Uma Estrutura de Reprogramação em Rede para Sistemas Operacionais Embarcados

Rodrigo Steiner, 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

{rodrigo,giovani,guto}@lisha.ufsc.br

Resumo. *Este artigo apresenta uma estrutura de reprogramação de software em rede para sistemas embarcados. Tal estrutura foi desenvolvida no sistema operacional EPOS e é formada por um protocolo de disseminação de dados e um ambiente de suporte de sistema operacional que isola os componentes do sistema em unidades físicas independentes de posição de memória, permitindo que sejam atualizados em tempo de execução. A estrutura foi testada em nodos reais e avaliada em termos de consumo de memória, tempo de disseminação e de reprogramação.*

Abstract. *This paper presents a software network reprogramming structure for embedded systems. Such a structure was developed in the EPOS operating systems and it is composed by a data dissemination protocol and an operating system support environment that isolates the system components in memory position independent physical units, allowing their updating at execution time. The structure was tested in real nodes and evaluated in terms of memory consumption, dissemination and reprogramming time.*

1. Introdução

Reprogramar o software de um programa em execução é uma tarefa presente na grande maioria dos ambientes computacionais. A gama de aplicações que utiliza algum meio de reprogramação varia desde browsers para internet à sistemas dedicados, como controladores em um veículo, por exemplo. Devido às características especiais destes sistemas dedicados (e.g. limitação de recursos), a estrutura de reprogramação de software é diferente daquela presente nos ambientes computacionais convencionais. Além disso, alguns desses sistemas dedicados, como Redes de Sensores Sem Fio (RSSF), são formados por uma grande quantidade de nodos, onde coletar todos para reprogramá-los é impraticável.

A estrutura de reprogramação de software em um sistema embarcado deve ser composta por um protocolo de disseminação de dados e uma estrutura que organize os dados de forma consistente na memória do sistema. Uma maneira de oferecer isso às aplicações embarcadas é ocultar tal estrutura em um sistema operacional. Geralmente, a estrutura de reprogramação utilizada em um SO embarcado é composta por módulos atualizáveis em tempo de execução. Esses módulos são independentes de posição de memória para as aplicações e podem ser substituídos em tempo de execução [Han et al. 2005, Dunkels et al. 2004].

Não obstante, é essencial que a totalidade dos novos dados de um ou mais módulos cheguem corretamente a todos os nodos envolvidos na reprogramação. Por este motivo, um protocolo de disseminação deve ser usado juntamente com a estrutura do SO. De forma simplificada, um protocolo funciona da seguinte maneira: a disseminação começa através de uma estação base responsável por transmitir os dados aos seus nodos vizinhos. Uma vez que um nodo recebe os dados ele passa a ser capaz de retransmiti-los aos seus próprios vizinhos. Assim, se um nodo A tem B como vizinho, e o nodo B tem A e C como vizinhos, ao receber os dados de A o nodo B vai retransmiti-los para C. O processo se repete para todos os nodos, de forma que a rede inteira receba os dados [Thanos Stathopoulos 2003, Hui and Culler 2004].

Este artigo apresenta o projeto e implementação de uma estrutura de reprogramação de software eficiente para sistemas embarcados transparente para aplicações. A estrutura é implementada no EMBEDDED PARALLEL OPERATING SYSTEM (EPOS) [Fröhlich 2001] e é composta por um sistema que isola os componentes do sistema em unidades físicas da memória e um protocolo de disseminação de dados que garante a entrega dos novos dados para os nodos envolvidos na reprogramação. A estrutura foi avaliada em termos de consumo de memória, tempo disseminação e de reprogramação, apresentando bons resultados.

O restante deste artigo está organizado da seguinte maneira. A seção 2 apresenta os principais trabalhos relacionados a protocolos de disseminação e sistemas operacionais que suportam reprogramação. A seção 3 mostra o projeto do protocolo de disseminação usado neste trabalho e compara suas características com os protocolos relacionados. A seção 4 apresenta a integração do protocolo de disseminação com a estrutura do SO. A avaliação da estrutura de reprogramação completa é realizada na seção 5 e finalmente, a seção 6 conclui o artigo.

2. Trabalhos Relacionados

2.1. Protocolos de Disseminação

Multi-hop Over the Air Programming é um mecanismo de distribuição de código projetado priorizando o consumo de energia e memória em detrimento da latência [Thanos Stathopoulos 2003]. O MOAP utiliza um mecanismo de disseminação chamado *Ripple*, que distribui os dados de vizinhança em vizinhança. Em cada vizinhança apenas um pequeno subconjunto de nodos (de preferência apenas um) funcionam como emissores, enquanto os restantes são receptores. Quando os nodos recebem todos os dados eles podem se tornar emissores para seus próprios vizinhos (que estavam fora do alcance do emissor original). Para evitar que nodos se tornem emissores em uma vizinhança que já possua um, o mecanismo utiliza uma interface divulga/inscreve, nodos emissores divulgam sua versão e todos os interessados se inscrevem. Caso um emissor não receba inscrições ele fica em silêncio.

Deluge é um protocolo de disseminação projetado para propagar uma grande quantidade de dados de forma rápida e confiável. Ele compartilha várias ideias com o MOAP, como o uso de NACKs, pedidos de retransmissão *unicast* e transmissão de dados *broadcast* [Hui and Culler 2004]. Com o intuito de limitar a quantidade de informações que um receptor deve manter, possibilitar atualizações incrementais e permitir que os nodos continuem a disseminação antes de possuírem todos os dados, o protocolo utiliza o

conceito de páginas. Os dados são divididos em *P* páginas, sendo que uma página nada mais é do que um conjunto de *N* pacotes. Utilizando um vetor de idades para descrever a idade de cada página, os nodos são capazes de determinar quando uma página mudou e se necessitam ou não requisitá-la. Exigindo que os nodos recebam uma página por vez, pode-se utilizar um mapa de bits de apenas *N* bits para gerenciamento de segmentos, pois não é mais necessário manter registros de todos os pacotes ao mesmo tempo.

Multi-hop Network Programming (MNP) é um protocolo de reprogramação em rede cujas principais características incluem um mecanismo para seleção de emissor e uma abordagem para reduzir o uso da memória RAM [Wang 2004]. Assim como no MOAP a disseminação ocorre de vizinhança em vizinhança e um nodo só pode se tornar emissor após receber todos os dados. Através do algoritmo para seleção de emissor, um nodo decide se deve transmitir o código ou não. O objetivo deste algoritmo é o de garantir que a qualquer momento apenas um nodo esteja transmitindo os dados por vez, e que este transmissor seja o que vai causar maior impacto, em outras palavras, o que tiver um maior número de receptores. É importante ressaltar que o algoritmo não garante encontrar o emissor ideal, todavia, ele seleciona "bons" emissores e reduz o número de colisões.

Infuse é um protocolo de disseminação de dados baseado em uma comunicação sem colisões devido ao uso do MAC (*Medium Access Control*) TDMA (*Time Division Multiple Access*) [Arumugam 2004]. Este protocolo requer que os nodos conheçam tanto a sua localização como a de seus vizinhos, classificando-os em predecessores e sucessores. Assim um nodo ouve durante a faixa de tempo de seus predecessores para receber os dados e transmite durante a sua. A Tabela 1 apresenta as prioridades dos protocolos analisados.

2.2. Sistemas Operacionais Embarcados

Alguns SOs embarcados são projetados com uma abstração para atualização de software. Através desta abstração é possível realizar reprogramações em tempo de execução sem a necessidade de reiniciar o sistema, desta forma, evitando perda de dados.

TINYOS é um sistema operacional constituído de componentes reutilizáveis que são usados em conjunto formando uma aplicação específica [Levis et al. 2005]. Este SO suporta uma ampla gama de plataformas de hardware e tem sido utilizado em várias gerações de nodos sensores, podendo ser considerado o SO mais usado na área de RSSF. Ele apresenta um modelo de concorrência orientado a eventos e originalmente não suporta reconfiguração de software. Contudo, todos os protocolos apresentados anteriormente foram implementados utilizando o TINYOS, desta forma possibilitando a reconfiguração.

SOS é um sistema operacional constituído por módulos dinamicamente carregáveis e um *kernel* [Han et al. 2005]. Esses módulos enviam mensagens e se comunicam com o *kernel* através de uma tabela do sistema que contém *jumps* relativos. Desta forma o código em cada módulo torna-se independente de posição da memória, possibilitando alterações no software de maneira mais eficiente.

CONTIKI é um SO que possui uma estrutura de reconfiguração semelhante a do SOS. Ele implementa processos especiais, chamados *serviços*, responsáveis por prover funcionalidades a outros processos [Dunkels et al. 2004]. Esses serviços podem ser 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. A Tabela 2 resume o processo de reconfiguração nos SOs analisados.

Tabela 1. Prioridades dos protocolos analisados.

Protocolo	Energia	Latência	Memória
MOAP	1º	3º	2º
Deluge	3º	1º	2º
MNP	2º	3º	1º
Infuse	1º	2º	3º

Tabela 2. Processo de reconfiguração nos SOs analisados.

SO	Processo de Reconfiguração
TINYOS	Sem suporte direto.
SOS	Módulos reconfiguráveis.
CONTIKI	Módulos reconfiguráveis.

3. Reprogramação em Rede

Em geral o processo de reprogramação em rede é dividido em três etapas, como ilustrado na Figura 1. A primeira é responsável pela preparação dos dados a serem disseminados. A segunda etapa engloba todo o processo de disseminação, onde os dados são enviados e armazenados pelos nodos pertencentes à rede. Por fim, o mecanismo de reconfiguração do SO interpreta os dados recebidos e os utiliza para atualizar a memória de programa.

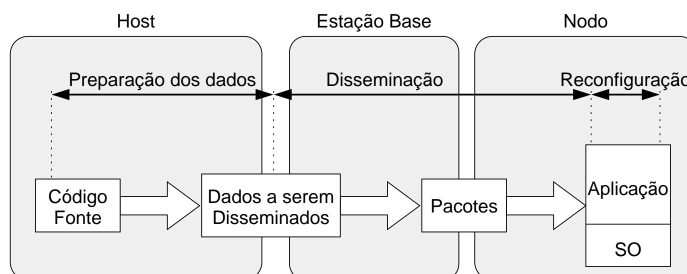

Figura 1. Processo de programação em rede.

3.1. Protocolo de Disseminação

Protocolos de disseminação de dados são utilizados para propagar dados pela rede utilizando seus próprios nodos para isso. Em especial, um protocolo utilizado por um mecanismo de reprogramação em rede deve ser confiável, ou seja, garantir a entrega correta de todos os dados a todos os nodos. Abaixo, as propriedades que devem ser levadas em conta ao se projetar um protocolo de disseminação [Lanigan et al. 2005]:

Baixa Latência: como a atualização é um serviço secundário o protocolo não deve interromper a aplicação principal por muito tempo.

Baixo Consumo de Memória: os dados necessários para a atualização devem ser armazenados até que a transmissão termine, entretanto o protocolo deve requisitar pouco espaço de armazenamento de forma a não restringir a quantidade de memória disponível para a aplicação principal.

Confiabilidade: ao contrário de algumas aplicações tradicionais onde a perda de um pacote é tolerável devido ao fato de que os dados são redundantes e correlacionados, na reprogramação cada pacote é crucial e todos devem ser recebidos para que a atualização possa ocorrer. Sendo assim o protocolo deve possuir uma política de retransmissão permitindo a recuperação de pacotes perdidos.

Eficiência Energética: o protocolo deve minimizar seu consumo de energia de forma a não diminuir severamente o tempo de vida do nodo.

Tolerância a Inclusão/Remoção de nodos: é possível que um nodo falhe durante um período de tempo e depois volte a funcionar, ou até mesmo que novos nodos sejam incluídos na rede. Desta forma a disseminação não deve ser severamente afetada pela inclusão ou remoção de nodos.

Uniformidade: para garantir que a rede inteira seja atualizada, todos os dados devem ser entregues a todos os nodos da rede. Nodos incluídos na rede durante ou depois de uma atualização também devem ser capazes de receber os dados da atualização.

As propriedades de **confiabilidade** e **uniformidade** são obrigatórias, uma vez que garantem o funcionamento correto do protocolo. Já as propriedades de baixa latência, baixo consumo de memória, eficiência energética e tolerância a inclusão ou remoção de nodos são apenas desejáveis, pois não garantem corretude. Entretanto um protocolo que as ignore seria de pouca utilidade na prática [Thanos Stathopoulos 2003].

3.1.1. Características de um Protocolo

A Figura 2 apresenta o diagrama de características de um protocolo de disseminação de dados. Este tipo de diagrama possibilita caracterizar as propriedades de um determinado conceito, apresentando seus pontos de variação [Czarnecki and Eisenecker 2000]. As características são representadas como nodos de uma árvore, cuja raiz é o conceito, e só estão presentes se seu nodo pai está presente. Características obrigatórias e opcionais são representadas por um círculo no final de suas arestas, preenchido e vazio respectivamente. Características alternativas são conjuntos do qual apenas uma característica pode estar presente e são representadas por um arco ligando suas arestas.

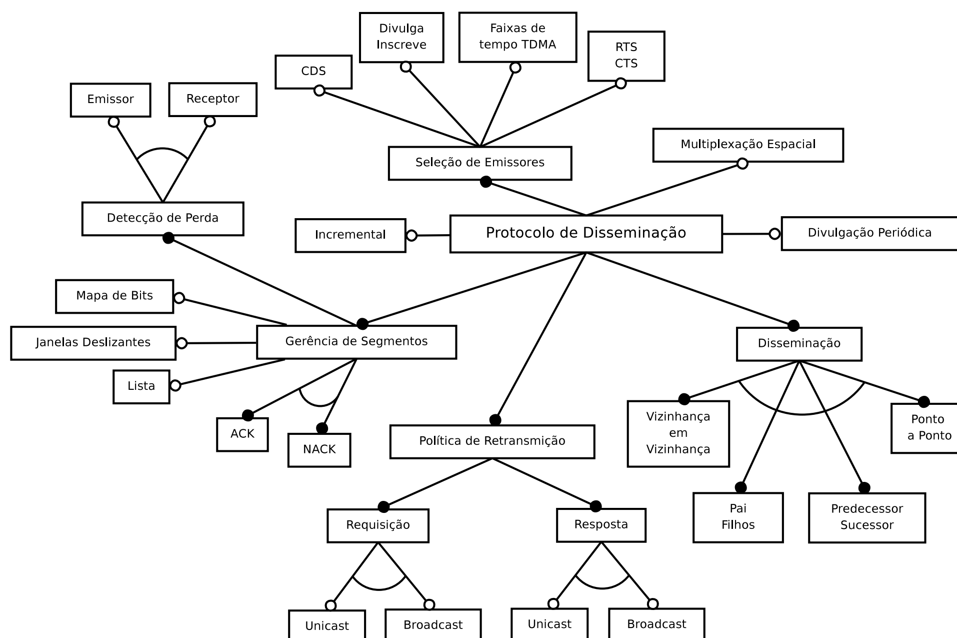

Figura 2. Diagrama de características.

Incremental: um protocolo com esta característica envia somente as diferenças entre os novos dados e os antigos. Desta forma diminui-se a quantidade total de dados a serem transmitidos, consequentemente diminuindo o consumo de energia.

Disseminação: a forma pela qual os dados são propagados pela rede.

Gerência de Segmentos: mecanismo utilizado para detectar perdas de pacotes.

Seleção de emissor: a forma como o protocolo decide quais nodos se tornarão emissores pode diminuir tanto o número total de colisões quanto o de mensagens transmitidas.

Política de Retransmissão: a forma em que são feitas as requisições por pacotes perdidos e retransmissões.

Divulgação Periódica: um protocolo com esta característica requer que todos os nodos divulguem suas versões periodicamente, possibilitando a nodos que de alguma forma perderam a operação de disseminação receberem os dados necessários para a reprogramação.

Multiplexação Espacial: um protocolo com esta característica não exige que os nodos recebam todos os dados para tornarem-se emissores, desta forma possibilitando que os dados sejam transmitidos em paralelo por toda a rede.

3.1.2. Escolhas de Projeto

Como algumas propriedades desejáveis entram em conflito com outras, os protocolos existentes realizam escolhas de projetos dando preferência a umas em detrimento de outras. Quanto as escolhas realizadas no protocolo desenvolvido neste trabalho:

1. A propriedade não obrigatória considerada mais importante foi a de eficiência energética, uma vez que todas as operações realizadas necessitam de energia e, em muitos sistemas embarcados, há apenas uma quantidade finita disponível.
2. Consumo de memória foi a segunda propriedade considerada mais importante, visto que o protocolo de disseminação não é a aplicação principal do nodo, mas apenas um serviço oferecido pelo sistema operacional. Desta forma, não se deve limitar a quantidade de memória disponível para as aplicações.
3. Por fim, a latência. Para poder otimizar o consumo de energia e memória algumas propriedades que diminuiriam a latência não foram utilizadas (e.g. multiplexação espacial).

3.1.3. Implementação

A Figura 3 apresenta a máquina de estados do protocolo desenvolvido. Ele distribui os dados de vizinhança em vizinhança, utiliza um mecanismo de seleção de emissores baseado no MNP (divulga / inscreve), responsabiliza os receptores por detectar perdas, realiza requisições *unicast* e retransmissões *broadcast* e utiliza o mecanismo de janelas deslizantes para gerência de segmentos.

Nodos divulgam suas versões, periodicamente, e todos os interessados a requisitam. Um potencial emissor mantêm uma variável *ReqCtr*, inicializada com zero, e a incrementa para cada nova requisição recebida, destinada a ele, vinda de um nodo ainda não computado. As mensagens de divulgação tem duas funções: anunciar uma nova versão e

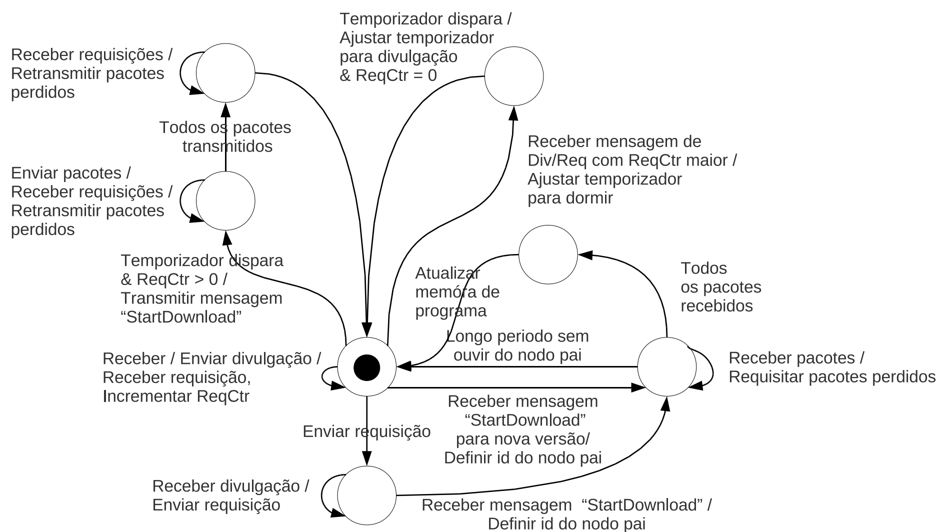

Figura 3. Máquina de estados do protocolo desenvolvido.

prevenir que nodos com menos requisições virem emissores; elas possuem o número da versão, o *id* do emissor e sua variável *ReqCtr*. Quando um nodo recebe uma mensagem de divulgação que contenha uma nova versão, ele irá enviar uma requisição *broadcast* contendo seu *id*, o do transmissor e o valor da *ReqCtr* recebida. Como as divulgações e requisições são *broadcasts* outros nodos que estão na disputa para se tornarem emissores também as recebem e caso possuam um *ReqCtr* menor vão para o estado *sleep*. Como critério de desempate é utilizado o *id* dos nodos.

Ao virar emissor um nodo transmite uma mensagem *broadcast "StartDownload"* e passa a enviar os dados, pacote por pacote. Os receptores definem este nodo como seu "pai" e só aceitam pacotes vindo dele. Cada pacote possui um identificador único sequencial e os receptores mantêm o número do último pacote recebido. Assim, ao receber um novo pacote é verificado se há uma lacuna entre estes dois números e os pacotes intermediários são considerados perdidos. Ao detectar uma perda, o receptor envia um pedido de retransmissão para o emissor, utilizando um pacote *unicast*. Os pedidos de retransmissão possuem uma prioridade maior que pacotes normais, então um emissor irá primeiro responder a todas as requisições antes de continuar com a transmissão.

4. Integração com o Sistema Operacional

EPOS é um sistema operacional multi-plataforma baseado em componentes, onde serviços tradicionais do SO são implementados através de componentes do sistema independentes de plataforma [Fröhlich 2001]. O suporte a serviços específicos de plataforma é realizado através de mediadores de hardware [Polpeta and Fröhlich 2004]. Mediadores são funcionalidades equivalentes a drivers de dispositivos em plataformas UNIX, mas não são camadas de abstração de hardware tradicionais. Ao contrário, os mediadores fazem uso de interfaces independentes de plataforma para sustentar suas interfaces de contrato entre componentes de hardware. Devido ao uso de metaprogramação estática em C++ e funções *inlining*, o código do mediador é dissolvido nos componentes em tempo de compilação.

Reprogramação no EPOS é suportada através do EPOS LIVE UPDATE SYSTEM

(ELUS) [Gracioli 2009]. O ELUS modificou o framework de componentes do EPOS, mais especificamente, o aspecto de invocação remota [Fröhlich 2001] para suportar também reconfiguração do software. A Figura 4 demonstra a nova estrutura do framework. Ao invés dos elementos `Proxy` e `Agent` estarem em diferentes espaços de endereçamento (e.g. diferentes nodos), ambos estão presentes no mesmo nodo. Desta forma, somente o `Agent` tem conhecimento sobre a posição de memória de um componente, podendo assim atualizar o código e dados deste componente. A Figura também mostra como o processo de reconfiguração é habilitado ou não para um componente. Isso é realizado através da classe *Trait* do componente. Habilitando a opção de reconfiguração (*reconfiguration*) irá adicionar ao sistema em tempo de compilação o suporte à reprogramação ao componente. Somente os componentes habilitados suportam reprogramação. O elemento `Adapter` é usado para adaptar o componente aos diferentes cenários de execuções, aplicando os correspondentes aspectos suportados pelo `Scenario` antes e depois da chamada real do método do componente [Fröhlich and Schröder-Preikschat 2000].

As invocações dos métodos entre o `Proxy` e o `Agent` acontece através de uma função que possui uma tabela de métodos, chamada de `Dispatcher`, que contém os endereços dos métodos do `Agent`. Essa função garante que não aconteça chamadas ao componente enquanto este estiver sendo atualizado utilizando um Semáforo. O `Agent` armazena os objetos do componente em uma tabela hash, e usa a tabela de métodos virtuais do objeto para fazer a chamada ao método real. Ao se atualizar métodos de um componente, basta atualizar os métodos dentro da tabela virtual.

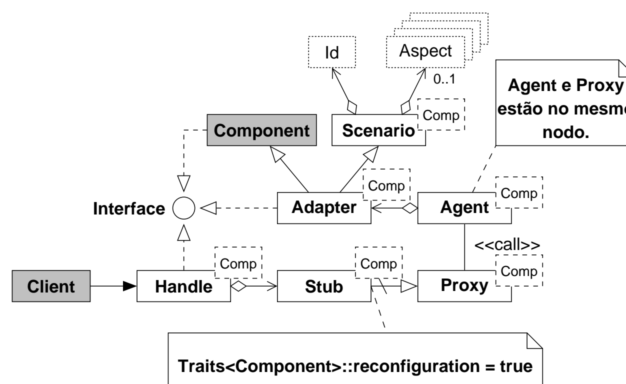

Figura 4. Framework do EPOS modificado para reprogramação de software.

O ELUS recebe requisições de reprogramação de um componente através de um protocolo, chamado de ELUS TRANSPORT PROTOCOL (ETP). A Figura 5 apresenta as mensagens disponíveis pelo ETP. Os 4 bits menos significativos do campo de controle definem o tipo da mensagem e os 4 bits mais significativos definem a quantidade de campos que a mensagem contém, pois isso varia de acordo com o tipo de mensagem. Uma *Thread* criada na inicialização do sistema, chamada de RECONFIGURATOR, cria uma instância do protocolo de disseminação e após o recebimento dos dados, inicia uma reprogramação. A escrita dos dados na memória de programa é abstraída por um gerenciador de código (*Code Manager*).

A mensagem (a) informa uma adição de método a um componente. A mensagem (b) informa que um método está sendo removido de um componente. A mensagem (c) requisita a atualização de todos os métodos do componente. Para isso, além do novo

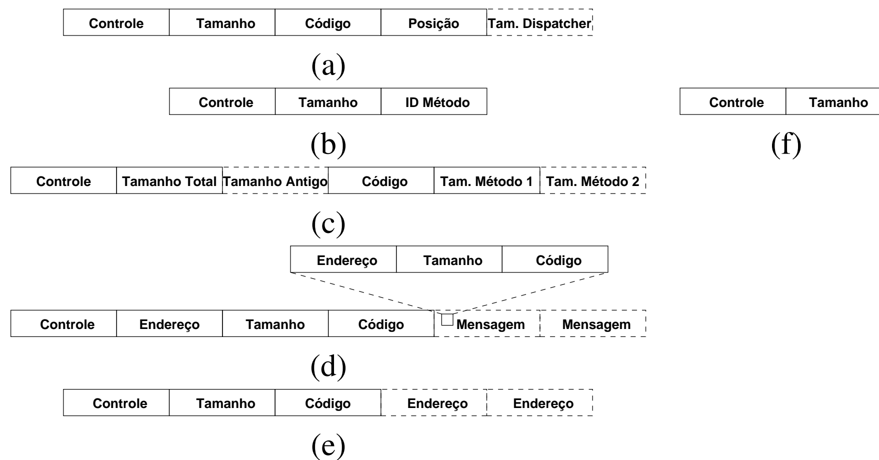

Figura 5. Mensagens de reprogramação do ETP. (a) Adição de método (b) Remoção de método (c) Atualização do componente (d) Atualização de endereço (e) Atualização da aplicação (f) Adição de atributos.

tamanho do código, são informados o tamanho antigo, o novo código e o novo tamanho dos métodos do componente. Todas essas informações são utilizadas pelo `Agent` para decidir se há necessidade de alocar novo espaço de memória ou se o novo código cabe no espaço antigo. O ETP ainda permite atualizar um endereço específico (d). Várias mensagens podem ser concatenadas, e o número de mensagens é controlado pelo campo de controle. A mensagem (e) informa a atualização de uma aplicação e a mensagem (f) requisita a adição de atributos, enviando para o `Agent` o tamanho do objeto que deve ser criado somando os tamanhos dos atributos antigos com os novos. O `Agent` irá alocar espaço para o novo objeto contando com os novos atributos, transferir o estado (dados) do objeto antigo para o novo e apagar o objeto antigo. Os atributos só podem ser acessados através dos métodos `set` e `get`, por isso uma mensagem de adição de atributos também deve ser seguida por uma mensagem de adição de métodos.

A estrutura de mensagens criada pelo ELUS permite que um protocolo de disseminação de dados seja facilmente integrado ao sistema operacional. O protocolo de disseminação deve, portanto, criar mensagens no formato ETP e realizar uma chamada ao `Agent` informando uma atualização. Essa simples estrutura é um diferencial não encontrado nos trabalhos relacionados, o que torna o processo de atualização simples e abstrai do desenvolvedor detalhes de como a reprogramação acontece efetivamente. Além disso, não é necessário reinicializar o sistema, evitando perda de dados, diferentemente dos protocolos de disseminação apresentados nos trabalhos relacionados.

A Figura 6 apresenta o diagrama de sequência do processo de atualização, demonstrando como é realizada a integração entre o protocolo de disseminação de dados e a estrutura do ELUS. O RECONFIGURATOR inicia o protocolo através da chamada ao método *run*. Este método fica bloqueado até que uma atualização (nova versão) requisitada pelo nodo seja recebida. Após o recebimento dos novos dados, o RECONFIGURATOR cria uma mensagem no formato ETP e passa os dados para o `Agent` através da chamada ao *trapAgent*. Por fim, o `Agent` realiza a escrita dos dados na memória de código.

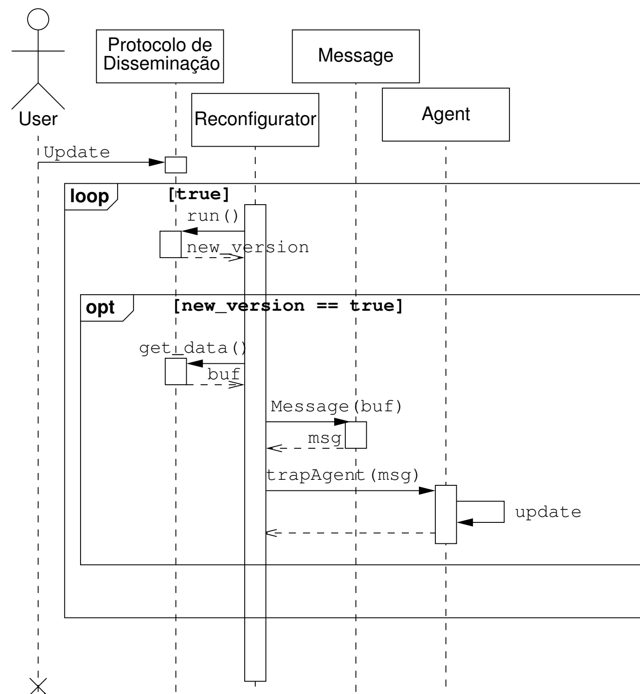

Figura 6. Integração do protocolo de disseminação com a estrutura do ELUS.

5. Avaliação

A estrutura de reprogramação formada pelo ELUS e o protocolo de disseminação foi avaliada pelo consumo de memória, latência de envio de dados para toda a rede e tempo de reconfiguração, este considerando o tempo de recebimento dos dados pela rede e o tempo de escrita na memória de programa. Estes testes foram realizados utilizando-se nodos sensores Mica2 [Crossbow Technology Inc.], que possuem um microcontrolador ATMega128, 4kB de memória ram, 128kB de memória flash, 4kB de EEPROM e comunicação via rádio. O sistema foi gerado com o compilador GNU g++ 4.0.2 e o consumo de memória medido utilizando-se a ferramenta GNU *objdump* 2.16.1. A latência e tempo de reconfiguração foram medidos através do temporizador do microcontrolador.

5.1. Memória

A Tabela 3 apresenta o consumo de memória de todos os elementos da estrutura. Para este teste, o suporte a reconfiguração foi habilitado para um componente que possui 4 métodos. O protocolo de disseminação ocupa 2534 bytes na área de código e 21 bytes na área de dados não inicializados. Vale-se ressaltar que um buffer é criado dinamicamente na recepção do novo código pelo protocolo, sendo assim dependente do tamanho do código a ser enviado. A estrutura do framework do ELUS consome 2076 bytes de código, 94 bytes de dados e 99 bytes de não inicializados devido às tabelas e variáveis necessárias para armazenar os objetos e métodos.

Além disso, quando um novo componente tem o suporte à reprogramação habilitado, este deve ter seus métodos incluídos no framework do ELUS. A Tabela 4 mostra o consumo de memória necessário pelo ELUS quando um novo componente é adicionado no sistema. Os métodos `Create`, `Destroy` e `Update` representam o construtor, destrutor do objeto do componente e o método de atualização e devem, portanto, sempre

Tabela 3. Consumo de memória da estrutura: ELUS e protocolo de disseminação.

Elementos da Estrutura	Tamanho da Seção (bytes)			
	.text	.data	.bss	.bootloader
Protocolo de Disseminação	2536	0	21	0
RECONFIGURATOR	166	0	4	0
AVR Code Manager	36	0	0	416
ELUS	2076	94	74	0
Total	4814	94	99	416

Tabela 4. Consumo de memória do ELUS ao habilitar o suporte à reconfiguração em um componente.

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)
Semáforo	0	18
Tamanho mínimo	1662	26

estar presentes. Para cada componente também é necessário um semáforo para controlar o acesso exclusivo ao componente e impedir que uma atualização ocorra ao mesmo tempo que seu código esteja executando. O total de consumo mínimo para um novo componente adicionado, composto pelo construtor, destrutor, método `Update` e um método que não receba parâmetros e não tenha nenhum valor de retorno é de 1662 bytes de código e 26 bytes de dados.

A Equação 5.1 sumariza o sobrecusto de memória. O tamanho de um componente "C" é o somatório dos tamanhos de todos os métodos conforme a Tabela 4 somados com o tamanho da implementação dos métodos pelo próprio componente. Ainda, somam-se a este valor, o tamanho dos métodos `Create`, `Destroy` e `Update`. Já o tamanho dos dados é a soma dos dados do componente com os valores do `Dispatcher` e `Semáforo` que são utilizados pelo ELUS.

$$Tamanho_c = \sum_{i=1}^{n}(Método_i) + Create + Destroy + Update \quad (1)$$

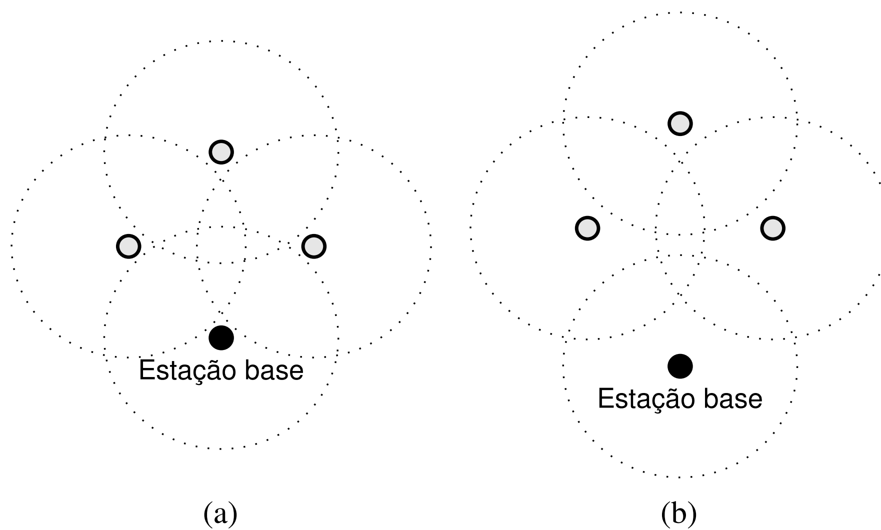

Figura 7. Topologias utilizadas para medição de latência. (a) Estação base possui comunicação com todos os nodos (b) Estação base não possui comunicação com todos os nodos.

5.2. Latência

Foram utilizadas duas topologias para medir a latência do protocolo de disseminação, ilustradas na Figura 7: (a) a estação base possui comunicação com todos os nodos, (b) a estação base não possui comunicação com um nodo, que está ao alcance dos outros. Em ambas topologias repetiu-se o processo de disseminação vinte vezes, de forma a atualizar um método de um componente do sistema, propagando 10 bytes de dados (utilizados na atualização do método) e 6 bytes de informações de controle (utilizadas pelo protocolo).

A Figura 8 apresenta a média do tempo que a estação base leva para propagar os dados aos nodos a sua volta. Foi observado um desvio padrão de 0,0233 segundos. Este tempo não mudou alterando o número de receptores entre um e três, isto porque pacotes perdidos estão altamente correlacionados, ou seja, vários receptores perdem o mesmo conjunto de pacotes [Wang 2004].

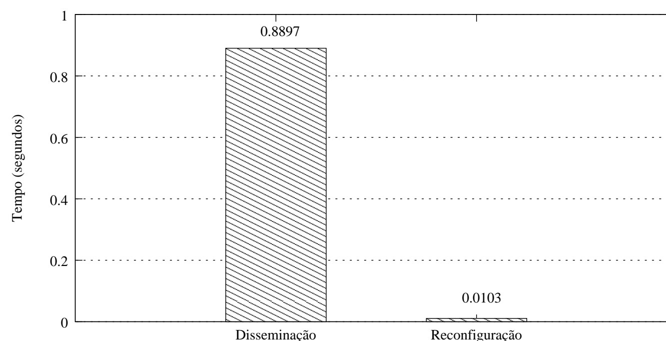

Figura 8. Tempo de disseminação e reconfiguração.

A Figura 9 mostra a média de tempo que os dados levam para serem propagados da estação base para nodos intermediários, e destes para o nodo fora de alcance da

estação base (segunda topologia). É possível perceber que o tempo necessário para propagar dados entre nodos normais da rede é aproximadamente quatro vezes maior que o tempo gasto pela estação base, isto se deve ao fato que a estação base não executa a etapa de seleção de emissor, ou seja, não perde tempo divulgando sua versão e recebendo requisições para enfim se tornar um emissor e começar a disseminar os dados. O tempo de disseminação dos nodos intermediários apresentou um desvio padrão de 1,1288 segundos.

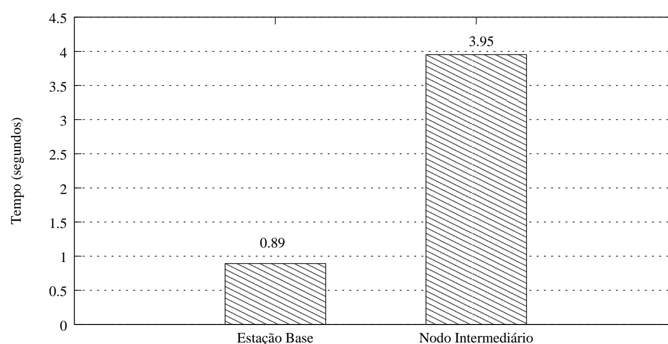

Figura 9. Tempo de disseminação.

5.3. Tempo de Reconfiguração

Foi considerado como tempo de reconfiguração, o tempo que o nodo leva para atualizar sua memória de programa após ter recebido todos os dados necessários, via o protocolo de disseminação. Na estrutura proposta este tempo engloba: a chamada do RECONFIGURATOR, os métodos `p` e `v` do semáforo, a chamada para o método `Update`, a recuperação dos argumentos passados na mensagem ETP, a recuperação do objeto a ser atualizado em uma tabela hash, descobrir o endereço da `vtable` e a escrita dos dados na flash. A Figura 8 mostra a média do tempo de reconfiguração obtido, sendo que o mesmo apresentou um desvio padrão de 0,0103 segundos.

Uma característica da arquitetura utilizada é que não é possível mudar apenas um byte por vez em sua memória flash. Esta memória só permite a escrita em páginas, cujo tamanho é de 256 bytes, e antes de reescrever uma página é necessário apagar seu conteúdo. Desta forma para atualizarmos uma parte da memória é necessário ler o conteúdo da página e armazená-lo em um buffer temporário, modificar apenas a parte desejada para enfim escrever na flash.

6. Conclusões

Este artigo apresentou uma estrutura de reprogramação em rede para sistemas operacionais embarcados que permite reconfiguração do software em tempo de execução. Esta estrutura é composta por um protocolo de disseminação de dados e um ambiente de suporte de sistema operacional que isola os componentes do sistema em unidades independentes de posição de memória. O protocolo garante a entrega correta de todos os dados para todos os nodos da rede e o ambiente de suporte permite que o sistema seja reconfigurado em tempo de execução sem a necessidade de reinicialização. A estrutura foi testada em uma RSSF utilizando nodos sensores reais, Mica2, e avaliada em termos de consumo de memória, tempo de disseminação e de reprogramação.

Referências

Arumugam, M. U. (2004). Infuse: a tdma based reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 281–282, New York, NY, USA. ACM.

Crossbow Technology Inc. *MICA2 Datasheet*.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

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.

Fröhlich, A. (2001). Application-Oriented Operating Systems. *Sankt Augustin: GMD-Forschungszentrum Informationstechnik*, 1.

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. (2009). Reconfiguração dinâmica de software em sistemas profundamente embarcados.

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 Press.

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.

Lanigan, P., Gandhi, R., and Narasimhan, P. (2005). Disseminating Code Updates in Sensor Networks: Survey of Protocols and Security Issues. Technical report, School of Computer Science, Carnegie Mellon University, PA.

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

Polpeta, F. V. and Fröhlich, A. A. (2004). Hardware mediators: a portability artifact for component-based systems. In *International Conference on Embedded and Ubiquitous Computing*, volume 3207 of Lecture Notes in Computer Science, pages 271–280, Aizu, Japan. Springer.

Thanos Stathopoulos, John Heidemann, D. E. (2003). A remote code update mechanism for wireless sensor networks. Technical report.

Wang, L. (2004). Mnp: multihop network reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 285–286, New York, NY, USA. ACM.