexte selon qu'on utilise des *protothreads*, des coroutines classiques ou des coroutines à pile partagée. Dans le cas des coroutines à pile partagée, la mesure a été réalisée pour différentes tailles de contexte, en concordance avec les mesures réalisées sur Contiki, c'est-à-dire allant de 16 à 256 octets. La figure 5.8 donne le résultat de nos mesures, obtenues en moyennant 100 000 changements de contexte pour chaque technique.

Dans le meilleur des cas, le changement de contexte est environ 6 fois plus lent avec les coroutines à pile partagée qu'avec les *protothreads*. Dans le cas de contextes applicatifs de 256 octets, ce ratio s'élève à 30. Les coroutines classiques, quant à elles, ont un coût intermédiaire entre *protothreads* et coroutines à pile partagée.

Si on considère les valeurs mesurées de manière absolue, un changement de contexte dans le pire cas (contexte de 256 octets, jamais rencontré lors de l'exécution d'applications de Contiki) nécessite 0,36 ms. Dans le cas de contextes de 32 octets (taille non dépassée dans 56 % des cas selon nos mesures), le coût est de 0,09 ms. Rappelons que dans notre noyau, on a au maximum un changement de contexte par segment généré (comme l'illustre la figure 5.2). Le surcoût impliqué par nos coroutines est tout à fait acceptable si on le compare au temps processeur requis pour la construction d'un segment (*c.f.* figure 5.4), de 2,5 ms pour uIP et 0,3 ms pour notre prototype.

L'impact de la ré-invocation

Dans le cadre de la stratégie d'émission S5, le gestionnaire de services copie une continuation pour chaque segment émis. Lorsqu'une retransmission TCP est requise, il ré-invoque la routine pour générer à nouveau les données. Le concept des continuations ré-invocables existe dans la littérature. Il a été théoriquement exposé par Thielecke *et al.* [Thi98]. Étonnamment, les langages de programmation qui permettent au programmeur de manipuler les continuations, tels que Scheme ou Python, ne supportent pas la ré-invocation. Cette fonctionnalité est pourtant envisageable pour les coroutines classiques, les *protothreads* comme pour nos coroutines à pile partagée.

Dans le cas d'une coroutine classique, les continuations contiennent une pile d'exécution

	protothreads	coroutines	coroutines à pile partagée
coût d'ordonnancement	appel	appel	copie contexte + appel
coût de ré-invocation	copie variables + appel	copie pile + appel	copie contexte + appel
mémoire/continuation	$variables+$ $(profondeur \times 2)$	max. pile	$variables+$ $(profondeur \times appel)$
limite de profondeur	figée par le code source	taille arbitraire des piles	quantité de mémoire totale
spécificités du code	absence de locales, absence de `switch`, manipulation explicite des sous-routines	aucune	aucune

Table 5.2 – Caractéristiques des *protothreads*, des coroutines et des coroutines à pile partagée

complète. La sauvegarde d'une continuation nécessite une copie de cette pile. En cas de ré-invocation, il est nécessaire de recopier la pile à son emplacement d'origine ; l'activation d'une pile d'exécution hors de son emplacement d'origine est en effet impossible car elle invaliderait les pointeurs ciblant des données locales.

Dans le cas d'un *protothread*, l'activation consiste simplement à appeler ce dernier avec pour paramètres un pointeur sur l'instruction à exécuter et un pointeur sur une structure contenant les données locales. Dans le cas de la ré-invocation, afin d'éviter l'invalidation de pointeurs sur des données locales, il est là encore nécessaire de copier à leur emplacement d'origine l'ensemble des variables (stockées dans une structure externe) du *protothread.*

Pour les coroutines à pile partagée, l'activation nécessite systématiquement une copie du contexte à son emplacement d'origine. La ré-invocation n'est qu'un cas particulier d'activation, elle est donc naturellement supportée. En guise de synthèse, la table 5.2 rappelle les particularités des *protothreads*, des coroutines et des coroutines à pile partagée.

La technique des coroutines à pile partagée permet ainsi d'envisager la gestion simultanée d'un grand nombre de fils d'exécution applicatifs au sein du gestionnaire de services, avec une maîtrise fine de la consommation mémoire. Il est alors possible de se focaliser sur les stratégies d'ordonnancement dans le contexte d'un serveur du Web des objets.

5.3 Ordonnancement des requêtes

Nous nous intéressons ici à l'ordonnancement des requêtes à traiter dans le macro-noyau. Comme nous l'avons vu dans la figure 5.2, c'est le gestionnaire d'évènements qui désigne en permanence le prochain traitement à effectuer. Lorsque plusieurs requêtes sont présentes dans le système, il en élit une puis demande au gestionnaire de services d'émettre les données associées à la ressource ciblée. Il s'agit alors soit d'accéder au système de fichiers, soit de lancer ou de débloquer une coroutine applicative.

Les travaux conduits sur l'ordonnancement de requêtes au sein de serveurs Web montrent que la politique utilisée a potentiellement un impact important sur la performance ou l'équité du système [HBSBA03, LSD04, GW04, SHB06]. Dans les serveurs de production, typiquement fondés sur un système d'exploitation à noyau monolithique, le serveur Web s'appuie sur des *sockets* de Berkeley. Les développeurs de ces serveurs ne se soucient pas de la politique d'ordonnancement utilisée, car les *sockets* ne permettent pas de contrôler l'ordre de traitement des connexions.

Nous souhaitons bénéficier de l'architecture de notre macro-noyau pour concevoir un ordonnanceur qui exploite les données applicatives, comme la taille des ressources ou la mémoire consommée pendant le blocage d'une routine. L'objectif est alors d'optimiser performances et charge mémoire tout en garantissant une équité auprès des clients.

5.3.1 Identification des métriques appropriées

L'ordonnancement des requêtes d'un serveur Web est un exemple classique d'application de la théorie de l'ordonnancement. Dans ce contexte, il s'agit de partager les ressources du serveur Web entre les requêtes présentes dans le système, afin d'optimiser une métrique en particulier. Les bases du formalisme de ce problème ont été introduites dans l'état de l'art, en sous-section 2.3.3, page 33.

Dans la plupart des travaux la métrique considérée est le temps de séjour moyen des requêtes dans le système. De nombreux résultats permettent l'optimisation de cette métrique sous diverses conditions ; pourtant, les serveurs de production n'adoptent pas les politiques SRPT ou FSP, préférant la classique technique du tourniquet. Nous tentons ici de comprendre cet état de fait et nous discutons les métriques appropriées à l'évaluation d'un ordonnancement dans le cadre d'un serveur du Web des objets.

Limites de la métrique du temps de séjour

Les travaux s'appuyant uniquement sur la métrique du temps de séjour font l'hypothèse que l'état d'avancement intermédiaire des requêtes n'importe pas [FH03]. Seules les dates de complétion des tâches sont prises en compte par cette métrique. Cet état de fait est illustré dans l'état de l'art, sous-section 2.3.3, page 33. Afin de minimiser le temps de séjour des requêtes, les algorithmes comme SRPT ou FSP minimisent le partage des ressources entre les tâches, qui ne ferait que retarder la prochaine date de complétion [Sch68, FH03]. En conséquence, il arrive fréquemment que le traitement d'une tâche soit interrompu pendant une durée arbitrairement longue. De son côté, l'algorithme PS (approximant la politique classique du tourniquet) partage équitablement les ressources du système à chaque instant. Ainsi, l'état intermédiaire des tâches évolue en permanence.

Dans le cadre de serveurs du Web des objets, nous pensons que la métrique du temps de séjour moyen, à elle seule, n'est pas appropriée, ce qui explique pourquoi les algorithmes SRPT et FSP ne sont toujours pas adoptés par les serveurs Web. Nous avons identifié quatre raisons pour lesquelles l'absence de partage de ressource pénalise l'ensemble du système :

Capacité physique des chemins En pratique, le fait de partager une liaison entre plusieurs clients est sensiblement plus efficace que de charger au maximum chaque chemin réseau consécutivement. En effet, les connexions vers les clients sont hétérogènes et le fait de surcharger un chemin conduit à saturer ce dernier, à provoquer des pertes de données, *etc.*. En partageant la liaison entre toutes les connexions, on évite ces phénomènes ;

Inactivité des connexions Le fait de conserver certaines connexions totalement inactives pendant une longue durée risque de conduire à des déconnexions automatiques, initiées soit par l'hôte distant, soit par un pare-feu ou un serveur mandataire intermédiaire ;

Applications affichables progressivement Les applications et les contenus Web sont construits de telle manière qu'ils sont affichables progressivement, au fil de la réception des

données. Le rendu graphique de l'application par le navigateur Web se fait progressivement, permettant au client de commencer la consultation et l'interaction. Cela suggère que l'état intermédiaire des requêtes devrait être pris en compte ;

Expérience utilisateur Le fait de bloquer le service de certaines requêtes pendant un certain temps dégrade l'expérience utilisateur. Après avoir attendu quelques secondes, l'utilisateur du navigateur va conclure que la page requise n'est pas joignable et risque de lancer une nouvelle requête vers le serveur.

Malgré ces défauts, la métrique du temps de séjour moyen a tout de même quelques avantages dans le cas de serveurs Web embarqués. En optimisant cette métrique, SRPT ou FSP minimisent le nombre de tâches présentes dans le système à chaque instant. On réduit ainsi la charge mémoire du serveur, qui est particulièrement critique dans notre contexte. On peut également espérer améliorer les performances, dans le cas où des routines dont la complexité dépend du nombre de requêtes dans le système sont utilisées (ce qui est typiquement le cas de la routine `select()` utilisée pour écouter les évènements [BM98, BDM98]).

Pour ces raisons, nous pensons que la métrique du temps de séjour moyen doit être utilisée avec prudence dans le cadre du Web des objets. Elle doit être complétée par une métrique évaluant le niveau de partage des ressources du système.

Prise en compte le partage des ressources

Dans des travaux récents, Raz *et al.* ont proposé une métrique novatrice pour la mesure de l'équité considérant à la fois l'ancienneté des tâches et leur durée d'exécution [RLAI04]. Pour arriver à ses fins, cette métrique, nommée RAQFM, s'intéresse au partage des ressources du système. Elle se fonde sur la notion de discrimination individuelle, correspondant à l'intégration, pendant le séjour d'une tâche, de la différence entre la quantité de ressources allouées à la tâche et la quantité de ressource moyenne allouée à toutes les tâches. Soit a_i la date d'arrivée dans le système de la tâche i, d_i sa date de départ, $s_i(t)$ la part de ressources allouée à la tâche i au temps t et $N(t)$ le nombre de tâches dans le système au temps t. La discrimination individuelle D_i est définie ainsi :

$$D_i = \int_{a_i}^{d_i} \left(s_i(t) - \frac{1}{N(t)} \right) dt$$

L'équité d'un ordonnancement est ensuite calculée comme la variance des discriminations individuelles des tâches du système. On appelle l'*équité instantanée* le niveau de partage des ressources entre les tâches à chaque instant. En mesurant l'équité instantanée, la métrique RAQFM prend également en considération l'état intermédiaire des requêtes et semble appropriée à notre contexte.

La politique PS, en partageant équitablement les ressources à chaque instant, conduit à des discriminations individuelles systématiquement nulles. Le fait que PS, utilisée dans l'industrie, optimise l'équité RAQFM confirme la pertinence de cette métrique. La politique SRPT ne permet pas une bonne équité instantanée, puisqu'elle utilise l'intégralité des ressources du serveur pour une unique requête jusqu'à sa complétion.

Cependant, SRPT produit des temps de séjour plus courts que PS, diminuant le nombre de tâches dans le système et en conséquence la charge mémoire du serveur. Dans le cadre de notre système d'exploitation dédié, la charge mémoire est un critère particulièrement important. Nous proposons de concevoir dans un premier temps un ordonnanceur qui considère à la fois

le temps de séjour moyen des requêtes et l'équité instantanée des connexions. Dans un second temps, nous prendrons en considération plus finement la charge mémoire du noyau.

5.3.2 Un compromis entre rapidité et partage des ressources

Puisque la rapidité pure et le partage des ressources semblent antagonistes, nous proposons un algorithme d'ordonnancement paramétrable, permettant au concepteur du système de privilégier l'une ou l'autre des métriques.

L'algorithme β-SRPT

L'algorithme que nous proposons, appelé β-SRPT, a pour objectif de réaliser un compromis continu entre SRPT et PS, en fonction de la valeur d'un paramètre β. Lorsque $\beta = 0$, l'algorithme doit se comporter comme SRPT, alors que lorsque $\beta = 1$, il doit être équivalent à PS. Pour cela, on base β-SRPT sur la politique PS généralisée, qui sert chaque tâche de manière continue en considérant un poids individuel [1]. Comme SRPT, β-SRPT favorise les tâches dont le temps restant est le plus court.

À tout instant, les N tâches du système sont triées par ordre croissant de temps restant (taille de requête restant à servir). Le poids w_j de la tâche de rang j est définie comme suit :

$$w_j = \frac{\beta^j}{\sum_{i=0}^{N-1} \beta^i}$$

En d'autres termes, le poids des tâches décroît exponentiellement (base β) avec leur rang (selon le classement par temps restant croissant). C'est ce qui permet de doser, en fonction de valeur de β, l'importance que l'on donne aux tâches qui sont proches de la terminaison. On a :

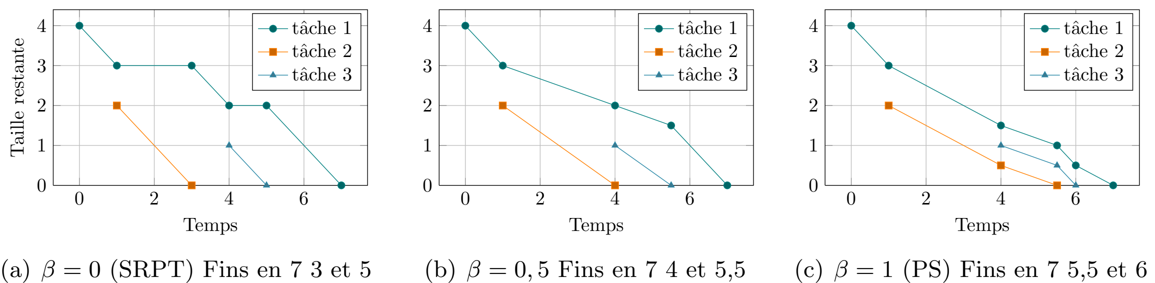
$$w_j = \begin{cases} \frac{1}{N} & \beta = 1 \\ \beta^j \times \frac{1-\beta}{1-\beta^N} & \beta \neq 1 \end{cases}$$

Lorsque $\beta = 1$, chaque tâche a le même poids $\frac{1}{n}$, si bien que la politique est strictement équivalente à PS. Avec $\beta = 0$, on a $w_0 = 1$ et $w_j = 0$ pour tout $j > 0$. Comme les tâches sont triées par ordre croissant de temps restant, dans cette situation, cette politique est strictement équivalente à SRPT.

En fonction de la valeur du paramètre β, l'algorithme β-SRPT se positionne entre SRPT et PS, à la fois en termes de temps de séjour moyen et d'équité instantanée. Quelle que soit la valeur de β, la $j + 1^{eme}$ requête est servie β fois plus lentement que le j^{eme} requête. En augmentant β, on réduit le temps de séjour moyen et on diminue l'équité instantanée. La figure 5.9(b) illustre le comportement de β-SRPT pour différentes valeurs de β.

Plus la valeur de β est élevée, plus on évite la stagnation des requêtes, et moins on favorise les tâches proches de se terminer. Le comportement dans les cas $\beta = 0$ et $\beta = 1$ est bien respectivement équivalent à SRPT et PS. Notons qu'en pratique, la taille exacte des requêtes à servir n'est pas nécessairement connue *a priori*. Dans ce cas, il est possible d'estimer la taille en moyennant les résultats d'exécutions précédentes des mêmes tâches avec les mêmes paramètres (Lu *et al.* ont montré que la précision d'une telle approximation avait un impact sensible sur les performances des politiques telles que SRPT ou FSP [LSD04]).

1. L'implémentation d'une politique PS généralisée requiert de considérer la granularité des paquets, ce qui peut être réalisé précisément avec la politique WFQ [2] [PG93]



(a) $\beta = 0$ (SRPT) Fins en 7 3 et 5 (b) $\beta = 0,5$ Fins en 7 4 et 5,5 (c) $\beta = 1$ (PS) Fins en 7 5,5 et 6

FIGURE 5.9 – Exemples d'ordonnancement avec l'algorithme β-SRPT.

Paramètres de simulation

Nous mesurons l'impact du paramètre β sur le temps de séjour moyen et l'équité RAQFM par l'intermédiaire de simulations. Suivant plusieurs études empiriques, il est largement accepté que les trafics Web peuvent être modélisés précisément en utilisant une file M/G/1 dite à *longue traîne*. Le processus d'arrivée des tâches est un processus de Poisson de paramètre λ. La distribution des tailles des tâches (*i.e.* des requêtes) est une distribution de Pareto bornée de forme α légèrement supérieure à 1. Avec une telle distribution, les petits contenus sont nombreux mais les grands contenus sont suffisamment importants pour représenter la majeure partie du volume total transféré.

Comme dans [BHB01], nous utilisons $\alpha = 1,1$ pour la distribution de Pareto bornée. La taille minimale des tâches x_{min} est choisie de telle manière que la taille moyenne des tâches soit de 1, est que la taille maximale des tâches soit $x_{max} = 10^5$. Le choix d'une moyenne de 1 est arbitraire ; c'est en choisissant une unité appropriée qu'on fait correspondre la simulation à un cas réel. La charge globale ρ est calculée comme le produit entre le paramètre λ du processus de Poisson et la taille moyenne des tâches, on a $\rho = \lambda$.

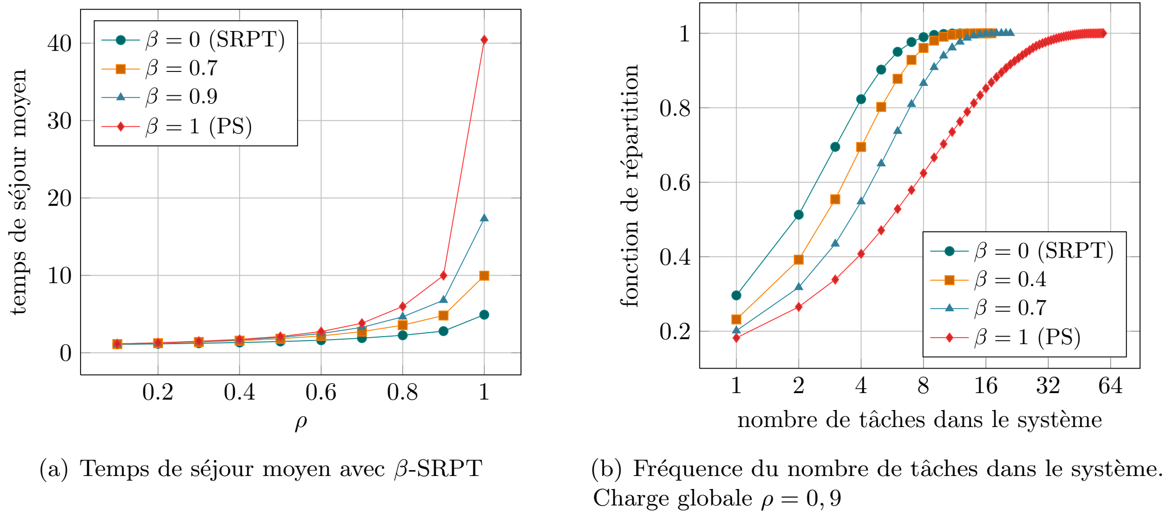
Nous avons écrit un simulateur capable d'exécuter PS, SRPT ou β-SRPT pour tout paramètre $0 \leq \beta \leq 1$ et pour toute charge globale $0 < \rho \leq 1$. Chaque simulation a été conduite sur 1 million de tâches. Nous avons validé notre simulateur en :

- S'assurant pour toute simulation que la loi de Little [Lit61] est systématiquement valide ;
- Reproduisant les simulations décrites dans l'article de Friedman *et al.* [FH03]. Nous avons obtenu des résultats quasiment identiques pour SRPT comme pour PS (les simulations sont stochastiques).

Impact sur la rapidité

Nous nous intéressons tout d'abord à la rapidité du service des requêtes en fonction du paramètre β. La figure 5.10(a) montre le temps de séjour moyen des tâches pour différentes valeurs de β et sous des charges globales allant de 0,1 à 1. À mesure que la charge augmente, le temps de séjour des tâches s'allonge logiquement. Comme nous l'attendions, l'augmentation du paramètre β allonge le séjour des tâches. On remarque que l'écart entre différentes valeurs de β est d'autant plus important que le système est chargé.

La figure 5.10(b) montre quant à elle l'évolution du nombre de tâches actives en fonction des paramètres de simulation. Comme mentionné précédemment, il y a une relation directe entre le temps de séjour des tâches et le nombre de tâches dans le système. Ainsi, en diminuant la valeur du paramètre β, on diminue le nombre de connexions actives sur le serveur, donc



(a) Temps de séjour moyen avec β-SRPT

(b) Fréquence du nombre de tâches dans le système. Charge globale $\rho = 0,9$

FIGURE 5.10 – Impact de β et ρ sur la rapidité de service avec β-SRPT

la quantité de mémoire consommée par ce dernier. Par exemple, avec un paramètre $\beta = 1$ (PS), la probabilité d'avoir 4 tâches ou moins dans le système est d'environ 0,4, contre 0,9 avec $\beta = 0$ (SRPT).

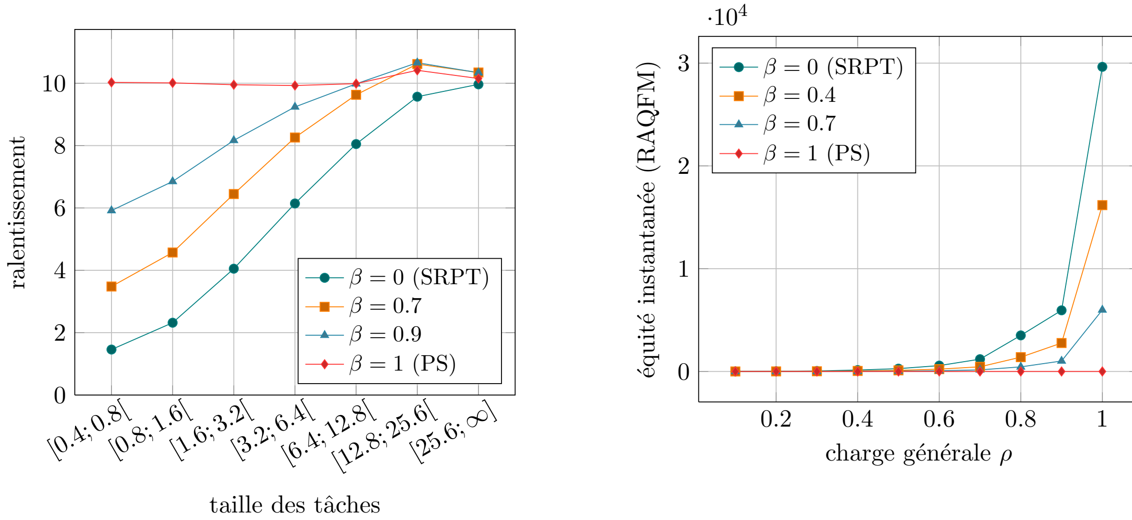
Impact sur l'équité et le partage de ressources

Usuellement, l'équité de l'ordonnancement de requêtes Web est évaluée en analysant la corrélation entre la taille des tâches et leur ralentissement moyen (rapport entre le temps de séjour et la taille initiale de la tâche) [HBSBA03, FH03, LSD04, GW06]. Si la taille des tâches a un impact sur leur ralentissement moyen, l'ordonnancement souffre d'iniquité endogène. La figure 5.11(a) montre le ralentissement moyen en fonction de la taille des tâches, pour diverses valeurs de β sous une charge globale $\rho = 0,9$. Ici, l'iniquité endogène est d'autant plus importante que β est proche de 0, ce qui est cohérent avec le compromis que β-SRPT est supposé réaliser entre PS et SRPT.

La figure 5.11(b) montre quant à elle l'évolution de la métrique RAQFM selon la valeur de β, sous diverses charges globales. Lorsque β s'approche de 0, la note RAQFM, augmente donc l'équité instantanée diminue. La distribution à longue traîne des tailles de tâches provoque l'obtention de notes très élevées dans certaines situations, ici jusque 30 000. Cela est dû à l'importance des tâches longues sur la charge globale. La métrique RAQFM, considérant à la fois le partage de ressources et l'état intermédiaire des requêtes, confirme que β-SRPT représente un compromis entre PS et SRPT, en fonction du paramètre β.

Caractérisation du compromis

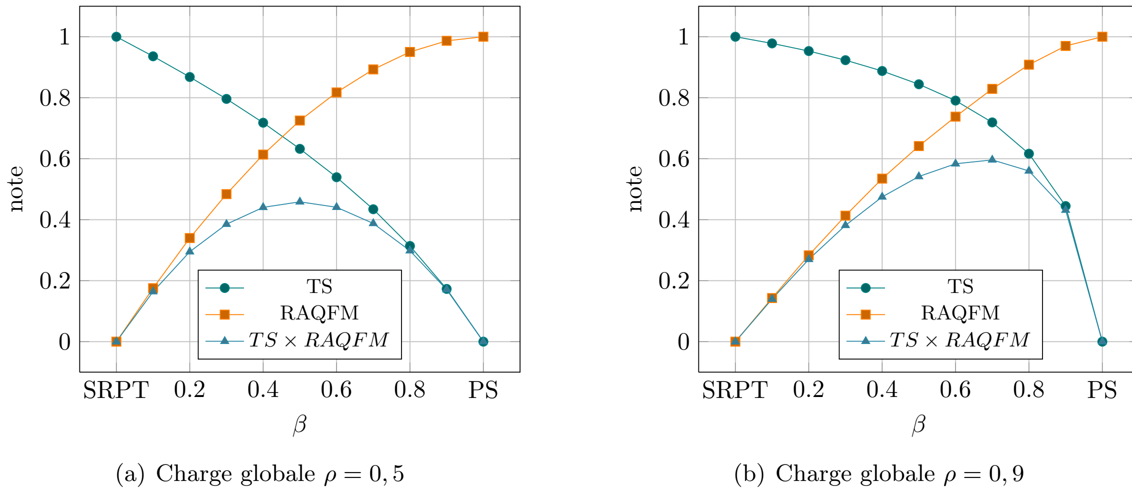
Lors de la conception de l'ordonnanceur, le choix du paramètre β détermine l'importance que l'on souhaite donner aux temps de séjour ou au partage des ressources. Afin de caractériser le compromis entre ces deux critères, nous affectons une note normalisée entre 0 et 1 pour le temps de séjour moyen ainsi que pour RAQFM. Les algorithmes extrêmes SRPT et PS permettent la calibration de ces notes. La figure 5.12 montre l'évolution de ces deux notes en



(a) Ralentissement en fonction de la taille des tâches β-SRPT. Charge globale $\rho = 0,9$

(b) Équité instantanée avec β-SRPT

FIGURE 5.11 – Impact de β sur l'équité et le partage des ressources avec β-SRPT



(a) Charge globale $\rho = 0,5$

(b) Charge globale $\rho = 0,9$

FIGURE 5.12 – Compromis entre performances et équité instantanée avec β-SRPT. TS : temps de séjour

fonction de β pour une charge globale de 0,5 ou 0,9.

Le croisement entre les deux courbes indique la valeur de β pour laquelle on atteint un équilibre entre les deux métriques. On constate que cet équilibre n'est pas atteint pour la même valeur de β en fonction de la charge globale. Les courbes tracent également le produit entre les deux notes, représentant le compromis obtenu entre les deux métriques. Ce compromis dépend lui aussi de la charge globale. Il est maximisé à $\beta = 0,5$ pour une charge $\rho = 0,5$ et à $\beta = 0,7$ pour une charge $\rho = 0,9$. Il est également intéressant de constater que l'évolution des courbes n'est pas linéaire, si bien que le produit des notes peut dépasser 0,25 et que le croisement entre les deux courbes se situe au-dessus de 0,5.

Le compromis permis par β-SRPT semble donc particulièrement avantageux. Par exemple, avec une charge $\rho = 0,9$, si on a besoin d'un ordonnanceur qui partage la liaison réseau équitablement (proche de PS) tout en prioritisant légèrement les requêtes courtes afin de réduire le nombre de connexions dans le serveur, on peut choisir $\beta = 0,9$. On a alors un compromis intéressant, avec une note RAQFM de 0,97 et une note sur le temps de séjour de 0,44.

5.3.3 Considération affinée de la charge mémoire

Comme nous l'avons mentionné précédemment, la minimisation du temps de séjour moyen permet de réduire le nombre de requêtes présentes simultanément sur le serveur. Dans le noyau de notre système d'exploitation dédié, chaque requête sur le serveur nécessite une quantité de mémoire différente, selon la nature des contenus qu'elle cible et selon la taille de contexte nécessaire à l'éventuel stockage des continuations. Nous proposons de nous focaliser sur la consommation mémoire du serveur en s'appuyant sur la notion de charge mémoire, présentée précédemment en sous-section 4.1.4.

On affecte à chaque tâche j une consommation instantanée de mémoire m_j. La charge mémoire c_j d'une tâche est calculée comme $c_j = (C_j - r_j) \times m_j$ (produit du temps de séjour et de la consommation instantanée de mémoire). On se propose alors de minimiser la charge mémoire globale du système. Soit $A(t)$ l'ensemble des tâches présentes dans le système au temps t. La charge mémoire globale du serveur est :

$$C = \int_0^\infty \sum_{j \in A(t)} m_j, dt = \sum_j (C_j - r_j) \times m_j$$

La minimisation de la charge mémoire du système est strictement équivalente au problème de minimisation du temps de séjour pondéré $1|r_i, pmtn| \sum w_i C_i$, où le poids w_j des tâches correspond à la consommation mémoire m_j. Ce problème est NP-dur (*c.f.* sous-section 2.3.3). La version pondérée de SRPT, WSRPT, optimise cependant le cas sans date d'entrée des tâches et sans préemption.

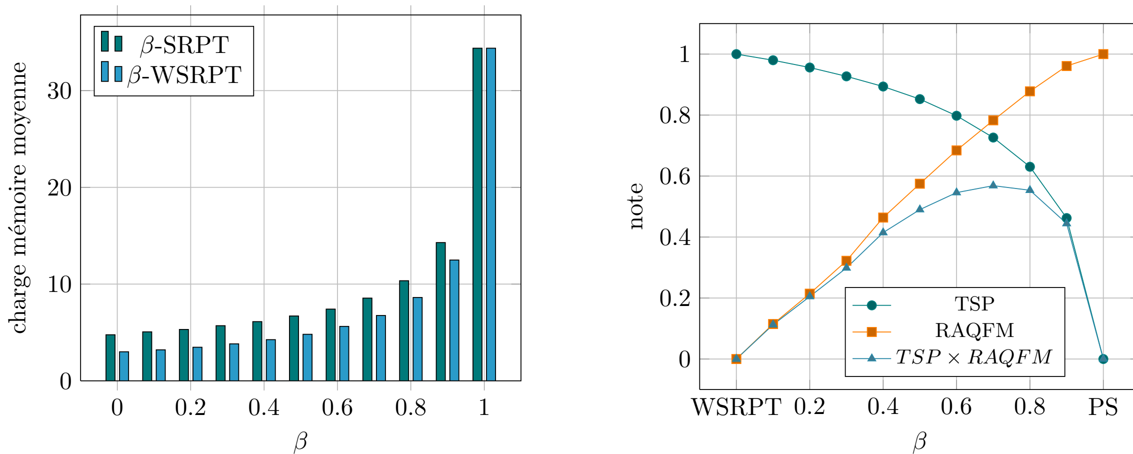
L'algorithme β-WSRPT

Nous proposons d'adapter l'algorithme β-SRPT afin d'y considérer la consommation mémoire des tâches. Le nouvel algorithme, β-WSRPT, se situe alors entre PS et WSRPT. Comme β-SRPT, l'algorithme est fondé sur un PS généralisé avec des poids dépendant de β. Au lieu de trier les tâches par ordre croissant de temps restant, β-WSRPT réalise un tri par ordre croissant de rapport $\frac{R_j}{m_j}$. Les poids sont ensuite calculés avec les mêmes formules que dans β-SRPT, si bien que la tâche $j + 1^{eme}$ est servie β fois moins rapidement que la j^{eme}.

En pratique, le poids m_j de chaque requête n'est pas nécessairement connu *a priori*. Il est toutefois possible, comme pour la taille des requêtes, de réaliser des estimations fondées sur les exécutions précédentes.

Résultats de simulation

Nous évaluons les résultats de β-WSRPT en utilisant le même simulateur que dans le cas non pondéré. Le nouveau paramètre m_j suit une distribution de Pareto de paramètre $\alpha = 1,1$ identique à celle de la taille des tâches (les lois de puissance sont généralement appropriées à la



(a) Temps de séjour pondéré moyen avec β-SRPT et β-WSRPT. Charge globale $\rho = 1$

(b) Compromis entre charge mémoire et équité instantanée avec β-WSRPT. Charge globale $\rho = 0,9$

FIGURE 5.13 – Bénéfices et compromis permis par β-WSRPT. TSP : temps de séjour pondéré

modélisation de phénomènes résultants d'actions humaines). Au lieu de considérer le temps de séjour des tâches, on s'intéresse à leur temps de séjour pondéré, qui correspond dans notre cas à leur charge mémoire individuelle. Ainsi, en minimisant le temps de séjour pondéré moyen de tâches, on minimise la charge mémoire globale du serveur C. La figure 5.13(a) montre la charge mémoire moyenne obtenue avec β-SRPT et β-WSRPT pour différentes valeurs de β.

L'écart entre les deux algorithmes est d'autant moins important que β est proche de 1, cas dans lequel les deux algorithmes équivalent à PS. Dans le cas où $\beta = 0$, β-WSRPT a une charge mémoire 10 fois moins importante qu'avec PS et réduite de 37 % en comparaison à β-SRPT. Une telle économie de mémoire peut être utilisée par le noyau pour faciliter le passage à l'échelle ou pour augmenter la taille des tampons utilisée, donc augmenter le débit espéré.

Nous montrons en figure 5.13(b) le compromis entre charge mémoire et équité instantanée en fonction de β, avec des notes normalisées entre 0 et 1, pour une charge globale $\rho = 0,9$. L'équilibre entre les deux courbes est obtenu pour une note de 0,7 et le produit des notes dépasse 0,5. En positionnant de manière appropriée la valeur de β, on peut doser efficacement la charge mémoire du serveur et l'équité instantanée des connexions.

Remarques et synthèse

Les résultats de simulation que nous avons obtenus montrent que les algorithmes β-SRPT et β-WSRPT permettent un compromis intéressant entre temps de séjour (ou charge mémoire dans le cas pondéré) et équité instantanée (qui assure un partage des ressources efficace et considère les états intermédiaires des requêtes). Dans notre système d'exploitation dédié, ces algorithmes prennent place dans le gestionnaire d'évènements. Alliés au gestionnaire de services à base de coroutines à pile partagée, ils permettent une gestion efficace des applications et un contrôle fin de la charge mémoire.

Les résultats de simulation gagneraient toutefois à être complétés par des expérimentations réelles. En effet, plusieurs facteurs ne sont pas pris en compte dans le simulateur. Par exemple, l'abandon de requêtes n'est pas pris en compte. Dans un système réel, lorsque la quantité

de mémoire disponible sur le serveur est insuffisante, certaines requêtes sont défaussées. Les variations de charge du réseau, les pertes de paquets et les coûts non prédictibles du système ne sont également pas pris en compte par ces simulations.

Le choix de la valeur optimale du paramètre β dépend des exigences et des paramètres de l'environnement dans lequel le serveur Web est déployé. Par exemple, comme le montre la figure 5.12, le compromis obtenu en fonction de β dépend de la charge du système. Il serait intéressant d'adapter dynamiquement le paramètre β lors de l'exécution afin d'assurer une qualité de service dépendante des performances pures et/ou du partage des ressources. Par exemple, on pourrait imposer un seuil minimal sur l'équité instantanée, laissant l'algorithme choisir la valeur appropriée de β, qui dépend du trafic entrant et des propriétés du réseau.

5.4 Conclusion

Dans ce chapitre, nous avons montré qu'un macro-noyau permettait la conception de techniques nouvelles en vue d'améliorer les performances du système et d'en minimiser la consommation de ressources. La construction évènementielle allant des couches basses du logiciel au support applicatif de haut niveau permet d'envisager des optimisations transversales particulièrement efficaces [DGV-Icess09, DGV-Imis09]. En réalisant un grand nombre de pré-traitements sur les applications avant leur déploiement, on allège considérablement la charge de la pile de communication lors de l'exécution.

Nous avons ensuite proposé une solution pour le support des stratégies décrites dans le chapitre précédent. À l'aide de coroutines à pile partagées, le gestionnaire de services est capable de supporter plusieurs fils d'exécution applicatifs et d'en stocker de multiples continuations en exploitant efficacement la mémoire disponible. On permet alors la ré-invocation des routines, nécessaires à l'émission efficace de données volatiles ou idempotentes, tout en manipulant un code applicatif non outillé, dont l'implémentation est donc rendue plus accessible.

Finalement, nous avons discuté de l'ordonnancement des requêtes, qui, dans notre macro-noyau, peut être réalisé en prenant en compte les informations applicatives au sein du moteur d'évènements. Couplées à la technique des coroutines à pile partagée, ces politiques permettent de doser précisément le temps de séjour des requêtes, leur charge mémoire et l'équité instantanée des connexions, garantissant un bon usage des liaisons réseau utilisées et des ressources disponibles [DG-Mascots10].

Dans le prochain chapitre, nous nous intéressons au système d'exploitation dédié dans son ensemble, constituant un serveur du Web des objets. Nous discutons tout d'abord de l'implémentation réelle d'un tel système, en termes de portabilité, de modularité et d'intégration des applications, nécessitant une chaîne de compilation dédiée. Nous présentons alors les résultats d'un large jeu d'expérimentations visant à mesurer et caractériser les bénéfices de notre approche en comparaison aux solutions de l'état de l'art. Empreinte mémoire, performances, charge mémoire et passage à l'échelle sont discutés.

Sixième chapitre

Mise à l'épreuve expérimentale

Dans ce chapitre, nous mettons à l'épreuve expérimentalement l'architecture de système d'exploitation à macro-noyau dédié qui constitue le cœur de ce mémoire. Nous prouvons l'applicabilité, au regard des caractéristiques des matériels visés, de l'approche présentée et des optimisations qui l'accompagnent. Dans un premier temps, nous nous intéressons à notre prototype, Smews. Nous en décrivons la structure, la chaîne de compilation associée et l'interface qu'il offre aux applications. Nous comparons ensuite les résultats obtenus par notre prototype aux solutions de l'état de l'art. L'objectif est de montrer que l'effort nécessaire à la conception d'un tel noyau est compensé par l'amélioration de ses performances et la réduction de sa consommation de ressources. Enfin, nous nous intéressons au cas exigeant de la notification d'évènements, et évaluons le comportement des différentes techniques d'interaction lorsqu'il est nécessaire de passer à l'échelle.

6.1 Smews : un macro-noyau pour un micro-serveur

La structure d'un système à macro-noyau impose, par construction, de développer un système d'exploitation dédié pour chaque famille d'applications visée. Nous nous intéressons ici à la conception d'un prototype de système dédié au support d'applications Web suivant l'architecture, l'interface applicative et les optimisations décrites dans les chapitres précédents.

6.1.1 Présentation de Smews

Notre prototype porte le nom de Smews, pour *Smart and Mobile Embedded Web Server*. Pour en garantir la portabilité sur des cibles matérielles diverses, il est écrit en langage C et le code source de son noyau ne dépend pas de l'architecture sur laquelle il doit être déployé. Chaque portage vers une cible matérielle particulière est constitué d'un répertoire définissant un ensemble de macros et de fonctions requises par le noyau. Dans sa version actuelle, Smews a été porté sur des matériels variés tels qu'une carte à puce à la configuration très modeste ou des capteurs communicants. Leurs caractéristiques techniques ont été décrites en sous-section 3.2.1, page 42. Smews est un logiciel libre, diffusé sous une licence CeCILL. Il est donc publiquement accessible [1], si bien que les expérimentations conduites dans ce chapitre sont toutes reproductibles.

1. Code source de Smews, incluant portages et applications, disponible à : `http://smews.gforge.inria.fr`

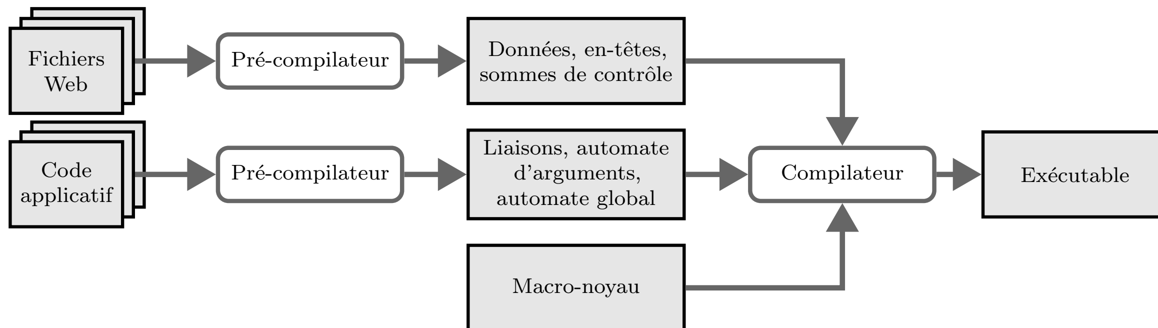


Figure 6.1 – Chaîne de compilation de Smews

Smews se fonde sur une chaîne de compilation inspirée de celles utilisées par les serveurs MiniWeb, iPic ou webAce, décrite en sous-section 2.2.3, page 26. Cependant, au lieu de traiter de simples pages Web, le pré-compilateur prend en charge des applications Web à part entière, telles que nous les avons décrites dans le chapitre 4. À partir d'un répertoire représentant une application Web, il génère du code C qui sera ensuite compilé puis lié avec le noyau, comme l'illustre la figure 6.1. Cela permet à Smews de supporter les optimisations fondées sur le traitement hors-ligne des applications. Lorsqu'il s'agit de fichiers statiques, le pré-compilateur se charge de calculer les sommes de contrôle par blocs et de générer l'en-tête de la réponse HTTP. Lorsqu'il s'agit de contenus dynamiques, il se charge de lier les fonctions de rappel entre elles, de gérer les canaux de notification d'évènements et de générer les structures contenant les méta-données applicatives (persistance des données, schéma d'interaction, ...), qui seront déployées en mémoire non volatile. C'est également ce pré-compilateur qui est en charge de générer les automates de décodage d'arguments et de chemins d'URLs, eux aussi stockés en mémoire persistante.

Le noyau de notre prototype gère un allocateur de mémoire, ce qui permet d'adapter la consommation mémoire du système à son activité réelle. Cet allocateur est basique et n'optimise pas le placement des allocations ; ce qui minimise la taille de son code. Ainsi, les connexions, données associées aux segments en vol ou tampons d'émission sont tous alloués dynamiquement. Smews implémente toutes les optimisations que nous avons décrites dans ce mémoire, à l'exception des politiques d'ordonnancement (le tourniquet est utilisé, ce qui correspond à la politique PS) et du choix dynamique entre stratégie S2/S4 ou S3/S5 pour les contenus idempotents. Ces contenus sont constamment pris en charge avec S5, c'est-à-dire comme des contenus volatils, ce qui permet de gérer un grand nombre de segments en vol. La prise en charge d'une ressource idempotente avec S2/S4 n'est en effet intéressante que dans le cas où la génération des données est très consommatrice de ressources et où les pertes de paquets sont fréquentes. Notons enfin qu'il ne supporte que les requêtes HTTP de type *get*. Comme nous l'avons discuté dans le chapitre précédent, en sous-section 5.1.4, le support des autres commandes HTTP est possible en suivant les mêmes techniques, qui pourraient toutefois être améliorées pour plus d'efficacité dans les situations ou les données reçues par le serveur sont volumineuses et destinées à un stockage en mémoire non volatile.

6.1.2 Interface applicative

Dans Smews, une application est constituée d'un ensemble de fichiers, comprenant des ressources Web statiques ou du code source applicatif écrit en langage C. Les fichiers de code ne sont pas nécessairement associés à des ressources qui seront accessibles sur le serveur ; ils peuvent réaliser divers traitements applicatifs (bibliothèque utilitaire ou code associé à un timer périodique). Les fichiers source associés à une ressource Web contiennent en commentaire un ensemble de méta-données applicatives au format XML. L'interface applicative de Smews est détaillée en annexe B, comprenant la liste des balises XML et des fonctions de rappel disponibles ainsi que les appels système fournis. On se contente ici d'expliquer un exemple de code, présenté en figure 6.2. Il est extrait d'une application Web destinée à un capteur, en charge de notifier les clients du fait que la température ambiante a dépassé un seuil.

Dans l'en-tête XML, les fonctions de rappel `init`, `initGet` et `doGet` sont renseignées, afin de cibler trois fonctions définies dans la suite du fichier. La ressource, d'identifiant `tempAlert`, est déclarée *volatile*, car les données qu'elle génère contiennent une information périssable, contenant la dernière température ayant dépassé le seuil contrôlé. Le schéma d'interaction est de type *alert*, correspondant à une émission utilisant Comet avec un *polling* long. Enfin, le format des arguments d'URL est décrit. Ici, deux arguments de nom `minThres` et `maxThres` de type entier sont attendus. Un client pourra par exemple accéder à cette ressource en ciblant l'URL `/tempAlert?minThres=20&maxThres=140`. À partir de ces informations, la chaîne de compilation génère l'automate de décodage HTTP. Elle déclare également la structure `struct args_t` comme une transposition exacte des arguments décrits, visible localement par chaque fichier source.

Lors de l'initialisation (qui a lieu au moment du déploiement), le convertisseur analogique-numérique (ADC) est configuré, puis on arme une alerte périodique, demandant au noyau d'appeler la fonction `check_temp()` tous les 200 ms. Lors de la réception d'une nouvelle requête, la fonction `set_thresholds()` se charge de positionner la valeur de seuil passée en argument d'URL. Cette valeur est directement accessible comme un champ de la structure `struct args_t`, portant le nom défini lors de la description des arguments. La fonction `check_temp()` compare périodiquement la température ambiante aux seuils courants. Lorsqu'une alerte est nécessaire, elle déclenche le canal `tempAlert` à l'aide de la routine système `trigger_channel()`. Enfin, la fonction `send_temperature()`, appelée par le gestionnaire de services après la levée de l'alerte, se contente de générer une réponse contenant la dernière température ayant dépassé le seuil.

6.2 Consommation mémoire et performances

À présent, nous nous intéressons aux bénéfices de notre approche dans son ensemble, en comparaison à deux solutions de l'état de l'art, la pile généraliste uIP et le serveur MiniWeb. La comparaison avec la pile uIP permet de mesurer les bénéfices de l'intégration de HTTP et du conteneur d'applications au sein du noyau. uIP [Dun03] est ici le représentant des piles IP embarquées génériques telles que CMX MicroNet [Sys02] ou TI MSP430 TCP/IP [Ins01]. Le serveur MiniWeb [Dun05], qui suit une architecture totalement intégrée, sert quant à lui de borne inférieure de référence ; il s'agit d'un prototype aux fonctionnalités extrêmement restreintes, simplement capable de servir quelques pages Web, mais consommant très peu de ressources. Il est le représentant des serveurs Web embarqués intégrés tels que webACE [Whi99] ou iPic [Shr02].

```
/*
<generator>
        <handlers init="init_adc_timer"
              initGet="set_threshold" doGet="send_temperature"/>
        <properties name="tempAlert"
              persistence="volatile" interaction="alert"/>
        <args>
              <arg name="minThres" type="uint16" />
              <arg name="maxThres" type="uint16" />
        </args>
</generator>
*/

static uint16_t min_thes = 128;
static uint16_t max_thres = 896;
static uint16_t curr_sample;

/* fonction de rappel associee au timer,
realise une verification periodique de la temperature */
static void check_temp() {
        uint16_t tmp_result = get_adc_val(ADC_TEMP);
        if(tmp_result < min_thres || tmp_result > max_thres) {
              curr_sample = tmp_result;
              trigger_channel(&tempAlert);
        }
}

/* fonction d'initialisation, configure l'ADC et le timer */
static char init_adc_timer(void) {
        /* retourne 1 en cas de succes, 0 en cas d'echec */
        return init_adc(ADC_TEMP) && set_timer(&check_temp,200);
}

/* fonction de rappel associee a l'arrivee d'une requete, initialise
les seuils a partir de la structure args_t genere a partir de la
balise XML <args>, qui contient deux champs minThres et maxThres */
static char set_thresholds(struct args_t *args) {
        if(args != NULL) {
              min_thres = args->minThres;
              max_thres = args->maxThres;
              return 1; /* succes */
        } else return 0; /* echec */
}

/* fonction de rappel associee au service d'une reponse,
realise l'emission de la mesure ayant depasse les seuils */
static char send_temperature(struct args_t *args) {
        out_uint(curr_sample);
        return 1; /* succes */
}
```

FIGURE 6.2 – Exemple de code d'une ressource Web contrôlant la température

	mémoire volatile (octets)			mémoire persistante (octets)		
Seveur	pile	données	total vol.	code	données	total pers.
Microprocesseur msp430 16 bits						
MiniWeb	36	62	**98**	3 k	710	**3,7 k**
Smews	100	174	**274**	7,8 k	240	**8,1 k**
uIP	156	834	**990**	10,8 k	1,2 k	**11,2 k**
Microprocesseur AVR 8 bits						
MiniWeb	52	52	**104**	3,7 k	696	**4,3 k**
Smews	118	172	**274**	9,7 k	240	**9,9 k**
uIP	184	803	**987**	12,4 k	908 k	**13,3 k**

Table 6.1 – Comparaison des empreintes mémoire de uIP, MiniWeb et Smews

6.2.1 Empreinte Mémoire

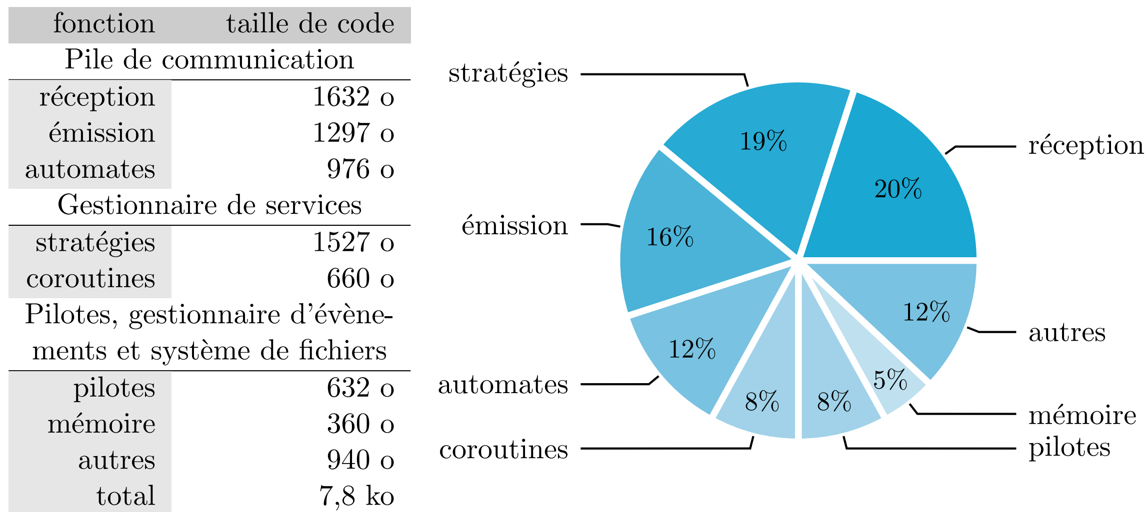
Dans les systèmes que nous visons, la mémoire est une des ressources les plus contraintes, qu'il s'agisse de mémoire volatile (typiquement de la RAM, utilisée comme mémoire de travail) ou non volatile (typiquement une ROM ou EEPROM, contenant le code exécutable ou d'autres données persistantes). La consommation mémoire dépend du langage machine utilisé, donc du type de processeur sur lequel on déploie le logiciel.

Comparaison à l'état de l'art

Nous nous focalisons sur l'empreinte mémoire de notre prototype, Smews, en comparaison à MiniWeb et à uIP accompagné de son serveur Web. Nous avons porté les trois logiciels sur les capteurs WSN430 et MicaZ, utilisant respectivement un processeur msp430 16 bits et AVR 8 bits. Les communications se font *via* une liaison série, prise en charge par des pilotes similaires dans les trois cas. Les trois solutions sont utilisées dans leur configuration consommant un minimum de mémoire. La pile uIP est compilée sans le support d'UDP, le routage IP ni le ré-assemblage de segments, elle utilise un tampon de données de seulement 200 octets. Smews et MiniWeb sont quant à eux capables de travailler sur de grands segments même avec des tampons de très petite taille (ici de 4 octets), car ils traitent les données en flux. Toutes les optimisations définies dans les chapitres précédents sont activées dans Smews. Afin de connaître précisément la consommation de mémoire vive totale, nous avons mesuré la taille maximale de la pile pendant l'exécution du logiciel. Cette mesure est réalisée en remplissant à l'initialisation toute la mémoire de la pile avec un marqueur, puis en recherchant la présence du marqueur après l'exécution. La table 6.1 synthétise l'empreinte mémoire des trois solutions, en distinguant la mémoire volatile de la mémoire persistante.

En ce qui concerne la mémoire volatile, MiniWeb et Smews nécessitent de 98 à 274 octets, alors que uIP consomme près d'1 kilo-octet. Si on veut permettre à uIP de gérer des segments de taille classique, c'est-à-dire de 1460 octets, sa consommation mémoire augmente significativement, pour atteindre 3,3 ko sur msp430 et 3.2 ko sur AVR [2]. En consommant jusqu'à 12 fois plus de mémoire vive que Smews, uIP et son serveur monopolisent une part importante de la mémoire disponible sur les équipements que nous visons, ce qui restreint directement la mémoire disponible pour l'exécution des applications Web. Sur une cible telle que la Funcard 7, disposant de 512 octets de RAM, le déploiement de uIP n'est pas envisageable.

2. Dans uIP, le tampon global partagé et celui des pilotes doivent tous deux supporter un segment entier



fonction	taille de code
Pile de communication	
réception	1632 o
émission	1297 o
automates	976 o
Gestionnaire de services	
stratégies	1527 o
coroutines	660 o
Pilotes, gestionnaire d'évènements et système de fichiers	
pilotes	632 o
mémoire	360 o
autres	940 o
total	7,8 ko

Figure 6.3 – Répartition des tailles de code des fonctionnalités de Smews sur msp430

En terme de mémoire persistante, on observe que MiniWeb possède de loin l'empreinte la plus faible. Smews est légèrement plus compact que la pile généraliste uIP et son serveur Web. Pour les trois solutions, la consommation de mémoire persistante est raisonnable en comparaison aux contraintes des cibles matérielles qui nous intéressent, disposant typiquement de quelques kilo-octets ou quelques dizaines de kilo-octets d'EEPROM ou de mémoire Flash. Excepté dans le cas de MiniWeb, qui ne supporte que le service de pages Web, tout espace non occupé par le serveur est à disposition des applications supportées.

Si on observe de plus près la composition du code de uIP, on remarque que le noyau de la pile IP représente 8,9 ko sur msp430, ce qui est supérieur aux 7,8 ko consommés par le noyau de Smews. La plupart des fonctionnalités de uIP sont désactivées ici (absence de routage et de fragmentation IP), alors que Smews offre des optimisations supplémentaires (stratégies d'émission, coroutines à pile partagée, automate de décodage, ...), dont l'empreinte mémoire est discutée plus loin. En termes de taille de TCB il est important de constater que l'intégration de HTTP et du conteneur d'applications au sein du macro-noyau de Smews a été compensée par la spécialisation au service d'applications Web et par la conception transversale.

Composition du code embarqué

À présent nous discutons de la composition du code exécutable constituant notre prototype. Les fonctionnalités que nous distinguons sont orthogonales à l'empilement des couches constituant la pile de communication, en raison de la construction transversale du système étudié. On retrouve donc les différents modules constituant notre architecture, présentée en sous-section 5.1.2, page 71. La figure 6.3 distingue la taille du code requise par les différentes fonctionnalités et optimisations de Smews, sur un processeur 16 bits msp430.

Les optimisations que nous avons introduites représentent une part non négligeable de l'image finale. Les stratégies d'émission adaptées à la nature des données représentent 19 % du code, celle des automates de décodage des données entrantes, 12 %, et celle des coroutines à pile partagée, 8 %. Les parties fondamentales de la pile de communication, c'est-à-dire la réception et l'émission des données, les pilotes, l'allocateur mémoire et quelques fonctionnalités diverses, représentent 4,7 ko, soit 61 % de la totalité du code. C'est ainsi qu'il a été possible

ressource	taille	nature	stratégie et description
index.html	831 o	statique	S1 – squelette de l'application
style.css	443 o	statique	S1 – mise en forme de la page
scripts.js	3,5 ko	statique	S1 – scripts d'interaction AJAX
logo.png	22,3 ko	statique	S1 – logo du carnet d'adresses
check	2 o	volatile	S3 – teste la présence d'un contact, retourne ok ou ko
get	288 o	volatile	S5 – retourne les données d'un contact
extract	4.5 ko	volatile	S5 – retourne l'ensemble du carnet d'adresses
del	2 o	persistant	S2 – supprime un contact, retourne ok ou ko
add	288 o	persistant	S4 – ajoute un contact, retourne les données ajoutées
dummy	4.5 ko	persistant	S4 – retourne l'ensemble du carnet d'adresses

Table 6.2 – Ressources constituant l'application Web utilisée pour les expérimentations

de porter Smews sur une carte à puce Funcard 7, ne disposant que de 8 ko d'EEPROM, en désactivant lors de la compilation certaines optimisations afin de réduire l'empreinte mémoire du logiciel embarqué.

6.2.2 Résultats à l'exécution

On souhaite mesurer les gains en performances et en charge mémoire permis par notre approche. Pour cela, on s'intéresse au comportement de Smews, uIP et MiniWeb après déploiement sur une même cible matérielle, lors du service d'une même application Web.

Application testée

Nous prenons l'exemple d'une application Web permettant la gestion d'un carnet d'adresses personnel. L'application est construite avec la technique AJAX, son chargement commence donc par l'accès à quelques ressources statiques, puis des requêtes asynchrones sont envoyées par le client permettant de manipuler le carnet d'adresses. La table 6.2 décrit chacune des ressources constituant cette application, qui constitue un choix intéressant pour nos expérimentations car elle inclut des contenus de taille et de nature variées, mettant à l'épreuve l'ensemble des stratégies d'émission que nous avons présentées. La ressource `dummy` est insérée artificiellement afin de disposer de contenus persistants de grande taille, qui sont les plus délicats à prendre en charge (S4), mais qui n'ont que peu de cas d'usage concrets.

Cette application est typique du Web des objets, par exemple dans le cas d'un déploiement sur carte à puce. On a alors un accès permanent à ses contacts, qui permet l'accès hors ligne ou l'enrichissement d'applications Web en ligne par l'intermédiaire de *mashup*. Les mesures de performances que nous proposons sont cependant réalisées sur un capteur WSN430, car la carte à puce dont nous disposions – la Funcard 7 – est trop contrainte en mémoire pour exécuter uIP.

Protocole expérimental

Pour réaliser nos mesures, nous connectons un client à ce capteur *via* une liaison série à 115 200 bauds, en utilisant le protocole SLIP [3]. Deux cas sont considérés. Dans le premier, le

3. SLIP : *Serial Line Internet Protocol*

client est connecté directement au capteur, produisant une latence d'en moyenne 6 ms [4]. Dans le second, on introduit une latence artificielle de 50 ms, simulant une situation dans laquelle le client se connecte à distance au serveur embarqué.

Le client est constitué d'un ordinateur personnel classique (dont les caractéristiques matérielles ont un impact négligeable sur ces expériences), utilisant le navigateur Mozilla Firefox 3.5.8 et un noyau Linux 2.6.31. On utilise volontairement un système d'exploitation dont la pile IP n'implémente pas les accusés différés sur les connexions courtes, ce qui pénaliserait de manière démesurée les performances obtenues par la pile uIP, limitée en permanence à un unique segment en vol. Le propos de ces expériences n'est pas de discuter l'adaptation d'une implémentation ou d'une autre au regard de cette politique en particulier, mais de comparer les performances obtenues par les architectures diverses des serveurs Web utilisés. Notons que les résultats obtenus avec Linux sont très proches de ceux obtenus avec Windows en désactivant les accusés différés.

Pour uIP et Smews, on limite le tampon d'émission des données à 200 octets. Lors de l'émission de données statiques, Smews n'utilise pas de tampon en mémoire vive et peut émettre des segments sans limite de taille fixée *a priori*. En pratique, le client négocie une MSS de 1460 octets. En travaillant sur des blocs de données de 16 octets, Smews est alors limité à des segments de 1456 octets (*c.f.* sous-section 5.1.3, page 73). MiniWeb n'est quant à lui capable d'émettre que des contenus statiques, avec une limitation constante à 200 octets.

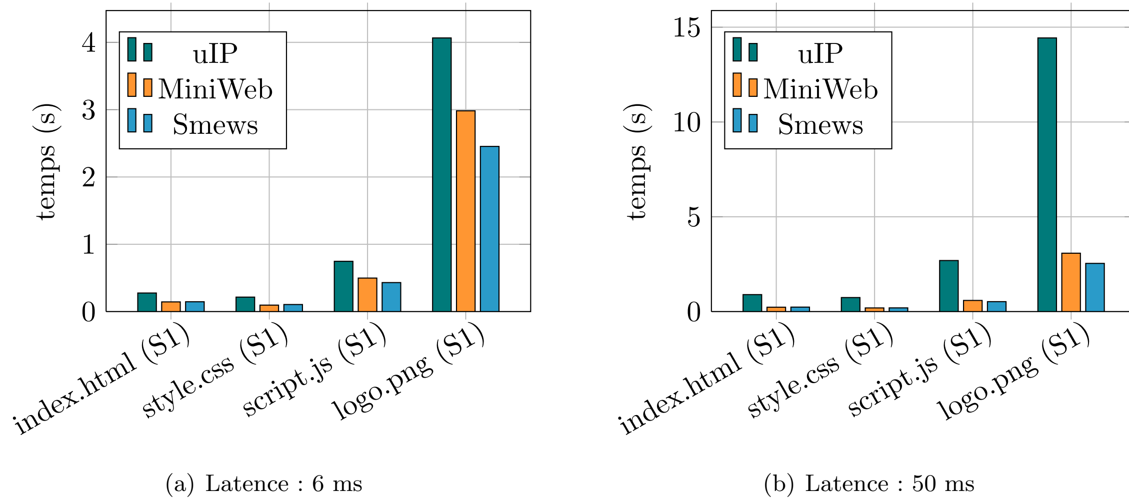
La performance du service d'une ressource est mesurée comme l'intervalle de temps entre le début de l'émission de la requête par le client et la fin de la réception du dernier segment accusant la réception de la réponse entière. Les temps d'ouverture et de fermeture de connexions ne sont pas considérés ici (seul Smews supporte les connexions persistantes). Le navigateur Web est utilisé dans sa configuration d'origine, il émet des requêtes d'une taille comprise entre 400 et 600 octets. Étant donnée la grande régularité des expériences conduites ici, les intervalles de confiance ne sont pas indiqués ; les résultats présentés sont simplement obtenus en moyennant cinq mesures.

Performances

Nous avons mesuré performances obtenues par les trois serveurs lors du service des différentes ressources de l'application. Dans un premier temps, nous nous concentrons sur la durée moyenne d'émission des contenus statiques selon qu'on utilise uIP, MiniWeb ou Smews. C'est ce que présente la figure 6.4, dans le cas d'une latence de 6 ou 50 ms.

Pour toutes les ressources et dans toutes les situations, uIP est plus lent que Smews et MiniWeb. L'écart obtenu est particulièrement significatif (rapport atteignant 5,7 avec Smews) dans le cas où la latence est de 50 ms, puisque uIP est limité à un unique segment en vol à cause de son mécanisme de retransmissions (*c.f.* sous-section 2.2.2, page 25). MiniWeb est la plupart du temps légèrement plus lent que Smews, car il se limite à l'émission de segments de 200 octets, en raison du calcul hors-ligne de tous les segments à émettre et des contraintes de négociation de MSS. Dans la configuration à faible latence et lors du service des fichiers `index.html` et `style.css` (tous inférieurs à un kilo-octet), MiniWeb est légèrement plus rapide que Smews. Il y a deux raisons à cela : MiniWeb construit un en-tête HTTP plus petit que

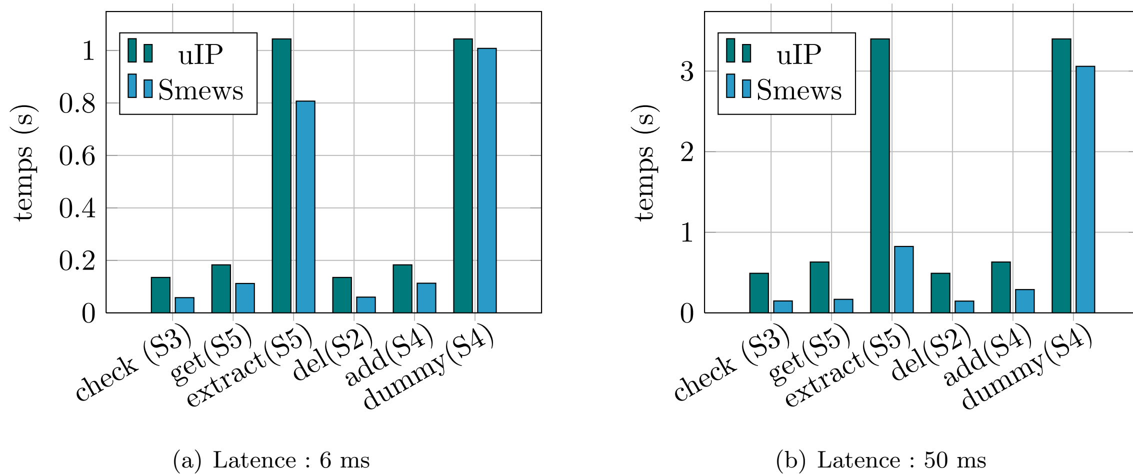
4. La latence est mesurée comme la moitié de l'intervalle de temps entre le début de l'émission d'une donnée et la fin de la réception d'une réponse. Elle inclut, en plus du délai de transmission induit par la connexion physique, certains coûts fixes matériels et logiciels subis par les deux hôtes.



(a) Latence : 6 ms

(b) Latence : 50 ms

FIGURE 6.4 – Comparaison des performances sur le service des contenus statiques



(a) Latence : 6 ms

(b) Latence : 50 ms

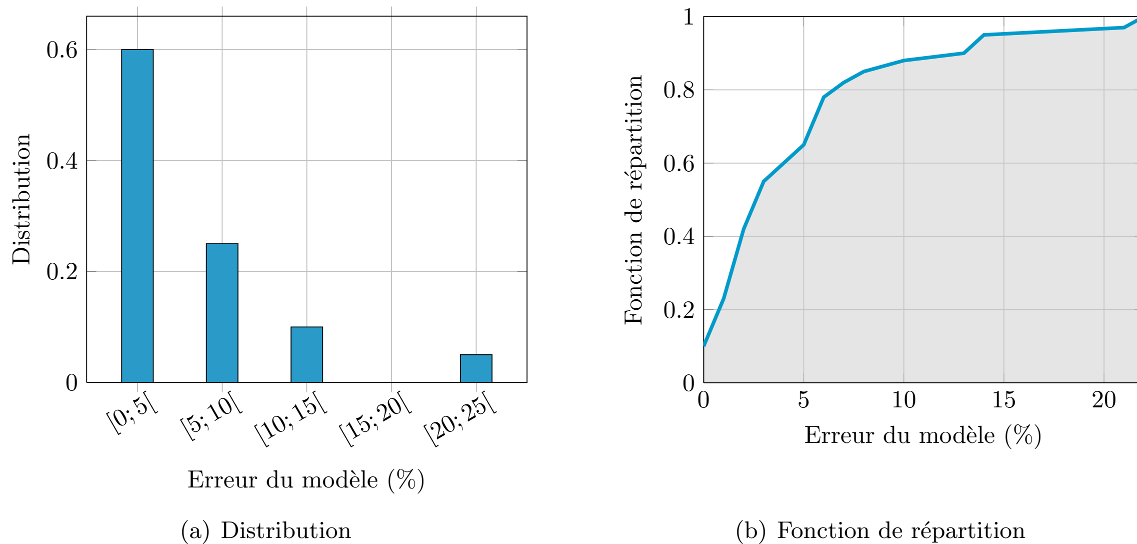
FIGURE 6.5 – Comparaison des performances sur le service des contenus dynamiques

Smews et commence à émettre sa réponse avant même d'avoir décodé la requête entière (avec une MSS de 200 octets, la requête est constituée de plusieurs segments).

Nous comparons dans la figure 6.5 les performances obtenues par uIP et Smews lors du service des contenus dynamiques. MiniWeb est exclu de cette comparaison car il ne supporte que l'émission de fichiers statiques.

Dans le cas de données volatiles, Smews est significativement plus rapide que uIP, car il supporte plusieurs segments en vol (maximum de respectivement 2 et 7 segments de 200 octets dans le cas d'une latence de 6 et 50 ms). De plus, les contraintes des *protothreads* sur lesquels uIP s'appuie incitent son serveur Web à émettre l'en-tête HTTP en plusieurs segments précédant les données de la réponse. Il aurait été tout à fait possible de regrouper ces données mais cela aurait engendré une expansion du code du serveur Web. Dans le cas d'une latence de 50 ms, le rapport de performances entre les deux serveurs atteint 4,1 sur les contenus volatiles.

L'émission de données persistantes se fait de manière comparable dans Smews et uIP,



(a) Distribution

(b) Fonction de répartition

FIGURE 6.6 – Caractérisation de l'écart entre le modèle et les performances mesurées

puisqu'il est alors nécessaire de conserver les données des segments en vol. Le fait d'imposer un tampon d'émission de seulement 200 octets limite Smews à un unique segment en vol, limitation subie par uIP dans le cas général. Les performances obtenues par les deux serveurs sont ici comparables pour les données persistantes, avec toutefois un avantage en faveur de Smews, qui gère plus efficacement l'émission des en-têtes HTTP, nécessitant un nombre moins important d'allers-retours.

Précision du modèle de trafic

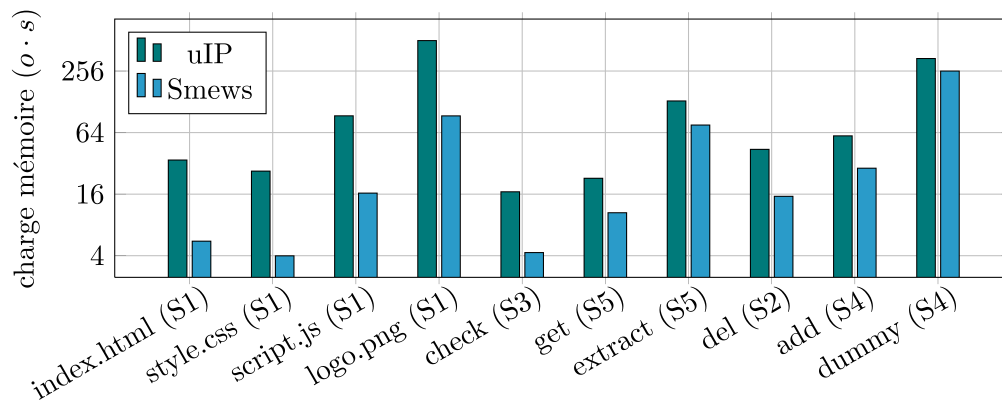
La mesure des performances obtenues avec les trois serveurs nous permet d'estimer la précision du modèle de trafic présenté en section 4.1.1, page 51. Nous avons extrait de nos mesures la partie concernant l'envoi des données et l'avons comparée aux durées d'émission estimées par le modèle. La figure 6.6(a) donne la distribution des erreurs commises. Elle est complétée d'une fonction de répartition par la figure 6.6(b).

L'erreur maximale mesurée est de 22 %. Dans 80 % des cas, elle n'excède pas 7 %, ce qui valide le modèle théorique qui a motivé et permis la construction des stratégies d'émission. Ainsi, nous pouvons être confiants en la qualité des propositions que nous avons élaborées sur la base de ce modèle.

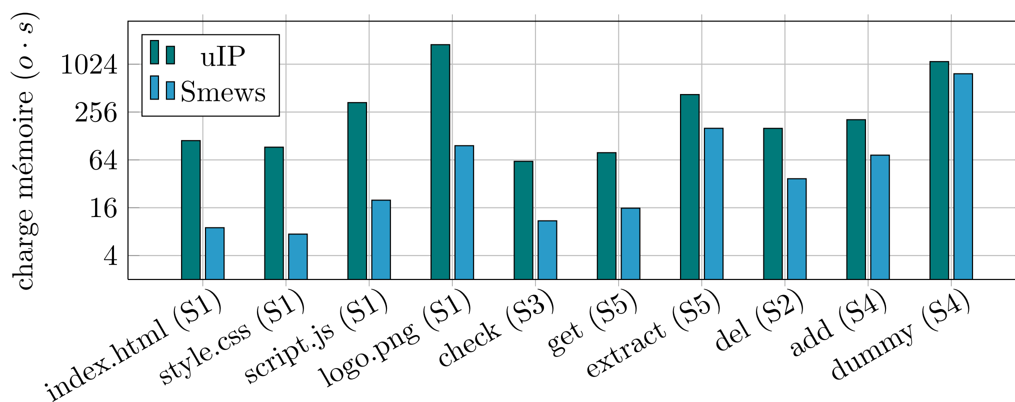
Charge mémoire

Les mesures réalisées, associées à l'analyse du code source du logiciel embarqué, permettent d'estimer la charge mémoire subie par les serveurs lors du traitement de chaque requête. Nous n'incluons pas MiniWeb dans cette analyse, car sa gestion entièrement statique de la mémoire rend non pertinente la notion de charge mémoire. MiniWeb stocke simplement en permanence les données de l'unique connexion éventuellement active.

La première étape, pour cette estimation, a été d'évaluer la quantité de données allouées au traitement de chaque requête. Dans uIP, chaque connexion active requiert 125 octets. Lorsque



(a) Latence : 6 ms



(b) Latence : 50 ms

FIGURE 6.7 – Comparaison de la charge mémoire pour chaque requête

uIP a besoin de retransmettre le segment courant, la reconstruction du dernier segment émis est à la charge de l'application (*c.f.* sous-section 2.2.2, page 25). Dans le cas des contenus persistants, cette dernière doit conserver en permanence les données du segment courant, consommant ainsi 200 octets supplémentaires. Dans Smews, 38 octets sont nécessaires à la gestion de chaque connexion, auxquels on ajoute 16 octets dans le cas où la génération d'un contenu dynamique est en cours. Pour chaque segment en vol, la quantité de mémoire consommée dépend de la nature des données, elle est de 20 octets pour les données volatiles ou idempotentes (stockage d'une continuation) et de 200 octets pour les données persistantes (stockage d'un segment). La figure 6.7 montre une estimation de la charge mémoire imposée à uIP et Smews pour chaque requête. Notons que l'axe des ordonnées suit une échelle logarithmique.

C'est dans le cas de contenus persistants que l'écart entre Smews et uIP est le moins important (rapport d'au moins 1,3) ; il est en fait principalement dû à la différence de consommation de mémoire instantanée. Dans le cas de données volatiles ou statiques, on cumule l'accélération et la réduction de la consommation de mémoire instantanée permises par Smews. Pour les données volatiles, le rapport d'amélioration est compris entre 1,7 et 5,6. Pour les données statiques, il oscille entre 5,5 et 18,7.

La très faible consommation mémoire de Smews permet d'espérer une bonne capacité de

passage à l'échelle, nécessaire dans le Web des objets en raison de l'augmentation permanente du nombre de connexions utilisées par les applications et du nombre potentiellement important de clients connectés simultanément.

6.3 Schémas d'interaction et passage à l'échelle

Nous étudions la capacités de passage à l'échelle permise par les différents schémas d'interaction identifiés dans notre taxinomie, dans le cas particulièrement exigeant de la notification d'évènements. Les trois schémas d'interaction correspondent à trois techniques de notification d'évènements en contexte Web : le *polling* (interrogations répétées), le *polling* long (alerte) et le *streaming* (émission en flux). Ces trois techniques ont été détaillées précédemment en sous-section 2.3.2.

Nous appuyons cette étude sur le travail de Bozdag *et al.* [BMD09], qui ont comparé l'efficacité de techniques et implémentations variées permettant la notification d'évènements depuis des applications Web. Leurs observations mènent à la conclusion que Comet (données poussées par le serveur vers les clients, par opposition au *polling*) est efficace en termes de réactivité, de cohérence des données chez le client et d'usage du réseau. Comet semble cependant causer des problèmes de passage à l'échelle. Les expérimentations conduites dans ces travaux n'imposent pas une charge suffisante pour observer une saturation des ressources du serveur, mais il a été observé que Comet engendrait une consommation processeur sensiblement plus importante que le *polling*. Notons que dans ces travaux, la technique de l'émission en flux n'est pas évaluée, en raison de la rareté des conteneurs d'application la supportant.

Les objectifs de l'étude présentée ici sont multiples. On souhaite tout d'abord décliner le travail de Bozdag *et al.* dans le cas de serveurs embarqués en y intégrant l'émission en flux, afin de caractériser les bénéfices et les inconvénients des différents schémas d'interaction. On souhaite également évaluer l'applicabilité de Comet (alerte et flux) dans le cas de serveurs fortement contraints, et analyser la capacité du système à passer à l'échelle.

6.3.1 Description des expérimentations

Comme dans la section précédente, Smews est utilisé sur un capteur WSN430, connecté à un ordinateur sous Linux par une liaison série à 115 200 bauds. L'application prise en exemple est très simple : à un intervalle de temps donné, le serveur notifie l'ensemble des clients en renseignant l'heure actuelle. L'application est déclinée en trois versions : *polling*, alerte et émission en flux.

Dans le cas du *polling*, le client interroge le serveur à un intervalle donné pour éventuellement récupérer une notification. Dans le cas de l'alerte, la réponse à chaque requête est retardée jusqu'à l'occurrence d'un évènement et est simplement constituée d'une notification. Avec l'émission en flux, chaque client envoie une unique requête pour s'enregistrer auprès du serveur, puis reçoit des notifications consécutives par l'intermédiaire d'une réponse HTTP potentiellement infinie. Dans chaque expérience, un nombre donné de clients se connecte au serveur afin de recevoir les notifications en utilisant tous sur la même technique.

En s'inspirant des travaux de Bozdag *et al.*, nous conduisons un ensemble d'expériences faisant varier les paramètres suivants :

Nombre de clients concurrents Cette variable permet d'évaluer les capacités de passage à l'échelle du serveur. Dans les travaux de Bozdag *et al.*, le nombre de clients est compris

dans l'intervalle [100; 10 000]. En raison des contraintes des matériels que nous visons, le nombre de clients dans nos expérimentations est de 1, 16, 32, 64, 128 ou 256 ;

Intervalle de publication d'évènements Nous utilisons les mêmes valeurs que Bozdag *et al.* : 1, 5, 15, 30 et 50 secondes. Nous avons inséré une sixième configuration où la publication ne se produit plus à intervalles réguliers, mais aléatoirement (durée comprise entre 1 et 50 secondes), afin d'évaluer l'adaptation à la fréquence d'occurrence des évènements ;

Mode applicatif Dans les travaux de Bozdag *et al.*, les différents modes applicatifs testés sont le *polling*, Cometd et DWR. Ces deux derniers sont évalués comme deux implémentations différentes du *polling* long. Il n'existe à notre connaissance aucun serveur Web destiné à l'embarqué supportant Comet. Smews est donc constamment utilisé en tant que serveur Web. Les trois modes applicatifs utilisés sont le *polling*, l'alerte et l'émission en flux ;

Intervalle des requêtes Ce paramètre n'est utilisé que dans le cas de *polling*, il s'agit de la période de temps entre chaque requête émise par un client. Comme Bozdag *et al.*, nous utilisons les valeurs suivantes : 1, 5, 15, 30 et 50 secondes.

La combinaison de ces variables produit 252 configurations expérimentales distinctes. Chaque expérience est réalisée pendant une durée variant entre 5 et 10 minutes et a été exécutée 10 fois. Les valeurs que nous présentons sont une moyenne des résultats obtenus, après exclusion des 4 mesures extrêmes.

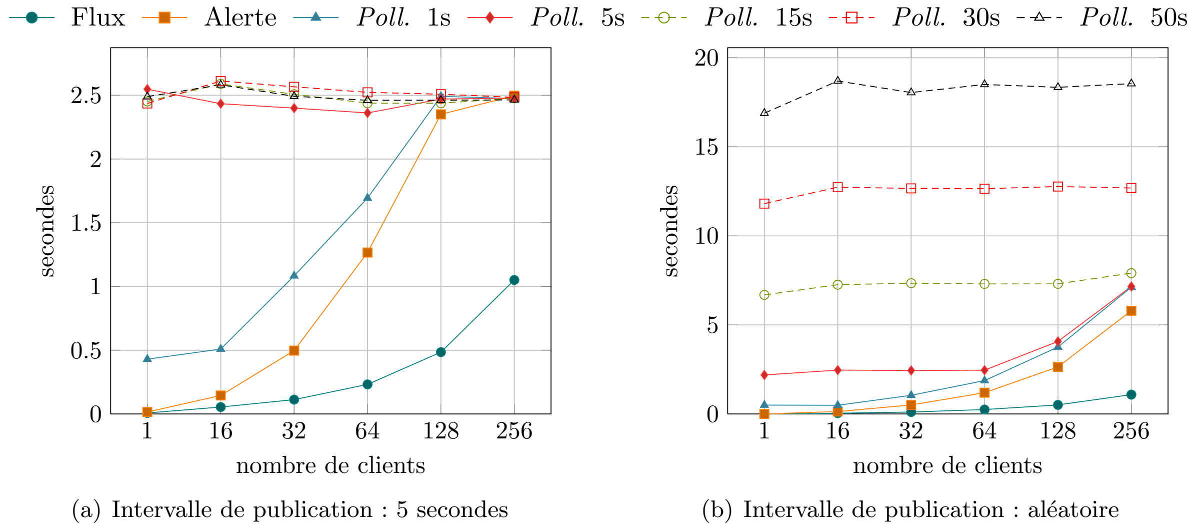
6.3.2 Résultats

Nous évaluons les expériences que nous avons conduites en utilisant les métriques introduites dans les travaux de Bozdag *et al.*. Les résultats exhaustifs sont présentés en annexe C. On se contente ici d'en commenter un sous-ensemble pertinent, métrique par métrique.

LMN – Latence moyenne de notification

La latence moyenne de notification est calculée comme la moyenne des délais séparant l'occurrence d'un évènement sur le serveur de sa notification au client. Dans leurs travaux, Bozdag *et al.* ont montré que la technique de l'alerte permettait d'obtenir une latence moyenne plus courte que le *polling*, quelque soit l'intervalle utilisé. La figure 6.8 montre le LMN que nous avons mesuré, pour un intervalle de publication de 5 secondes ou aléatoirement choisi entre 1 et 50 secondes.

Avec un unique client et un intervalle de publication de 5 secondes, l'alerte et l'émission en flux fournissent tous deux une latence moyenne très faible en comparaison au *polling*. Avec Comet, les notifications sont en effet poussées au client aussi tôt que possible, après que l'évènement ait eu lieu. L'alerte génère plus de trafic que l'émission en flux car elle impose aux clients d'émettre une requête HTTP pour s'enregistrer (ici d'environ 600 octets) entre chaque notification. Avec un intervalle de publication de 5 secondes et avec 64 clients ou plus, l'alerte ne fournit pas un LMN significativement meilleur que le *polling*, car les demandes d'enregistrement des clients causent une saturation du trafic. L'émission en flux fournit la latence la moins élevée dans toutes les configurations, car elle génère un trafic significativement plus léger que l'alerte ou le *polling*. Le flux produit des latences faibles sans problèmes de passage à l'échelle. Par exemple, avec un intervalle de publication de 5 seconde et 128 clients, il maintient une latence de 0,5 secondes contre 2,5 secondes pour l'alerte ou le *polling*.



(a) Intervalle de publication : 5 secondes

(b) Intervalle de publication : aléatoire

FIGURE 6.8 – LMN – Latence moyenne de notification

CPU – Usage moyen du processeur du serveur

Dans leurs travaux, Bozdag *et al.* mesurent l'usage du processeur afin d'estimer la charge du serveur. En analysant la croissance de l'usage du CPU en fonction du nombre de clients, ils concluent que Comet passe moins efficacement à l'échelle que le *polling*. Dans notre configuration, les principaux goulots d'étranglement sont la liaison série et la mémoire disponible. En effet, dans nos expérimentations, la charge imposée au serveur est suffisante pour observer la manière dont ce dernier passe à l'échelle sans conduire le processeur à saturation ; c'est pourquoi nous n'avons pas considéré cette métrique.

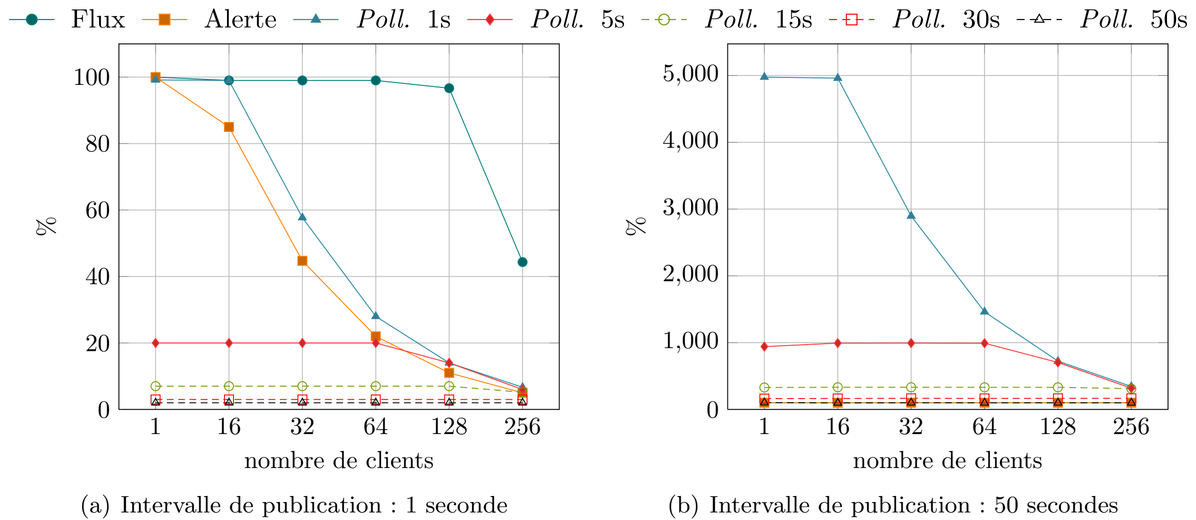
PNR – Pourcentage de notifications reçues

Le PNR représente la quantité moyenne de messages reçus par les clients. Il est représenté comme un pourcentage du nombre d'évènements produits par le serveur. Sa valeur peut dépasser 100 % lorsque les notifications sont reçues plusieurs fois par les mêmes clients. Cela permet de déceler un éventuel excédent de trafic. La figure 6.9 montre l'évolution du PNR.

Avec Comet (alerte comme émission en flux), la valeur de 100 % n'est jamais dépassée, puisque chaque évènement déclenche l'émission d'une notification auprès de tous les clients en écoute (après une unique génération de la réponse HTTP dans un tampon partagé). Dans le cas du *polling*, la quantité de messages reçus croît lorsque l'intervalle entre les requêtes diminue. Avec 16 clients ou moins, un intervalle de *polling* de 1 seconde et un intervalle de publication de 50 secondes, le PNR atteint 5000 % ; en d'autres termes, 98 % des requêtes sont superflues et résultent en l'émission de doublons par le serveur. On constate également une saturation de la liaison lorsque le taux de *polling* est élevé et le nombre de clients important, ce qui se traduit par une diminution importante du nombre de notifications reçues.

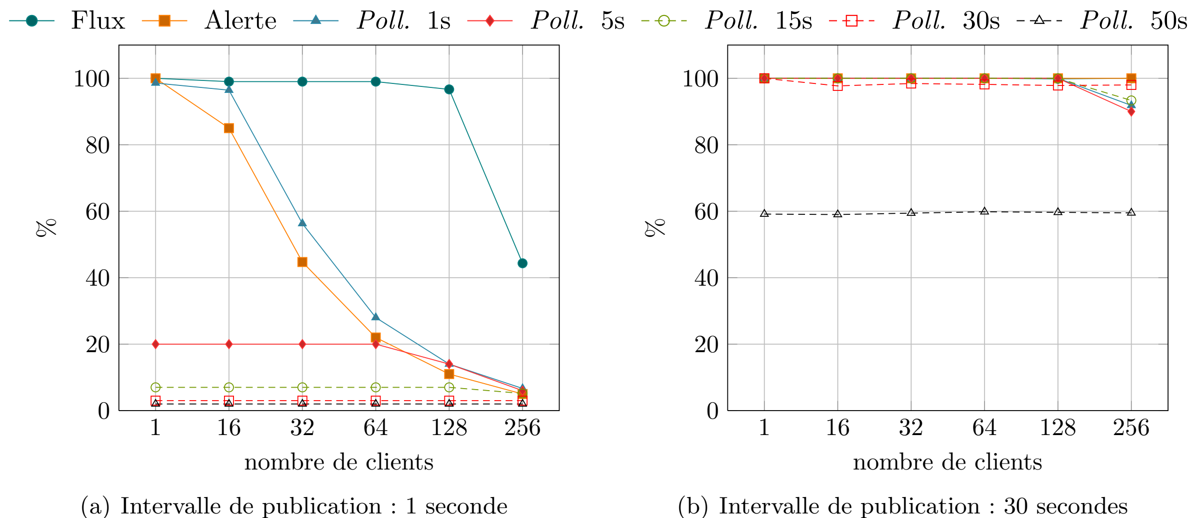
PNUR – Pourcentage de notifications uniques reçues

Le PNUR représente la quantité moyenne de notifications différentes acheminées vers les clients. Contrairement au PNR, il ne peut dépasser la valeur de 100 %. Il permet d'évaluer



(a) Intervalle de publication : 1 seconde (b) Intervalle de publication : 50 secondes

FIGURE 6.9 – PNR – Pourcentage de notifications reçues

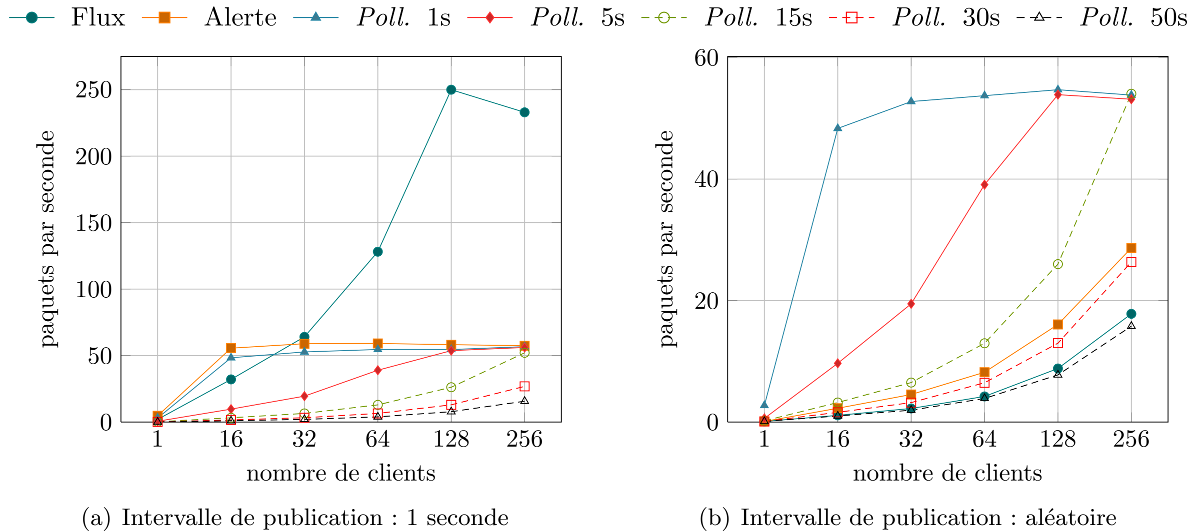


(a) Intervalle de publication : 1 seconde (b) Intervalle de publication : 30 secondes

FIGURE 6.10 – PNUR – Pourcentage de notifications uniques reçues

le nombre de notifications dont les clients ont été informés. Son évolution est montrée par la figure 6.10.

Avec un intervalle de *polling* trop élevé, les clients passent logiquement à coté de certaines notifications. Lorsque le trafic est saturé par un nombre trop important de clients, on constate également une diminution du nombre de notifications reçues. L'alerte fournit un PNUR comparable (en fait, légèrement moins bon) au *polling* utilisant un intervalle égal au taux de publications. Cela s'explique par le fait que l'alerte, comme le *polling*, génère une requête pour chaque notification. Lorsque l'émission en flux est utilisée, le PNUR est proche de 100 % même avec 128 clients et un intervalle de publication de 1 seconde. L'unique configuration dans laquelle le pourcentage de notifications reçues est significativement inférieur à 100 % avec l'émission en flux est celle d'un intervalle de 1 seconde avec 256 clients.



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : aléatoire

FIGURE 6.11 – TPS – Trafic en paquets par seconde

PMR – Pourcentage de messages reçus

Cette métrique est utilisée par Bozdag *et al.* comme un indicateur de pertes de paquets. Dans notre configuration, les communications se font par une liaison série sur laquelle aucune perte de paquets ne se produit. Nous ne nous intéressons donc pas au PMR, constamment égal à 100 %. Néanmoins, lorsque le nombre de données échangées croît, on observe une saturation de la liaison, limitant la qualité du service fourni à chaque client.

TPS – Trafic en paquets par seconde

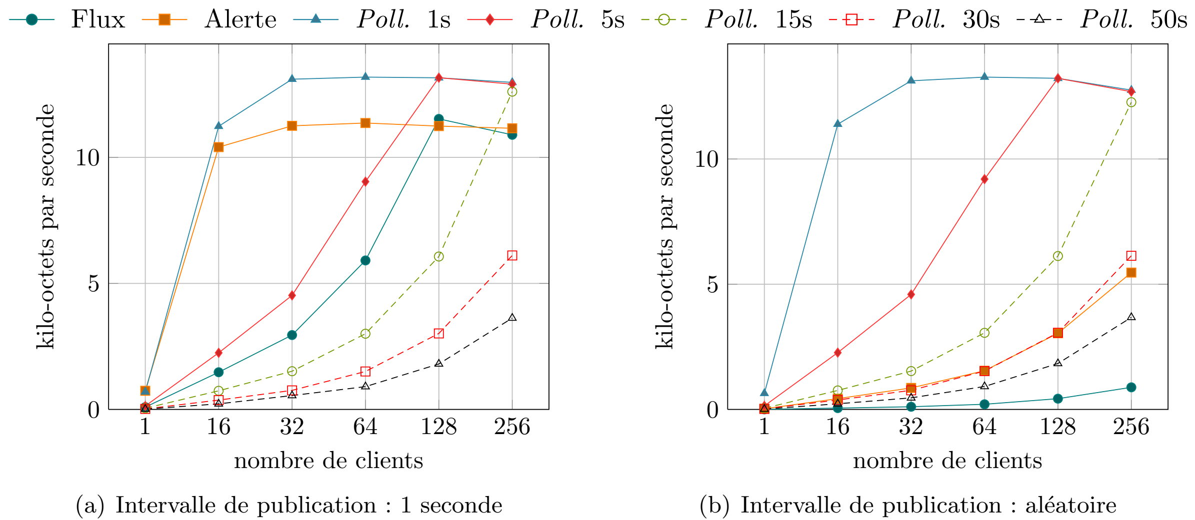
Dans les travaux de Bozdag *et al.*, le nombre de paquets par seconde transitant est utilisé comme indicateur de la charge du réseau. Dans l'optique d'estimer l'usage de la liaison en fonction du mode applicatif, la figure 6.11 montre l'évolution du TPS pour des intervalles de publication de 1 seconde ou aléatoirement choisi entre 1 et 50 secondes.

Avec le *polling* ou l'alerte, la liaison est saturée lorsque le nombre de paquets par seconde dépasse 50. L'émission en flux atteint quant à elle 250 paquets par seconde dans le cas d'un intervalle de publication de 1 seconde et de 128 clients ou plus. Cela s'explique par le fait qu'avec le flux, le trafic est constitué de petits segments TCP, présents en plus grand nombre à débit égal. Cette observation suggère que le TPS n'évalue pas efficacement la charge du réseau.

TDS – Trafic en volume de données par seconde

Afin d'estimer plus précisément la charge du réseau, nous mesurons le volume de données échangées sur la liaison, indépendamment du nombre de paquets. La figure 6.12 montre le TDS mesuré dans nos expérimentations.

On observe ici que le *polling* et l'alerte génèrent un trafic important, à cause du grand nombre de requêtes émises par le client. L'émission en flux permet de réduire fortement le trafic. Par exemple, avec un intervalle de publication aléatoire et 256 clients connectés, l'émission en flux génère un trafic de 0.9 ko/s contre 5.5 ko/s pour l'alerte. La régularité permise par la technique du *polling* permet quant à elle de contrôler parfaitement le trafic produit, qui



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : aléatoire

FIGURE 6.12 – TDS – Trafic en volume de données par seconde

ne dépend que de l'intervalle entre les requêtes et peut être choisi en fonction du degré de cohérence souhaité pour les clients. Par exemple, dans le cas d'un intervalle de publication de 1 seconde, le *polling* avec une période supérieure où égale à 15 secondes produit un trafic plus léger que l'émission en flux.

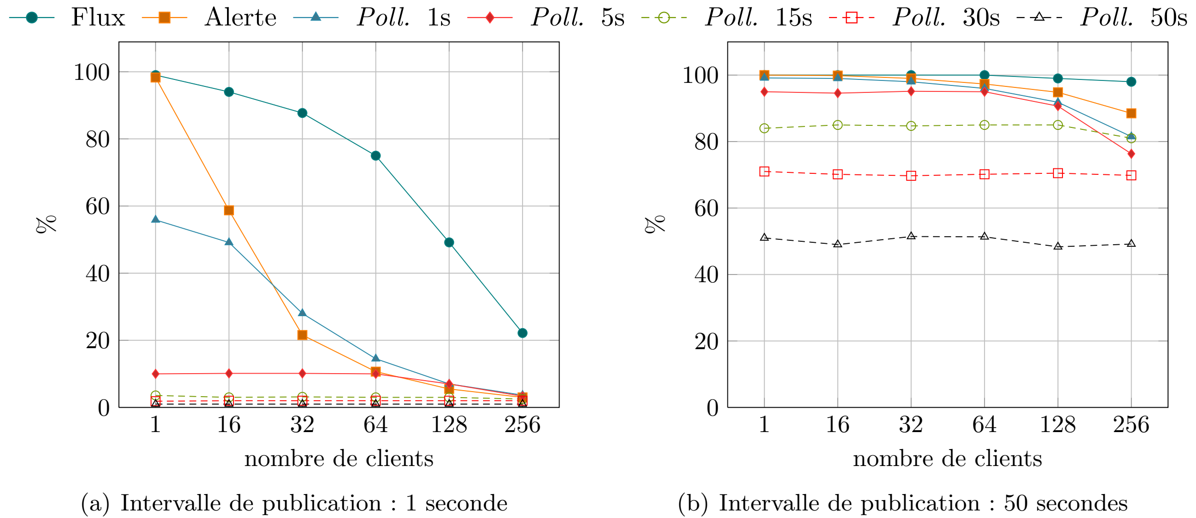
DMC – Degré moyen de cohérence

Nous proposons de synthétiser les résultats obtenus à l'aide d'une nouvelle métrique mesurant le degré de cohérence des données du client. Le DMC est calculé comme la moyenne de la part de temps passé par le client en concordance avec le serveur, c'est-à-dire où le client a reçu la notification du dernier évènement produit. Dans leurs travaux, Bozdag *et al.* évaluent ce degré de cohérence à partir de la latence moyenne de notification (LMN). Cette métrique n'est pas satisfaisante car elle ne considère que les notifications qui atteignent effectivement les clients. La figure 6.13 montre l'évolution du degré de cohérence moyen des clients.

Avec un intervalle de publication de 1 seconde, l'émission en flux permet un degré de cohérence significativement supérieur au *polling* ou à l'alerte. Avec 128 clients, il atteint 50 %, en comparaison des valeurs comprises entre 1 % et 7 % obtenues par le *polling* ou l'alerte. Avec un intervalle de 50 secondes, l'émission en flux conserve un degré de cohérence d'au moins 98 %. Dans le cas d'un *polling* utilisant un intervalle égal à celui des notifications, le degré de cohérence est de 50 %, ce qui s'explique par l'absence de synchronisation entre le serveur et les clients. Pour certains clients, les données sont envoyées juste après l'occurrence d'un évènement ; pour d'autres, elle ne sont envoyées que quelques secondes avant leur expiration.

6.3.3 Synthèse

La notification représente un besoin typique des applications du Web des objets, exigeant de maintenir une bonne réactivité auprès de nombreux clients. Les techniques Comet, avec lesquelles le serveur pousse les données aux clients, sont connues pour être réactives mais souffrant de problèmes de passage à l'échelle. C'est ce que les travaux de Bozdag *et al.* ont caractérisé dans le contexte de serveurs puissants, dont la consommation de ressources augmente



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 50 secondes

FIGURE 6.13 – DMC – Degré moyen de cohérence

plus rapidement avec Comet qu'avec le *polling*. Les contraintes matérielles des serveurs du Web des objets poussent à penser *a priori* que Comet n'est pas adapté aux serveurs embarqués. La spécialisation de notre macro-noyau au support d'applications du Web des objets a cependant permis d'y inclure, dans toutes les étapes de la conception, une gestion efficace de la notification d'évènements.

Différences observées pour l'alerte Nos observations diffèrent sensiblement de celles issues des travaux de Bozdag *et al.*. On remarque que la technique de l'alerte n'est pas très efficace en termes de nombre de notifications reçues et de trafic. Sur ces points, cette technique est légèrement moins pertinente que le *polling* utilisé avec un intervalle égal au taux de publications. Le *polling* est cependant moins efficace en termes de délais de notification et de degré de cohérence des clients. L'alerte est donc à réserver aux situations dans lesquelles l'intervalle de publication est irrégulier et où il est nécessaire de minimiser les latences de notification. Dans le cas contraire, on lui préférera le *polling*, plus léger en termes de traitements (absence d'enregistrement et de recherche des connexions à notifier) et de mémoire consommée sur le serveur (possibilité de ne pas maintenir les connexions ouvertes entre les notifications). Cette distinction avec les travaux en contexte non embarqué est principalement due à la différence des caractéristiques matérielles. Dans notre contexte, la liaison réseau à très faible débit constitue le principal goulot d'étranglement des performances.

Bénéfices de l'émission en flux Contrairement aux travaux de l'état de l'art, nous avons été en mesure de comparer l'émission en flux à l'alerte et au *polling*. Nous avons observé une domination de l'émission en flux sur la quasi-totalité des métriques et dans les 252 configurations expérimentales utilisées. Le flux nécessite très peu de paquets, génère des segments de petite taille et impose une faible charge au serveur, qui ne reçoit qu'un nombre très limité de requêtes HTTP. Après l'enregistrement initial des clients, tous les traitements sont particulièrement rapides et les émissions réactives. L'émission en flux domine largement les autres techniques sur la latence, sur le nombre de notifications reçues, l'absence de redondance, la cohérence des données du client et la capacité de passage à l'échelle.

Le contrôle offert par le polling La technique classique du *polling* peut tout de même se révéler plus pertinente que l'émission en flux dans certaines situations. Elle fournit un contrôle fin du trafic, ne dépendant que du degré de cohérence exigé sur les clients et non de la fréquence d'occurrence des évènements. Ainsi, si une faible cohérence des données est tolérable et si on a besoin de réduire au minimum le trafic produit par les notifications (pour consacrer le trafic à d'autres applications ou réduire la consommation d'énergie du serveur), il est pertinent d'utiliser le *polling* avec un intervalle de temps élevé entre chaque requête.

L'applicabilité de Comet à l'embarqué Notre prototype a été en mesure de supporter efficacement jusqu'à 256 clients avec seulement 10 kilo-octets de RAM et un processeur 16 bits à 8 MHz, ce qui est particulièrement intéressant dans le cas de Comet où les connexions simultanément ouvertes sont potentiellement nombreuses. Cette faible consommation de ressources s'explique par la spécialisation de la pile de communication au support d'applications Web. Ainsi, Smews gère des connexions dédiées, connaît statiquement une part des en-têtes HTTP qu'il envoie et, dans le cas de Comet, il partage un unique tampon de données pour les réponses de tous les clients en attente. Avec un intervalle de publication de 15 secondes, il garantit un taux de cohérence de 92 % à 256 clients par l'intermédiaire d'émission en flux.

6.4 Conclusion

En implémentant un prototype fondé sur un macro-noyau dédié, nous avons montré la faisabilité et évalué les bénéfices réels de notre approche. Smews fournit une interface applicative en concordance avec l'analyse des applications proposée en chapitre 4 et supporte les optimisations décrites en chapitre 5. Il montre qu'il est possible de construire un système à macro-noyau dédié portable vers des cibles matérielles variées.

Notre prototype ne nécessite que quelques centaines d'octets de mémoire volatile et quelques kilo-octets de mémoire persistante pour s'exécuter. Sa consommation de ressources est ainsi comprise entre celle d'un logiciel totalement intégré et celle d'une pile IP générique supportant un serveur [DGV-Icess09]. La taille de la TCB de Smews est inférieure à celle de la pile générique uIP (utilisée dans une configuration minimale, sans support UDP, routage ni fragmentation IP). L'intégration des couches logicielles hautes au sein du noyau a donc été compensée par la spécialisation au support d'applications Web et par la conception transversale. Une autre conséquence de cette intégration est qu'elle permet de fournir un service poussé aux applications, comme par exemple la possibilité d'utiliser Comet.

Par construction, un système d'exploitation dédié doit cependant être développé pour chaque famille d'application visée, requérant un effort important. Celui-ci doit être contrebalancé par des gains significatifs sur le comportement du système produit. En comparaison à l'état de l'art, notre prototype permet une amélioration importante des performances et de la charge mémoire [DGV-Emsoft09], atteignant un facteur d'amélioration de respectivement 5,7 et 18,7.

Nous avons finalement évalué les différents schémas d'interaction et leur capacité à passer à l'échelle dans le cas de la notification d'évènements. Cette étude complète les travaux de l'état de l'art réalisés en contexte non embarqué. Nous avons ainsi caractérisé les propriétés de chaque approche et notamment montré l'applicabilité avec efficacité de la technique d'émission en flux, habituellement considérée comme particulièrement coûteuse pour les serveurs. Notre prototype a ainsi été capable de supporter jusqu'à 256 clients avec seulement 10 kilo-octets de mémoire volatile [DGV-Wse09].

Septième chapitre

Conclusion et perspectives

Nous concluons ce mémoire en résumant les contributions de cette thèse, en en discutant les limites puis en ouvrant des perspectives aux travaux présentés.

7.1 Synthèse

Dans cette thèse, nous avons présenté et étudié une approche consistant à dédier un système d'exploitation à une famille d'applications de haut niveau. Nous nous sommes intéressés au cas particulier du serveur d'applications Web en environnement contraint. Ce cas d'étude a d'intéressant qu'il pousse à l'extrême à la fois les exigences applicatives fonctionnelles (les serveurs d'applications Web fournissent un haut niveau d'abstraction) et les contraintes matérielles (les matériels visés ne sont pas de puissants serveurs, mais sont des cartes à puce ou des capteurs). Les bénéfices qui se dégagent de nos travaux sont les suivants :

Efficacité du noyau L'intégration des couches logicielles hautes au sein du macro-noyau, alliée à la spécialisation de l'interface applicative, produit un logiciel efficace en termes de consommation de ressources et en particulier de performances, d'empreinte mémoire et de charge mémoire. De plus, l'extension verticale du noyau est compensée en terme de taille par sa construction transversale et fonctionnellement dédiée ;

Richesse du support applicatif En factorisant un grand nombre de traitements au sein du noyau et en présentant une interface dédiée, il est possible de supporter des fonctionnalités riches, performantes et accessibles. Cette factorisation se ressent également en termes de génie logiciel : l'effort nécessaire à la conception du noyau n'est à fournir qu'une fois mais outille la construction d'un nombre illimité d'applications qui n'ont plus qu'à se concentrer sur des traitements purement fonctionnels.

La contrepartie de l'approche présentée est qu'elle nécessite un effort d'analyse, de conception et de développement pour chaque famille d'application visée. Les qualités du système résultant sont la conséquence d'un travail qui ne peut être automatisé. En effet, il est intéressant de constater que les bénéfices obtenus découlent principalement de changements radicaux d'architecture et d'une analyse dédiée de l'environnement d'exécution. On citera l'interface dérivée de la taxinomie, l'adaptation des stratégies d'émission tirée du modèle de trafic, la gestion en flux des couches hautes du logiciel au sein du noyau, les co-routines ré-invocables à pile partagée ou les politiques d'ordonnancement minimisant la charge mémoire.

7.2 Résumé des contributions et publications

Nous résumons ici les contributions de cette thèse et faisons référence aux publications qui en découlent, détaillées dans la **Bibliographie personnelle**, page 117.

Dans un premier temps, nous nous sommes intéressés à la conception du support applicatif. Nous avons proposé un modèle de trafic et présenté une taxinomie des ressources constituant les applications Web. Nous avons introduit une nouvelle métrique, la *charge mémoire*, capable de synthétiser performances et consommation mémoire. En se fondant sur cette analyse, nous avons proposé d'adapter le comportement d'une pile IP aux propriétés des applications et avons formalisé les gains permis par cette adaptation [DGV-Emsoft09].

Dans un second temps, nous nous sommes focalisés sur les mécanismes internes du macro-noyau dédié. Nous avons proposé un ensemble d'optimisations permises soit par la connaissance globale de la pile protocolaire, soit par un outillage de l'interface entre noyau et applications [DGV-Icess09, DGV-Imis09]. Nous avons ensuite adressé le support de coroutines à faible consommation mémoire et permettant la ré-invocation. Afin d'améliorer le support de ces coroutines, nous avons présenté deux politiques d'ordonnancement dont l'objectif est de paramétrer un compromis entre performances, charge mémoire et équité de service auprès des clients [DG-Mascots10].

Enfin, nous nous sommes intéressés à la construction d'un prototype, Smews, montrant la faisabilité de notre approche. Nos expérimentations évaluent les bénéfices de nos propositions, qui, en comparaison aux solutions de l'état de l'art, améliorent les performances, diminuent la consommation de ressources et étendent les fonctionnalités du système. Nous avons étudié le cas particulièrement exigeant de la notification d'évènements [DGV-Wse09]. Nos résultats montrent que Comet, et en particulier la technique de l'émission en flux, est efficace en contexte embarqué ; il a été possible de gérer plusieurs centaines de clients en ne disposant que de quelques kilo-octets de mémoire vive. Enfin, en validant expérimentalement le modèle de trafic sur lequel repose une part importante de nos propositions, nous avons confirmé la pertinence de ces dernières.

7.3 Perspectives

Les perspectives aux travaux présentés sont doubles. Elles concernent soit l'approche du système d'exploitation à macro-noyau dédié, soit son application au cas du Web des objets.

Problèmes spécifiques aux réseaux *ad hoc* sans fil Les travaux que nous avons présentés se focalisent sur des nœuds reliés directement à une infrastructure. On cible ainsi des cartes à puces, des capteurs ou des systèmes de domotique. Les problèmes spécifiques aux réseaux *ad hoc* multi-saut sans fil tel que des réseaux de capteurs n'ont pas été abordés. Dans ce contexte, les pertes de paquets sont fréquentes et il est nécessaire de supporter un routage efficace en temps, en mémoire et en énergie. L'efficacité de nos stratégies d'émission, optimistes par construction, est alors remise en cause. Un compromis est probablement à réaliser au niveau de la couche de liaison, afin de masquer les pertes de paquets aux couches hautes (à la manière des caches de segments distribués [DVA04]) tout en limitant la consommation de mémoire et d'énergie. D'autres aspects sont susceptibles de soulever de nouvelles questions, comme l'intégration efficace de 6lowPAN [HC08], la gestion des cycles d'activité de la puce radio ou la sécurisation des échanges *via* HTTPS ou IPsec.

Méthodologie de conception d'applications Web Nous avons montré qu'en adaptant le comportement de la pile de communication aux propriétés des applications de haut niveau, il était possible de servir les clients plus efficacement tout en réduisant la consommation mémoire. Ces résultats se fondent sur une taxinomie qui vise à extraire un maximum de propriétés sur chaque ressource constituant les applications Web. Il serait alors pertinent de discuter de la méthodologie de conception d'applications Web, voire de proposer des outils en permettant l'optimisation. Par exemple, les données persistantes sont celles qui sont servies le moins efficacement. On pourrait en réduire la proportion en affinant la granularité du découpage des applications. En isolant les traitements persistants, on augmenterait la part de ressources statiques, idempotentes et volatiles, ce qui accélérerait les performances globales de l'application.

Application aux stations serveur Il est intéressant de ramener les résultats que nous avons obtenus avec Smews en termes de passage à l'échelle aux limites atteintes par les stations serveur lors de l'énoncé du célèbre *C10K problem* [Keg99]. En 1999, Kegel y dénonçait l'incapacité des serveurs équipés de 2 giga-octets de RAM à supporter plus de 10 000 clients. Aujourd'hui, les serveurs de production gérant de fortes charges ont évolué. Ils sont le plus souvent implémentés de manière évènementielle, ce qui leur permet de faire un meilleur usage des ressources. Cependant, leur consommation mémoire par client est toujours de l'ordre de plusieurs dizaines de kilo-octets, contre seulement 38 octets dans Smews. En transposant nos travaux au cas des stations serveur, on pourrait espérer en réduire significativement le coût ou la consommation énergétique. On libérerait également une quantité importante de mémoire vive, qui permettrait une augmentation de la taille des caches de système de fichier. D'autres questions seraient alors à aborder, comme l'adaptation efficace de notre architecture aux processeurs multi-cœurs. L'accélération de systèmes évènementiels sur plusieurs cœurs est un problème qui connait déjà ses premières solutions [GGL+10].

Application à d'autres domaines L'approche présentée dans cette thèse est *a priori* adaptée à tout système supportant une famille d'applications particulière dans un contexte où l'on souhaite minimiser toute consommation de ressources, en termes de performances, de mémoire, d'énergie ou de coût du matériel. Le cas d'étude auquel nous nous sommes intéressés – celui de serveurs du Web des objets – nous a permis de travailler en particulier sur l'efficacité des communications en réseau. Dans ses travaux sur les exo-noyaux [KEG+97], Engler a montré que de nombreux aspects d'un système d'exploitation pouvaient bénéficier d'un support adapté aux traitements applicatifs. C'est par exemple le cas de la gestion de la mémoire virtuelle ou du système de fichiers. On peut espérer une transposition efficace de notre approche à des domaines variés, tels qu'un système voué à l'exécution d'applications java (ou Java Card) ou un serveur de base de données.

Bibliographie personnelle

Conférences internationales

[DGV-Emsoft09] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : Serving embedded content via web applications : model, design and experimentation. *Dans ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, octobre 2009.

[DGV-Wse09] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : Consistency and scalability in event notification for embedded web applications. *Dans 11th IEEE International Symposium on Web Systems Evolution (WSE'09)*, Edmonton, Canada, septembre 2009.

[DGV-Icess09] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : The web of things : interconnecting devices with high usability and performance. *Dans 6th IEEE International Conference on Embedded Software and Systems (ICESS'09)*, HangZhou, Zhejiang, China, mai 2009.

Ateliers et posters internationaux

[DG-Mascots10] Simon Duquennoy et Gilles Grimaud : Efficient web requests scheduling considering resources sharing. *Dans 18th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'2010)*, Miami Beach, Florida, août 2010. Poster paper.

[DGV-Imis09] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : Smews : Smart and mobile embedded web server. *Dans 3rd International Workshop on Intelligent, Mobile and Internet Services in Ubiquitous Computing (IMIS'09)*, Fukuoka, Japan, mars 2009. **Best Paper Award**.

Conférences nationales

[DGV-Cfse09] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : Servir du contenu embarqué via des applications web : modèle, conception et expérimentation. *Dans 7ème Conférence Française en Systèmes d'Exploitation (CFSE'7)*, Toulouse, France, septembre 2009.

[DGV-Jdir08] Simon Duquennoy, Gilles Grimaud et Jean-Jacques Vandewalle : Haute performance pour serveurs web embarqués. *Dans 9èmes Journées Doctorales en Informatique et Réseau (JDIR'08)*, Villeneuve d'Ascq, France, janvier 2008.

Bibliographie

[AAB05] Eitan Altman, Konstantin Avrachenkov, and Chadi Barakat. A stochastic model of tcp/ip with stationary random losses. *IEEE/ACM Trans. Netw.*, 13(2) :356–369, 2005.

[Agr98] I. D. Agranat. Engineering web technologies for embedded applications. *Internet Computing, IEEE*, 2(3) :40–45, May 1998.

[AKK04] J. N. Al-Karaki and A. E. Kamal. Routing techniques in wireless sensor networks : a survey. *IEEE Wireless Communications*, 11(6) :6–28, December 2004.

[All00] AllegroSoft. Rompager embedded web server toolkit, April 2000.

[Apa95] Apache. Apache httpd, 1995. `http://httpd.apache.org/`.

[Bar09] Richard Barry. Using the freertos real time kernel - a practical guide, January 2009. `http://www.freertos.org/`.

[Bas08] Alessandro Bassi. Internet of things in 2020 – roadmap for the future. May 2008.

[BCB03] Rob Von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers), 2003.

[BCD+05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os : An embedded multithreaded operating system for wireless micro sensor platforms. In *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, page 2005, 2005.

[BD07] Nikhil Bansal and Kedar Dhamdhere. Minimizing weighted flow time. *ACM Trans. Algorithms*, 3(4) :39, 2007.

[BD08] Engin Bozdag and Arie van Deursen. An adaptive push/pull algorithm for ajax applications. In *Third International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'08)*, pages 95–100, July 2008.

[BDM98] Gaurav Bangs, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. *SIGMETRICS Perform. Eval. Rev.*, 26(3) :23–30, 1998.

[Bha06] Sapan Bhatia. *Optimisations de Compilateur Optimistes pour les Systèmes Réseaux.* PhD thesis, Université Sciences et Technologies - Bordeaux I, June 2006.

[BHB01] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling : Investigating unfairness. pages 279–290, 2001.

[BM98] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *ATEC '98 : Proceedings of the annual conference*

on USENIX Annual Technical Conference, pages 1–1, Berkeley, CA, USA, 1998. USENIX Association.

[BMD07] Engin Bozdag, Ali Mesbah, and Arie van Deursen. A comparison of push and pull techniques for ajax. In *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE)*, pages 15–22. IEEE Computer Society, 2007.

[BMD09] Engin Bozdag, Ali Mesbah, and Arie van Deursen. Performance testing of data delivery techniques for ajax applications. *Journal of Web Engineering (JWE)*, 2009.

[Bra89] R. Braden. Rfc 1122 : Requirements for internet hosts - communication layers, 1989.

[CB97] Mark E. Crovella and Azer Bestavros. Self-similarity in world wide web traffic : evidence and possible causes. *IEEE/ACM Trans. Netw.*, 5(6) :835–846, 1997.

[CGV10] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine based embedded systems. *ACM Transactions on Embedded Computing Systems*, 2010.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CJ00] M. J. Choi and H. T. Ju. An efficient embedded web server for web-based network element management. *Network Operations and Management Symposium, IEEE/IFIP*, pages 187–200, April 2000.

[CJRS02] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *Communications Magazine, IEEE*, 40(5) :94–101, 2002.

[CKZ01] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *STOC '01 : Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 84–93, New York, NY, USA, 2001. ACM.

[Coo02] G. H. Cooper. Tinytcp, 2002. http ://www.csonline.net/bpaddock/tinytcp/.

[DAV+04] Adam Dunkels, Juan Alonso, Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. Connecting wireless sensornets with tcp/ip networks. In *In Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004*, pages 143–152, 2004.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of LCN '04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[DKP+01] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull : disseminating dynamic web data. In *WWW '01 : Proceedings of the 10th international conference on World Wide Web*, pages 265–274, New York, NY, USA, 2001. ACM.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. Ip header compression, February 1999.

[Doj08a] Dojo. Cometd the scalable comet framework, 2008. http ://cometd.com/.

[Doj08b] Dojo. Dojo fundation bayeux protocol, 2008. `http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html`.

[Dom03] M. Domingues. A simple architecture for embedded web servers. *ICCA'03*, 2003.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads : simplifying event-driven programming of memory-constrained embedded systems. In *Proc. of SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[Dun03] Adam 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.

[Dun05] Adam Dunkels. The proof-of-concept miniweb tcp/ip stack, 2005. http ://www.sics.se/ adam/miniweb/.

[Dun07] Adam Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, Swedish Institute of Computer Science, February 2007.

[DVA04] Adam Dunkels, Thiemo Voigt, and Juan 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, January 2004.

[DWR07] DWR. Direct web remoting, 2007. http ://directwebremoting.org/dwr/reverse-ajax.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel : An operating system architecture for application-level resource management. pages 251–266, 1995.

[Fal03] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03 : Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY, USA, 2003. ACM.

[FH03] Eric J. Friedman and Shane G. Henderson. Fairness and efficiency in web server protocols. *SIGMETRICS Perform. Eval. Rev.*, 31(1) :229–237, 2003.

[FHL99] S. Freyder, D. Helland, and B. Lightner. A $25 web server. *Circuit Cellar*, July 1999.

[Fie00] Roy Thomas Fielding. *REST : Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[For08] UPnP Forum. Upnp forum, 2008. http ://www.upnp.org/.

[Gar05] Jesse J. Garrett. Ajax : A new approach to web applications, 2005.

[GGL+10] Fabien Gaud, Sylvain Geneves, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quema. Efficient workstealing for multicore event-driven systems. *Distributed Computing Systems, International Conference on*, 0 :516–525, 2010.

[Gla06] Glassfish. Glassfish, 2006. `https://glassfish.dev.java.net/`.

[GLLRK79] Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and A. G. H. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *andm*, 5 :287–326, 1979.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language : A holistic approach to networked embedded

systems. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[Gri00] Gilles Grimaud. *Camille : Un Système d'Exploitation Ouvert pour Carte à Microprocesseur*. PhD thesis, Université des Sciences et Technologies de Lille, December 2000.

[GS06] Lin Gu and John A. Stankovic. t-kernel : providing reliable os support to wireless sensor networks. In *SenSys '06 : Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14, New York, NY, USA, 2006. ACM.

[GT09] Dominique Guinard and Vlad Trifa. Towards the web of things : Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009.

[GTPL09] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, June 2009.

[GW03] M. Gong and C. Williamson. Quantifying the properties of srpt scheduling. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, volume 0, pages 126–135, Los Alamitos, CA, USA, October 2003. IEEE Computer Society.

[GW04] Mingwei Gong and Carey Williamson. Simulation evaluation of hybrid srpt scheduling policies. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 355–363, October 2004.

[GW06] Mingwei Gong and Carey Williamson. Revisiting unfairness in web server scheduling. *Comput. Netw.*, 50(13) :2183–2203, 2006.

[HAN05] Ahmad Al Hanbali, Eitan Altman, and Philippe Nain. A survey of tcp over ad hoc networks. *IEEE Communications Surveys and Tutorials*, 7(1-4) :22–36, 2005.

[HBSBA03] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2) :207–233, 2003.

[HC08] Jonathan W. Hui and David E. Culler. 6lowpan : Extending ip to low-power, wireless personal area networks. *Internet Computing, IEEE*, 12(4) :37–45, July-Aug. 2008.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05 : Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[HZW+03] Guang-jie Han, Hai Zhao, Jin-dong Wang, Tao Lin, and Ji-yong 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, 2003.

[Ins01] Texas Instruments. Msp430 internet connectivity. Technical report, Texas Instruments, November 2001.

[Jac90] V. Jacobson. Compressing tcp/ip headers for low-speed serial links, February 1990.

[JCH00] Hong-Taek Ju, Mi-Joung Choi, and James W. Hong. An efficient and lightweight embedded web server for web-based network element management. *Int. J. Netw. Manag.*, 10(5) :261–275, 2000.

[JKN+01] Philippe Joubert, Robert B. King, Richard Neves, Mark Russinovich, and John M. Tracey. High-performance memory-based web servers : Kernel and user-space performance. In *USENIX Annual Technical Conference, General Track*, pages 175–187, 2001.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP '97*, pages 52–65, New York, NY, USA, 1997. ACM Press.

[Keg99] Dan Kegel. The c10k problem, 1999. `http://www.kegel.com/c10k.html`.

[KEGW96] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. In *EW 7 : Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 141–148, New York, NY, USA, 1996. ACM.

[Lab98] Jean J. Labrosse. *Microc/OS-II.* R & D Books, 1998.

[Lie95] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5) :237–250, 1995.

[Lig03] Lighttpd. Lighttpd web server, 2003. `http://www.lighttpd.net/`.

[Lit61] John D. C. Little. A proof for the queuing formula : L= λ w. *Operations Research*, 9(3) :383–387, 1961.

[LLLRK84] J. Labetoulle, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[LM97] T. V. Lakshman and Upamanyu Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3) :336–350, 1997.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos : An operating system for sensor networks. pages 115–148. 2005.

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2) :3–19, 1979.

[Log05] Real Time Logic. Introduction to the barracuda embedded web-server, April 2005.

[LSD04] Dong Lu, Huanyuan Sheng, and P. Dinda. Size-based scheduling policies with inaccurate scheduling information. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 31–38, October 2004.

[LY90] J. Y. T. Leung and G. H. Young. Preemptive scheduling to minimize mean weighted flow time. *Inf. Process. Lett.*, 34(1) :47–50, 1990.

[LZW+04] Tao Lin, Hai Zhao, Jiyong Wang, Guangjie Han, and Jindong Wang. An embedded web server for equipments. *ispan*, 00 :345, 2004.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4BSD operating system.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[Mic09a] Micrium. uc/tcp-ip protocol stack, June 2009.

[Mic09b] Microchip. The microchip tcp/ip stack, July 2009.

[Mor07] Mortbay. Jetty web server, 2007. `http://jetty.mortbay.org/`.

[MSM97] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3) :67–82, 1997.

[NCS93] NCSA. Ncsa httpd, 1993. `http://hoohoo.ncsa.illinois.edu/`.

[Net95] Netscape. An exploration of dynamic documents, 1995.

[Ope03] OpenTCP. Opentcp, January 2003. `http://sourceforge.net/projects/opentcp/`.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Technical Conference*, January 1996.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash : an efficient and portable web server. *Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference table of contents*, pages 15–15, 1999.

[PF94] Vern Paxson and Sally Floyd. Wide-area traffic : the failure of poisson modeling. In *SIGCOMM '94 : Proceedings of the conference on Communications architectures, protocols and applications*, pages 257–268, New York, NY, USA, 1994. ACM.

[PFTK00] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp reno performance : a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2) :133–145, 2000.

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks : the single-node case. *IEEE/ACM Trans. Netw.*, 1(3) :344–357, 1993.

[PKGZ08] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services : design and implementation of interoperable and evolvable sensor networks. In Tarek F. Abdelzaher, Margaret Martonosi, and Adam Wolisz, editors, *SenSys*, pages 253–266. ACM, 2008.

[Pos81] J. Postel. Rfc 793 : Transmission control protocol, September 1981.

[PSSAM09] Ian Paterson, Dave Smith, Peter Saint-Andre, and Jack Moffitt. Xep-0124 : Bidirectional-streams over synchronous http (bosh), November 2009. url-http ://xmpp.org/extensions/xep-0124.html.

[RBF+89] Richard Rashid, Robert Baron, Alessandro Forin, Ro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach : A foundation for open systems. In *In Proceedings of the 2nd Workshop on Workstation Operating Systems. IEEE*, pages 109–113, 1989.

[Riv87] Wind River. Vxworks, 1987. `http://www.windriver.com/`.

[RLAI04] David Raz, Hanoch Levy, and Benjamin Avi-Itzhak. A resource-allocation queueing fairness measure. In *SIGMETRICS '04/Performance '04 : Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 130–141, New York, NY, USA, 2004. ACM.

[RMS+01] J. Riihijarvi, P. Mahonen, M. J. 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, October 2001.

[Rus06] A. Russell. Comet : Low latency data for the browser, 2006.

[Sch68] L. E. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16 :678–690, 1968.

[SD95] David C. Sastry and Mettin Demirci. The qnx operating system. *Computer*, 28 :75–77, 1995.

[SHB06] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload : How scheduling can help. *ACM Trans. Internet Technol.*, 6(1) :20–52, 2006.

[Sho05] Sugoog Shon. Protocol implementations for web based control systems. *International Journal of Control, Automation, and Systems*, 3 :122–129, March 2005.

[Shr02] H. Shrikumar. Ipic - a match head sized webserver., 2002.

[SKR03] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-rto : an enhanced recovery algorithm for tcp retransmission timeouts. *SIGCOMM Comput. Commun. Rev.*, 33(2) :51–63, 2003.

[SLR98] R. Srinivasan, Chao Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 60–70, December 1998.

[Smi56] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3 :59–66, 1956.

[SSM03] Ranjani Sridharan, Ramalingam Sridhar, and Sumita Mishra. Poster : A robust header compression technique for wireless ad hoc networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(3) :23–24, 2003.

[Sti08] Vlad Stirbu. Towards a restful plug and play experience in the web of things. In *ICSC '08 : Proceedings of the 2008 IEEE International Conference on Semantic Computing*, pages 512–517, Washington, DC, USA, 2008. IEEE Computer Society.

[Sys02] CMX Systems. Cmx micronet true tcp/ip networking, October 2002.

[Tec06] InterNiche Technologies. Interniche's nichelite tcp/ipv4 datasheet, November 2006.

[Thi98] Hayo Thielecke. Using a continuation twice and its implications for the expressive power of call/cc, 1998.

[Utr02] Jim Utrley. The two percent solution. Embedded Systems Design. TechInsights (United Business Media), December 2002.

[W3C09] W3C. The web sockets api, 2009. `http://dev.w3.org/html5/websockets/`.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda : an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5) :230–243, 2001.

[WHB03] Adam Wierman and Mor Harchol-Balter. Classifying scheduling policies with respect to unfairness in an m/gi/1. *SIGMETRICS Perform. Eval. Rev.*, 31(1) :238–249, 2003.

[Whi99] Fredric White. Webace server, October 1999. `http://d116.com/ace/`.

[Woo01] Adam D. Woodbury. Efficient algorithms for elliptic curve cryptosystems on embedded systems," http ://www.wpi.edu/pubs/etd/available. In *Worcester Polytechnic Institue, Massachusetts. Retrieved 10 June, 2003 from http ://www.wpi.edu/Pubs/ETD/Available*, pages 1001101–195321, 2001.

[XPMS01] G. Xylomenos, G. Polyzos, P. Mahonen, and M. Saaranen. Tcp performance issues over wireless links. *IEEE Communications Magazine*, 39(4) :52–58, 2001.

[YD09] Dogan Yazar and Adam Dunkels. Efficient application integration in ip-based sensor networks. In *Proceedings of ACM BuildSys 2009, the First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings*, Berkeley, CA, USA, November 2009.

[Zeu99] Zeus. Zeus web server, 1999. `http://www.zeus.com/`.

Liste des sigles et acronymes

6LoWPAN *IPv6 over LoW Power wireless Area Networks*
AJAX *Asynchronous Javascript And XML*
API *Application Programming Interface*
ADC *Analog-to-Digital Converter*
BSD *Berkeley Software Distribution*
CeCILL CEa Cnrs Inria Logiciel Libre
CSS *Cascading Style Sheets*
CPU *Central Processing Unit*
DHCP *Dynamic Host Configuration Protocol*
DPWS *Devices Profile for Web Services*
EEPROM *Electrically Erasable Programmable Read-Only Memory*
ICMP *Internet Control Message Protocol*
IGMP *Internet Group Management Protocol*
IP *Internet Protocol*
IPsec *Internet Protocol security*
IPv6 *Internet Protocol version 6*
IrDA *Infrared Data Association*
FSP *Fair Sojourn Protocol*
(X)HTML *(eXtensible) Hypertext Markup Language*
HTTP *HyperText Transfer Protocol*
HTTPS *HyperText Transfer Protocol Secure*
JSON *JavaScript Object Notation*
JS *JavaScript*
MMU *Memory Management Unit*
MSS *Maximum Segment Size*
PAN *Personal Area Network*
POPS Petits Objets Portables et Sécurisés
POSIX *Portable Operating System Interface*
PS *Processor Sharing*

REST	*REpresentational State Transfer*
RFC	*Request For Comments*
RFID	*Radio-Frequency IDdentification*
RAM	*Random Access Memory*
RAQFM	*Resource-Allocation Queueing Fairness Measure*
ROM	*Read-Only Memory*
RTOS	*Real-Time Operating System*
SIM	*Subscriber Identity Module*
Smews	*Smart and Mobile Embedded Web Server*
SLIP	*Serial Line Internet Protocol*
SOAP	*Simple Object Access Protocol*
SRPT	*Shortest Remaining Processing Time*
TCB	*Trusted Computing Base*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UPnP	*Universal Plug and Play*
URL	*Uniform Resource Locator*
WFQ	*Weighted Fair Queuing*
WSDL	*Web Services Description Language*
WSRPT	*Weighted Shortest Remaining Processing Time*
XML	*eXtensible Markup Language*
XSS	*Cross-site scripting*

Liste des figures

2.1 Structure des différents modèles de système d'exploitation 21
2.2 Chaîne de compilation typique d'un serveur Web intégré 27
2.3 Illustration des techniques de notification d'évènements pour le Web 32
2.4 Ordonnancement dans lequel SRPT est plus efficace que PS pour chaque tâche 34

3.1 Schéma de fonctionnement d'AJAX en deux phases 44
3.2 Illustration du *mashup* centré sur le serveur ou le client 45
3.3 Application des modèles de systèmes d'exploitation au cas du serveur Web . . . 48

4.1 Illustration du modèle de trafic 54
4.2 Durée d'émission en fonction des paramètres TCP 56
4.3 Caractérisation des tailles de contenus d'applications AJAX 58
4.4 Illustration des différentes stratégies d'émission 64
4.5 Illustration des routines impliquées dans les différents schémas d'interaction . . 67

5.1 Architecture du système d'exploitation à macro-noyau 71
5.2 Diagramme de séquence illustrant le comportement du système évènementiel . 72
5.3 Usage du processeur par le serveur Web de uIP 74
5.4 Comparaison de l'usage du processeur par uIP et notre prototype 76
5.5 Illustration de la réduction de mémoire consommée avec les *protothreads* 78
5.6 Caractérisation des tailles de contextes observées sur les applications de Contiki 80
5.7 Illustration de l'occupation mémoire selon le type de coroutine utilisée 81
5.8 Comparaison des temps processeur requis pour un changement de contexte . . 82
5.9 Exemples d'ordonnancement avec l'algorithme β-SRPT. 87
5.10 β-SRPT – Impact de β et ρ sur la rapidité de service 88
5.11 β-SRPT – Impact de β sur l'équité et le partage des ressources 89
5.12 β-SRPT – Compromis entre performances et équité instantanée 89
5.13 β-WSRPT – Bénéfices et compromis entre charge mémoire et équité instantanée 91

6.1 Chaîne de compilation de Smews 94
6.2 Exemple de code d'une ressource Web contrôlant la température 96
6.3 Répartition des tailles de code des fonctionnalités de Smews sur msp430 98
6.4 Comparaison des performances sur le service des contenus statiques 101
6.5 Comparaison des performances sur le service des contenus dynamiques 101
6.6 Caractérisation de l'écart entre le modèle et les performances mesurées 102
6.7 Comparaison de la charge mémoire pour chaque requête 103

6.8 Notification – Latence moyenne de notification 106
6.9 Notification – Pourcentage de notifications reçues 107
6.10 Notification – Pourcentage de notifications uniques reçues 107
6.11 Notification – Trafic en paquets par seconde 108
6.12 Notification – Trafic en volume de données par seconde 109
6.13 Notification – Degré moyen de cohérence 110

A.1 TCP – Ouverture et fermeture d'une connexion 133
A.2 TCP – Exemple de retransmission provoquée par un délai 133
A.3 TCP – Impact des accusés différés 134
A.4 TCP – Fenêtre glissante 134

C.1 Notification (exhaustif) – Latence moyenne de notification 138
C.2 Notification (exhaustif) – Pourcentage de notifications reçues 139
C.3 Notification (exhaustif) – Pourcentage de notifications uniques reçues 140
C.4 Notification (exhaustif) – Trafic en paquets par seconde 141
C.5 Notification (exhaustif) – Trafic en volume de données par seconde 142
C.6 Notification (exhaustif) – Degré moyen de cohérence 143

Liste des tableaux

2.1 Caractéristiques d'une sélection de piles IP embarquées 25
2.2 Caractéristiques d'une sélection de serveurs Web embarqués 28

3.1 Caractéristiques matérielles d'une sélection d'équipements 42

4.1 Nombre maximal de segments en vol en fonction des propriétés de la liaison . . 55
4.2 Affectation des stratégies S1 à S5 aux catégories A et B de la taxinomie 61
4.3 Synthèse des performances et charges mémoire des différentes stratégies 66

5.1 Impact de la taille des blocs des sommes de contrôle TCP calculées hors-ligne . 76
5.2 Caractéristiques des *protothreads*, des coroutines et des coroutines à pile partagée 83

6.1 Comparaison des empreintes mémoire de uIP, MiniWeb et Smews 97
6.2 Ressources constituant l'application Web utilisée pour les expérimentations . . 99

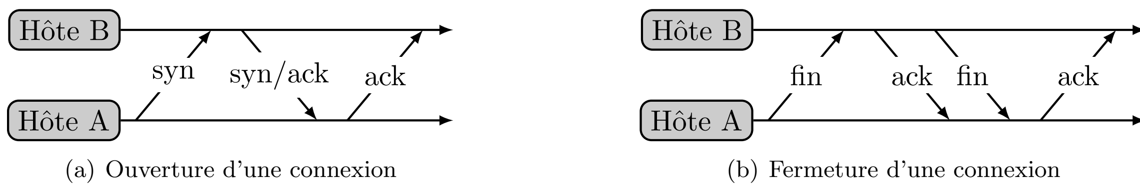
B.1 Smews – Balises XML disponibles 135
B.2 Smews – Fonctions de rappel disponibles 136
B.3 Smews – Appels systèmes fournis 136

Annexe A

Compléments sur le protocole TCP

TCP est un protocole de couche transport, en charge d'assurer une liaison fiable entre deux hôtes distants sur un réseau. Il a été défini en 1981 par la RFC 793 [Pos81]. Nous en détaillons les bases techniques, utiles à la compréhension de ce document.

Connexions Dans TCP, les communications entre deux hôtes se fondent sur une connexion. Une poignée de mains en trois temps permet d'établir une connexion (figure A.1(a)). Cette étape permet notamment de négocier la MSS (taille maximale des segments), comme la valeur minimale des MSS proposées par chaque hôte. La fermeture se fait quant à elle en quatre (parfois simplifiés en trois) échanges (figure A.1(b)).



(a) Ouverture d'une connexion

(b) Fermeture d'une connexion

FIGURE A.1 – Ouverture et fermeture d'une connexion

Retransmissions Une fois la connexion établie, le rôle principal de TCP est d'assurer que toutes les données transmises sont bien reçues en ordre. Pour cela, chaque segment émis contient un numéro de séquence et un numéro d'accusé de réception. Le numéro de séquence est un indice dans les données transmises. L'accusé informe l'hôte distant du prochain numéro de séquence attendu. La figure A illustre un échange. Le segment 7 est perdu. Après un délai, il

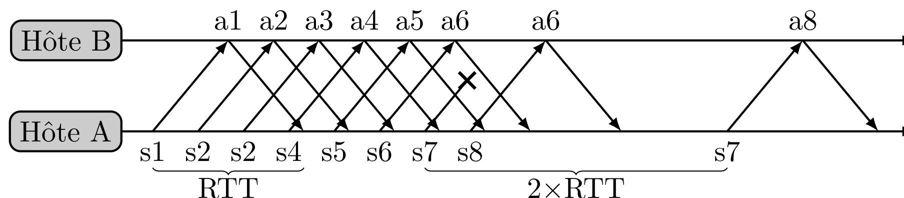


FIGURE A.2 – Exemple de retransmission provoquée par un délai

est retransmis par l'hôte A. L'hôte B émet alors un accusé indiquant qu'il a reçu les segments

1 à 8. Notons que pendant tout la durée de l'échange, chaque hôte calcule le temps moyen d'aller-retour sur la connexion, noté RTT. Le délai de retransmission d'un segment est calculé comme le double de ce RTT.

Accusés différés Afin de minimiser le nombre d'accusés émis, la politique des accusés différés [Bra89] est couramment utilisée. Un hôte TCP qui implémente les accusés différés n'accuse un segment que (i) 200 ms après sa réception ou (ii) dès qu'il reçoit un second segment. La figure A.3 illustre un échange sans puis avec cette politique.

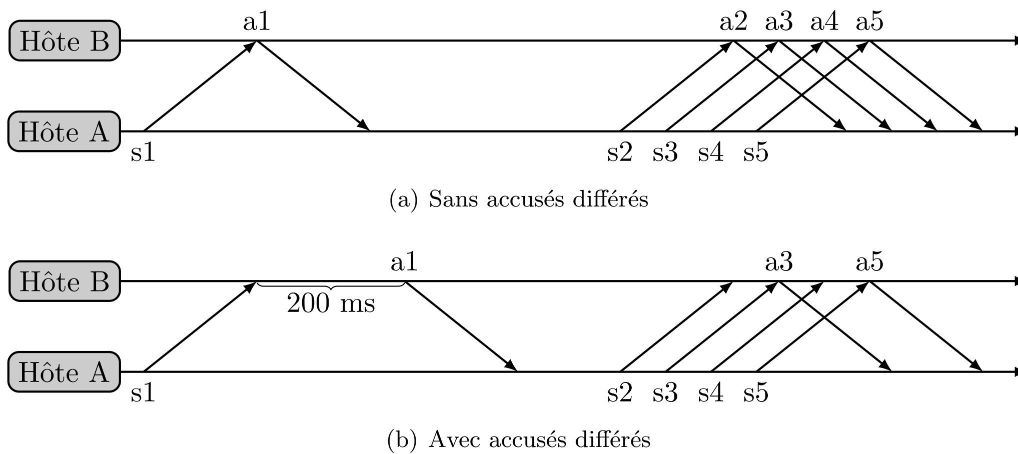


FIGURE A.3 – Impact des accusés différés

Fenêtre glissante Comme les schémas précédents l'illustrent, il arrive qu'un hôte émette plusieurs segments avant de commencer à recevoir un premier accusé. Les données émises mais non encore acquittées constituent la ≪ fenêtre glissante ≫ (wnd). Il est possible pour un hôte de limiter la taille de la fenêtre de son correspondant, par l'intermédiaire de la ≪ fenêtre annoncée ≫ ($awnd$). Afin de limiter les congestions du réseau, chaque hôte se régule en calculant sa propre ≪ fenêtre de congestion ≫ ($cwnd$). La taille de le fenêtre glissante est $wnd = min(awnd, cwnd)$. La figure A.4 illustre un cas où un hôte transmet 16 segments avec une fenêtre glissante de 4 segments.

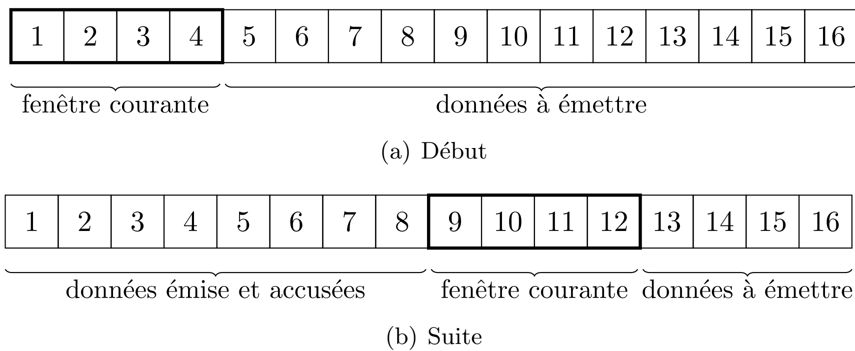


FIGURE A.4 – Fenêtre glissante

Annexe B

Interface applicative de Smews

Les applications Web pour Smews sont écrites en langage C. La déclaration d'une ressource Web se fait en incluant des méta-données au format XML en commentaire d'un fichier source. Les balises mises à disposition sont listées et décrites par la table B.1. Le gestionnaire de service de Smews fait appel aux applications par l'intermédiaire d'un ensemble de fonctions de rappel, définies dans la table B.2 et liées au code par interprétation des méta-données XML. Ces fonctions interagissent avec le noyau à l'aide des appels système présentés dans la table B.3.

`generator`

racine des méta-données applicatives

`handlers`

définition des fonctions de rappel applicatives par les champs *init*, *initGet* et *doGet*

`properties`

définition du nom de la ressource par le champ *name*, des propriétés de persistance des données *via* le champ *persistence* (valeur *persistent*, *volatile* ou *idempotent*) et *interaction* (valeur *rest*, *notification* ou *streaming*)

`args`

définition des arguments attendus dans l'URL par un ensemble de balises `arg`

`arg`

définition d'un argument par les champs *name*, *type* (valeurs *uint8*, *uint16*, *uint32* ou *str*) et éventuellement *size*

Table B.1 – Balises XML disponibles

`char (init_app_func_t)(void)`

fonction d'initialisation de l'application, retournant 1 en cas de succès, 0 en cas d'échec

`char (initget_app_func_t)(struct args_t *)`

fonction appelée lors de la réception d'une requête *get*, avec en paramètre un pointeur sur la structure de données contenant l'ensemble des arguments d'URL décodés. Retourne 1 en cas de succès, 0 en cas d'échec.

`char (doget_app_func_t)(struct args_t *)`

fonction appelée pour la génération de la réponse HTTP, ayant accès aux arguments décodés, retournant 1 en cas de succès, 0 en cas d'échec.

Table B.2 – Fonctions de rappel disponibles

`out_c(char c)`

à utiliser dans la fonction de rappel `doGet` afin d'ajouter l'octet `c` à la réponse HTTP qui est en train d'être générée

`out_uint(uint16_t i, char radix)`

à utiliser dans la fonction de rappel `doGet` afin d'ajouter l'entier `i` dans la base `radix` à la réponse HTTP qui est en train d'être générée

`out_str(const char str[])`

à utiliser dans la fonction de rappel `doGet` afin d'ajouter la chaîne de caractères `str` à la réponse HTTP qui est en train d'être générée

`out_flush()`

force l'émission d'un segment contenant des données générées jusqu'à présent

`trigger_channel(const struct output_handler_t *handler)`

utilisable dans une fonction quelconque, permet de déclencher un canal Comet, pour une notification ou une émission en flux

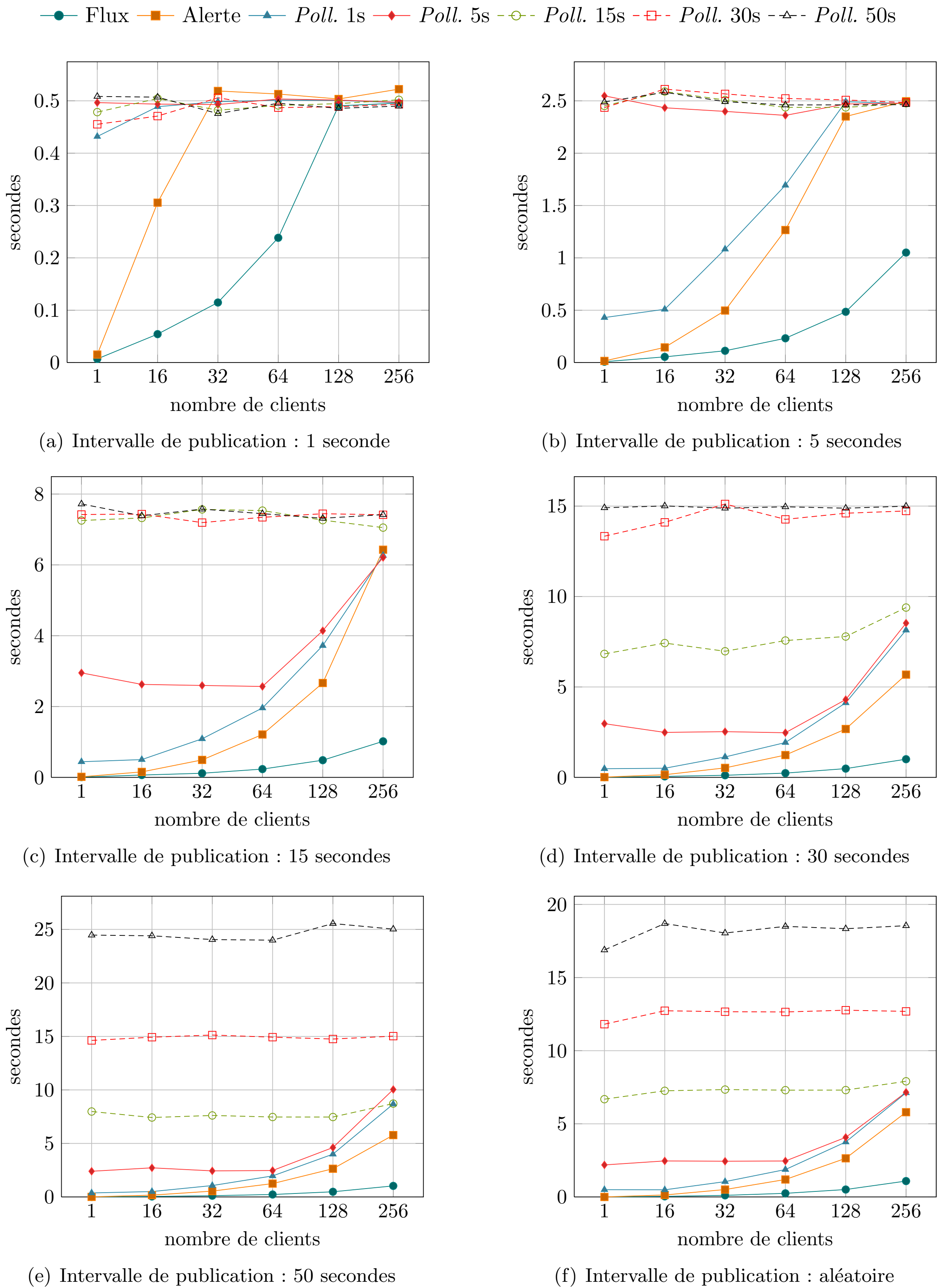
`set_timer(timer_func_t callback, uint16_t period_millis)`

utilisable dans une fonction quelconque, permet de demander l'appel d'une fonction de rappel à un intervalle de temps donné en paramètre

Table B.3 – Appels systèmes fournis

Annexe C

Résultats expérimentaux sur la notification d'évènements



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 5 secondes

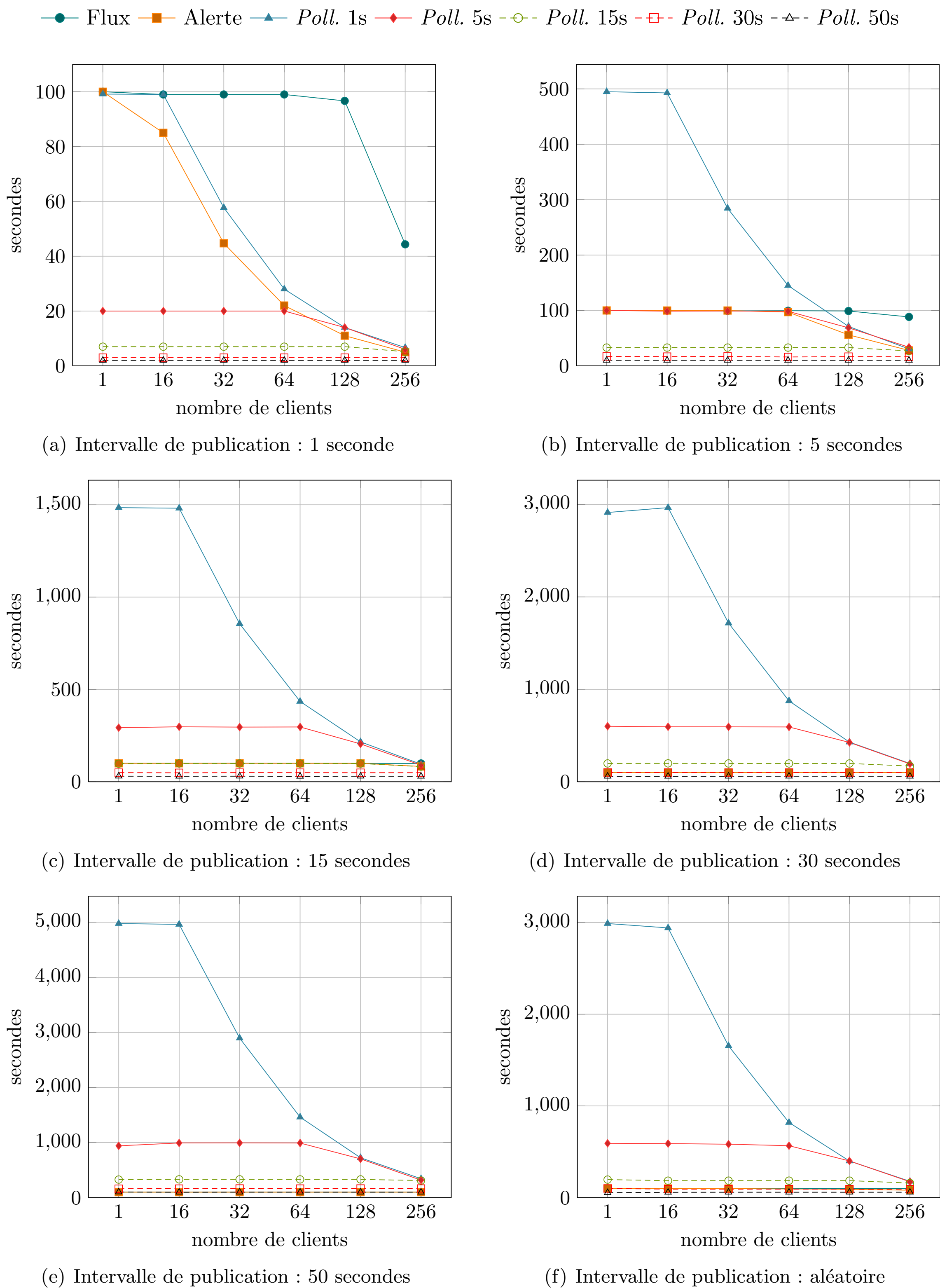
(c) Intervalle de publication : 15 secondes

(d) Intervalle de publication : 30 secondes

(e) Intervalle de publication : 50 secondes

(f) Intervalle de publication : aléatoire

Figure C.1 – LMN – Latence moyenne de notification



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 5 secondes

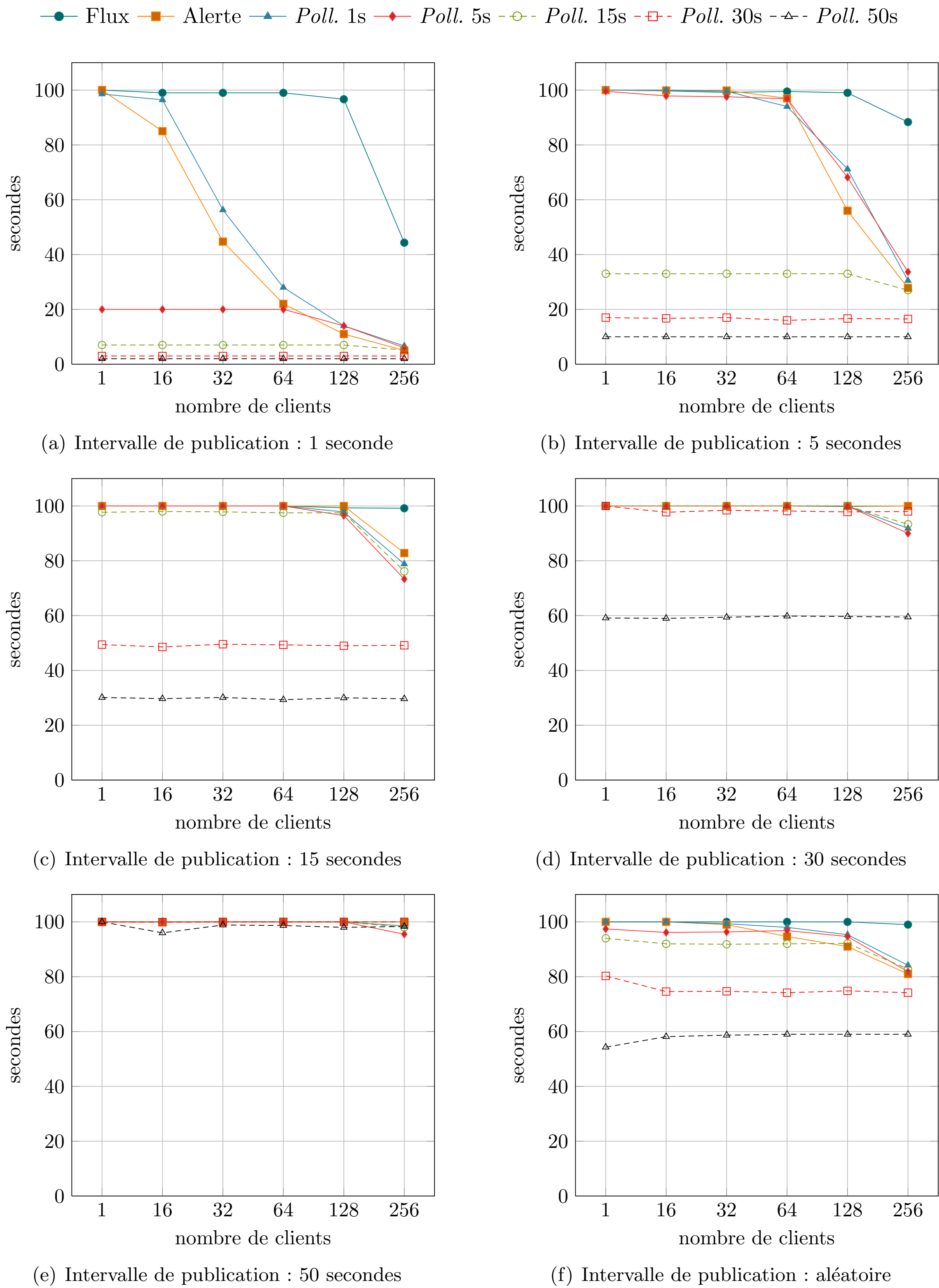
(c) Intervalle de publication : 15 secondes

(d) Intervalle de publication : 30 secondes

(e) Intervalle de publication : 50 secondes

(f) Intervalle de publication : aléatoire

Figure C.2 – PNR – Pourcentage de notifications reçues



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 5 secondes

(c) Intervalle de publication : 15 secondes

(d) Intervalle de publication : 30 secondes

(e) Intervalle de publication : 50 secondes

(f) Intervalle de publication : aléatoire

FIGURE C.3 – PNUR – Pourcentage de notifications uniques reçues

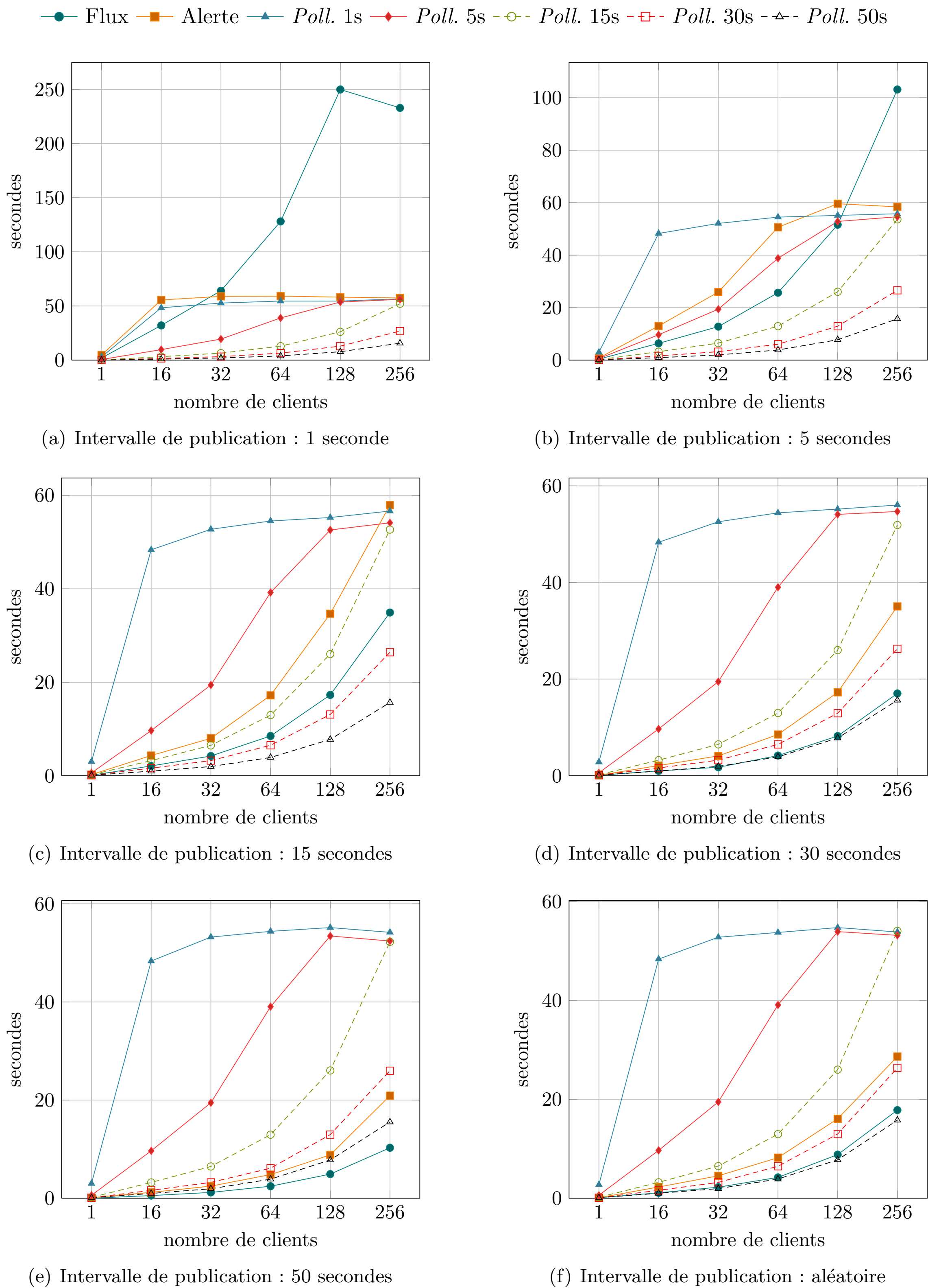
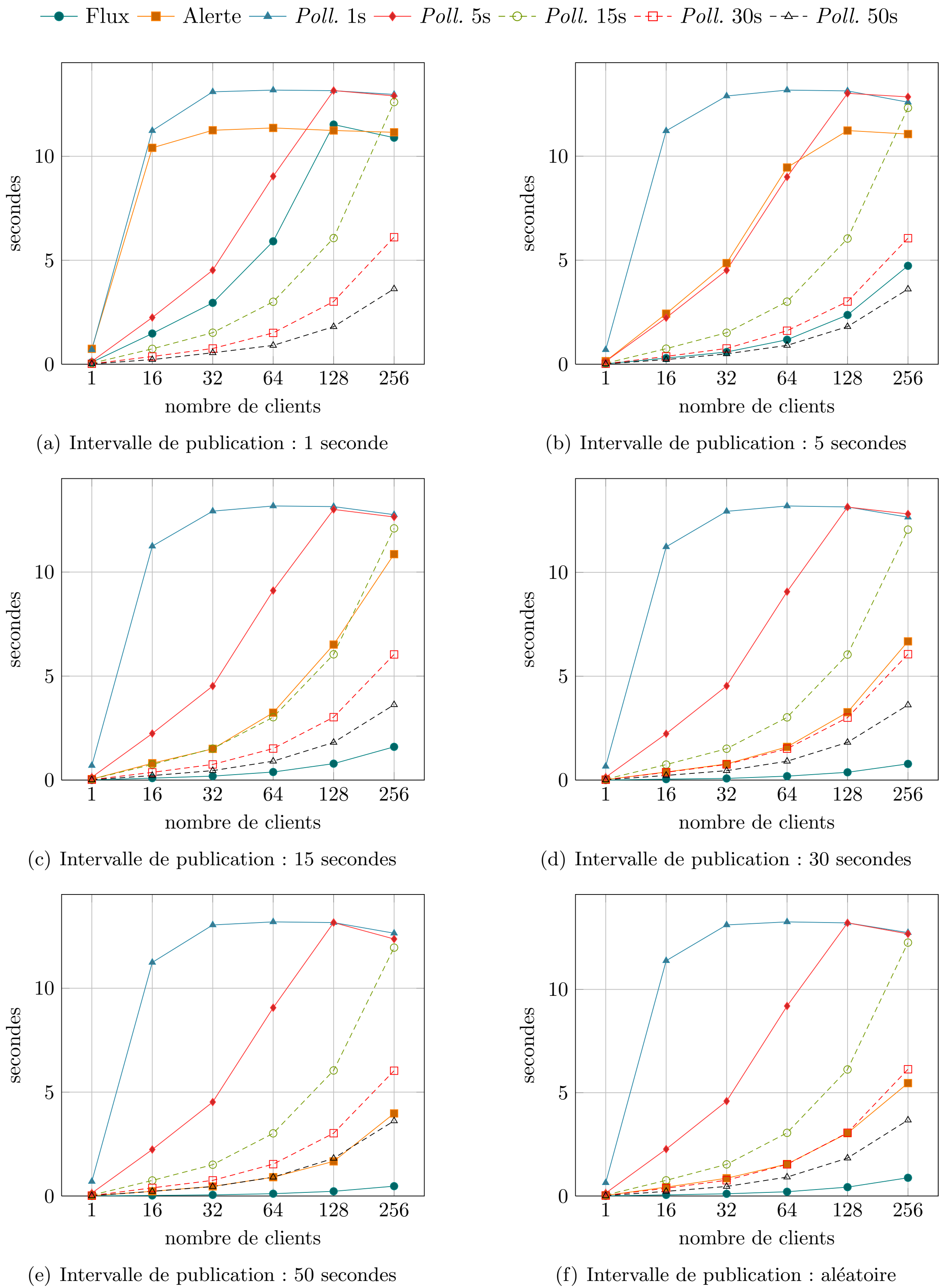


FIGURE C.4 – TPS – Trafic en paquets par seconde



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 5 secondes

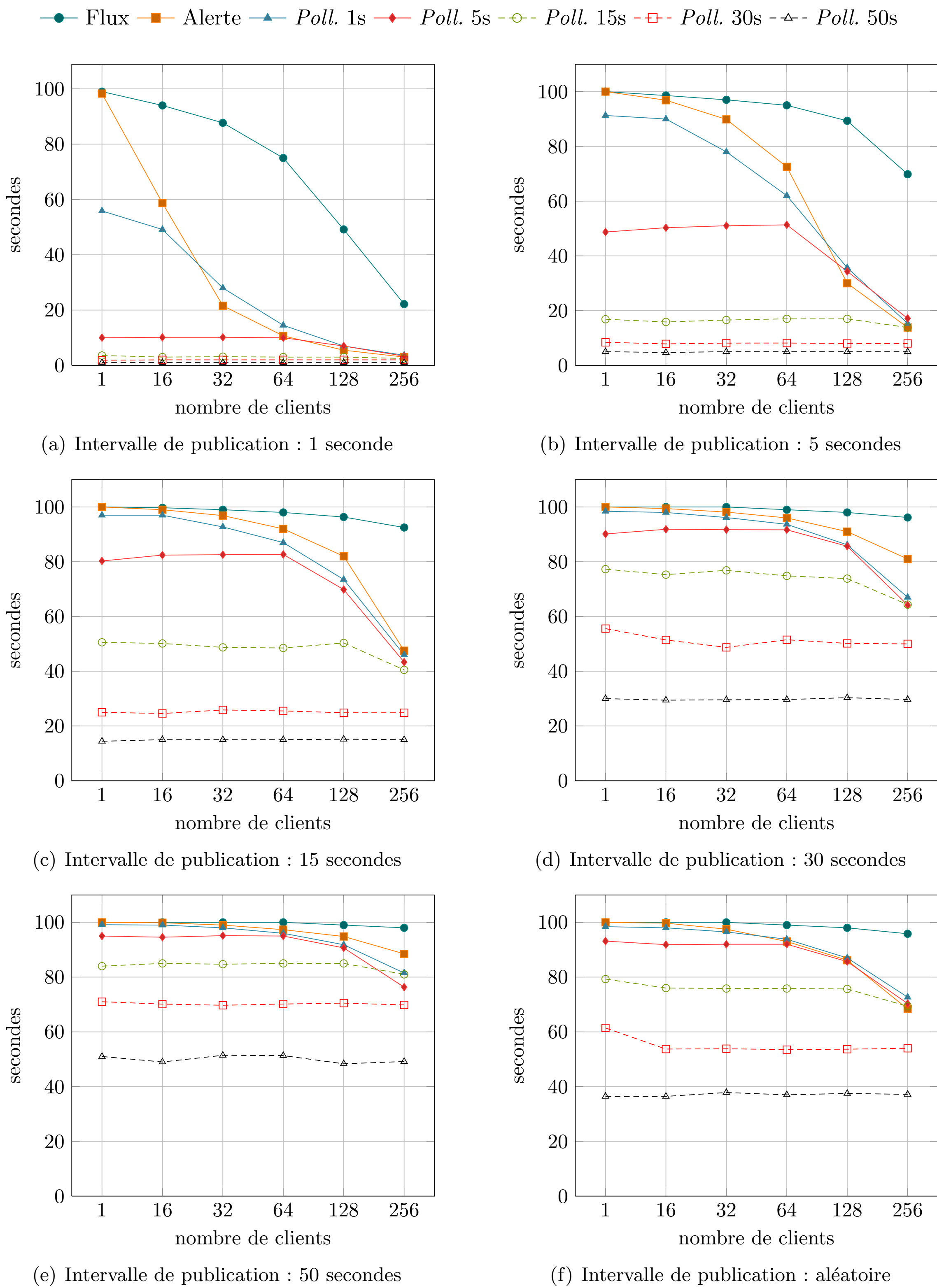
(c) Intervalle de publication : 15 secondes

(d) Intervalle de publication : 30 secondes

(e) Intervalle de publication : 50 secondes

(f) Intervalle de publication : aléatoire

FIGURE C.5 – TDS – Trafic en volume de données par seconde



(a) Intervalle de publication : 1 seconde

(b) Intervalle de publication : 5 secondes

(c) Intervalle de publication : 15 secondes

(d) Intervalle de publication : 30 secondes

(e) Intervalle de publication : 50 secondes

(f) Intervalle de publication : aléatoire

Figure C.6 – DMC – Degré moyen de cohérence

Résumé

Les travaux présentés dans ce mémoire se placent dans le contexte de l'extension des technologies du Web à l'informatique enfouie. Le *Web des objets* qui en résulte ouvre les portes à de nouvelles interactions en garantissant une interopérabilité aussi bien au niveau réseau qu'applicatif. Nous adressons la conception du système logiciel jouant le rôle de serveur Web au sein d'équipements fortement contraints, tels que des cartes à puce ou des capteurs. Les solutions de l'état de l'art permettant de supporter une pile protocolaire standard à faible empreinte mémoire exigent de sacrifier les performances et les fonctionnalités du système. La thèse défendue est qu'en dédiant un système d'exploitation au support d'une famille d'applications de haut niveau, nous sommes en mesure de produire un logiciel performant et consommant peu de ressources tout en offrant des fonctionnalités très riches. Nous étudions une architecture fondée sur un *macro-noyau* intégrant gestion du matériel, pile de communication et conteneur d'applications, présentant une interface adaptée aux besoins de ces dernières.

Les propositions que nous présentons sont fondées sur une analyse et une modélisation théorique des besoins des applications Web, puis sont validées expérimentalement à l'aide de notre prototype, Smews. Nous nous intéressons notamment à l'adaptation du comportement de la pile de communication aux propriétés des applications exécutées, à la construction transversale du noyau évènementiel permettant des traitements performants tout en consommant peu de ressources, au support de coroutines applicatives à bas coût, à l'ordonnancement efficace et équitable des requêtes, à la notification d'évènements ainsi qu'à la capacité de passage à l'échelle du système résultant.

Abstract

The context of this thesis is the extension of Web technologies to ambient computing. The resulting *Web of Things* allows novel interactions by guaranteeing interoperability at both network and applications levels. We address the design of the software system behaving as a Web server and embedded in strongly constrained devices such as smart cards or sensors. The state of the art solutions allowing to run a lightweight standard protocol stack involve poor performances and sacrifice the system features. The thesis is that by dedicating an operating system to the support of a high-level family of applications, we can produce an efficient software consuming a few resources while providing rich functionalities. We study an architecture based on an *macro-kernel* integrating the hardware management, the communications stack and the applications container, providing an interface that fits the applications needs.

The proposals we present are based on a theoretical analysis and modeling of Web applications needs and have been experimentally validated thanks to our prototype, Smews. The topics we address include the adaptation of the communication stack behavior depending on executed applications properties, the cross-layer design of the event-driven kernel which allows fast processing while consuming few resources, the low-cost applicative coroutines management, the efficient and fair requests scheduling, the support of event notification as well as the overall system scalability.