ELUS: Mecanismo de Reconfiguração de Software para Sistemas Operacionais Embarcados

Giovani Gracioli e Antônio Augusto Fröhlich

[1]Laboratório de Integração Software e Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476, 88049-900, Florianópolis, SC, Brasil

{giovani,guto}@lisha.ufsc.br

Resumo. *Reconfiguração Dinâmica de Software (RDS) é o processo de atualizar o software de um sistema em execução. Um mecanismo de RDS para sistemas embarcados deve ser simples, transparente para aplicações e usar o mínimo de recursos possíveis (e.g. memória, processamento), pois estará competindo com os recursos do próprio sistema embarcado. Este artigo apresenta o* EPOS LIVE UPDATE SYSTEM *(*ELUS*), uma infra-estrutura de sistema operacional que permite reconfiguração dinâmica de software em sistemas embarcados.* ELUS *faz uso de sofisticadas técnicas de metaprogramação estática em C++, consumindo pouca memória e tornando o processo de reconfiguração configurável, simples e totalmente transparente para as aplicações.*

Abstract. *Dynamic software reconfiguration is the process of updating the system software during its execution. A dynamic software reconfiguration mechanism for an embedded system must be simple, transparent to applications, and use the minimum amount of resources (e.g. memory, processing) possible, since it shares resources with the target embedded system. We present* EPOS LIVE UPDATE SYSTEM *(*ELUS*), an operating system infrastructure for resource-constrained embedded systems. Through the use of sophisticated C++ static metaprogramming techniques, unlike the previous software reconfiguration infrastructures,* ELUS *provides a low-overhead, simple, configurable, and fully transparent software reconfiguration mechanism.*

1. Introdução

Reconfiguração Dinâmica de Software (RDS) é uma característica presente na grande maioria dos sistemas atuais, desde ambientes de computação convencionais (e.g. computadores pessoais) até sistemas embarcados (e.g. sensores sem fio). Com a finalidade de corrigir bugs, adicionar e/ou remover funcionalidades e adaptar o sistema às variabilidades dos ambientes de execução, esta característica permite que o software do sistema seja atualizado em tempo de execução.

Redes de Sensores Sem Fio (RSSF), por exemplo, podem ser formadas por um grande número de sensores espalhados em um ambiente de monitoração. Geralmente, estes sensores possuem poucos kilobytes de memória e poder de processamento [Pottie and Kaiser 2000]. Frequentemente, coletar todos os sensores para reprogramação é impraticável devido à grande quantidade ou ao difícil acesso às áreas de monitoração. Para estes casos, um mecanismo eficiente de RDS poderia reprogramar

todos os nodos da rede remotamente. Além disso, o mecanismo de RDS também deve ser adaptável a grande variabilidade de plataformas de sistemas embarcados existentes em termos de recursos de memória, interconexões (e.g. RS-485, CAN, ZigBee) e poder de processamento.

Não obstante, a infra-estrutura de reconfiguração dinâmica de software para sistemas embarcados compartilha recursos com as aplicações do sistema e não deve influenciar sua operação [Felser et al. 2007]. Mecanismos para RDS em sistemas embarcados podem ser divididos em três categorias: (i) atualização de código binário [Reijers and Langendoen 2003, Marrón et al. 2006, Felser et al. 2007]; (ii) máquinas virtuais [Levis and Culler 2002, Koshy and Pandey 2005, Xie et al. 2006]; e (iii) sistemas operacionais [Dunkels et al. 2004, Han et al. 2005, Cha et al. 2007, Bagchi 2008]. Na primeira, um bootloader ou ligador é responsável por receber e atualizar a nova imagem do sistema. Na segunda, a reconfiguração é realizada através da atualização do script da aplicação que é interpretado pela máquina virtual. Finalmente, sistemas operacionais são projetados para abstrair uma atualização de software. Tais SOs são organizados em módulos reconfiguráveis e criam um nível de indireção entre a aplicação que utiliza o módulo e o módulo propriamente dito através de ponteiros e/ou tabelas. A reconfiguração neste caso ocorre mudando o endereço desses módulos dentro das tabelas/ponteiros.

Este artigo apresenta o EPOS LIVE UPDATE SYSTEM (ELUS), uma infra-estrutura de SO para sistemas embarcados com recursos limitados. O ELUS é construído dentro do framework de componentes do EMBEDDED PARALLEL OPERATING SYSTEM (EPOS), em torno do programa de aspecto de invocação remota de métodos [Fröhlich 2001]. As principais características que tornam o ELUS diferente das infra-estruturas de SO existes são:

- **Configurabilidade:** componentes do sistema[1] podem ser marcados como reconfiguráveis ou não em tempo de compilação. Para todos os componentes não-reconfiguráveis, nenhum sobrecusto em termos de processamento e memória é adicionado ao sistema.
- **Baixo Consumo de Memória:** ao marcar um componente como reconfigurável, são adicionados cerca de 1.6KB de memória e 26 bytes de dados por componente, o que presenta um baixo consumo comparado com os trabalhos relacionados.
- **Transparência e Simplicidade:** uma reconfiguração é realizada através de um simples protocolo, o ELUS TRANSPORT PROTOCOL (ETP). Além disso, tanto a infra-estrutura de reconfiguração como a reconfiguração de software são totalmente transparentes para as aplicações.
- **Estrutura de Mensagens:** através do uso da estrutura do framework do EPOS, a passagem de argumentos e valor de retorno entre os componentes do sistema é realizada eficientemente, cerca de 5 vezes mais rápida do que os trabalhos anteriores.
- **Reconfiguração:** ELUS permite que o desenvolvedor atualize ambos componentes do SO e aplicações. Em alguns SOs, como Contiki [Dunkels et al. 2004] por exemplo, somente aplicações ou partes do SO são reconfiguráveis.

[1]Componentes no EPOS são encapsulados em classes C++ com interface e comportamento bem definidos.

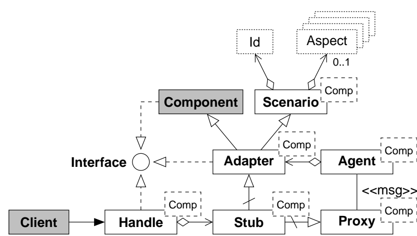


Figura 1. Framework metaprogramado do EPOS.

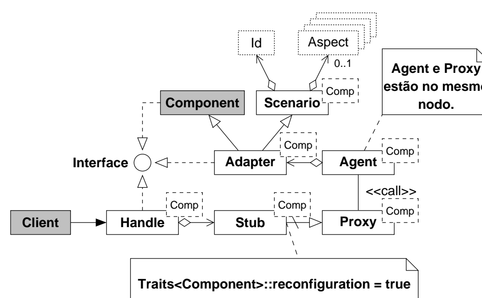


Figura 2. Framework para RDS.

O restante deste artigo está organizado da seguinte maneira: a seção 2 apresenta o framework metaprogramado do EPOS. A seção 3 explica o projeto e a implementação do ELUS. A avaliação experimental é relatada na seção 4. A seção 5 apresenta os trabalhos relacionados. A seção 6 compara os trabalhos relacionados com o ELUS. Finalmente, a seção 7 conclui o artigo.

2. Framework Metaprogramado do EPOS

EPOS é um SO para sistemas embarcados multi-plataforma baseado em componentes que possui uma arquitetura altamente escalável moldada para suportar as necessidades das aplicações [Fröhlich 2001]. O EPOS implementa aspectos[2] através de adaptadores de cenários [Fröhlich and Schröder-Preikschat 2000], que quando combinados com distintos componentes do sistema, formam diferentes arquiteturas de software. Neste contexto, o EPOS implementa um framework que define como os componentes são organizados de forma a gerar um sistema funcional. O framework é realizado por um metaprograma estático em C++, sendo assim executado na compilação do sistema, adaptando os componentes selecionados para coexistir entre eles, aspectos e as aplicações.

Uma visão geral do framework metaprogramado do EPOS é demonstrado na Figura 1. Um cliente (aplicação ou componente) deseja invocar um método de um componente (e.g. Thread, UART, etc). A classe parametrizada `Handle` recebe um componente do sistema como parâmetro. Ela verifica se o objeto foi corretamente criado e encaminha a invocação de método ao elemento `Stub` que verifica se o aspecto de invocação remota está ativo para o componente (o aspecto é selecionado pelo componente através de sua classe *Trait* [Stroustrup 1997]). Se o aspecto não está habilitado, `Stub` herdará o adaptador de cenário do componente, caso contrário herdará o `Proxy` do componente (*Stub<Componente, true>*). Consequentemente, se *Traits<Componente>::remote = false* então `Handle` usa o adaptador de cenário como `Stub`; se verdadeiro, `Handle` usa o `Proxy`.

O elemento `Proxy` é responsável por enviar uma mensagem com a invocação do método do componente para o seu `Agent`. A mensagem contém os identificadores do objeto, método e classe que são usados pelo `Agent` para invocar o método correto, associando-os a uma tabela de métodos. O elemento `Agent` recebe esta mensagem e invoca o método através do `Adapter`. A classe `Adapter` aplica os aspectos suportados pelo `Scenario` antes e depois da chamada real do método. Cada instância do

[2]Como na tradicional programação orientada a aspectos (AOP).

```
1  namespace Application {
2    typedef Handle<System::Componente> Componente;
3  }
```

Figura 3. Exportando um componente para o *namespace* da aplicação.

`Scenario` consulta o *Traits* do componente para verificar quais aspectos estão habilitados, agregando o aspecto de cenário correspondente. Quando um aspecto não é selecionado para o componente, uma implementação vazia é utilizada. Neste caso, nenhum código é gerado na imagem final do sistema, similar a um *aspect weaver* na programação orientada a aspectos tradicional.

A estrutura do framework é totalmente transparente para as aplicações devido ao uso de dois *namespaces*: um para as aplicações e outro para o sistema. Em tempo de compilação, os componentes do sistema são exportados para o *namespace* da aplicação, conforme demonstra a Figura 3. Consequentemente, uma invocação de método da aplicação para um componente do sistema é realizada da mesma maneira, mas ao invés de chamar o método real, a chamada é feita através do framework. Da mesma foram, quando componentes do sistema desejam invocar métodos de componentes reconfiguráveis também devem fazê-lo através do *namespace* da aplicação.

Foi observado neste trabalho que a estrutura `Proxy`/`Agent` do framework cria um nível de indireção entre as invocações de métodos do cliente para os componentes do sistema que possuem o aspecto de invocação remota habilitado, isolando os componentes e os tornando independente de posição na memória. Neste cenário, da perspectiva do cliente, uma invocação remota é realizada sem o conhecimento da localização do componente. Essa isolação pode também ser aplicada para RDS, pois somente o `Agent` é ciente da posição do componente na memória. Diferentemente do aspecto de invocação remota presente no EPOS, este trabalho propõe substituir a invocação de método entre o `Proxy` e `Agent` para uma simples invocação de método entre eles. Sendo assim, o mesmo nível de indireção pode residir no mesmo espaço de endereçamento de um nodo. A Figura 2 exemplifica este cenário na estrutura do framework.

3. EPOS LIVE UPDATE SYSTEM

Para que uma reconfiguração de software seja executada de maneira segura e sem comprometer o sistema como um todo, existem três **requisitos principais** que devem ser satisfeitos [Polakovic and Stefani 2008]:

- **Estado quiescente**: para que uma reconfiguração possa acontecer corretamente, o sistema deve atingir um estado de execução considerado consistente, passível de reconfiguração, chamado de estado quiescente [Bloom and Day 1993, Soules et al. 2003]. Em um ambiente com múltiplas threads, como o EPOS, o estado quiescente de um componente é atingido quando nenhuma das threads está invocando métodos deste componente [Polakovic and Stefani 2008].
- **Transferência de estado**: após o sistema atingir o estado quiescente, o estado (dados) do componente antigo deve ser transferido para o novo componente, se necessário. Isto pode ser realizado através da cópia dos dados privados do antigo componente para o novo, através da criação de um novo objeto passando os dados

do objeto antigo para o construtor do componente e deletando o objeto antigo ou ainda através dos métodos `set` e `get` [Polakovic and Stefani 2008].

- **Ajuste das referências**: após a transferência de estado, as referências que o sistema utilizava para acessar o antigo código do componente devem ser atualizadas para apontarem para o novo código, de forma que a aplicação continue sua execução invocando os métodos corretamente.

Adicionalmente a esses requisitos, sistemas embarcados ainda apresentam sérias limitações em termos de memória e, quando são alimentados por bateria, deve-se respeitar limitações no consumo de energia. Por esta razão, o próprio sistema de reconfiguração deve utilizar o mínimo de recursos possíveis, pois estará compartilhando recursos com a(s) aplicação(ões). Reduzir o número de transferência de dados pela rede e o número de escritas/leituras na memória do sistema são **requisitos desejáveis** para que o sistema consiga realizar as suas atividades por um maior período de tempo. Da mesma forma, reduzir a quantidade de memória utilizada pelo mecanismo de reconfiguração resultará em maior espaço de memória livre para a aplicação.

3.1. Premissas

Em função dos requisitos apresentados na seção anterior, o projeto do ELUS baseou-se nas seguintes premissas:

- A unidade de reconfiguração do sistema é componente. Os componentes são classes que devem ser implementados na linguagem C++ e poderão ser marcados como reconfiguráveis ou não em tempo de compilação. Para aqueles componentes não reconfiguráveis nenhum sobrecusto deverá ser adicionado ao sistema. Somente os componentes que possuem reconfiguração habilitada no sistema são passíveis de atualização.
- O estado quiescente do componente que está sendo reconfigurado deverá ser alcançado antes da sua reconfiguração. Desta forma, a reconfiguração de software será realizada de forma consistente e não comprometerá as atividades realizadas pela aplicação.
- Não são permitidas mudanças na API do sistema. As assinaturas dos métodos dos componentes devem permanecer as mesmas em ambas versões. Esta característica restringe o escopo de reconfiguração, mas por outro lado, devido a resolução das dependências entre os componentes e a aplicação e pelas otimizações realizadas pelo compilador na compilação do sistema, o código gerado apresentará um bom desempenho.
- O hardware do sistema não sofre mudanças. Tanto a versão que está sendo reconfigurada quanto a versão antiga executam sobre a mesma plataforma.
- É tarefa do desenvolvedor garantir que o novo componente não remova nenhum método que está sendo utilizado por outros componentes. O ELUS não será responsável por verificar dependências de chamadas de métodos entre componentes do sistema.
- Os métodos reconfiguráveis do componente deverão ter sido declarados como virtuais. Ao declarar um método como virtual, o compilador irá gerar uma tabela de métodos virtuais (`vtable`) que contém os endereços dos métodos do componente. A `vtable` será utilizada para ajustar as referências para os métodos após uma reconfiguração.

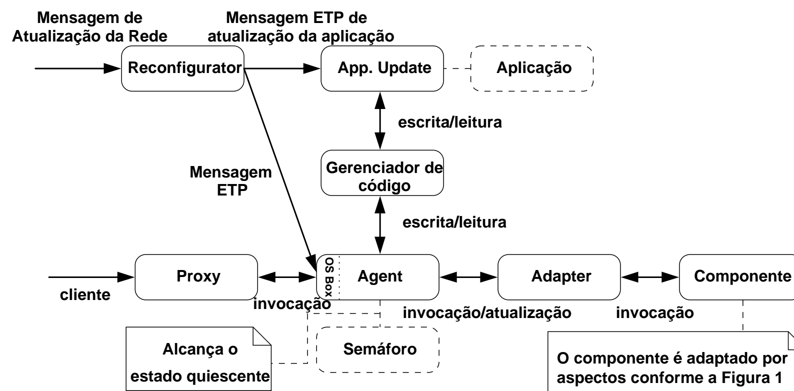


Figura 4. Arquitetura do ELUS.

- Os atributos de um componente reconfigurável deverão ser acessados exclusivamente através dos métodos `set` e `get` relacionados ao atributo.

3.2. Arquitetura

A arquitetura do ELUS é apresentada na Figura 4. O framework original foi estendido para suportar reconfiguração de software. O elemento `Agent` é o único ciente da posição dos componentes em memória. Este elemento possui uma "OS Box" que armazena as mensagens enviadas pelo `Proxy` e controle o acesso aos métodos do componente através de um sincronizador (Semáforo) para cada componente, evitando que um método de um componente que esteja sendo reconfigurado seja chamado ao mesmo tempo. Essa "Box" também funciona como um ponto de acesso, chamando os métodos do componente no `Agent` através de uma tabela de métodos. O `Agent` encaminha a chamada ao `Adapter` que invoca os métodos do componente através da `vtable` do objeto, após recuperá-lo em uma tabela hash.

O framework é um metaprograma, no qual todas as suas dependências entre um componente e os cenários de execuções são resolvidas durante a compilação do sistema. Portanto, o `Proxy` é dissolvido no código do cliente pelo compilador. Desta forma, chamadas do `Proxy` ao `Agent` são realizadas diretamente do cliente para a "OS Box". No código final gerado, os elementos `Proxy`, `Handle` e `Stub` não existem.

A Figura 5 mostra os novos elementos `Proxy` e `Agent` do framework do ELUS. Eles só estarão presentes se a reconfiguração estiver habilitada para um componente. A classe `Proxy` realiza os métodos do componente e usa a classe `Message` para encaminhar uma invocação de método ao `Agent`. Quando o `Proxy` recebe um pedido do `Handle`, ele preenche a mensagem com os IDs do método e componente e com os parâmetros do método. A classe `Message` possui uma interface para anexar e recuperar parâmetros e valor de retorno (métodos *in* e *out*). O método *invoke* da classe chama a função OS BOX, na qual usa um vetor (*Dispatcher*) para entregar as chamadas ao `Agent`. O `Agent`, por sua vez, reconfigura o componente através de uma chamada ao método **update** que utiliza um gerenciador de código para realizar escritas e leituras na memória de programa. Uma Thread, chamada de RECONFIGURATOR, criada na inicialização do sistema, recebe um pedido de reconfiguração e o novo código do componente por uma interface de rede (RS-232 ou rádio por exemplo). Esse pedido é enviado para o `Agent` através da "OS Box" em uma mensagem usando um formato específico, chamado de

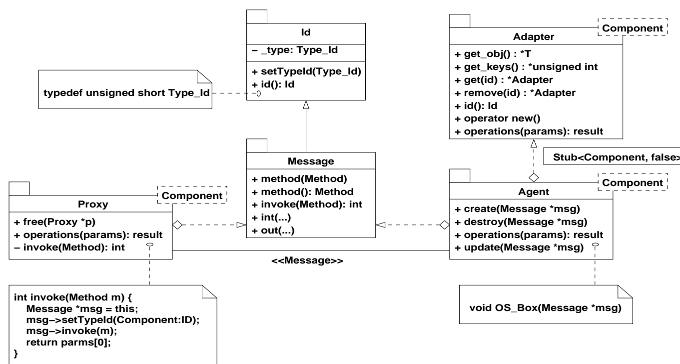


Figura 5. Elementos `Proxy`, `Agent`, `Adapter` e `Message` do framework do ELUS.

Controle	Tam Total	Tam Antigo	Código	Tam Met1	Tam Met2

Figura 6. Mensagem de atualização de componente.

Controle	Tamanho	Código	Endereço	Endereço

Figura 7. Mensagem de atualização de aplicação.

ELUS TRANSPORT PROTOCOL (ETP). Um novo componente ao ser atualizado é compilado e ligado com a imagem gerada do sistema e seu código enviado pela rede. Este novo código do componente é ligado usando o endereço da função OS BOX, na qual é o ponto de entrada para a estrutura do framework. Consequentemente, esta função é o único elemento do sistema que **não é** reconfigurável.

O ETP define 6 tipos de mensagens de reconfiguração: (a) adição de método em um componente; (b) remoção de método de um componente; (c) atualização de um componente; (d) atualização de um endereço de memória específico; (e) atualização da aplicação; e (f) adição de atributo. As Figuras 6 e 7 exemplificam as mensagens de atualização de componente e aplicação. O campo de controle identifica qual o tipo da mensagem. Um componente especial (*Application Update*) foi criado para atualizar a aplicação e reiniciar o sistema após a sua atualização.

4. Avaliação

A avaliação do ELUS considerou três métricas: consumo de memória, sobrecusto nas chamadas de métodos introduzido pelo nível de indireção e `vtable` e tempo de reconfiguração. Foi usado o compilador GNU g++ 4.0.2 e a ferramenta GNU *objdump* 2.16.1 para gerar o sistema e analisar as métricas. A plataforma utilizada na avaliação foi o Mica2, composto de um microcontrolador ATMega128, com 4KB de RAM, 128KB de Flash, 4KB de EEPROM e um conjunto de periféricos incluindo comunicação via rádio.

4.1. Consumo de Memória

Na primeira avaliação, o suporte à RDS foi habilitado somente para o componente *Chronometer* composto por 8 métodos atualizáveis [Gracioli et al. 2008]. O objetivo foi medir o sobrecusto de memória associado à infra-estrutura do framework. A Tabela 4.1 apresenta o consumo de memória dos elementos do ELUS. O framework necessitou de cerca de 2.6KB de memória de código, 42 bytes de dados para o *Dispatcher* e atributos de controle e 52 bytes para dados não-inicializados para tabela hash. RECONFIGURATOR precisou de 70 bytes de dados para o buffer de recebimento de dados via rede (40 bytes) e variáveis de controle. APPLICATION UPDATE usou 210 bytes de memória de código. Para esta configuração, o total de memória utilizada foi 3.5KB de código e 148 bytes de dados.

Tabela 1. Consumo de memória do ELUS.

ELUS Elementos	Tamanho Seção (bytes)			
	.text	.data	.bss	.bootloader
Reconfigurator	410	0	70	0
Code Manager	16	0	2	375
App. Update	210	0	0	0
OS Box	76	0	0	0
Framework	2620	43	52	0
Total	3452	43	124	375

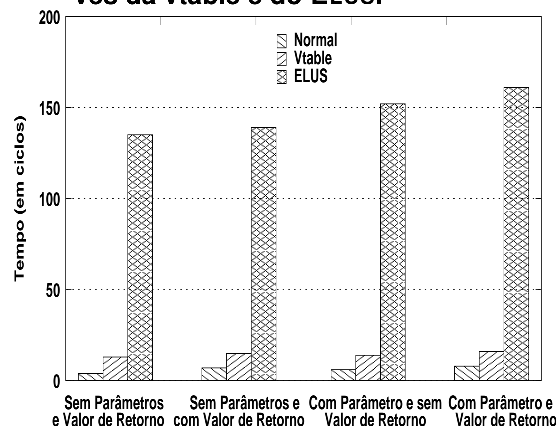
Tabela 2. Tempo de reconfiguração para um componente.

	Tempo (em ciclos)	
	Atualização mesma posição	Atualização nova posição
Agent Update	238	290
Reconfigurator	65	65
OS Box	59	59
Total	362	414

Figura 8. Consumo de memória dos métodos individuais do framework do ELUS.

Método do Framework	Tamanho da seção (bytes)	
	.text	.data
Create	180	0
Destory	138	0
Método sem parâmetro e valor de retorno	94	0
Método com um parâmetro e sem valor de retorno	98	0
Método sem parâmetro e com valor de retorno	112	0
Método com um parâmetro e valor de retorno	126	0
Update	1250	0
Dispatcher	0	2 X (n. de métodos)
Semaphore	0	18
Tamanho mínimo	1662	26

Figura 9. Comparação dos tempos de invocação de método entre uma invocação normal, através da vtable e do ELUS.



O segundo teste de memória avaliou o consumo dos métodos individuais do framework para um componente genérico. Este teste considerou somente o consumo de memória do framework, desprezando a memória necessária pelo componente. O objetivo foi medir o sobrecusto adicionado quando um componente é marcado como reconfigurável. A Figura 8 apresenta os valores dessa avaliação. O total mínimo com o construtor, destrutor, o método de atualização e um método sem parâmetro e valor de retorno para um componente foi de 1.6KB de código e 26 bytes de dados.

4.2. Tempo de Invocação de Método

O nível de indireção entre uma aplicação e um componente cria um sobrecusto de invocação. Antes da chamada real do método, o `Agent` deve recuperar os argumentos da mensagem, o objeto associado ao componente em uma tabela hash e finalmente chamar o método através da sua `vtable`. A Figura 9 apresenta uma comparação entre os tempos de invocação de métodos de um componente normal, através de uma `vtable` e através do ELUS, usando 4 tipos de métodos. O desempenho do ELUS foi cerca de 10 vezes pior que a `vtable`. Como será visto na Seção 6, esse desempenho é melhor que os trabalhos relacionados.

4.3. Tempo de Reconfiguração

A Tabela 2 apresenta os tempos de reconfiguração em 2 cenários de atualização: (i) quando o novo código de um componente é menor ou igual ao antigo, sendo assim uma

atualização na mesma posição do componente e (ii) quando o novo código de um componente é maior que o antigo, identificando uma atualização em uma nova posição. Por motivos de comparação com os trabalhos relacionados, este teste não considerou o tempo de recebimento de dados pela rede no RECONFIGURATOR e o tempo de escrita dos dados na flash. Foram considerados os tempos de chamada do RECONFIGURATOR para a "SO Box", o tempo na "SO Box" para acessar o *Dispatcher* e chamar o método *update* do `Agent` e o tempo para o `Agent` recuperar os dados e executar uma reconfiguração.

O RECONFIGURATOR consome 65 ciclos do microcontrolador para chamar a "SO Box". A "SO Box" consome 59 ciclos para atingir o estado quiescente (operações *p()* e *v()* no semáforo) e chamar o método *update* do `Agent`. Finalmente, o método de atualização gasta 238 ciclos para executar uma atualização no cenário (i) e 290 ciclos no cenário (ii).

5. Trabalhos Relacionados

TINYOS é considerado o SO mais popular para RSSF. É um SO orientado a eventos e originalmente não suporta reconfiguração de software [Hill et al. 2000]. Entretanto, diversos trabalhos têm sido realizados com o intuito de suportar reconfiguração no TINYOS [Levis and Culler 2002, Hui and Culler 2004, Marrón et al. 2006]. MANTISOS é outro SO para RSSF muito conhecimento no ambiente acadêmico, mas não suporta reconfiguração [Abrach et al. 2003].

NANO-KERNEL permite RDS das aplicações e dos componentes do kernel através da separação dos dados e dos algoritmos lógicos do kernel [Bagchi 2008]. É criado um nível de indireção entre as aplicações e os dispositvos do kernel (e.g. escalonador, gerenciador de memória, etc). O núcleo do NANO-KERNEL e os dispositivos do kernel se comunicam através de interfaces específicas que devem ser iniciadas na inicialização do sistema.

RETOS implementa RDS através de relocação dinâmica de memória e ligação em tempo de execução [Cha et al. 2007]. O processo de relocação extrai informações de variáveis e funções globais em tempo de compilação (meta-informações) que são colocadas em um arquivo no formato RETOS. Tais informações são utilizadas pelo kernel para substituir todo endereço acessível de um módulo quando o estiver carregando. O SO possui uma tabela com endereços para funções de outros módulos. Um módulo registra, desregistra e acessa funções através desta tabela.

CONTIKI é um SO para RSSF que implementa processos especiais, chamados de *serviços*, que provêem funcionalidades a outros processos [Dunkels et al. 2004]. Serviços são substituídos em tempo de execução através de uma *interface stub* responsável por redirecionar as chamadas das funções para uma *interface de serviço*, que possui ponteiros para as implementações atuais das funções do serviço correspondente. CONTIKI somente permite a atualização de algumas partes do sistema.

SOS é um SO para RSSF que permite RDS [Han et al. 2005]. O SO é construído em módulos que são inseridos, removidos ou substituídos em tempo de execução. Através do uso de chamadas relativas, o código em cada módulo torna-se independente de posição da memória.

ELUS é conceitualmente similar aos trabalhos relacionados apresentados, porém a infra-estrutura do framework possui a vantagem de eliminar parte do sobrecusto associado

às tabelas de indireção com o uso da metaprogramação estática. Tabela 3 revisa o processo de reconfiguração nos SO embarcados analisados.

Tabela 3. Características do processo de reconfiguração nos SOs analisados.

SO	Processo de Reconfiguração
TINYOS	Sem suporte direto
MANTISOS	Não há suporte
NANO-KERNEL	Módulos reconfiguráveis
RETOS	Relocação dinâmica e ligação em tempo de execução
CONTIKI	Módulos reconfiguráveis
SOS	Módulos reconfiguráveis
EPOS/ELUS	**Componentes reconfiguráveis selecionados em tempo de compilação**

6. Análise Comparativa

Esta seção discute os resultados do ELUS comparando-os aos trabalhos relacionados. O ELUS adiciona cerca de 1.6Kb de código e 26 bytes de dados por componente no sistema quando o suporte de reconfiguração está habilitado. CONTIKI necessita de 6Kb de código e 230 bytes de dados [Dunkels et al. 2004]. SOS ocupa mais de 5Kb de código para o gerenciamento dos módulos e a tabela de módulos ocupa 230 bytes de dados. ELUS apresenta bom desempenho devido a metaprogramação estática, na qual resolve as dependências entre o framework, componentes e aplicações em tempo de compilação. Além disso, ELUS permite que o desenvolvedor escolha os componentes reconfiguráveis sem introduzir sobrecusto na imagem final do sistema, gerando somente código e dados que o sistema realmente precisa.

A respeito do sobrecusto de invocação de métoso, SOS leva 21 ciclos para chamar uma função de um módulo reconfigurável e 12 para chamar uma função do kernel. Entretanto, a comunicação entre módulos (troca de dados) é realizada através do envio e recebimento de mensagens em um buffer, que leva cerca de 822 ciclos [Han et al. 2005]. CONTIKI tem um desempenho ainda pior, pois o *stub* deve encontrar a interface de serviço antes de chamar a função do módulo reconfigurável. Isso é feito comparando uma cadeia de caracteres [Dunkels et al. 2004]. Em uma comparação, CONTIKI foi cerca de 4 vezes mais lento que o SOS [Yi et al. 2008]. ELUS tem a vantagem de usar a estrutura de mensagem presente no framework para passar e receber parâmetros e valor de retorno, na qual é 7 vezes mais rápida que o SOS.

Em uma reconfiguração, o SOS remove o módulo antigo e registra o novo módulo e seu tratador de eventos. Tais tarefas demoram 612 cilcos, desconsiderando o tempo de receber os dados pela rede e escrevê-los na memória flash [Yi et al. 2008]. CONTIKI inicia uma reconfiguração enviando uma mensagem de remoção para a interface de serviço, que faz a transferência dos dados entre os módulos antigo e novo. Após, a nova interface é registrada no sistema. RETOS usa relocação dinâmica e ligação em tempo de execução através das meta-informações sobre os módulos. Essas informações extras também devem ser transmitidas juntamente com o novo código de um módulo, aumentando a quantidade de dados e o tempo de transmissão pela rede, o consumo de energia e o tempo de reconfiguração. O ELUS não necessita registrar ou remover os componentes em tempo de execução. Uma reconfiguração demora cerca de 415 ciclos. Além disso, ELUS utiliza o ETP que diminui os dados enviados pela rede em um pedido de reconfiguração.

7. Considerações Finais

Este artigo apresentou o ELUS, uma infra-estrutura de SO para sistemas embarcados com recursos limitados. O ELUS foi desenvolvido em torno do framework de componentes do EPOS, que foi modificado para habilitar o confinamento e isolação dos componentes do sistema em módulos físicos, tornado-os assim independentes de posição de memória. O ELUS também permite que o desenvolvedor, escolha em tempo de compilação, quais serão os componentes reconfiguráveis. Para todos os componentes marcados como não reconfiguráveis nenhum sobrecurso é adicionado na imagem final do sistema. Além disso, o ELUS é totalmente transparente para as aplicações e, usando o ELUS TRANSPORT PROTOCOL, a RDS se torna fácil de integrar com outros protocolos, tal como um protocolo de disseminação de dados. A avaliação mostrou que o ELUS consome menos memória, adiciona menos sobrecusto e tem melhor tempo de reconfiguração quando comparado com os trabalhos relacionados.

Referências

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. (2003). Mantis: system support for multimodal networks of in-situ sensors. In *Proc. of the 2nd ACM int. conf. on Wireless sensor networks and applications*, pages 50–59, NY, USA. ACM.

Bagchi, S. (2008). Nano-kernel: a dynamically reconfigurable kernel for wsn. In *Proceedings of the 1st MOBILWARE*, pages 1–6, ICST, Brussels, Belgium, Belgium.

Bloom, T. and Day, M. (1993). Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. (2007). Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157, New York, NY, USA. ACM.

Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA.

Felser, M., Kapitza, R., Kleinöder, J., and Preikschat, W. S. (2007). Dynamic software update of resource-constrained distributed embedded systems. In *International Embedded Systems Symposium 2007 (IESS '07)*.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Fröhlich, A. A. and Schröder-Preikschat, W. (2000). Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th WMSCI*, Orlando, U.S.A.

Gracioli, G., Santos, D., de Matos, R., Wanner, L., and Fröhlich, A. A. (2008). One-shot time management analysis in epos. In *Proc. of the XXVII SCCC*, Punta Arenas, Chile. IEEE.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). 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. ACM.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J. (2000). System architecture directions for networked sensors. In *Proc. of the IX ASPLOS, Cambridge, MA, USA*, pages 93–104.

Hui, J. W. and Culler, D. (2004). The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA. ACM.

Koshy, J. and Pandey, R. (2005). Vm*: Synthesizing scalable runtime environments for sensor networks. In *In Proc. SenSys 05*, pages 243–254. ACM Press.

Levis, P. and Culler, D. (2002). Mate: A tiny virtual machine for sensor networks. In *Proc. of the X ASPLOS, San Jose, CA, USA*.

Marrón, P. J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., and Rothermel, K. (2006). Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227.

Polakovic, J. and Stefani, J.-B. (2008). Architecting reconfigurable component-based operating systems. *Journal of Systems Architecture*, 54(6):562–575.

Pottie, G. J. and Kaiser, W. J. (2000). Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58.

Reijers, N. and Langendoen, K. (2003). Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA. ACM Press.

Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., Silva, D. D., Ganger, G. R., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003). System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley, 3 edition.

Xie, Q., Liu, J., and Chou, P. (2006). Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In *Proc. Fifth International Conference on Information Processing in Sensor Networks IPSN 2006*, pages 342–349.

Yi, S., Min, H., Cho, Y., and Hong, J. (2008). Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems. *Comput. Commun.*, 31(4):699–707.