Infra-estrutura de Sistema Operacional para Atualização de Código

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. *Diversos sistemas embarcados apresentam sérias limitações de processamento, memória e energia. Para permitir correções de bugs ou adição de novas funcionalidades, o software que executa sob esses sistemas deve ser capaz de prover um mecanismo de atualização de código que use o mínimo de recursos possíveis e não influencie nos serviços disponibilizados pelo sistema. Neste artigo é apresentada uma infra-estrutura de sistema operacional, ainda em desenvolvimento, para atualização de código. Os resultados preliminares mostram que essa infra-estrutura tem um baixo consumo de memória e adiciona pouco overhead para a aplicação.*

Abstract. *Several embedded systems present serious power processing, memory and energy limitations. In order to allow bug corrections or addition of new functionalities, the software that runs in these systems must provide a code update mechanism that uses the minimum available resources and does not influence the services provided by the system. This work presents an operating system infrastructure for remote code update. The preliminary results show that this infrastructure has low memory consumption and added little overhead to application.*

1. Introdução

Sistemas embarcados são projetados para executar um determinado conjunto de tarefas específicas com severas restrições computacionais, como processamento, memória e consumo de energia. Devido a correção de bugs, adição/remoção ou melhoramento de funcionalidades, extensões e mudanças no ambiente, o software que executa sob essas plataformas deve ser capaz de fornecer meios para atualização do código.

É importante que o próprio mecanismo de atualização de software use o mínimo de recursos possíveis e não influencie nos serviços disponibilizados pelo sistema [Felser et al. 2007]. Um exemplo típico são as Redes de Sensores Sem Fio (RSSF) que são formadas por pequenos sensores capazes de monitorar algum fenômeno físico e que apresentam baixo poder de processamento e pouca memória disponível. Tais RSSF são compostas por milhares de sensores, muitas vezes instalados em lugares inóspitos e de difícil acesso, onde a única forma de atualizar o software nos sensores é através de um mecanismo de atualização remota. Portanto, uma boa infra-estrutura para atualização de software em sistemas embarcados onde as restrições do sistema sejam atendidas é desejável.

Esse artigo apresenta uma infra-estrutura, ainda em desenvolvimento, para atualização de código em sistemas embarcados que adiciona pouco consumo de memória e sobrecusto à aplicação. A infra-estrutura é criada no EPOS (Embedded Parallel Operating System) [Fröhlich 2001], um sistema operacional orientado à aplicação e construído seguindo os conceitos da AOSD (Application-Oriented System Design) [Fröhlich 2001].

O restante deste artigo é organizado como segue. A seção 2 discute os trabalhos relacionados. Seção 3 apresenta a infra-estrutura para atualização de código. Seção 4 mostra os resultados preliminares. Finalmente, conclusões e trabalhos futuros são apresentados na seção 5.

2. Trabalhos Relacionados

Dynamic C++ Classes [Hjalmtysson and Gray 1998] permitem que novo código seja adicionado em nível de classe em um programa em execução desenvolvido em linguagem C++. Essa técnica usa uma classe *Proxy* que suporta atualização de versões e adição de novas classes. Para cada classe, o *Proxy* mantém uma lista de versão, um ponteiro para a versão ativa e um sincronizador. Isso aumenta o espaço de memória utilizado para cada classe, tornando a abordagem difícil de ser praticável em sistemas embarcados devido às restrições de memória.

Maté [Levis and Culler 2002] é uma máquina virtual que executa sob o sistema operacional TINYOS [Hill et al. 2000]. A máquina virtual disponibiliza 8 instruções (*bytecodes*) que são interpretados. Os *bytecodes* limitam o número de aplicações que podem ser construídas [Boulis et al. 2003] e possuem um tamanho menor do que o código nativo, diminuindo o consumo de energia na transferência dos dados. Entretanto, para aplicações que executam por um longo período a energia gasta para interpretar o código supera essa vantagem [Levis and Culler 2002]. Uma instrução *forw* é utilizada para enviar (*broadcast*) o código a ser instalado para a vizinhança do nodo. SensorWare [Boulis et al. 2003] provê uma máquina virtual na qual suporta a programação dos nodos através de uma linguagem de script para sensores com maior poder de processamento e memória. Existem comandos para replicar ou migrar o código e dados para outros nodos sensores da rede. As principais limitações no uso de máquinas virtuais em sistemas profundamente embarcados estão no sobrecusto introduzido pelo interpretador e na dependência das instruções com a plataforma alvo [Koshy and Pandey 2005].

MOAP [Stathopoulos et al. 2003] e *Deluge* [Hui and Culler 2004] são mecanismos de distribuição de código implementados no TINYOS e que enviam toda a nova imagem pela rede. São utilizadas técnicas para retransmissão de pacotes perdidos, multicast e confiabilidade com o intuito de garantir a entrega do novo código para todos os nodos da rede. *FlexCUP* [Marrón et al. 2006] é um sistema de atualização para o *TinyCubus* [Marrón et al. 2005]. São gerados meta-dados em tempo de compilação que descrevem os componentes compilados com informações, como a tabela de símbolos e relocação. Desta forma, *FlexCUP* deve estar envolvido no processo de compilação do código na estação base, tornando-se dependente das mudanças nas versões do compilador. Um ligador em cada nodo é responsável por unir o novo código e gerar a imagem final. O sistema é reiniciado após a atualização. Em [Felser et al. 2007] são usadas informações geradas pelo compilador na estação base para identificar situações onde é possível uma atualização com segurança. Quando é constatado uma atualização insegura, o sistema pergunta

ao administrador se a atualização pode ou não ser realizada, com isso pode-se preservar o estado do sistema. Koshy e Pandey [Koshy and Pandey 2005] tentam reduzir o sobrecusto e a computação particionando a atualização entre os nodos e uma estação base com maior poder de processamento. É usado um ligador incremental (*incremental linker*) que é capaz de controlar as posições das funções modificadas no nodo.

SOS [Han et al. 2005] é um sistema operacional para nodos sensores construído em módulos que podem ser atualizados e removidos em tempo de execução. Com o uso de *jumps* relativos, o código de cada módulo torna-se independente de posição. Por outro lado, limita o tamanho em bytes de cada módulo e a distância máxima (em termos da posição de memória) de *jumps* relativos na arquitetura alvo. Referências de funções e dados fora do módulo são implementados através de uma tabela de indireção ou não são permitidos. Esta solução é similar a proposta neste artigo, porém, a infra-estrutura apresentada na próxima seção ocupa menos espaço em memória e tem um sobrecusto em tempo de execução menor, além de não limitar a posição de memória.

3. Infra-estrutura pra Atualização de Código

EPOS (Embedded Parallel Operating System) [Fröhlich 2001] é um sistema operacional para sistemas embarcados desenvolvido seguindo os conceitos da AOSD (Application-Oriented System Design) [Fröhlich 2001]. Embora nenhuma infra-estrutura pra atualização de código exista, o EPOS apresenta um *framework* metaprogramado que permite invocação remota de métodos. Neste *framework*, características como confinamento e isolação são encontradas. Confinamento é importante para encapsular os componentes do sistema e isolação é importante para criar um nível de indireção entre as chamadas de métodos. Com base nessas duas características presentes no *framework*, foi possível criar uma infra-estrutura para atualização de código no EPOS. Nas próximas subseções o *framework* metaprogramado para invocação remota de métodos e a infra-estrutura para atualização remota de código são apresentados.

3.1. Invocação Remota no EPOS

Uma visão geral do *framework* metaprogramado para invocação remota de métodos é apresentada na figura 1. A classe parametrizada *Handle* recebe uma abstração do sistema como parâmetro. O *Handle* verifica se o objeto foi corretamente criado e repassa as invocações de métodos ao elemento *Stub*.

O elemento *Stub* é uma classe parametrizada que é responsável por verificar se o aspecto de invocação remota está ativo para abstração ou não. O aspecto de invocação remota é selecionado por uma abstração através da sua classe *Traits* [Stroustrup 1997]. Se o aspecto não estiver ativo, o *Stub* herdará o adaptador de cenário da abstração. Caso contrário, uma especialização do *Stub* (*Stub<Abstraction, true>*), herdará o *Proxy* da abstração. Consequentemente, quando *Traits<Abstraction>::remote = false* implica que o *Handle* seja implementado como adaptador de cenário, enquanto que *Traits<Abstraction>::remote = true* implica que o *Handle* seja um *Proxy*.

Proxy é responsável por enviar mensagens com a invocação de métodos para o *Agent*. Cada mensagem é composta pelos identificadores (IDs) do objeto, método e classe que são usados pelo *Agent* para invocar o método correto, associando os IDs com uma tabela de métodos. O ID do objeto é usado para recuperar o objeto correto antes da

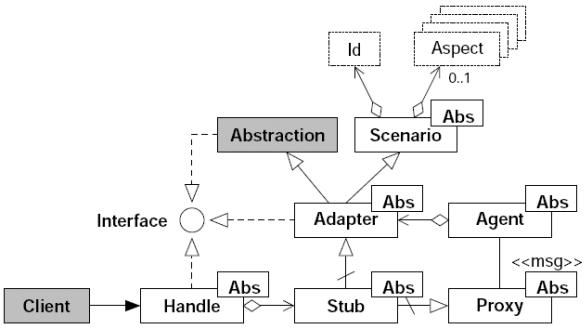

Figura 1. Framework metaprogramado do EPOS para invocação remota de métodos [Fröhlich 2001].

chamada do método. O *Agent* recebe a mensagem e invoca o método através do adaptador de cenário (*Adapter*) [Fröhlich and Schröder-Preikschat 2000].

A função da classe *Adapter* é aplicar os aspecto suportados pelo *Scenario* antes e depois da chamada real do método. Cada instância da classe *Scenario* consulta o *Traits* da abstração para verificar quais aspectos estão habilitados para aquela abstração, agregando o aspecto de cenário correspondente. Quando um aspecto não é selecionado para a abstração, uma implementação vazia é utilizada. Neste caso, nenhum código é gerado na imagem final do sistema.

Esta estrutura de *Proxy* e *Agent* descrita cria duas importantes características: o confinamento e a isolação dos componentes do sistema. Isso é conseguido pois toda chamada de método de um componente configurado com o aspecto de invocação remota passa pelo *Proxy*, criando um nível de indireção entre as chamadas de método e tornando o componente independente de posição na memória. Essas duas características encontradas no *framework* metaprogramado do EPOS são importantes para a criação da infra-estrutura de atualização remota de código, na qual é descrita abaixo.

3.2. Suporte para Atualização de Código

A estrutura de invocação remota do EPOS foi estendida conforme a figura 2. A invocação de um método de um componente da aplicação cliente com suporte a atualização remota[1] passa pelo *Proxy* que envia uma mensagem para o *Agent*. Esta mensagem é armazenada em uma "box" do sistema operacional. O *Agent* então lê a chamada de método da "SO Box" e invoca o método. Após a execução do método, uma mensagem com o valor de retorno é enviada para a aplicação. A "SO Box" controla o acesso aos métodos do componente através de um sincronizador (semáforo), somente permitindo a chamada de métodos do componente que não está sendo atualizado no momento. Com esta estrutura, é criado um nível de indireção entre as chamadas de métodos da aplicação, tornando os componentes independentes de posição na memória do sistema, sendo que somente o *Agent* possui ciência desta posição.

Uma *Thread* criada na inicialização do sistema é responsável por receber uma mensagem de solicitação de atualização. Esta mensagem é repassada para o *Agent* e

[1]O suporte a atualização remota de um componente é configurado através do seu *Traits* (*Traits<Component>::remote_update = true*).

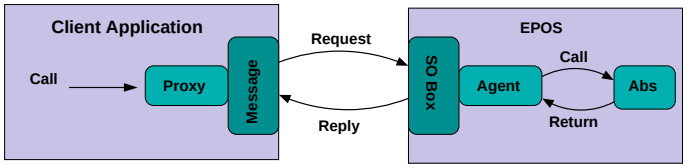

Figura 2. Cenário de invocação de métodos com suporte para atualização remota de código.

contém, além dos identificadores (IDs) da classe, método e objeto, o novo código do componente a ser atualizado, os novos endereços relativos dos métodos dentro do arquivo objeto e o tamanho do código. O *Agent* possui um array com as posições dos métodos referentes àquele componente (arquivo objeto). As referências externas utilizadas nesse novo componente são resolvidas efetuando a ligação do novo componente com o sistema antigo, que não sofreu mudanças. A unidade de atualização da estrutura é um componente e as assinaturas dos métodos deste componente devem ser a mesma nas duas versões.

No cenário de atualização remota, o ID do método é referente ao método *Update*. Dentro deste método, o *Agent* aloca memória para o novo código, copia o código recebido para esta nova posição, atualiza os novos endereços dos métodos recebidos na tabela daquele componente, destrói o objeto antigo, cria um objeto novo e adiciona este novo objeto na tabela de objetos.

A infra-estrutura do *framework* e o sistema de atualização são transparentes para a aplicação. Por outro lado, a cada adição de um novo componente ao sistema necessita que seus métodos sejam colocados na estrutura do *framework* para permitir o suporte a este novo componente. Com o suporte de atualização habilitado no sistema, as chamadas de métodos em cada componente sofrem um sobrecusto, devido ao nível de indireção, e também há um consumo de memória extra devido ao código e dados utilizados pelo *framework*. Estes dois parâmetros são analisados a seguir.

4. Resultados Preliminares

Com a finalidade de comprovar a aplicabilidade da infra-estrutura descrita, foi utilizada uma aplicação teste do jantar dos filósofos, na arquitetura IA-32. A partir desta aplicação de teste, foram avaliadas duas métricas: quantidade de memória necessária pelo *framework* e a perda de desempenho gerado pelo nível de indireção. A tabela 1 apresenta o consumo de memória obtidos através do compilador g++ na versão 4.0.2. O *framework* adiciona 1476 bytes para a aplicação, destes, 1008 são adicionados na seção .text devido as instruções necessárias ao *framework*, 404 na seção .data devido a ponteiros e tabelas usados pelo *framework* e finalmente 64 bytes na seção .bss devido a dados adicionais não inicializados. Quando um novo componente é selecionado para suportar a atualização, o espaço de memória adicionado é dependente do número de métodos desse novo componente, pois cada método deverá ter seu endereço armazenado, aumentando o tamanho da tabela que guarda estes endereços e também a quantidade de código do *framework*.

A tabela 2 mostra o sobrecusto gerado pela infra-estrutura de suporte a atualização quando um método é invocado. Estes valores foram medidos utilizando a abstração *Chronometer* do sistema. Os resultados são a representação da média de valores obtidos em 10 (dez) execuções, desconsiderando os maiores e menores valores encontrados. O

Tabela 1. Consumo de memória do suporte para atualização no componente Thread na aplicação teste.

Seção	*Sem Suporte (bytes)*	*Com Suporte (bytes)*	*Adicionado (bytes)*
.text	26896	27904	1008
.data	36	440	404
.bss	500	564	64
TOTAL	**27432**	**28872**	**1476**

tempo do construtor da Thread apresentou um sobrecusto de 156%, devido a maior passagem de parâmetros dentro do *framework* até a criação da *Thread*. Conforme o aumento do tempo de computação do método, a influência da infra-estrutura de atualização acaba sendo minimizada, o que pode ser notado na invocação do método da *Thread Pass*.

Tabela 2. Tempo para invocar um método com e sem suporte a atualização.

Método	*Sem Suporte (us)*	*Com Suporte (us)*	Sobrecusto(%)
Construtor	66.75	171.125	156%
Resume	27.85	41.125	47.66%
Pass	282.625	309.875	9.6%

5. Conclusões e Trabalhos Futuros

Este artigo apresentou uma infra-estrutura para atualização de código no sistema operacional EPOS. A infra-estrutura é composta por um *framework* metaprogramado, na qual possibilita o confinamento e isolação dos componentes do sistema, tornando-os independentes de posição na memória.

Os resultados preliminares mostram que a infra-estrutura adiciona pouca memória aos componentes configurados com suporte a atualização e o sobrecusto criado pelo nível de indireção na invocação do método não compromete os serviços disponibilizados pela aplicação. O baixo consumo de memória e sobrecusto são alcançados devido a metaprogramação estática, na qual todas as dependências entre os componentes do sistema e aplicação são resolvidas em tempo de compilação, não gerando código desnecessário na imagem final do sistema.

Como trabalhos futuros, o mecanismo de atualização de código será ampliado para suportar distribuição de código pela rede e somente enviar as mudanças entre o código antigo e o novo, melhorando assim sua aplicação em sistemas profundamente embarcados.

Referências

Boulis, A., Han, C.-C., and Srivastava, M. B. (2003). Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA. ACM Press.

Felser, M., Kapitza, R., Kleinöder, J., and Schröder-Preikschat, W. (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 World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A.

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 *Architectural Support for Programming Languages and Operating Systems*, pages 93–104.

Hjalmtysson, G. and Gray, R. (1998). Dynamic C++ classes—A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conf.*, pages 65–76.

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). Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 354–365.

Levis, P. and Culler, D. (2002). Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*.

Marrón, P. J., Lachenmann, A., Minder, D., Hähner, J., Sauter, R., and Rothermel, K. (2005). TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289.

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.

Stathopoulos, T., Heidemann, J., and Estrin, D. (2003). A remote code update mechanism for wireless sensor networks. Technical report, Los Angeles, CA, USA.

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