Support au déploiement de logiciel sur Smews

Sommaire

Introduction

Notre projet de fin d'étude nous a amené dans l'univers des systèmes embarqués. Un système embarqué est un dispositif souvent très petit qui intègre des sondes, des traitements de données et des capacités de communication.

Nous nous sommes intéressés à un serveur web pour les systèmes embarqués : Smews. Un tel serveur permet de consulter et d'agir sur ce dernier comme sur n'importe quel élément du web. On en retrouve dans des équipements de domotique.

Notre rôle était de permettre le déploiement d'une application quand le serveur est en fonctionnement. Pour ce faire, nous pouvions nous baser sur un travail réalisé pour un système d'exploitation destiné aux systèmes embarqués : Contiki.

Système embarqué

Les systèmes embarqués qui nous occupent dans le cadre de ce projet sont des capteurs de terrain. Les communications se font sans fil, ils disposent de peu de puissance et ont une faible mémoire.

Ce type de matériel est applicable à un large éventail d'applications critiques, y compris la lutte contre les incendies, les interventions d'urgence, surveillance de bâtiments, la surveillance militaire et des applications médicales. On peut aussi en retrouver tous les jours dans nos systèmes ménagers.

Travailler sur ce genre de système, nous impose toute une série de contraintes dû à leurs capacités limitées, au souci d'économie d'énergie (fonctionnement sur batterie), à l'espace mémoire restreint.

Contiki Operating System

Contiki est un système d'exploitation portable, multitâches et open source pour les systèmes embarqués et les réseaux de capteurs sans fil.

Contiki a été développé par l'Institute Swedish Institute of Computer Science (SICS). Il est écrit en C.

Plusieurs sociétés ont adopté des idées et des mécanismes de Contiki pour leur propre système. On peut par exemple le retrouver dans les capteurs de vibration de la Play Station 2.

Contiki gère les communications IP, l'interaction avec le capteur à l'aide d'un navigateur, un système de fichier en flash et beaucoup d'autres fonctionnalités comme un chargeur dynamique d'application sur lequel nous nous sommes basés pour réaliser notre projet.

Smews

Smews est un serveur web embarqué ne reposant sur aucun système d'exploitation. De ce fait, l'empreinte mémoire y est très minime. Le fait qu'il ne possède pas de système d'exploitation demande donc que toute la gestion du matériel et de la pile de communication soit gérée par celui-ci. De plus, Smews permet de gérer un ensemble d'application, pour faire simple, un ensemble d'URL.

Une application est donc un ensemble d'URLs, regroupé au sein d'un module pour facilité sa gestion. Une application peut être composée de fichiers statiques, que ce soit une page HTML, un fichier Javascript ou bien une image, et de contenu dynamique, généré lors de l'accès à une URL.

Une fois que Smews a été généré, avec son ensemble d'application, il n'est pas possible d'en rajouter sans régénérer à nouveau Smews. Cela est dû à une limitation actuelle de Smews et est l'objet de notre travail.

Démarche

Dans un premier temps, nous nous sommes plongés dans la lecture de l'article : "Run-time dynamic linking for reprogramming wireless sensor networks", Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt , 2006, ISBN:1-59593-343-3. Cet article reporte une expérience d'implémentation d'un chargeur dynamique et les tests de performances qui ont été effectués.

Ensuite, nous avons étudié d'une part le fonctionnement de Contiki et d'autre part celui de Smews. Nous avons cherché à exécuter des exemples utilisant le chargeur dynamique de Contiki. Ensuite en partant de cet exemple, nous avons recherché comment fonctionne le chargeur et quels étaient les éléments que nous pouvions reprendre et adapter à Smews.

Aussi bien pour Contiki que pour Smews, nous nous sommes intéressés aux fonctionnalités de ses systèmes pour en ressortir les caractéristiques et voir les éléments qui nécessiteront une adaptation.

Par la suite, nous avons inséré le loader de Contiki dans Smews et chercher à créer tous les liens nécessaire à son bon fonctionnement et avons adapté le système de placement en mémoire d'une application du chargeur.

Enfin, la dernière étape a été de créer notre propre programme qui, placé dans Smews, devait être chargé et exécuté.

Adaptation du chargeur dynamique

Recherche des fonctions utiles et compilation

Contiki possède un dossier contenant tous les fichiers du loader. Mais une partie de ces fichiers n'est pas nécessaire pour l'adaptation à Smews. Nous sommes donc partis du fichier central (elfloader.c) et nous avons petit à petit ajouté les éléments indispensables à son bon fonctionnement. Il s'agit des éléments permettant l'accès mémoire et de faire lien avec les fonctions de Smews. Et de le lier à la fonction gérant les interruptions de Smews.

Ensuite, une des particularités de Smews et qu'à la compilation en fonction de la cible certains fichiers vont être compilé ou non. Smews place ses fichiers dans un dossier targets et un sous-dossier au nom de la cible. Dans le cas du loader, Contiki place tous les fichiers dans le même dossier et en précisant la cible dans le nom du fichier.

Nous avons donc changé ces fichiers pour pouvoir les placer dans les dossiers de leur cible respective et pour que le compilateur de Smews les prenne en charge.

Pour compiler Smews utilise scons qui ressemble à make mais est écrit en python. Nous avons ajouté à Smews un dossier et nous avons dû demander à scons d'en tenir compte et donc de compiler les fichiers qui sont dedans.

Pour pouvoir générer notre image de Smews avec notre chargeur, nous avons effectué plusieurs compilations qui ont révélées des erreurs, que nous corrigions et ainsi de suite. Les erreurs nous demandaient d'inclure certaines fonctions qui manquaient ou d'en adapter d'autres.

Choix des fonctions

On a repris que le strict nécessaire, on est parti de rien, on a jute regardé les fonctions de elfloaders qui nous intéressaient. On a regardé les dépendances et ce qui existait déjà dans Smews à pu être réutilisé.

Exemples :

Les interruptions existent déjà dans Smews, pas besoin d'aller rechercher celles de Contiki.

Nous n'avons pas repris le système de descripteur de fichier de Contiki, cela alourdirait Smews et le rapprocherait plus de l'OS que du web serveur. Comme expliqué plus loin, nous avons utilisé un système d'adresse et pointeur dans la mémoire.

Tout le système de gestion des processus de Contiki n'a pas été repris non plus, Smews n'est pas un OS, il ne fonctionne donc pas de la même manière. Nous avons dû donc chercher comment l'adapter tout en pouvant appeler nos fonctions dans notre code que nous avons chargé dynamiquement. Pour ce faire, nous avons remarqué qu'il existait une fonction dans le loader qui nous permettait de retrouver nos fonctions et de les appelées.

Adaptation des fonctions d'accès à la mémoire

Le chargement dans Contiki se fait au travers d'un descripteur de fichier. Ce descripteur de fichier peut prendre plusieurs formes : un réel descripteur de fichier pour le système de fichier utilisé dans Contiki : Coffee, un accès à l'EEPROM, un système de fichier type POSIX ou en mémoire.

Dans Smews il n'y a pas de système de fichier. Les seuls accès possible sont en mémoire Flash et la mémoire RAM. Il faut donc adapter les fonctions de lecture de fichier du loader pour pouvoir lire en mémoire et non plus un fichier. De ce fait au lieu de prendre un descripteur de fichier nous avons ici un pointeur et toutes les opérations se font en mémoire.

Pour être plus complet, il y a 3 fonctions clés qu'il faut modifier et pour faire fonctionner le chargement de code. La fonction de lecture, la fonction d'écriture et la fonction de code relocation.

Comme expliqué précédemment, le seul accès possible est la mémoire et donc nous utilisons ici les fonctions de base comme memcpy afin de remplacer les fonctions de lecture d'un fichier, écriture d'un fichier. Il en a été de même pour la relocalisation d'adresse. Cependant c'est un point plus difficile étant donné qu'on ne se contente pas de recopier des bytes dans deux endroits différents de la mémoire mais on modifie le code afin de permettre la bonne exécution de celui-ci.

Création d'une application appelant les fonctions de Smews

Une fois que l'on possède un Smews compilé avec notre chargeur, nous devons tester son fonctionnement. Nous avons pour cela créé une application qui appelle une fonction de Smews permettant d'afficher un message à l'écran. Nous avons choisi d'afficher le traditionnel message « Hello World ».

Nous avons compilé notre application en dehors de Smews, faisant appel à une fonction extérieure, nous avons dû le compiler sans qu'il crée les liens, ceux-ci seront réalisés lors du chargement de l'application dans Smews.

Ensuite, nous avons dû faire reconnaître les fonctions de Smews à notre chargeur. Cela n'est possible qu'avec un Smews déjà compilé et à l'aide d'un script fourni par Contiki qui nous permet de créer le fichier dans lequel le chargeur recherche les fonctions. En fait, ce fichier crée une structure avec un tableau contenant les adresses d'accès à chaque fonction.

Une application ne peut être chargée que sous forme de tableau. En effet, dans Smews, il n'y a pas de système de fichiers et tout est donc dans la mémoire. Nous avons donc créé un script qui lit un fichier binaire et crée le tableau qui sera lu et chargé. De plus le loader recherche le nom d'une fonction qui servira de point de démarrage dans l'application actuellement il s'agit d'un main mais on peut très bien décider de l'appeler init ou autre.

Enfin nous plaçons notre tableau dans un fichier qui est lu par la fonction main de Smews et qui appel notre chargeur et recompilons Smews. A l'exécution, notre application est chargée et toute une série de test sont effectués pour vérifier sa conformité avec l'environnement.

Les tests sont :

- Vérifier que l'application passée est bien un fichier binaire en format elf ;
- Que les fonctions présentes dans le programme sont bien présentes dans Smews ou dans notre le nouveau programme ;
- Pas de point de démarrage dans le programme créé ;
- Vérification qu'il y a ait bien du code à exécuter

Recherche de l'adresse de la nouvelle fonction

Une fois le code chargé en mémoire et correctement lié avec Smews, on peut maintenant exécuter le code. A une chose prêt : qu'est-ce qu'on va exécuter?

Nous avons une liste de fonction il faut donc choisir la bonne et l'exécuter. Dans notre cas nous recherchons un point d'entrée ou d'initialisation, ici par exemple, la fonction main.

Une partie du chargement dynamique du code s'occupe donc de rechercher la fonction main dans ce qui a été chargé. En clair, nous obtenons un pointeur vers cette fonction, que l'on peut donc appeler comme n'importe quelle fonction C.

Création de la table des symboles

Pour qu'une application puisse être liée aux fonctions déjà existantes dans Smews, une table d'association est utilisée. Cette table fait l'association entre le nom d'une fonction et l'adresse pour accéder à cette fonction. On parle dans notre cas de table des symboles. Cette table est créée grâce à l'outil 'NM' dont le résultat est passé à un script qui crée le fichier symbols.c (placer dans les drivers de la cible).

Nous avons tenté d'inclure dans Smews la création de la table des symboles et la recompilation de Smews en tenant compte de cette table.

Notre première idée a été d'exécuter un script créant cette table et de ré exécuter la compilation. Pour ce faire, nous appelions le script à la fin de SConscript. Ensuite dans SConstruct, on rappelé le SConscript une deuxième fois pour qu'il prenne en compte la nouvelle table des symboles. Cette méthode n'a pas fonctionné car au moment de l'exécution du script, le binaire n'était pas encore créé.

En deuxième lieu, nous avons pensé utiliser la méthode addPostAction de env. En demandant comme post action l'exécution de notre script. Une nouvelle fois, nous avons eu les mêmes problèmes que pour notre première idée. Nous l'avons d'ailleurs vérifié en ayant comme post action l'affichage d'un message. On a alors remarqué que l'exécution se faisait bien avant la création du binaire.

Nous avons donc dans SConscript ajouter une commande qui créé un script contenant les instructions de création de la table des symboles et faisant une nouvelle compilation une fois la table créée.

Notre solution demande donc à l'utilisateur de penser à exécuter un script après la compilation. Afin de connaitre l'outil 'NM' à appeler, ce dernier est spécifique à chaque fichier objet généré (x86, arm, msp430, ...), un champ dans les SConscript des targets a été ajouté. Si ce champ n'est pas présent, la commande nm de gcc sera utilisée. De plus, la deuxième compilation tient compte des options qui ont été passé à scons par l'utilisateur. Enfin, le fichier symbols.c est placé dans les drivers de la bonne cible.

Conclusion

Nous arrivons à faire l'édition de liens quand l'application est déjà présente dans Smews. Il manque la création d'une application qui permet à un utilisateur de passer son application et de la télé versé dans Smews. C'est cette application qui doit ensuite lancer notre chargeur.

Seulement l'upload dans Smews ne se fait pas comme ça. On pourrait penser que comme c'est un serveur web, une simple page comme on a l'habitude de voir sur internet avec la possibilité de cliquer sur un bouton « upload » devrait suffire. Seulement, ce type de page demande que le serveur supporte la méthode http POST. Hors Smews ne la possède pas encore. Il faut donc rechercher quelle méthode utiliser pour pouvoir réaliser ce transfert de fichier. Nous avons exprimé une méthode mais il en existe beaucoup d'autres. Ce qu'il ne faut pas oublié c'est que Smews est destiné à des systèmes limités en ressources et sur batterie.

Annexe

Préparatifs pour lancer une application avec le loader

A la racine du projet :

gcc -c hello.c -> output hello.o

python arrayBuilder.py hello.o hello.h

Une fois cela fait :

Pour pouvoir utliser le fichier hello.h générer à l'aide du script arrayBuilder.py, il faut le placer dans */core* et modifier le main de la façon suivante :

```
int ret;
char *print, *symbol;
	elfloader_init();
	ret = elfloader_load(smews_to_load);
	switch(ret) {
		case ELFLOADER_OK:
			print = "OK";
			break;
		case ELFLOADER_BAD_ELF_HEADER:
			print = "Bad ELF header";
			break;
		case ELFLOADER_NO_SYMTAB:
			print = "No symbol table";
			break;
		case ELFLOADER_NO_STRTAB:
			print = "No string table";
			break;
		case ELFLOADER_NO_TEXT:
			print = "No text segment";
			break;
		case ELFLOADER_SYMBOL_NOT_FOUND:
			print = "Symbol not found: ";
			symbol = elfloader_unknown;
			break;
		case ELFLOADER_SEGMENT_NOT_FOUND:
			print = "Segment not found: ";
			symbol = elfloader_unknown;
			break;
		case ELFLOADER_NO_STARTPOINT:
			print = "No starting point";
			break;
		default:
			print = "Unknown return code from the ELF loader (internal bug)";
			break;
	}
	printf(print);
	printf("\n");
```

Création du binaire :

1) Compilation de smews
   scons target=linux apps=contactsBook,mycalendar:calendar,:welcome

2) Exécuter la commande : *./postInstallScript*
   Ces deux étapes permettent d'avoir un binaire contenant la table d'association avec les fonctions de Smews.