UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Giovani Gracioli

ELUS: Projeto e Implementação de um Mecanismo de Reconfiguração Dinâmica de Software para Sistemas Profundamente Embarcados

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Florianópolis, Setembro de 2009

ELUS: Projeto e Implementação de um Mecanismo de Reconfiguração Dinâmica de Software para Sistemas Profundamente Embarcados

Giovani Gracioli

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Dr. Mauro Roisenberg

Coordenador

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Orientador

Prof. Dr. Fabiano Passuelo Hessel

Prof. Dr. Rômulo Silva de Oliveira

Prof. Dr. Mario Antonio Ribeiro Dantas

Para os meus pais,

Leonildo Gracioli

Rosilene Marisa Bertochi

Agradecimentos

Gostaria de agradecer primeiramente ao apoio dos meus familiares, especialmente da minha mãe Rosilene Marisa Bertochi e do meu pai Leonildo Gracioli. Obrigado por tudo.

Agradeço ao Professor Antônio Augusto Fröhlich pela constante dedicação e auxílio na orientação deste trabalho e pela oportunidade de fazer parte do LISHA, que me deu todas as condições e um excelente ambiente para realizar pesquisa no mais alto nível.

Muito obrigado aos colegas do LISHA, pelas discussões técnicas e científicas que ajudaram na concretização deste trabalho e também pelos momentos de descontração em churrascos e festas.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Acrônimos

ABS	Anti-lock Brake System
ADL	Architecture Description Language
AMD	Advanced Micro Devices
AOP	Aspect-Oriented Programming
AOSD	Application-Oriented System Design
API	Application Programming Interface
CAN	Controller Area Network
C-MAC	Configurable Media Access Control
CPU	Central Processing Unit
DVM	Dynamic Virtual Machine
EDF	Earliest Deadline First
ELUS	Epos Live Update System
EPOS	Embedded Parallel Operating System
ETP	Elus Transport Protocol
FBD	Family-Based Design
HAL	Hardware Abstraction Layer

ID Identificador

MANTIS Multimodal Networks of In-situ Sensors

MMU Memory Management Unit

NIC Network Interface Card

PCM Powertrain Control Module

PDA Personal Digital Assistant

RSSF Redes de Sensores Sem Fio

RTC Real-Time Clock

SO Sistema Operacional

TDMA Time Division Multiple Access

TSC Time Stamp Counter

TSL Test and Set Lock

UML Unified Modeling Language

VTABLE Tabela de Métodos Virtuais

Resumo

Reconfiguração dinâmica de software em ambientes computacionais convencionais é o processo de atualizar o software de um sistema em execução. Esta atividade é extremamente importante para corrigir eventuais erros, adicionar e/ou remover funcionalidades e adaptar-se às mudanças que por ventura o sistema pode sofrer durante o seu tempo de vida. Reconfiguração dinâmica de software em sistemas profundamente embarcados torna-se um desafio ainda maior devido às características de tais sistemas, que apresentam sérias limitações de processamento, memória e, quando alimentados por bateria, de energia. Neste cenário, o próprio mecanismo de reconfiguração de software deve usar o mínimo de recursos possíveis pois estará competindo com os recursos do sistema e não deve influenciar os seus serviços.

Esta dissertação apresenta o EPOS LIVE UPDATE SYSTEM (ELUS), uma infra-estrutura de sistema operacional que permite reconfiguração dinâmica de software em sistemas profundamente embarcados. Através do uso de sofisticadas técnicas de metaprogramação estática em C++, o ELUS utiliza pouca memória e o processo de reconfiguração torna-se simples e totalmente transparente para as aplicações. O ELUS é construído dentro do framework de componentes do EPOS, em torno do aspecto de invocação remota, permitindo a seleção dos componentes reconfiguráveis em tempo de compilação, sendo que para todos os outros componentes não selecionados, nenhum sobrecusto em termos de memória e processamento é adicionado no sistema. As principais características que diferem o ELUS das outras infra-estruturas de sistemas operacionais para reconfiguração dinâmica de software existentes são a configurabilidade, o baixo consumo de memória, a simplicidade e a transparência para as aplicações.

Abstract

Dynamic software reconfiguration is the process of updating the system's software during its execution. This activity is highly desirable to be able to correct bugs, to add new and/or to remove features, and to adapt the system to varying execution environments. Dynamic software reconfiguration in deeply embedded system is even more challenging due to the characteristics of such systems, that are subject to severe resource limitation, like processing power, memory and energy. In this scenario, whichever dynamic software reconfiguration mechanism devised, it will have to operate under even more severe resource limitations, since it will compete for resources with the target embedded system and yet must not disrupt the embedded system operation.

This dissertation presents the EPOS LIVE UPDATE SYSTEM (ELUS), a low-overhead operating system infrastructure for dynamic software reconfiguration. By using sophisticate C++ static metaprogramming techniques, the infrastructure and the code update become fully transparent to applications. Moreover, the infrastructure allows the system's components to be marked as updatable or not at compilation time, and for those components that are not marked as updatable, no overhead is added. The main features that make ELUS different from the existing operating system infrastructures are the configurability, low memory consumption, simplicity, and transparency for the applications.

Capítulo 1

Introdução

Nos últimos anos, os avanços das tecnologias de hardware tornaram possível a produção de dispositivos de baixo custo e baixa potência cada vez menores, criando o conceito de sistemas profundamente embarcados. Tais sistemas são projetados para executar um determinado conjunto de tarefas específicas com severas restrições computacionais, como processamento, memória e consumo de energia, onde a utilização de maneira adequada dos recursos é um fator extremamente importante.

Como exemplos de sistemas profundamente embarcados podem ser citados as Redes de Sensores Sem Fio (RSSF) e os sistemas automotivos. As RSSF permitem o monitoramento de uma grande variedade de ambientes, tais como diagnóstico de falhas em máquinas, monitoramento ambiental, automação industrial e detecção química e biológica. Em muitos desses ambientes, surge a necessidade que os sensores da rede tenham uma grande vida útil, ou seja, sejam capazes de oferecer seus serviços por diversos meses ou até mesmo por diversos anos sem nenhum tipo de manutenção (i.e. troca de bateria) [HSW+00]. Sistemas automotivos modernos têm centenas de unidades de microcontroladores dedicadas que executam funções específicas, tais como o controle do ABS (*Anti-lock Brake System*) e motor (PCM - *Powertrain Control Module*).

Devido a correção de bugs, adição/remoção ou melhoramento de funcionalidades, extensões e mudanças no ambiente, o software que executa sob tais sistemas deve ser capaz de fornecer meios para atualização do código que está sendo executado.

As RSSF, por exemplo, muitas vezes estão localizadas em áreas de difícil acesso, onde coletar todos os sensores e reprogramá-los é impraticável. Um estudo realizado pela Daimler-Chrysler apontou que o tempo necessário para atualizar todo o sistema embarcado distribuído em um automóvel, em uma oficina autorizada, leva cerca de 8 horas e apresenta custos extremamente elevados [EN03]. Neste cenário, o mecanismo de atualização terá que operar ainda com mais restrições em termos de recursos, pois estará competindo pelos recursos do sistema e ainda não deve influenciar os serviços disponibilizados pelo sistema [FKKP07].

Em ambos exemplos, um *mecanismo de reconfiguração de software* remoto e eficiente seria ideal para lidar com as mudanças necessárias do software durante o tempo de vida do sistema. Além disso, sistemas profundamente embarcados apresentam uma grande variedade de plataformas, que podem variar desde um microcontrolador de 8-bits, com poucos *kilobytes* de memória e um sistema de transmissão (e.g RS-485, CAN, ZigBee), até um processador com grande capacidade de memória, processamento e comunicação.

Os mecanismos de reconfiguração de software em sistemas profundamente embarcados podem ser separados em dois grupos: sistemas que são baseados na atualização do código binário [RL03] e sistemas operacionais [DGV04]. No primeiro, um *bootloader* ou ligador é responsável por receber o novo código do sistema e fazer as alterações necessárias (e.g relocação de endereços, alocação de memória, etc). Após isso, o sistema é reiniciado com o novo código instalado. Já os SOs são projetados para que a reconfiguração se torne mais simples e fácil para os seus usuários. Normalmente, os SOs que permitem reconfiguração são organizados em módulos atualizáveis, como o SOS [HKS+05] e o RETOS [CCJ+07], ou criam um nível de indireção entre a chamada real da função/método e a aplicação através do uso de ponteiros ou tabelas, como no Contiki [DGV04] e no Nano-Kernel [Bag08]. Neste caso, a reconfiguração se dá através da atualização dos endereços dos ponteiros e/ou tabelas para a nova posição em memória.

O recebimento e a atualização da imagem completa do sistema apresenta um alto sobrecusto para a aplicação, pois será necessário o recebimento de grandes quantidades de dados pela rede e diversas escritas na memória (seja ela flash, RAM, etc),

gerando um grande consumo de energia. Para atenuar este problema, podem ser usadas técnicas que somente enviem a diferença do código entre a imagem nova e a antiga [RL03] e protocolos de disseminação de dados específicos [SHE03]. Da mesma forma, a reinicialização da aplicação pode levá-la a um estado inconsistente ou até mesmo a perda de dados que por ventura estiverem na memória RAM. Portanto, deve-se evitar o reinicio da aplicação após uma reconfiguração.

Esta dissertação apresenta o projeto e a implementação de uma infra-estrutura de sistema operacional que permite reconfiguração dinâmica de software, chamada de EPOS LIVE UPDATE SYSTEM (ELUS). Através do uso de sofisticadas técnicas de metaprogramação estática em C++, a infra-estrutura e a atualização de código tornam-se totalmente transparentes para as aplicações. O ELUS é construído dentro do framework de componentes do EPOS, em torno do aspecto de invocação remota [Frö01]. Com isso, existe a possibilidade de selecionar somente aqueles componentes que sejam reconfiguráveis e para todos os outros componentes não selecionados, nenhum sobrecusto é adicionado ao sistema. O framework torna um componente independente de posição de memória, eliminando a necessidade de um *bootloader* e/ou ligador instalado em cada nodo da rede para permitir a reconfiguração. Além disso, o sistema não precisa ser reiniciado após cada reconfiguração. Apesar de não ser o foco desta dissertação, a estrutura proposta pode ser integrada com qualquer protocolo de disseminação de dados, para que seja possível a atualização de diversos nodos em uma rede.

1.1 Objetivos

O principal objetivo desta dissertação é o desenvolvimento de uma infra-estrutura que dê suporte à reconfiguração dinâmica de software em sistemas embarcados.

A partir deste objetivo principal, são definidos os seguintes objetivos específicos:

- Estudar as principais soluções para a reconfiguração dinâmica de software no con-

texto de sistemas embarcados visando o levantamento completo do estado da arte.

- Analisar os principais sistemas operacionais que oferecem suporte para reconfiguração de software em sistemas embarcados. Este objetivo é importante para que a estrutura proposta neste trabalho possa ser comparada com os outros trabalhos que suportam atualização de software desenvolvidos em diferentes sistemas operacionais para sistemas embarcados.

- Modelar e implementar a infra-estrutura de reconfiguração dinâmica no sistema operacional EPOS. O sistema será composto por um framework metaprogramado desenvolvido na linguagem C++.

- Testar a infra-estrutura levando em consideração alguns parâmetros como consumo de memória, tempo de atualização e sobrecusto para a aplicação.

- Análise dos resultados obtidos e comparação com os trabalhos relacionados ressaltando as vantagens e desvantagens do trabalho proposto.

1.2 Organização do Texto

O restante desta dissertação está organizada da seguinte maneira:

Capítulo 2 é dedicado aos trabalhos relacionados. Serão apresentados os trabalhos mais relevantes associados a reconfiguração dinâmica de software em sistemas embarcados.

Capítulo 3 descreve o sistema operacional EPOS, sua organização e os principais conceitos envolvidos que serão importantes para a compreensão da estrutura de atualização proposta.

Capítulo 4 é dedicado a descrição do projeto e implementação do EPOS LIVE UPDATE SYSTEM (ELUS).

Capítulo 5 avalia a estrutura proposta em termos de consumo de memória, tempo de atualização e sobrecusto para a aplicação.

Finalmente, o **Capítulo 6** conclui a dissertação e apresenta os trabalhos futuros.

Capítulo 2

Reconfiguração Dinâmica de Software em Sistemas Embarcados

Este capítulo descreve os principais trabalhos relacionados com reconfiguração dinâmica de software em sistemas embarcados. A seguir serão apresentadas infra-estruturas, máquinas virtuais, protocolos de disseminação de dados e sistemas operacionais dentro deste contexto.

2.1 Infra-Estruturas para Reconfiguração Dinâmica de Software

Muitos trabalhos têm sido desenvolvidos com o intuito de criar infra-estruturas que suportem reconfiguração de software em sistemas de propósito geral [HW04, OMT98, HG98]. Outras soluções focam sistemas embarcados com maior poder de processamento e memória, como PDAs e celulares [KJP05, LM03]. Nesta seção são identificados os principais trabalhos relacionados a infra-estruturas para reconfiguração de código em sistemas profundamente embarcados. A grande maioria dessas infra-estruturas criam métodos para a atualização do código binário do sistema, necessitando de bootloader e relocação de endereços, ou criam um nível de indireção entra as chamadas de funções e dados.

Diff-like é uma estrutura que recebe somente a diferença entre a nova e a velha imagem do sistema, diminuindo assim a quantidade de código transferido pela rede e a energia consumida pelos nodos [RL03]. O processo de reconfiguração é dividido em quatro fases. Na primeira, chamada de inicialização, os nodos preparam a memória onde será escrita a nova imagem. Na segunda fase as mensagens são enviadas para que a imagem seja montada corretamente na área de memória previamente conhecida. Na terceira, é feita uma verificação dos dados recebidos para que haja a garantia de que a imagem foi totalmente recebida e montada corretamente na memória. E por fim, a imagem é carregada e o sistema reiniciado. O processo de atualização é realizado através de comandos baseado em linguagem de script, como por exemplo, *insert*, *copy* e *repair*.

FlexCUP é um sistema de atualização para o TinyCubus proposto por [MLM+05]. São gerados meta-dados em tempo de compilação que descrevem os componentes compilados incluindo 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. A atualização se dá através do armazenamento do novo código e meta-dados na memória, a união da tabela de símbolos do componente antigo com o novo, o processo de relocação de endereços e referências realizado por um ligador instalado nos nodos e por fim a cópia dos dados recebidos da memória flash para a memória de programa e a reinicialização do nodo [MGL+06].

A infra-estrutura proposta em [FKKP07] usa informações geradas pelo compilador na estação base para identificar situações onde é possível uma atualização com segurança. Quando é constatado uma atualização insegura, como a atualização de uma função em execução por exemplo, o sistema pergunta ao administrador se a atualização pode ou não ser realizada, com isso pode-se preservar o estado do sistema. Esta solução é dependente da versão do compilador e também necessita de uma pessoa com conhecimentos específicos em relação ao sistema e ao processo de atualização para tomar as decisões de quando uma atualização é considerada segura ou não.

O ligador incremental remoto tenta 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, deslocando ou alocando mais espaço para as funções que tiveram seu código acrescido. Somente as diferenças entre as imagens são enviadas pela rede através do algoritmo *Xdelta* [KP05a].

Molecule [YMCH08] é um mecanismo de reconfiguração dinâmica adaptativo e que pode ser usado por sistemas operacionais para RSSF. Neste mecanismo, as aplicações, o kernel e os drivers de dispositivos são considerados módulos. O sistema suporta duas abordagens diferentes para permitir a atualização dos módulos: ligação direta e indireta. Baseado em uma análise de custos e o tempo de execução esperado em cada módulo, o mecanismo escolhe entre o modo direto ou indireto. No método de ligação direto, cada endereço de função ou dado do módulo que está sendo chamado por outros módulos deve ser atualizado. Com isso, este método tem maior custo de atualização em termos de processamento e energia. Por outro lado, apresenta um menor sobrecusto nas chamadas de funções ou acesso aos dados entre os módulos. No modo de ligação indireto, toda chamada à função é realizada através de uma tabela que contém os endereços das funções/dados. Desta forma, ao ser realizada uma atualização, apenas os endereços nesta tabela deverão ser atualizados, obtendo um menor custo de atualização e tornando o processo de atualização menos complicado. Por outro lado, o tempo de execução e tamanho de código são maiores do que a ligação direta. Ambas as ligações direta e indireta são realizadas nos nodos da rede, na qual aumenta o consumo de energia, processamento e memória necessários. Para reduzir este consumo, os autores propuseram um processo de ligação remota, realizado em uma estação base com maior poder de processamento e memória. Assim, o processo de atualização é executado na estação base e apenas as modificações são enviadas para os nodos.

2.2 Máquinas Virtuais

Uma máquina virtual fornece um ambiente completo para a execução de programas escritos em linguagem própria, geralmente chamada de linguagem de script. A máquina virtual disponibiliza um conjunto de instruções da sua linguagem de script para

que sejam criadas aplicações e em tempo de execução, essas instruções são interpretadas e executadas na plataforma alvo. A reconfiguração de software em uma máquina virtual é realizada de forma simples, sendo necessário apenas o envio das instruções que compõem a aplicação. Geralmente o código da máquina virtual não é passível de reconfiguração.

Maté [LC02] é uma máquina virtual que executa sobre o sistema operacional TINYOS. A máquina virtual disponibiliza 8 instruções (*bytecodes*) que são interpretados. Os *bytecodes* limitam o número de aplicações que podem ser construídas [BHS03] 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 [LC02]. Uma instrução *forw* é utilizada para enviar (*broadcast*) o código a ser instalado para a vizinhança do nodo.

SensorWare 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 [BHS03]. SensorWare ocupa 179Kb de memória divididos no interpretador da linguagem baseado em Tcl, o núcleo do SensorWare e código dependente de plataforma. Existem comandos para replicar ou migrar o código e dados para outros nodos sensores da rede através de scripts.

DVM (*Dynamic Virtual Machine*) é uma máquina virtual construída sob o sistema operacional SOS que interpreta scripts de alto nível escritos em um formato de bytecode portável [BHR+06]. A arquitetura da máquina virtual é dividida em um núcleo responsável por interpretar e executar os scripts e a linguagem de script que é compilada para o conjunto de instruções específicos da arquitetura alvo. DVM utiliza a estrutura de módulos do SOS para carregar novas extensões em tempo de execução.

VM* é um framework que permite a construção de uma máquina virtual baseada em pilha para RSSF e capaz de interpretar aplicações escritas em linguagem Java [KP05b]. Otimizações são realizadas para reduzir o sobrecusto introduzido pela máquina virtual. O framework é formado por um interpretador que executa os bytecodes em Java, uma pequena API para acessar os dispositivos e um sistema operacional para suportar o escalonamento de tarefas e alocação dinâmica de memória. O suporte para re-

configuração dinâmica é fornecido por um ligador incremental [KP05a] e um *bootloader*, na qual somente são introduzidos no sistema se a atualização de código for necessária. Para reduzir o efeito de uma atualização quando ocorre deslocamento de código (e.g. aumento de código em uma função), são deixados espaços de memória em branco entre as funções. Assim, quando uma função cresce, não existe a necessidade de trocar a função de posição na memória. Por outro lado, esta abordagem aumenta o consumo de memória. A máquina virtual provê instruções (e.g. *copy, run* e *add*) pare executar uma atualização. Em comparações realizadas pelos autores, VM* apresentou um melhor desempenho em termos de tamanho de código e dados do que a Maté. Porém, foi pior que o TINYOS.

Tapper é uma máquina virtual baseada em pilha na qual interpreta comandos em uma linguagem de script própria (e.g. A1 - executa uma leitura no canal 1 do ADC) [XLC06]. A máquina virtual não necessita de um ambiente de execução, como um sistema operacional, para sua execução. Tapper é composta por um gerenciador de buffer, um interpretador e primitivas do sistema, tais como temporizador, ADC, SPI, etc. Tapper tem sido usada em uma grande variedade de plataformas de sensores e diversos tipos de microcontroladores, variando desde uma arquitetura AVR de 8-bits com 4KB de RAM até Freescale 16-bits rodando protocolo TCP/IP. O tamanho de código da máquina virtual varia de 3KB a 11KB e os dados de 230B a 1.5KB dependendo da arquitetura. O processo de reconfiguração é executado pelas instruções “+” e “-”. Por exemplo, +func carrega a função “func” e -func remove a função do sistema. Essas instruções são realizadas em uma máquina que possui o conhecimento sobre a memória de dados e código em cada nodo da rede (host). Nesta máquina, a instrução “+” é traduzida em um comando de memória “w:x:....”, onde x é o endereço de inicial da nova posição de memória para a nova função sendo adicionada. Após a alocação de memória para a nova função, o gerenciador de buffer grava o nome do script (nome da função), o endereço inicial na memória e o tamanho da função em uma tabela. Ao executar um script, o host traduz o nome do script em um comando “e:x:len”, onde x é o endereço inicial e len o tamanho do script. A máquina virtual ainda não possui suporte para distribuição do código para todos os nodos da rede.

As principais limitações no uso de máquinas virtuais em sistemas pro-

fundamente embarcados estão no sobrecusto em termos de energia e processamento introduzido pelo interpretador e na dependência das instruções com a plataforma alvo [KP05a].

2.3 Protocolos de Disseminação de Dados

Embora este assunto não seja diretamente tratado neste dissertação, protocolos de disseminação de dados são fundamentais para qualquer mecanismo de reconfiguração de código que tem como objetivo a atualização de diversos nodos em uma rede de sensores.

MOAP é um mecanismo de distribuição de código implementado no TINYOS que envia toda a nova imagem pela rede [SHE03]. O foco deste protocolo está na diminuição do consumo de energia, uso de memória e na latência (tempo para enviar a nova imagem pela rede). Este mecanismo usa uma interface *publish-subscribe*, nomeada como *Ripple*, para selecionar o emissor e propagar os novos dados. A nova imagem é separada em segmentos, sendo que o mecanismo de gerenciamento de segmentos é baseado em *sliding window* e quando é detectada a falta de um segmento em uma atualização, o nodo receptor envia um pedido de retransmissão *unicast* ao emissor (sempre a 1-hop de distância).

Deluge também é um mecanismo de distribuição de código implementado no TINYOS [HC04]. Este mecanismo compartilha algumas idéias com o *MOAP*, como o uso de NACKs na retransmissão de pacotes, transmissão dos novos dados através de *broadcast* e *sliding window* na gerência de segmentos. Além disso, *Deluge* faz uma checagem CRC de 16-bits para verificar se os pacotes foram recebidos corretamente. Os nodos possuem três estados: (i) MAINTAIN responsável pelo envio da nova imagem; (ii) RX responsável pelo pedido de retransmissão de pacotes e (iii) TX responsável por enviar os pacotes de um dada página até que todos os nodos recebam a imagem completa. A seleção do emissor é baseada no *Trickle* [LPCS04].

MNP divide a nova imagem em segmentos sendo que existe um mapa de bits (*bitmap*) para detectar perda de segmentos na transmissão, com isso os segmentos não precisam ser enviados em ordem [Wan04]. MNP utiliza um mecanismo de seleção

de emissor na qual evita a perda de mensagens por colisão e também reduz o tempo que o rádio fica ativo colocando o nodo em modo "sleep" quando seus vizinhos estão transmitindo uma imagem que o nodo já possui.

Infuse utiliza TDMA para garantir confiabilidade na entrega dos dados pelo suporte de não colisões oferecido pelo TDMA [Aru04]. Uma estação base é responsável por começar o envio de uma nova imagem, sendo que esta é dividida em tamanho de pacotes chamado de cápsulas. Infuse utiliza dois algoritmos de recuperação dos dados em caso de perda de mensagens ou erros de transmissão baseados em *sliding window* [CCVV96, Tan88]. O protocolo reduz a energia consumida selecionando somente um emissor para o envio da nova cápsula através da definição dos nodos vizinhos do receptor como predecessor ou sucessor, sendo que o nodo predecessor é responsável pelo envio dos dados.

Sprinkler utiliza o protocolo TDMA para garantir confiabilidade na entrega dos dados. O protocolo calcula um agendamento para transmissão das mensagens que diminui colisões e perda de dados [NASZ05, NASZ07]. A transmissão é dividida em duas fases. Na primeira, chamada de *streaming*, a imagem é enviada pela rede. Na segunda, chamada de *recovery*, os nodos que não receberam corretamente a imagem na primeira fase enviam uma mensagem *unicast* contendo a lista de todos os pacotes que não foram recebidos. A seleção do emissor das mensagens é feita através da construção de um grafo virtual. Para diminuir o consumo de energia, os nodos que não estão enviando ou recebendo mensagens são colocados em modo de consumo de energia reduzido.

2.3.1 Características Comuns aos Protocolos

Com base nos protocolos descritos, as principais caraterísticas presentes nos protocolos de disseminação são [LGN05, HKSS05]:

- **Armazenamento/Gerência de Segmentos:** os protocolos dividem a nova imagem em segmentos que são transmitidos quando uma nova atualização inicia. Estes segmentos podem ser armazenados e gerenciados utilizando mapa de bits, estruturas de dados hierárquicas ou ainda um mecanismo de *sliding window*;

- **Política de Retransmissão:** quando é detectada a falha no recebimento de algum segmento, existe um mecanismo de retransmissão, tal mecanismo pode ser baseado em *broadcast* ou *unicast*. Todos os protocolos analisados utilizam *unicast*;

- **Protocolo de Disseminação:** os dados podem ser propagados através de uma interface *publish-subscribe* onde o emissor envia uma mensagem informando que existe uma nova versão e os nodos que ainda não a possuem respondem com o pedido de transmissão desta nova imagem ou ainda utilizando estados (transmissor e receptor) como o protocolo Deluge;

- **Confiabilidade:** para garantir confiabilidade pode-se utilizar um protocolo de controle de acesso ao meio como TDMA ou garantir confiabilidade em software, fazendo uso de checagem de pacotes (CRC) ou um mecanismo de confirmação de recebimento NACK;

- **Seleção do Emissor:** a seleção do emissor que irá transmitir os novos dados é importante para diminuir o número de mensagens enviadas e consequentemente diminuir o consumo de energia. Pode-se usar uma interface *publish-subscribe*, grafo virtual ou um protocolo específico como o *Trickle* [LPCS04];

- **Abordagens baseadas em Diff:** todos os protocolos permitem a integração de uma abordagem baseada em *diff*, ou seja, enviando somente as mudanças entre a antiga e a nova versão do sistema.

2.4 Suporte à Reconfiguração Dinâmica de Software em Sistemas Operacionais

Sistemas operacionais de propósito geral geralmente fornecem mecanismos para reconfiguração de software, mas não levam em consideração as restrições existentes em sistemas embarcados. O Linux permite o carregamento de módulos em tempo de execução. O K42 [BHA+05] possui um mecanismo chamado de *Hot-Swapping* que permite que uma instância de um objeto seja transparentemente trocada em tempo de

execução. Nesta seção são identificados os principais sistemas operacionais que suportam reconfiguração de software em sistemas embarcados.

TINYOS pode ser considerado atualmente o sistema operacional para sistemas embarcados mais conhecido e utilizado, especialmente na área de Redes de Sensores Sem Fio [HSW+00]. O SO é baseado em eventos e suporta uma grande variedade de sensores, como, por exemplo, Telos, Mica e BTnode. O TINYOS por si só não suporta reconfiguração de software, porém, como já apresentado anteriormente, alguns trabalhos implementam máquinas virtuais [LC02], infra-estruturas [MGL+06] ou protocolos de disseminação de dados [HC04, SHE03] em cima do TINYOS.

Mantis (*Multimodal Networks of In-situ Sensors*) é outro sistema operacional amplamente conhecido na área RSSF. O SO é multithread, de código aberto e com API compatível ao padrão POSIX. O Mantis não tem suporte a reconfiguração de software [ABC+03].

Nano-Kernel é um SO que suporta reconfiguração dinâmica separando os dados dos algoritmos lógicos do kernel [Bag08]. O espaço de endereçamento do kernel é dividido em duas partes, o núcleo do Nano-Kernel e os dispositivos do kernel, como escalonador e gerenciador de memória. O núcleo é responsável por criar um nível de indireção entre as chamadas da aplicação e os dispositivos do kernel. Desta forma, quando uma aplicação faz uma chamada a um método do kernel ou quando um dispositivo do kernel chama um método de outro dispositivo, o núcleo redireciona a chamada para o dispositivo do kernel apropriado e retorna o resultado corretamente a quem fez a chamada. Além disso, o núcleo também verifica se as permissões de acesso ao dispositivo são satisfeitas, o que introduz um sobrecusto a cada chamada de método dos dispositivos. Ao inicializar o sistema, todos os módulos são carregados, registrados no sistema e suas interfaces inicializadas. O núcleo pode substituir um dispositivo do kernel dinamicamente, pois estes são vistos como módulos carregáveis, isolados e individuais. Uma reconfiguração começa com o recebimento de uma mensagem “reconfig”, após isso o módulo que esta sendo atualizado é parado através de uma mensagem “STOP”, é removido através de um comando “unload” e carregado através de um comando “load”. O próximo passo é registrar os serviços disponibilizados pelo módulo nas estruturas internas do kernel res-

ponsáveis por fazerem a interface entre a aplicação e os módulos. A partir deste momento, os serviços do dispositivo atualizado podem ser novamente acessados.

RETOS é um sistema operacional multithread para RSSF [CCJ+07]. O SO usa um mecanismo de reconfiguração dinâmica nos módulos do sistema através de relocação dinâmica de memória e ligação em tempo de execução. O processo de relocação extrai informações de variáveis globais e funções em tempo de compilação e a partir dessas informações são criados meta-dados, nos quais são dependentes de arquitetura, sobre os módulos do sistema. Essas informações são colocadas em um arquivo no formato RETOS, sendo que o kernel usa esse arquivo para substituir os endereços utilizados pelo módulo quando o módulo é carregado no sistema. O SO também armazena uma tabela que permite que os módulos e as aplicações acessem outras funções disponibilizadas por outros módulos. Um módulo pode registrar, desregistrar e acessar funções através dessa tabela. O acesso as funções pela aplicação também é realizado através desta tabela. Em termos de avaliação, RETOS apresentou um consumo de memória maior do que o TINYOS, cerca de 7kb a mais.

Contiki é um sistema operacional desenvolvido na linguagem C para RSSF que implementa um kernel baseado em eventos e oferece suporte a multithread através de uma biblioteca que somente é ligada à imagem final do sistema quando necessária [DGV04]. Contiki divide o sistema em núcleo e aplicações em tempo de compilação, sendo que no núcleo se encontra o kernel, o carregador de programa, suporte da linguagem em tempo de execução e o suporte à comunicação das aplicações com os dispositivos de hardware. O sistema operacional implementa processos chamados de serviços que disponibilizam funcionalidades a outros processos. Os serviços, como por exemplo um protocolo de comunicação, pode ser substituído em tempo de execução. Isso é possível pois para cada serviço existe uma Interface Stub que redireciona as chamadas de funções para uma Interface de Serviço que possui ponteiros para as implementações das funções relativas àquele serviço. A Interface de Serviço também mantém a versão atual do serviço para futuras atualizações. Ao receber uma mensagem de atualização de um serviço, o kernel envia um evento na qual o serviço se auto-removerá do sistema. Para transferir o estado do serviço antigo (i.e. dados e estruturas), o kernel disponibiliza uma

interface para a passagem de ponteiros entre as duas versões. As aplicações são atualizadas pelo carregador do programa, que aloca a memória para o novo código e carrega o novo módulo no sistema com base nas informações de relocação recebidas no arquivo binário. O tamanho de código da imagem final do Contiki é maior do que o TINYOS, mas menor do que o Mantis [DGV04].

SOS é um sistema operacional para nodos sensores construído em módulos que podem ser atualizados e removidos em tempo de execução [HKS+05]. O kernel do sistema provê funções como alocação de memória, temporizadores, E/S e suporte para carregar um módulo enquanto o sistema estiver em execução. Ao ser carregado, cada módulo publica uma lista de funções que podem ser utilizadas por outros módulos e uma lista de funções que utiliza de outros módulos. Essas funções são colocadas em uma tabela, criando um nível de indireção para cada chamada. Com o uso de *jumps* relativos (cada chamada passa pela tabela), 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 também são implementadas através da tabela de indireção ou não são permitidos. Durante uma atualização são usadas informações do módulo recebido (identificador do módulo, tamanho de memória necessário e versão do módulo) para que seja alocada memória suficiente para o novo código. Se o sistema não tem memória suficiente, a atualização é imediatamente cancelada. Por fim, os ponteiros na tabela dentro do kernel são atualizados e a função de inicialização do módulo é chamada. Em caso de remoção, o kernel envia uma mensagem ("final") que permite ao módulo liberar seus recursos. Após este ponto, o kernel ativa um coletor de lixo para liberar memória, temporizados e sensores que por ventura estavam sendo usados pelo módulo.

Think é um framework para a construção de sistemas operacionais baseado no modelo Fractal [FSLM02, PS08]. Fractal é um modelo de componentes hierárquicos onde a comunicação entre os componentes é feita através de interfaces cliente/-servidor. Os componentes disponibilizam interfaces de serviços que servem como ponto de acesso para as suas implementações e também exportam as funcionalidades requeridas de outros componentes através de uma interface cliente. O componente é dividido

em duas partes: conteúdo que implementa as funcionalidades e membrana que implementa o controle sobre o comportamento do componente. Desta forma, o modelo Fractal se transforma em reflexivo, pois as membranas podem controlar as estruturas e comportamento interno do componente [BCL+06]. Fractal, e consequentemente o Think, são configurados através de uma linguagem de descrição de arquitetura (ADL - *Architecture Description Language*). A partir de uma descrição, o compilador ADL gera código na linguagem C dos componentes e o código necessário para fazer o controle e comunicação dos componentes transparentemente. Think permite reconfiguração através de uma membrana especial que provê funções para a aplicação parar um componente específico, transferir o estado do componente antigo para o novo e atualizar as referências que estavam apontadas para o antigo componente. A maior deficiência do framework Think é a quantidade de memória necessária para a sua execução. Think ocupa 109kb de memória de código e 13kb de memória de dados, o que se torna inviável para sistemas profundamente embarcados com severas restrições de armazenamento [PMSD07].

Tabela 2.1: Resumo do processo de reconfiguração nos sistemas operacionais analisados.

Sistema Operacional	**Processo de Reconfiguração**
TINYOS	Sem suporte direto
MantisOS	Não tem suporte
Nano-Kernel	Módulos reconfiguráveis
RETOS	Relocação dinâmica e ligação em tempo de execução
Contiki	Módulos reconfiguráveis
SOS	Módulos reconfiguráveis
Think	Modelo reflexivo (Fractal)

A Tabela 2.1 resume o processo de reconfiguração dinâmica de software nos sistemas operacionais para sistemas embarcados analisados. MantisOS não possui suporte à reconfiguração. Reconfiguração de software no TINYOS não é suportada diretamente, apenas através de máquinas virtuais ou infra-estruturas adicionadas no sistema operacional. Nano-Kernel, Contiki e SOS possuem módulos reconfiguráveis que podem

ser adicionados e removidos no sistema através de uma interface de registro e remoção durante a execução do sistema. RETOS suporta reconfiguração de software através de relocação dinâmica e ligação de código/dados em tempo de execução. Finalmente, o sistema Think tem suporte à reconfiguração devido ao modelo reflexivo Fractal.

Capítulo 3

Embedded Parallel Operating System (EPOS)

EMBEDDED PARALLEL OPERATING SYSTEM (EPOS) nasceu como um estudo de caso do Projeto de Sistemas Orientados à Aplicação (AOSD) [Frö01]. A AOSD faz uso da Engenharia de Domínio para definir componentes que representam entidades significantes em diferentes domínios. As variações dentro de cada domínio são tratadas como definido no Projeto Baseado em Famílias (FBD - *Family-Based Design*) [Par76], na qual a AOSD modela as abstrações independentes e as organiza como membros de famílias. Mesmo independentes e separadas em famílias, essas abstrações podem ainda ter dependências e sua reusabilidade em diferentes cenários torna-se mais complicada. Para reduzir essas dependências e aumentar a reusabilidade, a AOSD utiliza um processo de decomposição baseado na Programação Orientada a Aspectos (AOP). Com isso, as variações não são modeladas dentro das abstrações, mas sim como aspectos de cenários [FSP00].

O EPOS é um framework multi-plataforma e baseado em componentes para geração de sistemas embarcados. Os serviços de sistema operacional tradicionais são implementados no EPOS através de abstrações de sistemas (também chamados de componentes) que são independentes de plataforma. O EPOS possui porte para diversas arquiteturas, como IA32, AVR, PPC, SparcV8 e MIPS. Pela importância neste trabalho e

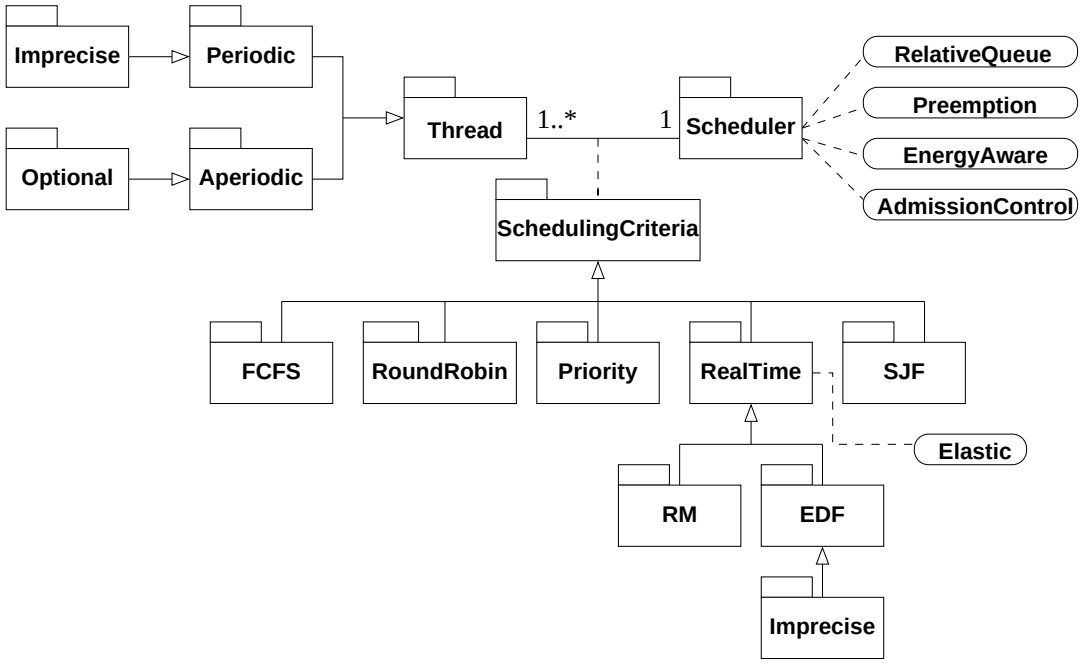

Figura 3.1: Threads, escalonador e critérios de escalonamento do EPOS.

também como forma de atualizar os recentes desenvolvimentos realizados, este capítulo apresentará um resumo dos principais componentes do EPOS.

3.1 Gerenciamento de Processos

Os processos no EPOS são gerenciados pelos componentes `Thread` e `Task`. O conceito de `Task` é aplicado às atividades específicas do programa, enquanto que as `Thread` são as entidades que executam tais atividades. Essa separação de conceito permite que uma `Task` seja executada por múltiplas `Threads` [Frö01, MHWF06]. A Figura 3.1 apresenta as famílias de componentes responsáveis pelo gerenciamento de processos no EPOS. Uma `Thread` pode ser aperiódica ou periódica, na qual é realizada através de uma especialização da classe `Thread` que agrega a esta mecanismos para sua reexecução através do uso do componente `Alarm` [MHWF06].

O escalonamento das Threads é realizado através dos componentes `Scheduler` e `SchedulingCriteria` na qual separa a política de escalona-

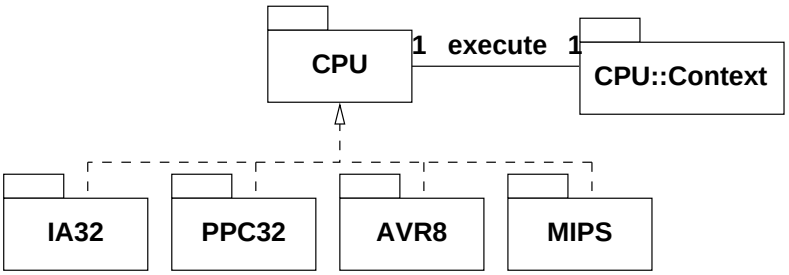

Figura 3.2: Mediador de hardware CPU.

mento (*scheduling criteria*) do seu mecanismo (implementação das filas de escalonamento) [MCSF09]. O escalonador também apresenta propriedades configuráveis, como preempção, controle de admissão de tarefas e parâmetros de energia, que são usados para o gerenciamento de energia do sistema com as threads `Imprecise` e `Optional`. O EPOS ainda permite que um escalonador seja implementado tanto em software quanto em hardware, justamente pela separação dos conceitos de política e mecanismo de escalonamento [MCSF09].

As principais dependências arquiteturais das threads, como o contexto de execução e inicialização da pilha, são abstraídas pelo mediador de hardware CPU, exemplificado na Figura 3.2 [MHWF06]. O mediador de hardware CPU também implementa funções utilizadas por outros componentes do sistema, como as operações TSL (*Test and Set Lock*) e conversões relacionadas ao formato da arquitetura (*endianess*).

3.2 Sincronização

Sincronizadores são usados para evitar condições de corrida. Uma condição de corrida acontece quando duas ou mais threads compartilham recursos e uma thread escreve em dados enquanto outra(s) thread(s) os acessa [Tan07]. Como um exemplo, considere uma variável inteira que é compartilhada entre duas threads (thread A e B). A thread A lê esta variável e encontra o valor 10. Ocasionalmente, o escalonador decide trocar a thread em execução, trocando para a thread B. A thread B por sua vez, realizada a leitura da variável, executa uma soma e escreve o valor 50 nesta mesma variá-

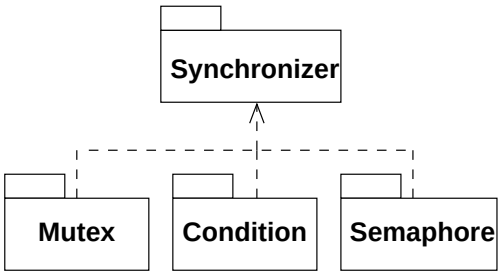

Figura 3.3: Família de componentes de sincronização.

vel. Após isso, a thread A retoma sua execução, sendo que neste ponto ela tem o valor 10 como sendo o valor da variável e não 50 como deveria ser. A variável inteira deveria ser protegida por uma seção crítica.

Seções críticas são protegidas por exclusão mútua, desta forma apenas uma thread pode entrar na seção crítica. O EPOS apresenta suporte a sincronização através da família de componentes descrita na Figura 3.3. O componente `Semaphore` implementa as operações de `p` e `v`. O componente `Mutex` implementa sincronização através dos métodos `lock` e `unlock`. Finalmente, o componente `Condition` suporta sincronização pelos métodos `wait`, `signal` e `broadcast`.

3.3 Gerenciamento de Memória

O projeto adequado da gerência de memória é um fator extremamente importante para que o sistema atinja um alto grau de portabilidade entre diversas plataformas, com hardware amplamente distintos [Frö01, Wan06, MHWF06]. A Figura 3.4 apresenta a família de componentes do EPOS responsável pelo gerenciamento de memória.

Para as aplicações, a memória disponível no sistema é vista como um segmento, implementado pela família `Segment`. O segmento é anexado a um determinado espaço de endereçamento pela família `Address_Space`. Esta família controla a alocação e a política de gerência de segmentos da memória física [Frö01]. O modo de endereçamento `Flat` define uma memória na qual os endereços físicos e lógicos são

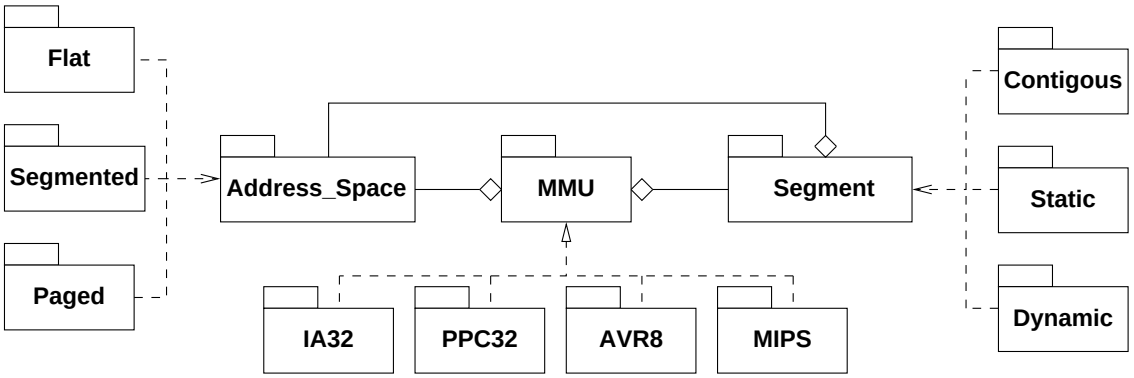

Figura 3.4: Família de componentes de gerenciamento de memória.

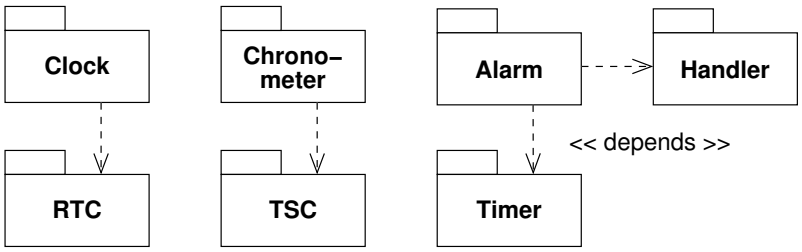

Figura 3.5: Componentes responsáveis pelo gerenciamento de tempo no EPOS.

os mesmos, eliminando assim a necessidade de uma unidade específica para o gerenciamento de memória (MMU). Para plataformas que não possuem MMU, o mediador de hardware MMU é apenas um simples artefato [Frö01, MHWF06].

3.4 Gerenciamento de Tempo

O tempo é gerenciado no EPOS pela família de componentes apresentada na Figura 3.5. O componente `Clock` é responsável por manter o tempo atual do sistema e é presente somente em sistemas que possuem `RTC` (*Real-Time Clock*) que é representado pela família de mediadores de hardware RTC. O componente `Chronometer` é usado para medir intervalos de tempo através do mediador do `TSC` (*TimeStamp Counter*).

O componente `Alarm` tem duas funções: gerar eventos de tempo e

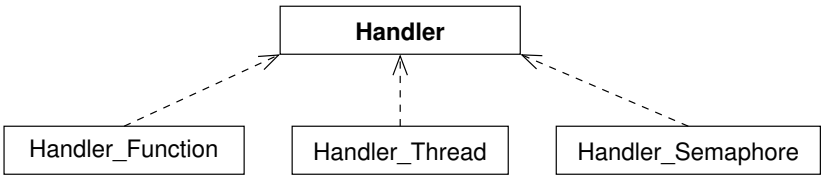

Figura 3.6: Tratadores de eventos suportados pelo EPOS.

colocar uma thread para "dormir" (*sleep*) por um certo período de tempo. Para isso, a aplicação instância um tratador (*handler*) e o registra com um Alarm especificando um período de tempo e o número de vezes que o objeto handler será chamado. O EPOS disponibiliza para a aplicação três tipos de tratadores, exemplificados na Figura 3.6. O membro `Handler_Function` associa uma função passada pela aplicação para o tratamento do evento. O membro `Handler_Thread` associa uma thread para o tratamento do evento. Esta thread já deve ter sido previamente criada pela aplicação no estado suspensa. Na ocorrência do evento, esta thread é liberada para execução, sendo que a escolha de qual thread irá executar é dever do escolador. Finalmente, o `Handler_Semaphore` associa um semáforo previamente criado pela aplicação ao tratamento do evento. O sistema operacional invoca o método `v()` deste semáforo na ocorrência do evento enquanto que a thread invoca o método `p()` para esperar por um evento.

A Figura 3.7 demonstra o uso da API de gerenciamento de tempo no EPOS. A aplicação começa lendo o tempo atual do sistema chamando o método `now` do componente `Chronometer`. Após isso, é criada uma função de tratamento de eventos, criado um `Alarm` que será executado 100 vezes e associada a função de tratamento a este `Alarm`. Finalmente, a aplicação chama o método `delay` para dormir um certo intervalo de tempo. Este intervalo é medido pelo componente `Chronometer` através dos métodos `start()` e `stop()`.

3.5 Gerenciamento de Energia

O EPOS disponibiliza gerenciamento de energia dirigido pela aplicação [JWF06, Jun07] ou dirigido pelo próprio sistema operacional [Wie08, WWGF08].

```
static int iterations = 100;
static Alarm::Microsecond time = 100000;

// ...

int main()
{
    OStream cout;
    Clock clock;
    Chronometer chron;

    // Realiza a leitura do tempo atual
    cout << "Tempo atual: " << clock.now() << endl;

    // Cria uma função de tratamento e a associa ao evento periódico
    Handler_Function handler(&func);
    Alarm alarm(time, &handler, iterations);

    // Inicia o chronometer e coloca a Thread para dormir
    // Depois, pára e lê o Chronometer
    chron.start();
    Alarm::delay(time * (iterations + 1));
    chron.stop();
    cout << "Tempo total: " << chron.read() << endl;

    return 0;
}
```

Figura 3.7: Exemplo do uso da API de gerenciamento de tempo.

```
System sys;
Thermometer therm;
UART uart;

void alarm_handler() {
    uart.put (therm.get()) ;
}

int main() {
    Handler_Function handler(&alarm_handler ) ;
    Alarm alarm(1000000 , &handler ) ;

    while ( 1 ) {
        sys.power(STANDBY) ;
    }
}
```

Figura 3.8: Exemplo do uso da API de gerenciamento de energia dirigido pela aplicação [JWF06].

No gerenciamento de energia dirigido pela aplicação, o SO disponibiliza uma interface pela qual a aplicação pode diminuir o consumo de energia dos componentes, tanto de software quanto de hardware. Esta interface permite que um componente seja colocado em um modo de operação que gaste menos energia, como por exemplo os modos *off*, *standby*, *light* e *full*. A interface tem dois métodos, um para obter o atual modo de operação do componente (`Power_Mode power()`) e outro para modificar o modo de operação do componente (`power(Power_Mode)`) [JWF06, Jun07].

A organização hierárquica dos componentes do SO permite que a aplicação mude o modo de operação em diferentes camadas do sistema. Um exemplo do uso da API do gerenciamento de energia dirigido pela aplicação é exemplificado na Figura 3.8 [JWF06]. A aplicação ao chamar o método `sys.power(STANDBY)` coloca

```
// ...
int main() {
    // ...
    Imprecise_Thread sensing(&mandatory_function, &optional_function,
        Criterion(150e3,170e3,INFINITE,0));
    // ...
}
```

Figura 3.9: Criação de uma thread imprecisa responsável pelo gerenciamento de energia dirigido pelo SO.

todos os componentes do sistema em modo STAND-BY, com exceção do componente **Alarm**.

O gerenciamento de energia dirigido pelo SO é executado no escalonamento do sistema. Com base no EDF [LL73] e computação imprecisa [LSL+94], uma thread é dividida em duas partes: obrigatória e opcional. A parte obrigatória sempre irá ter seus deadlines atendidos, independentemente dos níveis de energia. Já a execução da parte opcional só é realizada após a verificação dos níveis de energia e garantias tempo do sistema. O objetivo não é apenas economizar energia, mas sim atender uma quantidade de energia (e.g. tempo de vida da bateria), requisitada pela aplicação. A Figura 3.9 exemplifica o processo de criação de uma thread imprecisa no EPOS. O construtor recebe a função obrigatória, a função opcional e o critério, contendo o deadline, período, número de execuções e a fase da thread imprecisa. O sistema consegue estimar o seu tempo de vida através do monitoramento da carga da bateria [Gon07].

3.6 Comunicação

A Figura 3.10 demonstra a família de componentes responsáveis pela comunicação no EPOS. A comunicação realizada pelos processos do sistema é feita através do componente `Communicator`, que atua como uma interface para um canal de

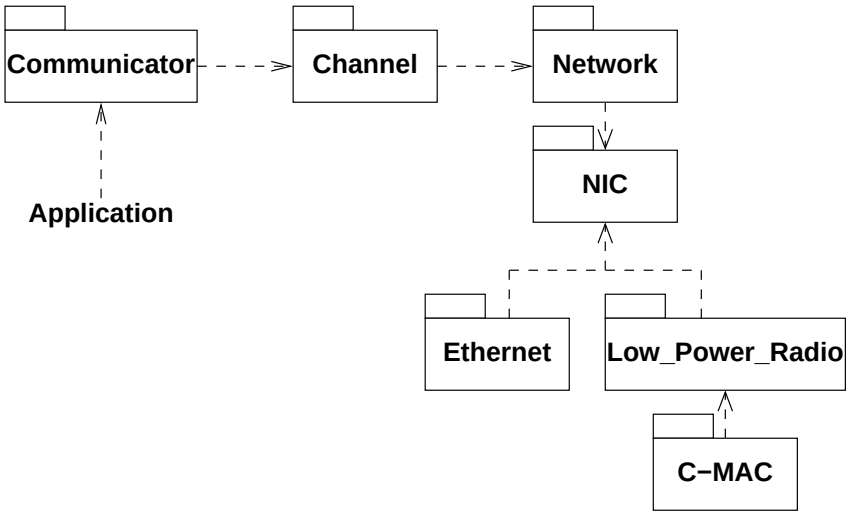

Figura 3.10: Família de componentes de comunicação.

comunicação (`Channel`) implementado sobre uma rede (`Network`) [Frö01, Wan06].

Os membros da família de componentes `Communicator` são o ponto de acesso para habilitar a troca de dados entre os processos. Esta família suporta, entre outras atividades, acesso assíncrono a segmentos de memória em um nodo remoto (*Asynchronous Remote Memory Segment*) e acesso a memória compartilhada (*Distributed Shared Memory*) [Frö01].

Os membros da família `Channel` são responsáveis pela comunicação inter-processos. Através de uma rede, é construído um canal lógico pelo qual as mensagens são enviadas e recebidas. Os membros da família implementam protocolos classificados como nível quatro no modelo OSI, incluindo membros como *Stream* e *Datagram* [Wan06].

Os membros da família `Network` disponibilizam os meios para a construção de uma rede física em cima dos canais lógicos. Os componentes dessa família abstraem as tecnologias de rede (NIC - *Network Interface Card*), sendo que para os componentes da família `Channel`, todas as redes são iguais [Frö01, Wan06].

A família `NIC` implementa as tecnologias de rede, entre elas ethernet e rádio com baixo consumo de energia. O membro `Low_Power_Radio` implementa um conjunto de métodos e estruturas comuns aos protocolos de controle de acesso ao meio

(MAC) para rádios de baixa potência [Wan06]. O protocolo C-MAC (*Configurable MAC*) utiliza esta estrutura para criar um protocolo de controle de acesso ao meio configurável pelo desenvolvedor [Wan06, WdOF07].

3.7 Mediadores de Hardware

Os mediadores de hardware [PF04] são artefatos que implementam funções dependentes de hardware, equivalentes a drivers de dispositivos em sistemas Unix, mas não são construídos como Camada de Abstração de Hardware (HAL - *Hardware Abstraction Layer*). Os mediadores de hardware implementam uma interface entre os componentes de software e os dispositivos de hardware através de técnicas de metaprogramação estática. Deste modo, o código do mediador é dissolvido no componente em tempo de compilação.

A Figura 3.11 mostra o componente `Chronometer` do EPOS na arquitetura AVR8. O componente utiliza o mediador de hardware `TSC` (*Time Stamp Counter*) que é responsável pela contagem de tempo no sistema. A título de exemplo, o método `start()` faz uma chamada ao método `time_stamp()` para verificar qual é o tempo atual. Esta chamada, por sua vez, não existe, pois o método `time_stamp()` é dissolvido em tempo de compilação dentro do método `start()` do componente `Chronometer`.

3.8 Framework Metaprogramado

O framework metaprogramado do EPOS é executado na compilação do sistema. O seu papel é unir os componentes do sistema e a aplicação em determinados cenários de execução para gerar a imagem final do sistema operacional. Através de regras de composição, o framework é capaz de adaptar e montar os componentes selecionados e satisfazer as dependências e limitações do sistema e também aplicar aspectos aos componentes, inclusive o aspecto de invocação remota de métodos. O framework é implementado através de técnicas de metaprogramação estática em C++ [Frö01].

Uma visão geral do framework é apresentada na Figura 3.12. A classe

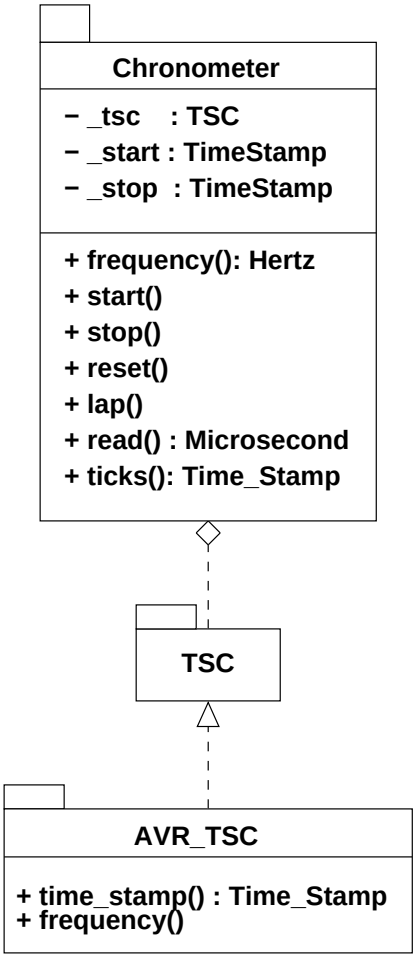

Figura 3.11: Componente Chronometer e o mediador de hardware AVR8_TSC do EPOS.

parametrizada `Handle` recebe um componente 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 o componente ou não. O aspecto de invocação remota é selecionado por um componente através da sua classe `Traits` [Str97]. Se o aspecto não estiver ativo, o `Stub` herdará o adaptador de cenário do componente [FSP00]. Caso contrário, uma especialização do `Stub` (*Stub<Componente, true>*), herdará o `Proxy` do componente. Consequentemente, quando *Traits<Componente>::remote* = *false* implica que o `Handle` seja implementado

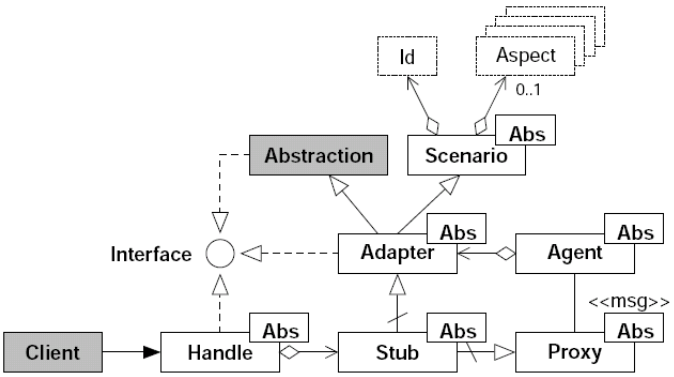

Figura 3.12: Framework metaprogramado do EPOS [Frö01].

como adaptador de cenário, enquanto que *Traits<Componente>::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 chamada do método. O `Agent` recebe a mensagem e invoca o método através do adaptador de cenário (`Adapter`).

A função da classe `Adapter` é aplicar os aspectos suportados pelo `Scenario` antes e depois da chamada real do método. Cada instância da classe `Scenario` consulta o `Traits` do componente para verificar quais aspectos estão habilitados para aquele componente, 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.

3.9 Considerações Parciais

Este capítulo apresentou um resumo dos principais elementos (componentes, mediadores de hardware e framework metaprogramado) que compõem o EPOS. Nesta dissertação, identificou-se que o framework metaprogramado possui duas impor-

tantes características: o confinamento e a isolação dos componentes do sistema.

A estrutura de `Proxy` e `Agent` encontradas no framework cria um nível de indireção entre as chamadas de métodos da aplicação para os componentes do sistema que possuem o aspecto de invocação remota habilitado. Esse nível de indireção isola os componentes, tornando-os independentes de posição na memória. Do ponto de vista da aplicação, uma chamada de método neste cenário é realizada normalmente, sem a ciência que o componente que está sendo invocado está localizado em outro nodo da rede. Da mesma forma, este mesmo conceito pode ser usado em um cenário de reconfiguração de software, pois apenas o framework conhece a localização dos componentes na memória do sistema, podendo atualizá-los transparentemente sem o conhecimento da aplicação. Essas características encontradas no framework metaprogramado do EPOS são importantes para a criação da infra-estrutura de reconfiguração dinâmica de software, na qual é descrita no próximo capítulo.

Capítulo 4

Epos Live Update System (ELUS)

Reconfiguração dinâmica de software é o processo de atualizar o software que está em execução em um determinado sistema. Esta atividade é extremamente importante quando se deseja corrigir erros, adicionar ou remover funcionalidades ou adaptar-se a mudanças durante o tempo de vida do sistema. Reconfiguração dinâmica de software em sistemas profundamente embarcados é um desafio ainda maior devido às próprias características desses sistemas, onde restrições de processamento, memória e energia influenciam diretamente no projeto e desenvolvimento do mecanismo de reconfiguração. Neste contexto, a estrutura de reconfiguração de software deve utilizar o mínimo de recursos possíveis para que não influencie no desempenho do sistema.

Conforme apresentado no capítulo 2, diversos trabalhos têm sido desenvolvidos para suportarem reconfiguração de software em sistemas profundamente embarcados. As infra-estruturas possuem como maior desvantagem o tempo necessário para executar uma reconfiguração, devido a relocação de código realizada no momento da atualização. Máquinas virtuais apresentam uma grande perda de desempenho devido à interpretação das instruções, aumentando o processamento e consequentemente o consumo de energia. Neste cenário, suporte à reconfiguração de software dentro do sistema operacional se torna uma boa alternativa pois abstrai a complexidade de uma reconfiguração.

Este capítulo apresenta o projeto e implementação do EPOS LIVE UP-

DATE SYSTEM (ELUS), uma infra-estrutura de sistema operacional que permite reconfiguração dinâmica de software em sistemas profundamente embarcados de maneira eficiente e segura. O ELUS possui algumas características, como configurabilidade, baixo consumo de memória, simplicidade e transparência, que o tornam diferente das infra-estruturas de sistema operacional existentes. Nas próximas seções serão apresentados os requisitos necessários em torno de uma reconfiguração de software que guiaram o desenvolvido do ELUS, bem como o projeto e a implementação dos elementos que compõem o sistema.

4.1 Requisitos

O EPOS possui um framework metaprogramado que permite invocação remota de métodos, conforme demonstrado no capítulo anterior. Neste framework, características como **confinamento** e **isolação** dos componentes do sistema são encontradas. Confinamento é importante para encapsular os componentes do sistema, criando um único ponto de acesso para invocação dos seus métodos. Isolação é importante para criar um nível de indireção entre as chamadas de métodos da aplicação para os componentes do sistema. Esse nível de indireção torna os componentes independentes de posição de memória, sendo que a forma que a aplicação invoca um método de um componente do sistema é realizada de forma transparente, como uma invocação de método normal. O ELUS utiliza esta duas características para criar o suporte à reconfiguração dinâmica de software no EPOS.

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 [PS08]:

- **Estado quiescente**: para que uma reconfiguração aconteça corretamente, o sistema deve atingir um estado de execução considerado consistente, passível de reconfiguração. Este estado consistente é chamado de estado quiescente [BD93, KM90, SAH+03], que é alcançado quando nenhuma atividade está sendo realizada no com-

ponente atualizado no momento da reconfiguração. Um componente é um artefato de software reusável, com uma interface claramente definida e que executa uma determinada função específica [Szy97]. Em um ambiente multithread, como o EPOS, o estado quiescente de um componente é atingido quando nenhuma `thread` está invocando métodos deste componente [PS08].

- **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` que podem ser disponibilizados pelos componentes [PS08].

- **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 profundamente 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 de software para sistemas profundamente embarcados deve utilizar o mínimo de recursos possíveis, pois estará compartilhando recursos com a(s) aplicação(ões). Em um nodo sensor MICA2 [HHKK04], por exemplo, a maior fonte de consumo de energia é o rádio, seguida por escritas e leituras da memória flash. 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.

4.2 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 EPOS LIVE UPDATE SYSTEM 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.

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

4.3 Arquitetura

O ELUS baseou-se nas características de confinamento e isolação dos componentes do sistema encontradas no framework metaprogramado do EPOS para construir o suporte à reconfiguração dinâmica de software. O framework metaprogramado foi estendido e a nova estrutura criada pelo ELUS é apresentada na Figura 4.1. A invocação de um método de um componente da aplicação cliente com suporte a reconfiguração de software 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) para cada componente reconfigurável, somente permitindo a chamada de métodos do componente que não está sendo atualizado no momento da sua invocação. Com isso o estado quiescente de cada componente é alcançado de forma segura. O suporte a reconfiguração de um componente é habilitado através dos seus `Traits` (*Traits<Componente>::reconfiguration = true*), semelhante ao suporte à invocação remota de métodos.

4.4 Tabela de Métodos Virtuais

A tabela de métodos virtuais (`vtable`) é gerada pelo compilador C++ para o(s) objeto(s) de uma classe que possui(em) métodos declarados como virtuais. A título de exemplo, considere o código C++ da Figura 4.2. A classe `Test` possui dois

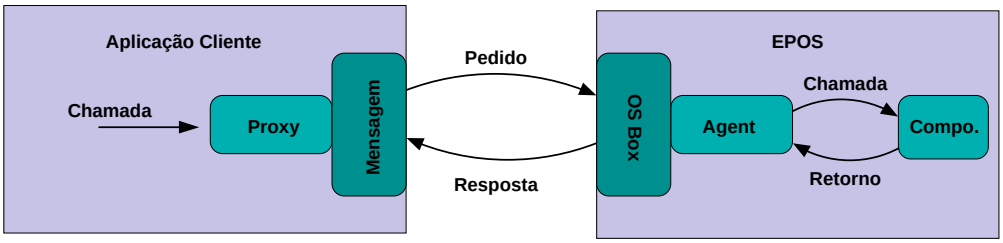

Figura 4.1: Visão geral da estrutura de invocação de métodos do ELUS.

métodos declarados como virtuais e dois atributos `a1` e `a2`. Após a criação dos objetos `t1` e `t2`, ambos fazem acesso a `vtable` para invocar os métodos virtuais `area` e `print`.

A `vtable` que seria criada pelo compilador GNU g++ 4.0.2 neste código é exemplificada na Figura 4.3. Os dois objetos `t1` e `t2` da classe `Test` acessam a mesma tabela de métodos virtuais para invocação dos métodos, porém, os dois objetos possuem os seus atributos separados. Por padrão, a ordem dos endereços dos métodos na `vtable` depende da ordem de declaração no código fonte, por isso neste caso o método `area` é seguido do método `print`. O primeiro argumento esperado pelos métodos `area` e `print` é o ponteiro para o objeto. Considerando o exemplo, seriam passados para os métodos os ponteiros para os objetos `t1` e `t2`, sendo que este código é gerado automaticamente pelo compilador.

O objeto de uma classe virtual possui o tamanho de todos os seus atributos privados somados com a posição necessária para armazenar o endereço da `vtable`, cujo tamanho é dependente de plataforma. Após a criação do objeto, o endereço da `vtable` é copiado para a primeira posição alocada na memória para o objeto, sendo que o endereço desta posição é utilizada para fazer o acesso a `vtable`.

Conforme já comentado na seção 4.2, a `vtable` do objeto é utilizada pelo ELUS para ajustar as referências dos métodos após uma reconfiguração. Considerando o exemplo anterior, se o método `area` fosse atualizado com um tamanho de código maior que o atual, seria alocado um espaço na memória para o novo código e o endereço do método atualizado na `vtable`.

Por padrão da linguagem C++ os construtores de uma classe não podem

```
// classe Test com dois métodos virtuais e dois atributos
class Test {
public:
    virtual void area();
    virtual void print();
public:
    int a1;
    short a2;
};

// ...

int main(void) {
    // declara objetos a e b da classe Test
    Test *t1 = new Test();
    Test *t2 = new Test();
    // ...
    t1->print();
    t2->area();
    // ...
}
```

Figura 4.2: Código exemplo da tabela de métodos virtuais.

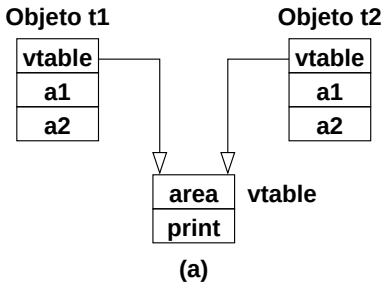

Figura 4.3: Tabela de métodos virtuais da Figura 4.2 demonstrando os objetos t1 e t2 e os endereços dos métodos area e print da classe Test

ser declarados como virtuais, pois no momento da execução de um construtor o objeto que aponta para a `vtable` ainda não existe, consequentemente não faz sentido um construtor ser declarado como virtual. Considerando um componente com suporte à reconfiguração dinâmica, se a atualização do código do(s) construtor(es) é necessária, todas as atividades realizadas por eles devem ser implementadas em um ou mais métodos normais (e.g. `create` ou `init`). Assim, a atualização desse código no ELUS é tratada normalmente.

4.5 Protocolo de Transporte

O ELUS utiliza um protocolo de transporte, chamado de ELUS TRANSPORT PROTOCOL (ETP), para o recebimento de mensagens solicitando reconfiguração. O primeiro tipo de mensagem (enviada pela aplicação através do `Proxy`) é simplesmente uma invocação de método normal. Neste caso, o `Agent` irá invocar o método do componente baseando-se nos IDs do componente e método recebidos, que são anexados na mensagem pelo `Proxy`, após recuperar o objeto referente àquele componente. A Figura 4.4 demonstra os campos deste tipo de mensagem. Os objetos são mantidos em uma tabela *hash*, cujo o identificador do objeto na tabela é o seu próprio ponteiro `this`. Ao invocar um método, o objeto usa os endereços da `vtable` para executar o código do método.

Os próximos tipos de mensagens são diferenciados por um byte chamado de controle. Os 4 bits menos significativos do controle definem o tipo de mensagem

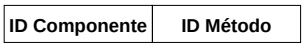

Figura 4.4: Mensagem de invocação de método do ETP.

e os 4 bits mais significativos definem a quantidade de campos que a mensagem contém, que varia de acordo com o tipo de mensagem. Toda a escrita e leitura da memória de código é realizada através de um gerenciador de código. Os tipos das mensagens são apresentados a seguir:

- **Adição de método sem relocação** `(código 0 - 0000)`**:** esta mensagem informa ao `Agent` que um pedido de adição de método foi requisitado. Um método adicionado em um componente implica também na sua adição no framework e na `vtable` dos objetos. Porém, em uma adição de método sem relocação não há a necessidade de relocação do `Dispatcher`, que é o vetor que contém os endereços dos métodos do framework. Além do byte de controle, a mensagem contém o tamanho do código, o código e a posição do método que está sendo adicionado dentro do código enviado (Figura 4.5).

- **Adição de método com relocação** `(código 1 - 0001)`**:** a diferença deste tipo de mensagem para o anterior é que o `Dispatcher` precisa ser relocado para outra posição, pois o tamanho do vetor não suporta a adição de mais um método. Para isso, a mensagem também contém o novo tamanho do `Dispatcher` (Figura 4.5).

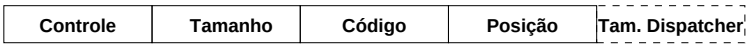

Figura 4.5: Mensagem de adição de método do ETP (códigos 0 e 1).

- **Remoção de método** `(código 2 - 0010)`**:** a remoção de um método é informada pelo código 0010 dos quatro primeiros bits do controle (Figura 4.6). A remoção do método implica na liberação da área de memória de código que estava sendo usada pelo método e a atualização dos ponteiros dentro do framework

e `vtable` para não mais apontarem para o método (endereço *null*). É tarefa do desenvolvedor garantir que o método removido não seja mais usado pela aplicação e/ou outros componentes.

Figura 4.6: Mensagem de remoção de método do ETP (código 2).

- **Atualização de componente com tamanho de código menor ou igual ao antigo** `(código 3 - 0011)`**:** esta mensagem (Figura 4.7) informa ao `Agent` que um componente está sendo atualizado, mas não é necessário alocar memória para o novo código do componente, pois este será colocado na mesma posição de memória do código antigo. O `Agent` irá atualizar os endereços dos métodos na `vtable` usando o tamanho de cada método enviado pela mensagem.

- **Atualização de componente com tamanho de código maior que o antigo** `(código 4 - 0100)`**:** neste tipo de mensagem (Figura 4.7), o `Agent` irá alocar memória para o novo componente através do gerenciador de código e atualizar as referências na tabela de métodos virtuais do objeto(s) para apontarem para os novos endereços. A área de memória antiga também é liberada para uso futuro e gerenciada pelo gerenciador de código.

- **Atualização de componente no framework com tamanho de código menor ou igual ao antigo** `(código 5 - 0101)`**:** neste tipo de reconfiguração (Figura 4.7), os métodos dentro do framework são atualizados com o novo código recebido. Não é necessária alocação de um novo espaço de memória para o componente. Após sua atualização, os endereços dos métodos do componente no `Dispatcher` são atualizados.

- **Atualização de componente no framework com tamanho de código maior que o antigo** `(código 6 - 0110)`**:** esta reconfiguração informa que um método dentro do framework está sendo atualizado e que o novo código é maior que o an-

tigo, necessitando alocação de memória e ajuste das referências no `Dispatcher` (Figura 4.7).

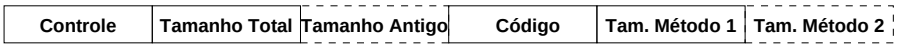

Figura 4.7: Mensagem de atualização de componente e framework do ETP (códigos 3 a 6).

- **Atualização de endereço específico** `(código 7 - 0111)`**:** a Figura 4.8 mostra este tipo de mensagem. Nesta mensagem, são enviados o endereço, o tamanho do código e o novo código a ser atualizado. O novo código é colocado a partir do endereço recebido. Este tipo de mensagem permite a atualização de endereços específicos, permitindo uma maior flexibilidade nas reconfigurações do ELUS. Os 4 bits mais significativos do controle representam o número total de mensagens extras que estão anexadas após o código. Essas mensagens extras possuem os próximos endereços a serem atualizados.

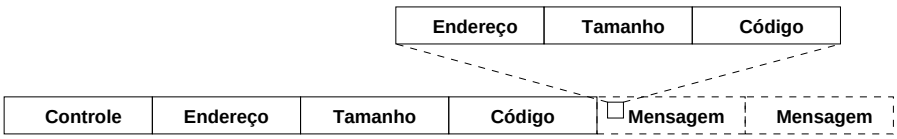

Figura 4.8: Mensagem de atualização de endereços do ETP (código 7).

- **Atualização da aplicação** `(código 8 - 1000)`**:** este tipo de mensagem é exemplificado pela Figura 4.9. Esta mensagem é enviada quando a aplicação é atualizada com um tamanho do seu novo código maior que o atual. Neste caso, é necessário alocar memória através do gerenciador de código para o novo código. Após a alocação, os endereços enviados na mensagem depois do código são atualizados com a nova posição alocada e assim as referências antigas para a aplicação são mantidas de forma consistente. Os 4 bits mais significativos do controle possuem o número de endereços que devem ser atualizados e que estão anexados na mensagem. Quando a aplicação precisa ser atualizada mas o tamanho do seu código é menor ou igual ao atual, é simplesmente utilizada uma mensagem de atualização

de endereço específico. O `Reconfigurator` não utiliza o `Agent` para atualizar a aplicação e sim um componente, chamado de `Application_Update`. O componente foi criado única e exclusivamente para permitir que a aplicação seja atualizada, pois todo o sistema deve ser interrompido antes da sua atualização. A seção 4.11 explica o processo de atualização da aplicação em maiores detalhes.

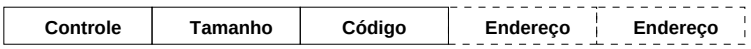

Figura 4.9: Mensagem de atualização da aplicação do ETP (código 8).

- **Adição de Atributos** `(código 9 - 1001)`**:** este tipo de mensagem suportado pelo ETP é exemplificado na Figura 4.10. Esta mensagem informa uma requisição para adição de atributos em um componente e envia para o `Agent` o tamanho do objeto que deve ser criado somados os 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 deletar 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.

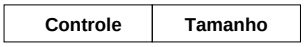

Figura 4.10: Mensagem de adição de atributos do ETP (código 9).

Todas as mensagens de reconfiguração são enviadas por uma `Thread`, chamada de `Reconfigurator`, criada na inicialização do sistema e transparente para aplicação, responsável por receber a solicitação de reconfiguração do ambiente externo, seja por rádio, UART, ethernet, etc. As referências externas utilizadas no novo código do componente são resolvidas efetuando a sua ligação com o sistema antigo, que não sofreu mudanças, durante a compilação do componente.

Os dados são recebidos pelo `Reconfigurator` que constrói uma mensagem no formato ETP e envia o pedido de reconfiguração para o `Agent` ou para o componente `Application_Update`. Os dados recebidos são armazenadas temporariamente até o recebimento completo do novo código, para que assim seja possível a realização da reconfiguração. O armazenamento temporário é dependente de plataforma, sendo que pode ser feito conforme a disponibilidade de memória, seja ela RAM, FLASH ou EEPROM. Se o código e as informações recebidas excederem o tamanho do armazenamento, uma mensagem de reconfiguração é enviada e o recebimento dos dados é prosseguido após a reconfiguração. Todas as mensagens também possuem o ID do componente que é anexado na mensagem pela chamada a um método específico (veja a Figura 4.18).

O ETP juntamente com a estrutura de passagem e recebimento de mensagens permite que o ELUS seja facilmente integrado com um protocolo de disseminação de dados. Para isso, basta que o protocolo seja adaptado para enviar mensagens no formato ETP.

4.6 Gerenciador de Código

O gerenciador de código é responsável pelo gerenciamento da memória de código do sistema, lendo, escrevendo, alocando e liberando espaços de memória através de uma interface simples, apresentada na Figura 4.11. O método `read` realiza a leitura de espaço de memória começando pelo endereço (*address*) e com tamanho (*size*) recebidos como parâmetro. O método `alloc` aloca um espaço de memória livre e retorna um ponteiro para a posição alocada. O método `free` libera um espaço de memória. Enquanto que o método `write` escreve os dados (*data*) com tamanho (*size)* em um endereço de memória (*address*) passados como parâmetros.

O gerenciador de código abstrai as diferenças de memória presentes nas diversas arquiteturas em que o EPOS executa através dos mediadores de hardware, conforme a plataforma (IA32, AVR8, PPC32, etc) e a memória utilizada (Flash, RAM, EEPROM). Por isso, para o `Agent`, a leitura/escrita, a alocação, a liberação e o *endianess* (*big* ou *little endian*) tornam-se transparentes de arquitetura e tecnologia da memória de

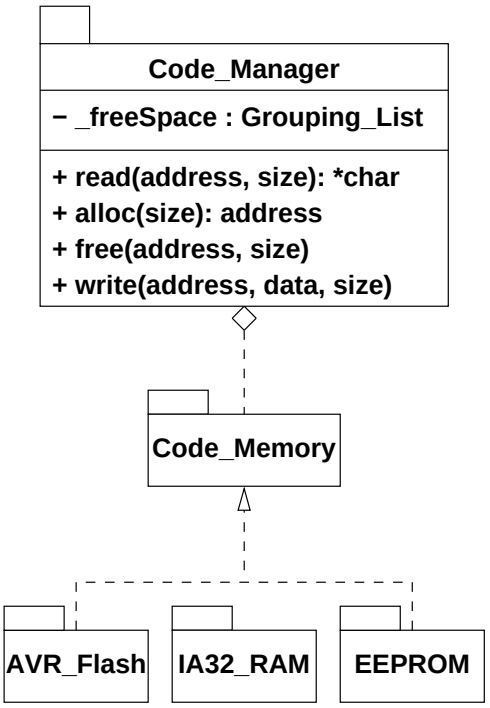

Figura 4.11: Gerenciador de código e os mediadores de hardware para escrita e leitura da memória de código.

código. A Figura 4.12 exemplifica a memória do sistema antes e depois da reconfiguração de um componente com aumento no tamanho do seu código. Os espaços de memória livre são controlados pelo gerenciador e utilizados para futuras reconfigurações.

4.7 Otimização do Código Enviado pela Rede

A fim de diminuir o número de bytes transmitidos pela rede e consequentemente economizar energia, pode-se enviar somente as diferenças entre o novo e o antigo código do componente. Esta abordagem é chamada de `diff` [KP05a] e também pode ser adicionada na estrutura de mensagens do ETP.

Os algoritmos de compressão de dados também podem ser usados para reduzir a quantidade de dados transmitidos pela rede. Um algoritmo de compressão reduz o número de bytes enviados realizando codificação nos dados que compõem o código que está sendo atualizado. Porém, há um aumento no processamento e energia por causa

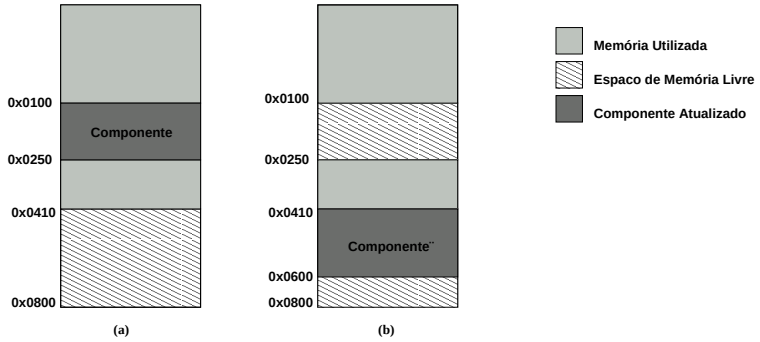

Figura 4.12: Exemplo da memória do sistema antes (a) e depois (b) da reconfiguração de um componente.

da execução do algoritmo de descompressão dos dados. O grande desafio é equilibrar o consumo de energia economizado pela redução dos bytes enviados e o consumo (tempo e processamento) necessário para realizar a descompressão. Um estudo realizado mostrou que é possível economizar em média 69% de energia e reduzir em 67% o tempo gasto em uma disseminação de dados em uma RSSF usando um algoritmo de compressão de dados como o GZIP [TDV08].

Tanto os mecanismos baseados em `diff` quanto os algoritmos de compressão podem ser envolvidos por um protocolo de disseminação de dados, que é responsável, entre outras funções, por garantir a entrega dos dados e reduzir o consumo de energia nas transmissões e recepções dos dados enviados pela rede. Como explicitado anteriormente, um protocolo de disseminação de dados pode ser facilmente adaptado para que receba os dados corretamente e construa mensagens no formato ETP, e assim possa ser integrado ao ELUS.

4.8 Visão Geral do Processo de Reconfiguração

A infra-estrutura do ELUS é transparente 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 reconfiguração habilitado no sistema, as chamadas de métodos em

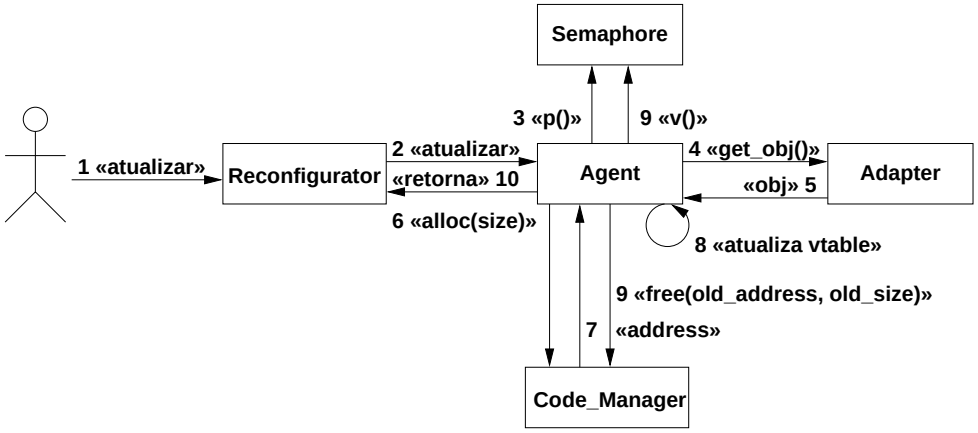

Figura 4.13: Resumo da sequência de atividades realizadas pelo ELUS em uma reconfiguração de software.

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. Esses dois parâmetros são mensurados no capítulo 5.

A Figura 4.13 resume de maneira geral a sequência de atividades executadas pelo ELUS em uma reconfiguração. Ao receber um pedido de reconfiguração e os dados do novo código, o `Reconfigurator` informa o `Agent` através de uma mensagem no formato ETP, que contém o identificador do componente, byte de controle, tamanho e o novo código do(s) método(s). O `Agent`, após o recebimento da mensagem, coloca o componente no seu estado quiescente através de um semáforo. Após isso, o `Agent` aloca memória para o novo código através do gerenciador de código, atualiza os endereços na `vtable` do objeto(s), libera o semáforo e retorna para o `Reconfigurator`.

4.9 Framework

4.9.1 Configurabilidade

O framework originalmente proposto para permitir invocação remota de métodos no EPOS foi modificado para suportar também reconfiguração dinâmica de software. O suporte à reconfiguração dinâmica em um componente é habilitado pelo seu

```
namespace Imp {
// Seleciona ou não o suporte à reconfiguração em um componente
template <> struct Traits<Component>: public Traits<void>
{
    static const bool framework = true;
    static const bool reconfiguration = true;
};
// ...
}

namespace System {
// Exporta para o namespace da aplicação o componente
// como um parâmetro para o Handle
typedef IF<Imp::Traits<Imp::Component>::framework,
    Handle<Imp::Component>,
    Imp::Component>::Result Component;
// ...
}
```

Figura 4.14: Código exemplo para habilitar o suporte à reconfiguração e exportar o componente para aplicação.

`Traits`. `Traits` são classes parametrizadas que descrevem propriedades relacionadas aos componente. Essas propriedades quando não selecionadas não adicionam nenhum sobrecusto para o código objeto do componente gerado, justamente pelo uso de metaprogramação estática e *inlining* de funções.

Os suportes ao framework e a reconfiguração dinâmica de software são verificados na compilação do sistema. Existem dois *namespaces*, um para a implementação dos componentes (*Imp*) e outro para a aplicação (*System*). Todos os componentes são exportados para o *namespace* da aplicação transparentemente, sendo que quando o framework está habilitado, o componente é exportado como um parâmetro para a classe

`Handle` (*Handle<Imp::Component>*), que é responsável por verificar se o suporte à reconfiguração está habilitado ou não para o componente. A comunicação entre componentes também deve ser realizada através do framework. Para isso, quando um componente utiliza outro componente, sua declaração deve ser feita adicionando o namespace *System* à frente do nome do componente (*System::Component*).

A Figura 4.14 exemplifica o processo de habilitação ou não do framework e do suporte à reconfiguração dinâmica de software em um componente e também o código que exporta o componente para o *namespace* da aplicação. O framework deve ser habilitado para também suportar invocação remota de métodos. A escolha entre invocação remota de métodos ou reconfiguração dinâmica é realizada pelo `Handle`.

O `Reconfigurator` também é habilitado no sistema a partir do `Traits`, conforme o código da Figura 4.15. Se pelo menos um componente possui suporte a reconfiguração dinâmica, o `Reconfigurator` deve estar habilitado no sistema, pois ele tem a função de receber uma requisição e os novos dados de uma reconfiguração. Com a `Thread` selecionada, sua criação é realizada na inicialização do sistema. Caso contrário, nenhum sobrecusto é introduzido no sistema em tempo de execução.

4.9.2 Handle

A classe `Handle` é responsável por verificar se o suporte à reconfiguração dinâmica está habilitado ou não para um determinado componente. Isso é feito consultando o `Traits` do componente (*Traits<Componente>::reconfiguration*). Os métodos construtores do `Handle` são parametrizados e podem receber de 0 a 4 parâmetros que são passados aos construtores da classe `Stub`, como por exemplo, *stub = new _Stub(a1)*. O tipo *_Stub* é definido de acordo com a habilitação ou não da opção de reconfiguração do componente no `Traits`:

```
typedef Stub<T, Imp::Traits<T>::reconfiguration> _Stub;
```

Sendo que *T* é o componente passado ao `Handle` como parâmetro do *template*.

```
template <> struct Traits<Reconfigurator>: public Traits<void>
{
    // habilita (true) ou não (false) o Reconfigurator
    static const bool enabled = true;
};

// Inicialização das Threads do sistema
// ...
if ( Traits <Reconfigurator>::enabled)
    Reconfigurator:: init () ;
// ...
```

Figura 4.15: Traits para habilitar ou não o suporte ao Reconfigurator e a sua criação na inicialização das Threads do sistema.

A Figura 4.16 apresenta a classe `Handle`. Os métodos implementados pelo componente são também implementados no `Handle`, porém, as invocações de métodos recebidas pelo `Handle` são apenas enviadas para o `Stub`. Por exemplo, a invocação do método `send` que recebe três parâmetros seria realizada da seguinte forma: *stub->send(p1, p2, p3)*.

4.9.3 Stub

A classe `Stub` é responsável por criar um `Adapter` ou `Proxy` conforme os parâmetros recebidos na sua criação via `Handle`. A Figura 4.17 mostra a implementação da classe `Stub`. A classe recebe um componente (*T*) e uma variável booleana como argumentos do *template*. Se a variável booleana for falsa, o `Stub` irá herdar a classe `Adapter` e simplesmente irá chamar os construtores do `Adapter`. Caso contrário, se o argumento booleano for verdadeiro, o `Stub` herdará a classe `Proxy` e chamará os construtores da mesma.

Desta forma, se o desenvolvedor marcar como verdadeiro o `Trait` de

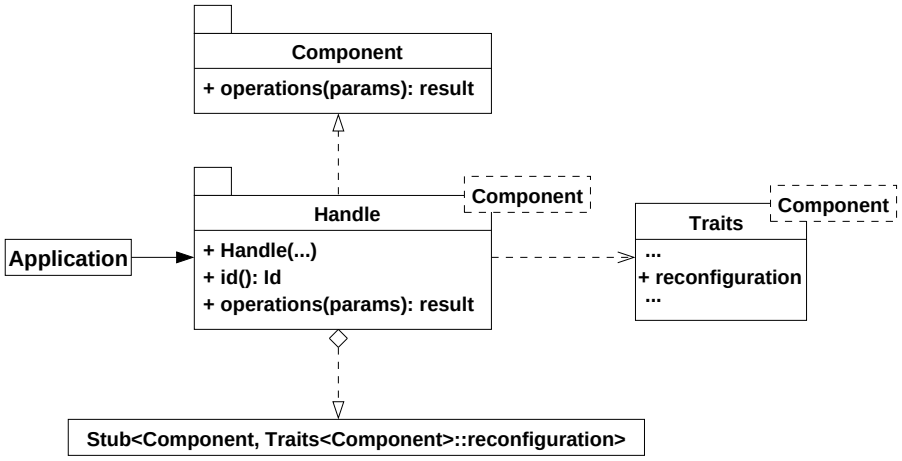

Figura 4.16: A classe Handle do ELUS.

reconfiguração de um componente (*Traits<Componente>::reconfiguration> = true*), o `Handle` irá criar um `Stub` que contém todos os métodos do `Proxy`, permitindo assim a comunicação do `Proxy` com o `Agent` e criando um nível de indireção entre as chamadas de métodos da aplicação para os componentes do sistema.

4.9.4 Proxy e Agent

A estrutura de `Proxy` e `Agent` somente estará presente se o suporte à reconfiguração dinâmica de software estiver habilitado para algum dos componentes do sistema. A Figura 4.18 mostra os elementos `Proxy` e `Agent` e a estrutura de comunicação entre os dois elementos.

A classe `Message` é o ponto comum entre o `Proxy` e o `Agent` para invocações de métodos, passagem de parâmetros e recebimento do valor de retorno após a execução de um método. A passagem de parâmetros é realizada através dos métodos `out` e `in`, conforme exemplifica o código da Figura 4.19. `out` anexa parâmetros na mensagem e o `in` realiza a leitura desses parâmetros. O método `out` é usado no `Proxy` e aceita qualquer tipo de parâmetro devido a metaprogramação estática. Os parâmetros recebidos são colocados dentro do vetor `params` e a variável "tamanho" (*length*) informa quantos argumentos estão sendo passados pela mensagem. O método `in` é usado no `Agent` para

```
template<class T, bool reconf>
class Stub: public Adapter<T>
{
public:
   Stub()  {}
   template<class T1>
   Stub(T1 &a1): Adapter<T>(a1) {}
   template<class T1, class T2>
   Stub(T1 &a1, T2 &a2): Adapter<T>(a1, a2) {}
\\   ...
};

template<class T>
class Stub<T, true>: public Proxy<T>
{
public:
   Stub()  {}
   template<class T1>
   Stub(T1 &a1): Proxy<T>(a1) {}
   template<class T1, class T2>
   Stub(T1 &a1, T2 &a2): Proxy<T>(a1, a2) {}
\\   ...
};
```

Figura 4.17: Implementação da classe Stub.

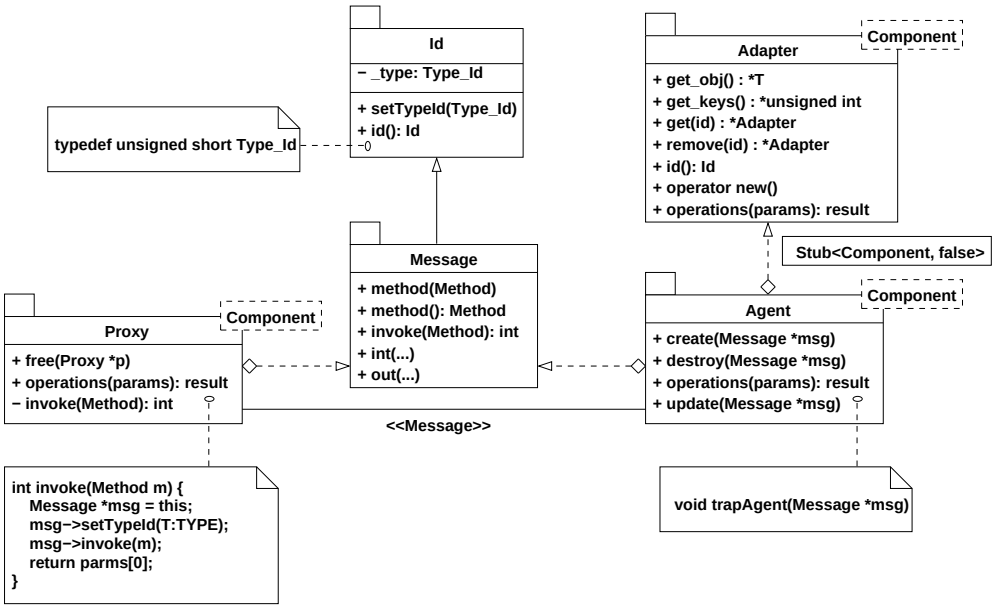

Figura 4.18: Os elementos Proxy e Agent e a estrutura de comunicação entre eles.

ler os parâmetros recebidos quando necessário. O `Agent` também usa o método `out` para anexar na mensagem o valor de retorno de um método. Esta estrutura de passagem e recebimento de parâmetros/valor de retorno é extremamente eficiente devido ao uso da metaprogramação estática, fazendo com que o código gerado não utilize a pilha como meio para passagem de argumentos e sim anexe os argumentos diretamente na mensagem.

O método `invoke` da classe `Message` é chamado pelo `Proxy` a fim de invocar um método do `Agent`. Este método simplesmente envia a mensagem para o `Agent` através da chamada à função `trapAgent`, que é a caixa de entrada (*SO Box*) para os métodos do `Agent`. O tipo `Method` representa a definição dos identificadores (IDs) dos componentes do sistema e dos métodos de cada componente declarados como virtuais e assim passiveis de reconfiguração. A última declaração dentro de cada componente é referente ao método de reconfiguração de código (*update*), que é enviado para o `Agent` através do `Reconfigurator`. A implementação do método `invoke` e a declaração do `Method` são também representados na Figura 4.19. Por fim, a classe `Message` generaliza a classe `ID`, que tem como objetivos identificar qual o componente que está sendo referenciado pela mensagem e qual é o identificador do objeto (o próprio ponteiro

```
enum Method
{
  CREATE = 0, DESTROY = 1, FAMILY = 2,
  THREAD = 0, THREAD_SUSPEND = FAMILY, THREAD_RESUME,...,
      THREAD_UPDATE, // ...
  SYNCHRONIZER = 1, SYNCHRONIZER_LOCK = FAMILY,
      SYNCHRONIZER_UNLOCK,..., SYNCHRONIZER_UPDATE // ...
};

class Message : public Id
{
public:
  int  invoke(const Method &m) {
      method(m); trapAgent(this); return method();
  }
  void method(const Method &m) { _method = m; }
  const Method &method() { return _method; }
  template<typename T1>
  void out(const T1 &v1) { reinterpret_cast<T1 &>(parms[0]) = v1; length = 1; }
  template<typename T1, typename T2>
  void out(const T1 &v1, const T2 &v2) {...}
  //  ...
  template<typename T1>
  void in(T1 &v1) { v1 = reinterpret_cast<T1 &>(parms[0]); }
  template<typename T1, typename T2> void in(T1 &v1, T2 &v2) { ... }
  //  ...
private:
  Method _method; int length; char parms[30];
};
```

Figura 4.19: Implementação do mecanismo de troca de mensagem da classe Message.

this).

O Proxy é uma classe parametrizada que recebe como parâmetro um ou mais componentes e que realiza os métodos presentes nestes componentes. Ao receber uma invocação de método via Handle, o Proxy usa a estrutura da classe Message para enviar a invocação de método para o Agent. A Figura 4.20 mostra a implementação do Proxy. Ao receber uma solicitação de criação de um objeto, por exemplo, o construtor da classe Proxy irá chamar o método invoke passando como argumento a opção CREATE (referente a criação de um objeto). O método invoke irá anexar na mensagem o ID do componente (*T::TYPE*) e enviar a mensagem para o Agent através do método invoke da classe Message (veja Figura 4.18).

O método free do Proxy simplesmente tem o objetivo de chamar o destrutor e deletar o objeto previamente criado. São utilizados os métodos out e in para a passagem e recebimento de parâmetros quando necessário.

O elemento Agent é uma classe parametrizada que recebe um ou mais componentes como parâmetro e realiza os métodos destes componentes. O Agent possui duas funções: (i) invocação de métodos e (ii) reconfiguração dos componentes e do próprio framework. A função trapAgent é o ponto de entrada para a realização destas duas atividades. A função, primeiramente, acessa um vetor bidimensional, chamado de Dispatcher, que possui ponteiros para os métodos implementados pelo Agent. Os índices usados para o acesso ao Dispatcher são os identificadores do componente e método, ambos anexados e enviados na mensagem pelo Proxy ou, no caso de uma reconfiguração, pelo Reconfigurator. Após, o Agent atinge o estado quiescente do componente através da chamada ao método p do semáforo relativo ao componente e realiza a chamada de método. A Figura 4.21 apresenta a implementação do Dispatcher e do trapAgent. O valor *MAX_METHODS* é definido pelo desenvolvedor em tempo de compilação do sistema.

Considere um exemplo em que a aplicação faz uma chamada aos métodos int get() e void set(int) de um componente hipotético. Após a chamada ao método pelo trapAgent, o Agent irá invocar o método através do Adapter (*Stub<Componente, false>*). No caso do método get, o valor de retorno é colocado na

```
template<class T>
class Proxy: public Message
{
public:
    Proxy() { invoke(CREATE); }
    template<class T1>
    Proxy(T1 &p1) { in(p1); invoke(CREATE); }
    // ...
    static void free(Proxy* p) {
        p->invoke(DESTROY);
        delete p;
    }
    // Thread
    void suspend() { invoke(THREAD_SUSPEND); }
    int join () { return invoke(THREAD_JOIN);}
    // ...
    // Synchronizer
    void lock() { invoke(SYNCHRONIZER_LOCK); }
    // ...
};
```

Figura 4.20: Implementação do Proxy.

```
Dispatcher* services[LAST_TYPE_ID + 1][MAX_METHODS] = {
    { &Agent<Component 1>::create,
      &Agent<Component 1>:::destroy,
      &Agent<Component 1>::method1,
      &Agent<Component 1>::method2,
      // ...
      &Agent<Component 1>::update },
    { &Agent<Component 2>::create,
      &Agent<Component 2>::destroy,
      &Agent<Component 2>::method1,
      &Agent<Component 2>::method2,
      // ...
      &Agent<Component 2>::update, }
     // ...
};

static Semaphore quiescent_state[LAST_TYPE_ID + 1];

void trapAgent(Message *msg) {
    Dispatcher *d;
    d = *services[msg->id().type()][msg->method()];
    quiescent_state[msg->id().type()].p();
    d(msg); // realiza a invocação do método
    quiescent_state[msg->id().type()].v();
}
```

Figura 4.21: Implementações do Dispatcher e da função trapAgent do Agent.

mensagem através do método `out` da classe `Message` e o objeto é recuperado através do método `get` do `Adapter` que recebe o ID como argumento. A chamada ao método pelo `Agent` é realizada da seguinte forma:

```
msg->out(_Stub::get(msg->id())->get());
```

Já no método `set`, o `Agent` precisa primeiro recuperar o argumento recebido através do método `in` da classe `Message`, para depois invocar o método. Sua chamada é realizada da seguinte forma:

```
int e;
msg->in(e);
_Stub::get(msg->id())->set(e);
```

No método `create`, o `Agent` cria um novo objeto do tipo `_Stub` e o adiciona em uma tabela *hash*. O objeto é recuperado antes de uma invocação de método usando o seu ID como chave única na tabela. O código do método `create` é mostrado abaixo:

```
static void create(Message* msg) {
     const _Stub *stub = new _Stub;
     _Stub::add(stub, msg->id());
}
```

Por fim, o método `update` é responsável pela reconfiguração de código dos componentes e é apresentado na seção 4.9.7.

4.9.5 Adapter

O elemento `Adapter` do ELUS é apresentado na Figura 4.22. O `Adapter` realiza a adaptação do componente que foi recebido como parâmetro pelo metaprograma para permitir sua execução em diferentes cenários [FSP00]. O `Adapter` funciona como um *wrapper* para as invocações dos métodos dos componentes, chamando

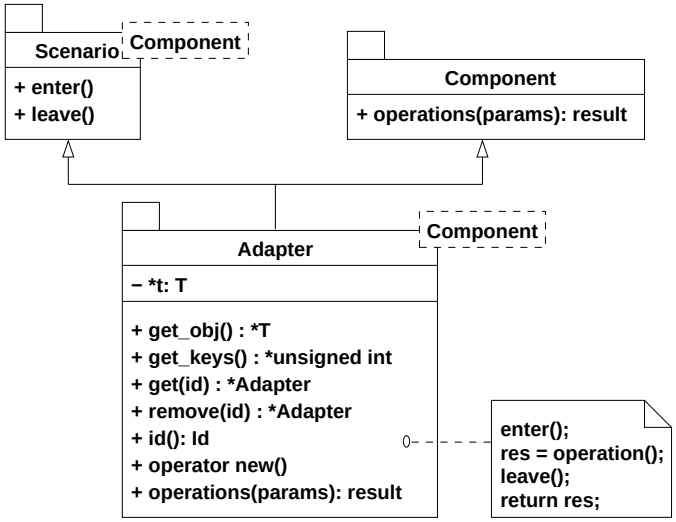

Figura 4.22: A classe Adapter do ELUS.

os métodos `enter` e `leave` do `Scenario`, antes e depois da invocação do método real do componente.

O `Adapter` também disponibiliza métodos (`get` e `remove`) para armazenar e recuperar os objetos previamente criados pelo `Agent` em uma tabela *hash* implementada no `Scenario`. O método `get_obj` retorna o objeto do tipo do componente recebido como parâmetro pelo metaprograma (tipo T) e criado por um dos construtores do `Adapter`. O método `get_keys` retorna todas as chaves (identificadores) armazenados pela tabela *hash*.

4.9.6 Scenario

O elemento `Scenario`, inicialmente responsável por aplicar os aspectos selecionados para cada componente recebido pelo metaprograma, foi estendido para também armazenar os objetos criados pelo `Agent` em tempo de execução. Para isso, foi criada uma classe, denominada `ScenarioHash`, que abstrai as operações de armazenamento, busca e recuperação de objetos através de uma tabela *hash*. A tabela é composta por um vetor de ponteiros sem tipo (*void*) para armazenar os objetos, um vetor de chaves para armazenar as chaves de cada elemento da tabela e um atributo inteiro que contém o

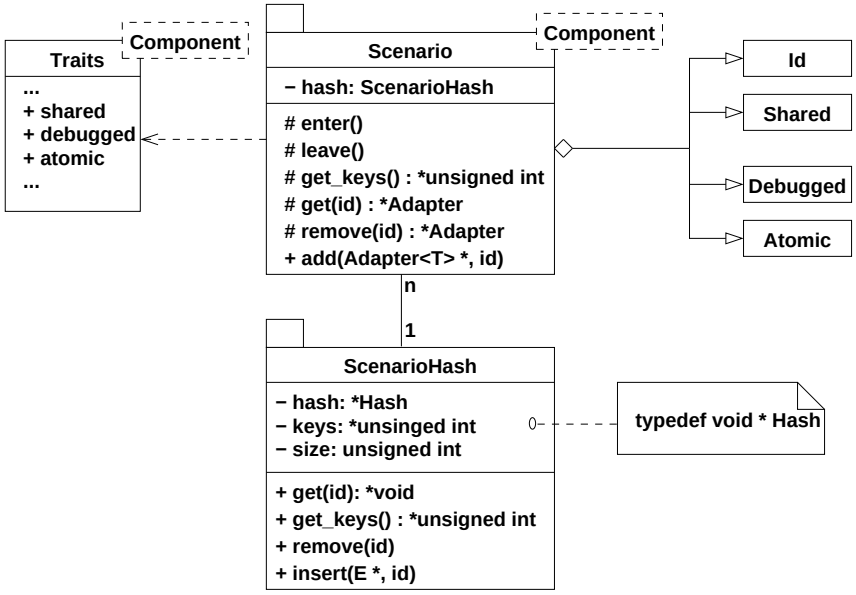

Figura 4.23: A classe Scenario do ELUS.

número atual de elementos na tabela em um determinado momento.

O método `get` retorna o objeto do tipo `Adater<T>`. O método `get_keys` retorna todas as chaves (identificadores) armazenados na tabela *hash*. O método `remove` libera uma posição na tabela removendo um objeto e recebendo o ID desse objeto como argumento para encontrá-lo na *hash*. O método `insert` é chamado através do método `add` do `Scenario` e insere um elemento na tabela, sendo que o primeiro argumento recebido pode ser de qualquer tipo devido a metaprogramação. A declaração do método é exemplificada a seguir:

```
template <typename E>
static void insert(E *e, unsigned int id) {
            //.....
}
```

O EPOS possui estruturas de dados como tabelas *Hash* e Listas metaprogramadas, as quais poderiam ser usadas para o armazenamento e recuperação de dados. A razão pelo não uso destas estruturas está justamente por elas serem metaprogramadas. A cada tipo diferente que é armazenado (seja ele inteiro, um objeto de uma

classe, etc), o compilador gera código específico para aquele tipo. Como o `Scenario` armazena diferentes tipos, pois é o mesmo metaprograma para diferentes componentes, o código gerado pelas estruturas de dados tornam-se grandes e difíceis de serem utilizados em sistemas com pouca memória.

4.9.7 Método de Reconfiguração do Agent

A reconfiguração de software é executada pelo método `update` do `Agent`. A Figura 4.24 apresenta o diagrama de sequência do método quando uma mensagem de reconfiguração referente à atualização de um componente é requisitada. O `Reconfigurator` fica aguardando por uma requisição de reconfiguração do ambiente externo. Esta requisição pode ser recebida por diversos meios como rádio, UART e ethernet, dependendo da implementação, arquitetura e plataforma utilizadas. Ao receber o pedido de reconfiguração, os dados (código), o byte de controle e também as informações sobre qual componente será atualizado e qual os tamanhos do novo código e dos métodos através do cliente, o `Reconfigurator` faz uma chamada a função `trapAgent` que é o ponto de entrada para os serviços disponibilizados pelo `Agent`.

O `Agent` então alcança o estado quiescente do componente antes de atualizá-lo através do método `p` do semáforo, lê os dados recebidos chamando o método `in` da classe `Message`, pega o adaptador de cenário do componente chamando o método `get` e recupera o objeto referente àquele componente através do método `get_obj` do `Adapter`. O acesso ao endereço da `vtable` na memória é realizado no próximo passo, através do seguinte código:

```
// acessa a vtable a partir do objeto "t"
unsigned int *addr = *reinterpret_cast<unsigned int>(&t);
```

O byte de controle informa qual o tipo de reconfiguração. Se a reconfiguração for do tipo de atualização do componente com tamanho do novo código maior que o antigo (*control_byte & 0x04*), o `Agent` irá alocar espaço na memória de código para o novo código do componente e irá escrevê-lo nessa posição, sendo que ambas operações

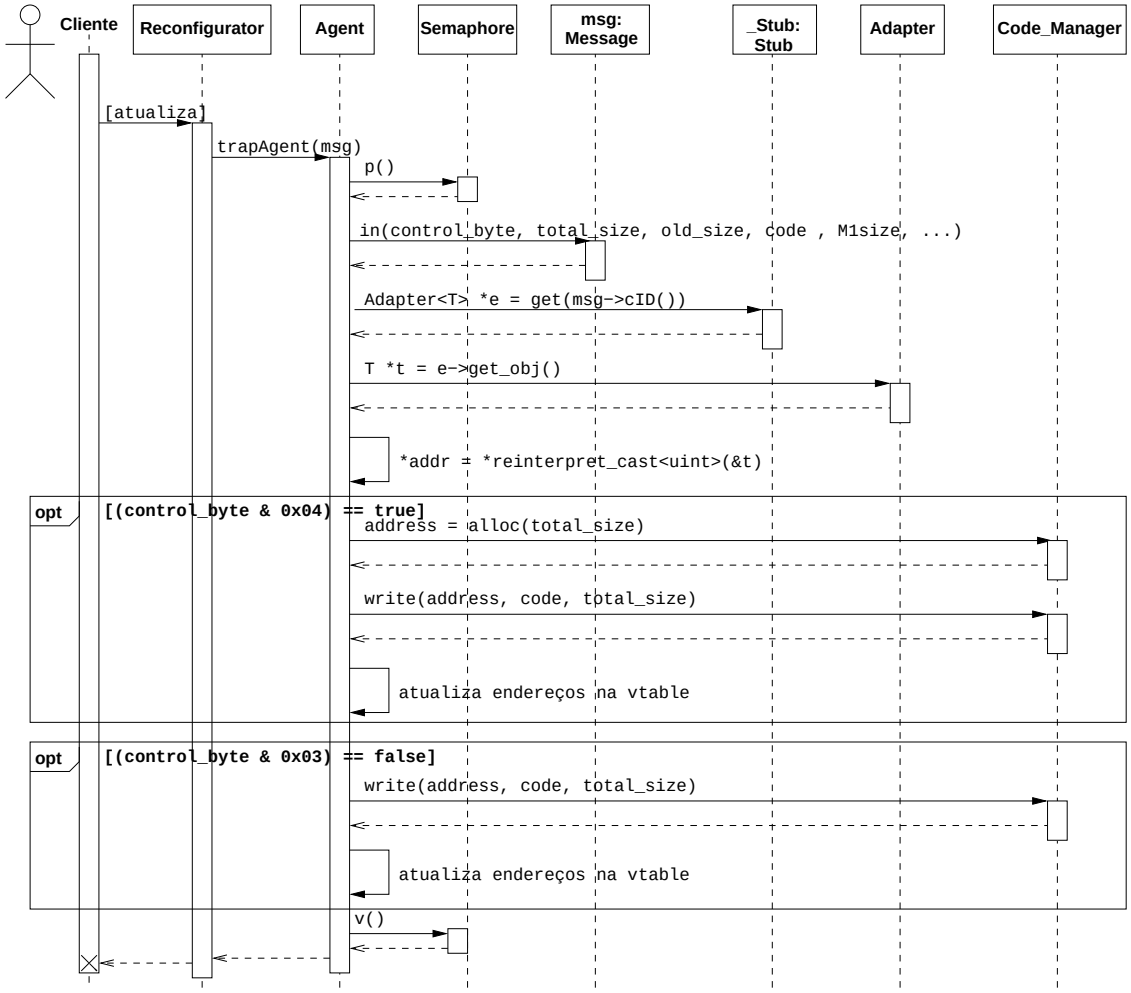

Figura 4.24: Diagrama de sequência do método update do Agent para mensagens do tipo de atualização de componente.

são realizadas através do gerenciador de código (`alloc` e `write`). O próximo passo é atualizar os novos endereços dos métodos na `vtable` para que possam ser acessados normalmente pela aplicação. Por outro lado, se o tipo de reconfiguração for uma atualização do componente com tamanho do novo código menor ou igual ao antigo, o novo código é escrito na mesma posição e os endereços dos métodos na `vtable` são atualizados com base no tamanho de cada método recebidos na mensagem. A última operação do método `update` é liberar o semáforo (`v`) e retornar para o `Reconfigurator`.

A título de exemplo, o código abaixo exemplifica como é realizada a

atualização dos endereços da `vtable` pelo gerenciador de código (método `write`) na arquitetura IA32 que possui o tamanho da palavra de 4 bytes:

```
// atualiza o endereço do primeiro método da vtable
memcpy((void *)*addr, (const void *)&new_addr, 4);
// atualiza o endereço do segundo método da vtable
memcpy((void *)*(addr + 4), (const void *)&new_addr, 4);
```

O acesso aos atributos de um componente reconfigurável deve ser feito através dos métodos `set` e `get` relacionados ao atributo, como manda os princípios da programação orientada a objetos. Para a adição de atributos, o `Agent` recebe uma mensagem de adição de atributos, que contém, além dos IDs do componente e do método `update` referente ao componente, o byte de controle e o tamanho do novo objeto a ser criado. O tamanho é a soma dos atributos antigos com o(s) novo(s). Em uma reconfiguração, quando um componente adiciona um ou mais atributos, os métodos `set` e `get` do(s) atributo(s) também devem ser adicionados no `Agent`. Após a adição dos métodos, o `Agent` cria espaço para o novo objeto (tamanho dos atributos privados somados com o novo(s) atributo(s) que é recebido na mensagem), copia os atributos antigos para o novo objeto através dos métodos `set` e `get` e destrói o objeto antigo. Isso é realizado para todos os objetos existentes do mesmo componente.

Quando uma mensagem do tipo de remoção de método é recebida pelo `Agent` (código 0010 do byte de controle), que contém os IDs do componente, do método `update` e do método a ser removido, além do tamanho do método, o `Agent` libera a memória ocupada pelo método, e atualiza a `vtable` e o `Dispatcher` para apontarem para uma posição nula (*null*).

4.10 Atualização do Framework

A função `trapAgent`, que é o ponto de entrada para os serviços do `Agent`, não é passível de reconfiguração pois seu endereço deve ser conhecido à priori para que os componentes reconfiguráveis possam ter ciência de qual é o seu endereço

e assim invocar os métodos corretamente através do `Agent`. Se a função `trapAgent` tiver seu lugar modificado na memória de código, este novo endereço não será conhecido e assim o novo código do componente não será compilado corretamente. Porém, todos os outros métodos do `Agent` são passíveis de reconfiguração. São três tipos de mensagens que podem implicar em uma atualização do framework:

- **Atualização de componente no framework com tamanho de código menor ou igual ao antigo:** neste caso o `Agent` simplesmente irá copiar o código recebido para a posição do componente e atualizar os endereços no `Dispatcher`, que contém os ponteiros para todos os métodos reconfiguráveis do componente no framework.

- **Atualização de componente no framework com tamanho de código maior que o antigo:** neste caso o `Agent` irá alocar memória para o novo código, atualizar os endereços dos métodos no `Dispatcher` e liberar a memória antiga.

- **Adição de métodos:** o `Dispatcher` é um vetor declarado estaticamente, conforme mostra a Figura 4.21. Se o componente reconfigurável não estiver utilizando todas as posições existentes dentro do vetor (*MAX_METHODS* posições - valor este que é configurado em tempo de compilação), isto significa que o `Dispatcher` tem espaço para o novo método e não necessita ter sua posição trocada em memória. Sendo assim, o `Agent` aloca memória para o novo método e insere o endereço do método no `Dispatcher`, na posição que foi enviada pela mensagem (ID método). Este tipo de mensagem é uma mensagem de adição de método sem relocação do `Dispatcher` que é informada pelo byte de controle do ETP (código 0000).

  Se o componente estiver usando todas as posições dentro do `Dispatcher`, ele precisa ser relocado para uma área de memória maior. Após isso, os dados do vetor antigo são copiados para o novo, o endereço do novo método é inserido na posição (ID método recebido pela mensagem) e a memória antiga liberada. Por fim, o `Dispatcher` é atualizado para apontar para a nova posição e assim manter a consistência do sistema. Este tipo de mensagem é definida pelo ETP como uma

mensagem de adição de método com relocação (código 0001).

A adição de um método no componente também implica no aumento da sua `vtable`. Com isso, deve ser alocado espaço para conter mais um método dentro da tabela, deve-se também copiar os endereços e liberar o espaço de memória antigo. Ainda, o novo endereço da `vtable` deve ser copiado para a primeira posição de memória do objeto que aponta para a tabela de métodos virtuais e é usado para fazer as invocações de métodos.

Somente os componentes marcados como reconfiguráveis em tempo de compilação são passíveis de atualização. Não é permitido a adição de outros componentes em tempo de execução. Porém, se isso for necessário por algum motivo, pode ser utilizado o tipo de mensagem do ETP referente a atualização de endereços e assim enviar o novo código do sistema que será reiniciado após sua atualização.

4.11 Atualização da Aplicação

Existem duas maneiras de atualizar a aplicação conforme o tamanho do seu novo código. Quando o tamanho do novo código da aplicação é menor ou igual ao código antigo, a atualização é realizada através de uma mensagem de atualização de endereço, conforme o tipo de mensagem ETP, mostrado na Figura 4.9. Neste caso, são enviados, além do novo código, o endereço inicial da aplicação (função `main`) e o tamanho do novo código.

Quando o novo código da aplicação é maior que o antigo, a aplicação necessita que seja alocado um novo espaço em memória. A atualização neste caso é realizada através de uma mensagem de atualização da aplicação. Além do novo código e seu tamanho, a mensagem também contém os endereços que faziam referência para a função `main`. Esses endereços são atualizados com a nova posição da função, mantendo assim a consistência do sistema.

Após a atualização da aplicação, o sistema deve ser reiniciado para que as novas atividades da aplicação possam ser realizadas. As referências externas da apli-

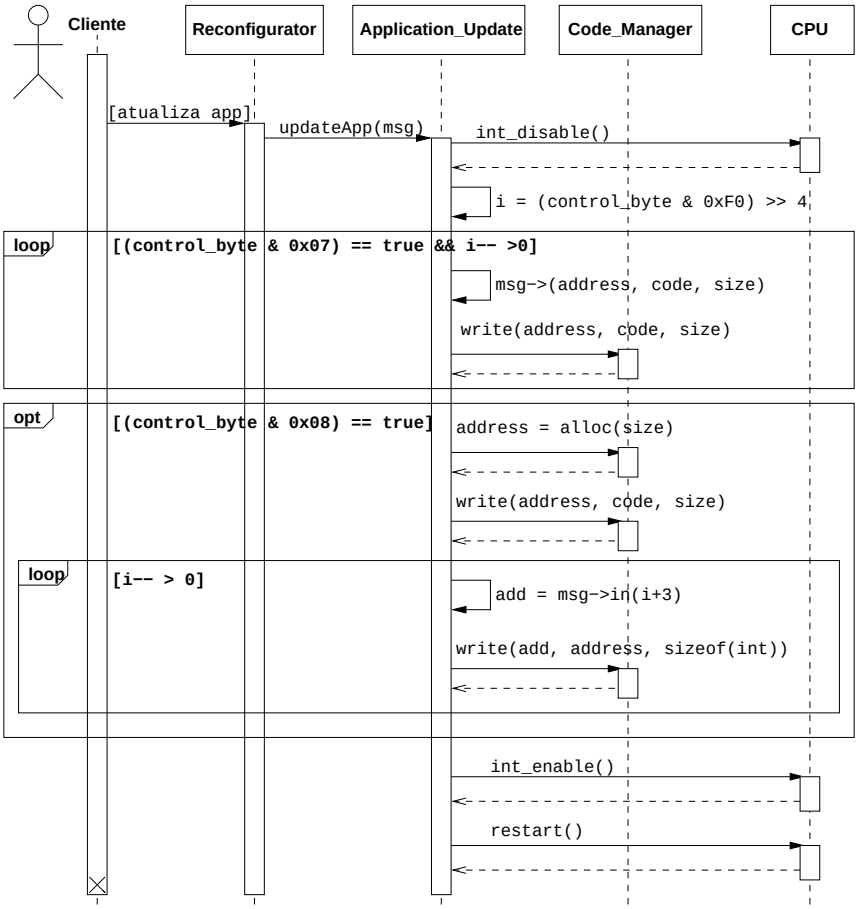

Figura 4.25: Diagrama de sequência para uma atualização da aplicação.

cação são resolvidas efetuando a sua ligação com o sistema antigo durante a compilação da aplicação.

A Figura 4.25 apresenta o diagrama de sequência quando um pedido de atualização da aplicação é recebido pelo `Reconfigurator`. O `Reconfigurator` recebe o pedido, monta a mensagem de acordo com o ETP e invoca o método `updateApp` do componente `Application_Update`. Este componente foi criado única e exclusivamente para suportar a atualização da aplicação e inclusive pode ser passível de reconfiguração. Para isso é necessário que o seu suporte à reconfiguração seja habilitado no sistema e seus métodos adicionados ao framework.

O componente `Application_Update` atinge o estado quiescente do sistema desabilitando as interrupções (`int_disable()`). Após, verifica se a men-

sagem recebida é uma simples atualização de endereço ou uma atualização da aplicação e executa as ações conforme o tipo de atualização. No final, o sistema deve ser reiniciado para que a nova aplicação comece sua execução.

O ELUS, através do framework metaprogramado e a infra-estrutura que permite reconfiguração dinâmica, é totalmente transparente para a aplicação. Por outro lado, é adicionado um sobrecusto para o sistema, por causa do nível de indireção criado entre as invocações de métodos da aplicação para os componentes do sistema e o código e dados necessários para que o ELUS consiga gerenciar uma reconfiguração. Estes dois parâmetros (sobrecusto na invocação de métodos e consumo de memória extra), juntamente com o tempo para reconfiguração de um componente são mensurados no próximo capítulo.

4.12 Considerações Parciais

Este capítulo apresentou o projeto e a implementação do EPOS LIVE UPDATE SYSTEM (ELUS). As principais contribuições do ELUS são:

- **Configurabilidade:** através do uso do framework metaprogramado do EPOS é possível marcar um componente como reconfigurável em tempo de compilação. Somente os componentes marcados como reconfiguráveis são passíveis de atualização e para todos os outros nenhum sobrecusto em termos de memória e processamento é adicionado no sistema. Essa configurabilidade permite que apenas alguns componentes específicos do sistema sejam selecionados como reconfiguráveis, não agregando nenhum sobrecusto ao sistema além do necessário.

- **Transparência e Consistência:** a invocação de um método de um componente reconfigurável, do ponto de vista da aplicação, é realizada de forma transparente, ou seja, a maneira como a aplicação faz a chamada a um método permanece a mesma. Além disso, o processo de reconfiguração de um componente também é realizado de forma transparente. Quando a aplicação invoca um método de um componente que está sendo reconfigurado naquele momento, a chamada é suspensa

até a reconfiguração ser concluída, garantindo a consistência do sistema após uma reconfiguração.

- **Protocolo de Transporte:** o ELUS utiliza o ELUS TRANSPORT PROTOCOL (ETP) para receber requisições de reconfiguração e invocações de métodos de um determinado componente reconfigurável. Este protocolo permite que o ELUS seja facilmente integrado com outros protocolos, como por exemplo, um protocolo de disseminação de dados.

- **Reconfiguração Total:** o ELUS permite que todos os componentes e mediadores do sistema sejam reconfigurados, bem como as aplicações. Existe apenas uma única função no sistema que não é passível de atualização, a função `trapAgent`. Esta função é o ponto de entrada entre as aplicações e o framework. Seu endereço deve ser conhecido à priori pela aplicação para que seja compilada de maneira correta. Se o endereço desta função fosse modificado, uma aplicação, após uma atualização, apontaria para um endereço incorreto, pois foi compilada e ligada com o endereço da função antigo. Esta situação causaria uma incosistência no sistema. Por outro lado, o código da função `trapAgent` não deverá sofrer mudanças, pois o seu objetivo é apenas fazer as chamadas aos métodos invocados conforme o ETP.

Capítulo 5

Avaliação da Proposta

A avaliação do ELUS foi realizada com o objetivo de corroborar as decisões de projeto apresentadas no capítulo 4. Para tal, definiu-se um conjunto de testes para mensurar o consumo extra de memória necessário para a infra-estrutura, a perda de desempenho na invocação dos métodos devido a introdução do nível de indireção e da estrutura de troca de mensagens e por fim o tempo gasto em uma reconfiguração de um componente.

5.1 Configuração dos Experimentos

A avaliação do ELUS foi realizada na arquitetura IA32, sendo que o sistema foi gerado para a mesma usando o compilador *GNU g++* na versão 4.0.2. As avaliações relativas ao consumo de memória usaram a ferramenta *GNU objdump* na versão 2.16.1 para analisar o consumo da memória de código (seção *text*), dados não inicializados (seção *bss*) e dados (seção *data*) nas imagens do sistema geradas pelo compilador.

O desempenho de invocação de métodos e tempo de reconfiguração foram mensurados em termos de ciclos de processador necessários para a execução das instruções. Os ciclos de cada instrução foram somados conforme o manual da *AMD* [Dev02].

5.2 Consumo Extra de Memória

Para a primeira avaliação, que mediu o consumo extra de memória do ELUS, o suporte à reconfiguração dinâmica de software foi habilitado somente para o componente `Chronometer` que possui 7 métodos passíveis de reconfiguração (`frequency`, `reset`, `start`, `stop`, `lap`, `read` e `ticks`) conforme exemplifica a Figura 3.11. O sistema foi gerado para a arquitetura IA32 com o compilador *GNU g++* 4.0.2 e o consumo de memória medido com o uso da ferramenta *GNU objdump* na versão 2.16.1. A Tabela 5.1 mostra o consumo de memória de cada elemento do ELUS para o cenário descrito.

Tabela 5.1: Consumo de memória do ELUS.

Elementos do ELUS	**Tamanho da Seção (bytes)**		
	.text	**.data**	**.bss**
Reconfigurator	224	0	72
Code_Manager IA32	816	0	0
Application_Update	208	0	0
Framework	3152	72	116
Total	4400	72	188

O ELUS como um todo utilizou 4400 bytes de memória de código, 72 bytes de dados e 188 bytes de dados não inicializados. Destes valores, o `Reconfigurator` consumiu 224 bytes de código e 72 bytes de dados não inicializados, sendo que 40 destes bytes são referentes ao tamanho do buffer utilizado na recepção dos dados (este buffer pode ter seu tamanho ajustado conforme a disponibilidade de memória e/ou tecnologia de transmissão utilizada). Neste caso, não foi computado o consumo de memória necessário pela tecnologia de transmissão, haja visto que é altamente dependente de plataforma e implementação. Para exemplificar o quanto esses valores podem variar, o protocolo de controle de acesso ao meio do EPOS (C-MAC) consome em torno de 4kb de código e 100 bytes de dados [Wan06] e o uIPv6, a menor implementação da

pilha de protocolos IPv6 até o momento, tem cerca de 11.5kb de código e 1.8kb de dados [DAW+08].

O gerenciador de código para a arquitetura IA32 foi responsável pelo consumo de 816 bytes de memória de código. O gerenciador não adicionou dados no sistema pois, através do mediador de hardware, utilizou a própria estrutura da MMU para a arquitetura IA32 que controla a alocação e liberação de espaços na memória. Por fim, toda a estrutura do framework, composta pelos elementos `Handle`, `Stub`, `Proxy`, `Agent`, `Scenario` e `ScenarioHash`, ocupou 3152 bytes de memória de código, 72 bytes de dados (44 bytes do `Dispatcher`) e 116 bytes de dados não inicializados. Destes, 84 bytes são utilizados pelo `ScenarioHash` para armazenamento e controle da tabela *hash* e 32 bytes pelo semáforo que controla o estado quiescente do componente. O tamanho da tabela *hash* foi configurado para armazenar até 10 objetos, sendo que esse valor pode ser ajustado em tempo de compilação para melhor se adequar aos requisitos da aplicação. Para o armazenamento de cada objeto são necessários 8 bytes.

Tabela 5.2: Consumo de memória na adição de métodos de um componente no framework do ELUS.

Método do framework	**Tamanho da Seção (bytes)**		
	.text	**.data**	**.bss**
Create	160	0	0
Destory	176	0	0
Método sem parâmetro e sem retorno	96	0	0
Método com parâmetro	112	0	0
Método com valor de retorno	128	0	0
Método com parâmetro e com retorno	144	0	0
Update	1232	0	0
Dispatcher	0	44	0
Semaphore	0	0	32
Total Mínimo	1568	44	32

A segunda avaliação mensurou qual o sobrecusto adicionado ao sistema quando um componente é marcado como reconfigurável, desconsiderando obviamente código e dados adicionados pela própria implementação do componente. O código também foi gerado para a arquitetura IA32 utilizando o compilador *g++* 4.0.2 e os valores mensurados através da ferramenta *GNU objdump* 2.16.1.

A Tabela 5.2 apresenta os valores obtidos. Os métodos `Create` (construtor) e `Destroy` (destrutor) adicionam 160 e 176 bytes de código respectivamente. Método que não possui passagem de parâmetros mas possui valor de retorno adiciona 128 bytes de código. Método que possui passagem de parâmetro mas não possui valor de retorno consome 112 bytes de código. Método com passagem de parâmetro e com valor de retorno adiciona 144 bytes. Método sem passagem de parâmetro e sem valor de retorno consome 96 bytes. São necessário apenas 16 bytes para cada parâmetro passado pela estrutura e 32 bytes para o valor de retorno. O método `update`, responsável pela reconfiguração de software, tem um sobrecusto de 1232 bytes de código. Por fim, o `Dispatcher` e o `Semaphore` utilizam 44 bytes de dados e 32 bytes de dados não inicializados, respectivamente. São necessários 1568 bytes para memória de código, 44 bytes de dados e 32 bytes de dados não inicializados para ter o mínimo do suporte à reconfiguração dinâmica (métodos `create`, `destroy` e `update`) habilitado para um componente do sistema. É importante relembrar que o tamanho do `Dispatcher` também é configurado em tempo de compilação pelo desenvolvedor, sendo que neste teste foi usado um vetor com capacidade para armazenar 11 endereços (4 bytes por endereço).

A Equação 5.2 mostra qual é o sobrecusto em termos de memória de código quando um componente é marcado como reconfigurável. O tamanho do componente C é o somatório dos tamanhos de todos os métodos conforme a Tabela 5.2 mais a implementação do método real pelo próprio componente. 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` (44) e `Semaphore` (22) que são utilizados pelo ELUS.

$$Tamanho_c = \sum_{i=1}^{n}(Método_i) + Create + Destroy + Update \quad (5.1)$$

5.3 Desempenho na Invocação de Métodos

O ELUS adiciona um sobrecusto na invocação dos métodos somente naqueles componentes que têm o suporte à reconfiguração dinâmica de software habilitado. Essa perda de desempenho é causada pelo nível de indireção criado entre a aplicação e a invocação do método real através do framework metaprogramado. O framework é responsável pelo armazenamento dos objetos reconfiguráveis criados pela aplicação, pelo controle de acesso aos métodos para evitar que uma reconfiguração aconteça no meio da execução de um método e assim garantir o estado quiescente do componente e pela passagem de parâmetros e valor de retorno através da estrutura de mensagem. Todas essas atividades realizadas antes ou depois da invocação do método são as causas da perda de desempenho e o preço pago pelo suporte à reconfiguração em um ou mais componentes do sistema.

A Tabela 5.3 apresenta a comparação de desempenho na invocação de métodos. São comparadas as invocações de métodos normais, através da `vtable` e através do ELUS. Nesta comparação não é considerado o tempo de execução do método, apenas o tempo gasto antes e depois da sua execução, com passagem de parâmetros e/ou valor de retorno. O teste foi realizado na arquitetura *x86*, utilizando o compilador *g++* 4.0.2. Os valores apresentados são a contagem do tempo, em ciclos de processador, necessários para a execução de cada instrução conforme o manual da *AMD* [Dev02].

Quatro tipos de invocações de métodos foram mensuradas: método que não recebe parâmetro e não tem valor de retorno, método com valor de retorno mas que não recebe parâmetro, método que recebe parâmetro mas não tem valor de retorno e método que recebe parâmetro e retorna um valor. A invocação de um método no ELUS foi de 6 a 12 vezes pior que a invocação através da `vtable` e de 8 a 28 vezes mais lenta que a invocação de um método normal, considerando apenas o número de ciclos.

Tabela 5.3: Comparação do desempenho da invocação de métodos normal, utilizando a vtable e através do ELUS.

Tipo do Método	Tempo (em ciclos)
Normal sem parâmetro e sem retorno	4
vtable sem parâmetro e sem retorno	9
ELUS sem parâmetro e sem retorno	112
Normal sem parâmetro e com retorno	5
vtable sem parâmetro e com retorno	10
ELUS sem parâmetro e com retorno	132
Normal com parâmetro e sem retorno	8
vtable com parâmetro e sem retorno	12
ELUS com parâmetro e sem retorno	123
Normal com parâmetro e com retorno	17
vtable com parâmetro e com retorno	22
ELUS com parâmetro e com retorno	143

O `trapAgent` que é responsável pela invocação dos métodos do `Agent` através do `Dispatcher` e pela garantia do estado quiescente do componente, consome 62 ciclos do processador. Esse valor está diluído em todos os quatro tipos de invocações de métodos avaliados no ELUS e representa a causa da perda de desempenho de até 55% dos ciclos gastos em uma invocação. Além desse fator, a busca na tabela *hash* e a invocação do método através do `Adapter` corroboram para a perda de desempenho. Para cada parâmetro passado através da estrutura de mensagem do ELUS é consumido 11 ciclos e para o valor de retorno são consumidos 20 ciclos.

Embora esses valores possam parecer grandes, o tempo de execução é consideravelmente baixo. Por exemplo, suponha que a velocidade de um processador seja de 10 Mhz. Cada ciclo neste caso seria executado em 0.1 microsegundos e no pior caso do ELUS com relação a invocação de método normal (quando nenhum parâmetro é passado e nenhum valor de retorno devolvido), a invocação de método levaria apenas 11.2

```
8290 <main>:
// ...
82c7: ,mov %eax, %esi
82c9: call a160: <Test_Component::abc>
//...

a160 <Test_Component::abc>
//...
```

(a) Invocação de método normal.

```
8290 <main>:
// ...
// coloca em %eax o endereço da vtable
82b4: $0xd470, (%eax)
// chama o segundo método da vtable
82d3: call *0x4(%eax)
//...

// vtable
d470:  60 a1 00 00
d474:  70 a1 00 00 // a170 endereço do método
// ...

a170 <Test_Component::abc>
//...
```

(b) Invocação de método através da vtable.

```
82c9 <main>:
// ...
8339: call a200 <trapAgent>
// ...
a200 <trapAgent>:
// ...
// chama o método através do dispatcher
a237: call *%esi
// ...

aae0 <Agent<Test_Component>::abc>
// ...
// chama o método do Adapter
ab0a: jmp *%ecx
// ...

a500 <Adapter<Test_Component>::abc>
// ..
// invoca o método do componente
a512: jmp *%ecx
// ...
```

(c) Invocação de método através do ELUS.

Figura 5.1: Código exemplo da comparação dos diferentes tipos de invocação de método. (a) invocação de um método normal. (b) invocação de método através da vtable e (c) invocação de método através do ELUS.

microsegundos.

A Figura 5.1 exemplifica o código em linguagem de montagem gerado nos três tipos de invocação de método comparados. Ao invocar um método pelo ELUS, a aplicação faz uma chamada ao `trapAgent` que primeiro alcança o estado quiescente do componente através do semáforo e depois chama o método dentro do `Agent` através do `Dispatcher`. O `Agent` recupera o objeto armazenado dentro do `Scenario`, passa os parâmetros recebidos e/ou o valor de retorno após a execução do método se necessário, e pula (*jmp*) para o método do `Adapter` que faz a invocação do método do componente utilizando a `vtable` do objeto.

5.4 Tempo de Reconfiguração

O tempo necessário para executar uma reconfiguração no ELUS foi medido em número de ciclos gasto pelo processador para executar cada instrução. O sistema foi gerado para a arquitetura IA32 com o compilador *g++* 4.0.2 e a ferramenta *objdump* 2.16.1 usada para verificar o código em linguagem de montagem do sistema gerado. O número de ciclos por instrução foi contabilizado conforme o manual da *AMD* [Dev02].

Nesta avaliação, não foram levados em consideração o tempo necessário para a recepção dos dados pelo `Reconfigurator`, haja visto que esse fator é altamente dependente de implementação e tecnologia de transmissão utilizada e também o tempo necessário para a escrita do código atualizado na memória, que da mesma forma, é dependente do tipo de memória, barramento e arquitetura. Desta maneira, foram mensurados apenas os ciclos gastos pelo `Reconfigurator` para fazer a chamada e passar os dados para o `trapAgent` e o número de ciclos que o `Agent` consome para efetuar uma reconfiguração, contando as chamadas das funções de escrita de memória mas desconsiderando o tempo de uma escrita.

A Tabela 5.4 mostra o tempo para realizar os dois tipos de atualização em um componente (códigos do ETP 3 e 4). O primeiro, quando o novo código recebido é menor ou igual ao código antigo, consome 102 ciclos. O segundo, quando o novo código é maior que o antigo e é necessário alocar um novo bloco de memória e liberar o bloco antigo, consome 353 ciclos. O `Reconfigurator` leva 34 ciclos para passar os dados recebidos em uma mensagem para o `trapAgent`. O `trapAgent`, que é responsável por garantir o estado quiescente do componente e invocar o método `update` do `Agent` consome 62 ciclos do processador. Somando-se esses valores com os valores de reconfiguração, o primeiro tipo de reconfiguração leva 198 ciclos e o segundo tipo consome 449 ciclos.

É importante salientar que o `trapAgent` faz uma chamada ao método `p()` do semáforo e se por acaso alguma thread esteja acessando algum método do componente no momento da reconfiguração, acontecerá uma condição de corrida. Neste caso, a reconfiguração terá que aguardar até a liberação do semáforo para prosseguir sua

Tabela 5.4: Tempo (em ciclos do processador) gasto para realizar uma atualização no código de um componente no ELUS.

	Tempo (em ciclos)	
	Atualização mesma posição	**Atualização nova posição**
	102	353
Reconfigurator	34	34
trapAgent	62	62
Total	198	449

execução.

5.5 Análise dos Resultados

Esta seção apresenta a análise dos resultados, comparando o desempenho do ELUS com os trabalhos relacionados. Os três tipos de sistemas que permitem reconfiguração dinâmica de código (infra-estruturas, máquinas virtuais e sistemas operacionais conforme o capítulo 2) são analisados em termos de consumo de memória, desempenho na invocação de métodos ou chamada de funções e o tempo necessário para que uma reconfiguração dinâmica seja realizada. Por fim, é realizada uma comparação em termos de tempo para alocação e liberação de memória em alguns sistema operacionais embarcados.

5.5.1 Consumo de Memória

Para cada componente com o suporte à reconfiguração dinâmica habilitado, o ELUS consome pouco mais de 1.5kb de código e 76 bytes de dados, além de algumas dezenas de bytes para o armazenamento dos objetos na tabela *hash*. Em comparação com as infra-estruturas que suportam reconfiguração dinâmica, o consumo de memória do ELUS é inferior. FlexCUP usa um buffer de 3kb para fazer a ligação do módulo recebido com o sistema. O buffer é utilizado para armazenar os meta-dados e as informações

de relocação recebidas no momento de uma reconfiguração. Além disso, FlexCUP deve armazenar as informações e meta-dados de todos os componentes que estão executando, aumentando consideravelmente o tamanho de código [MGL+06]. O ligador incremental remoto [KP05a] e a estrutura proposta em [FKKP07], embora não apresentam valores, também ocupam uma quantidade de memória considerável devido ao *bootloader* e as informações necessárias para fazer uma atualização e que devem ser armazenadas em tempo de execução. Em geral, o consumo de memória das infra-estruturas é maior do que os sistemas operacionais e menor ou equivalente, em alguns casos, às máquinas virtuais.

Considerando as máquias virtuais, o consumo de memória é maior em alguns casos, devido ao código da própria máquina virtual ou devido a necessidade de um sistema operacional para a sua execução. Maté consome 16kb de código e 800 bytes de dados, além do código e dados utilizados pelo sistema operacional TINYOS [LC02]. DVM necessita de 13kb de código e 727 bytes de dados e mais 33kb de código e 156 bytes de dados do sistema operacional SOS [BHR+06]. O tamanho de código da VM* varia de 10kb a 26kb e os dados de 500 bytes a 2kb dependendo da arquitetura alvo [KP05b]. A máquina virtual com o melhor desempenho considerando o consumo de memória é a Tapper. O tamanho do código também varia conforme a arquitetura, entre 3kb a 11kb e os dados entre 230 bytes a 1.5kb [XLC06]. Mesmo com o melhor desempenho da Tapper, o ELUS possui um consumo inferior do que as máquinas virtuais analisadas.

Em comparação com os sistemas operacionais, o ELUS também possui um bom desempenho. Contiki consome em torno de 6kb de código e 230 bytes de dados para ter o suporte à reconfiguração dinâmica no sistema [DGV04]. O sistema operacional SOS como um todo necessita de 20kb de código e em torno de 3kb de dados [HKS+05], somente o código responsável pelo gerenciamento dos módulos ocupa mais de 4kb e a tabela de módulos do kernel ocupa 224 bytes de dados. Think é o sistema que apresenta os maiores valores em termos de consumo de memória, 109kb de código e 13kb de dados, sendo impraticável sua utilização em sistemas profundamente embarcados [FSLM02].

A maior razão para o bom desempenho do consumo de memória do ELUS é explicada pelo uso da metaprogramação estática, na qual resolve todas dependências entre o framework e os componentes, os cenários de execuções e as aplicações em

tempo de compilação. Desta forma, o sistema gerado contém apenas o código e dados realmente necessários para sua execução. Outro diferencial do ELUS é a opção de escolha dos componentes reconfiguráveis em tempo de compilação, não agregando nenhum sobrecusto àqueles componentes não selecionados.

5.5.2 Desempenho na Invocação de Métodos

As infra-estruturas em geral não apresentam perda de desempenho nas invocações dos métodos ou chamadas a funções. A perda de desempenho é ocasionada devido a criação de um nível de indireção entre as invocações da aplicação para o método/função e esse nível de indireção não existe nas infra-estruturas analisadas (exceto no modo indireto do Molecule). O sistema é compilado e as instruções executadas normalmente sem nenhuma perda de desempenho. A maior perda de desempenho dessas estruturas está justamente no consumo de memória, como apresentado na seção anterior, e no tempo gasto em uma reconfiguração, como será apresentado na próxima seção. A perda de desempenho ocasionada pelo método indireto do Molecule é cerca de 50% quando comparado a uma chamada de função no método direto [YMCH08].

As máquinas virtuais em geral têm uma grande latência devido ao interpretador da linguagem de script. Uma instrução antes de ser executada deve ser lida da memória, decodificada e, com base em um código, executada. Uma única instrução da linguagem de script da máquina virtual pode ser traduzida em diversas instruções do processador. Para exemplificar a perda de desempenho, uma simples instrução como *call* ou *and*, que normalmente são realizadas entre 1 a 4 ciclos dependendo do processador, na máquina virtual Maté essas instruções levam no mínimo 14 ciclos [LC02]. O desempenho da DVM é ainda pior. Uma única instrução de incremento leva 550 ciclos [BHR+06]. O tempo necessário pela VM* para a invocação de um método virtual em Java é de 564 ciclos [KP05b], sendo que para uma invocação no ELUS são necessários pouco mais de 100 ciclos. Tapper leva 181 ciclos para escrever e 20 ciclos para ler em uma porta de E/S, sendo que essas operações geralmente são realizadas em poucos ciclos de processamento. A grande perda de desempenho das máquinas virtuais fica evidente com esses números

apresentados.

Os sistemas operacionais obviamente apresentam um melhor desempenho do que as máquinas virtuais. No SOS, uma chamada de função de um módulo reconfigurável através da tabela que o SO cria o nível de indireção demora 21 ciclos. Uma chamada a uma função do kernel leva 12 ciclos. A comunicação entre os módulos (troca de dados) é realizada através do envio e recepção de mensagem em um buffer, sendo que são necessários 833 ciclos para que o dado seja entregue [HKS+05]. O desempenho do Contiki é ainda pior, pois para fazer uma chamada de função o *stub* deve primeiro localizar qual a interface de serviço que está sendo chamada, sendo que a interface é localizada através de uma comparação de uma sequência de caracteres. Após a localização da interface de serviço, as funções são chamadas através de ponteiros [DGV04]. Em uma comparação realizada, o Contiki apresentou um desempenho em torno de 4 vezes pior do que o SOS em uma chamada de função [YMCH08]. Da mesma forma, RETOS [CCJ+07] e Nano-Kernel [Bag08] têm uma perda de desempenho considerável pela criação do nível de indireção através de uma tabela.

A perda de desempenho do ELUS é similar aos sistemas operacionais analisados. Ao invocar um método, o ELUS atinge o estado quiescente do componente através de chamadas aos métodos `p` e `v` de um semáforo. Nos sistemas baseados em módulos, como o SOS e o Contiki, o estado quiescente é alcançado somente no momento da reconfiguração através de uma mensagem que informa ao módulo que ele mesmo deve se remover do sistema. Essa solução torna a chamada a função mais rápida, porém a reconfiguração é mais lenta devido a troca de mensagens. O SOS ainda tem uma grande perda de desempenho na troca de dados entre os módulos (833 ciclos), enquanto que no ELUS, através da estrutura de troca de mensagem (classe `Message`), são necessários apenas 11 ciclos para cada parâmetro em uma invocação de método. A maior perda de desempenho do Contiki é explicada pela comparação da sequência de caracteres realizada para a localização da interface de serviço. No ELUS, a localização de um objeto é feita através de uma tabela *hash*, que é a estrutura de dados mais rápida para a busca de elementos, e a invocação do método realizada através do objeto.

5.5.3 Tempo de Reconfiguração

As infra-estruturas têm o pior desempenho em termos de tempo de reconfiguração. A necessidade da ligação do código recebido com o sistema em execução através de relocação e da utilização de meta-dados compromete o tempo de reconfiguração. FlexCUP, por exemplo, precisa de 5 segundos gasto apenas em processamento para atualizar 637 bytes em um microcontrolador ATMega128 de 8Mhz [MGL+06]. Em geral, essas estruturas realizam uma grande quantidade de processamento para unir o código recebido com o sistema antigo e também são necessárias várias escritas e leituras na memória.

As máquinas virtuais apresentam o melhor desempenho quando se tratando de tempo de reconfiguração. Por serem baseadas em scripts, o tamanho de uma aplicação é bastante reduzido, isto é, a aplicação é formada por apenas algumas poucas instruções. Em uma reconfiguração, somente essas instruções precisam ser enviadas. A título de exemplo, em uma reconfiguração de uma aplicação que realiza a leitura de um sensor e o envio do dado lido pelo rádio na máquina virtual Maté, necessitou apenas da transferência de 17 bytes [HKS+05]. A reconfiguração do código da própria máquina virtual geralmente não é permitida. A grande limitação no uso das máquinas virtuais em sistemas profundamente embarcados está no sobrecusto introduzido pelo interpretador (demonstrado na seção anterior), na dependência da linguagem de script com as instruções da arquitetura alvo, na grande quantidade de memória utilizada e no grande consumo de energia devido ao processamento [KP05a].

Os sistemas operacionais possuem um tempo de reconfiguração intermediário, são mais rápidos que as infra-estruturas mas, por outro lado, são mais lentos que as máquinas virtuais. O sistema SOS, em uma reconfiguração, necessita que o módulo antigo seja removido do sistema, que o novo módulo seja registrado e que o tratador (*handler*) dos eventos do módulo seja recuperado. Para tanto, são gastos 230 ciclos para a remoção do módulo, 267 ciclos para efetuar o registro do novo módulo e mais 124 ciclos para o tratador, totalizando 621 ciclos, desconsiderando os tempos de recepção do código e os tempos de escrita do código em memória [YMCH08]. Contiki também apresenta

desempenho similar, pois o kernel envia uma mensagem de remoção para o serviço, é realizada a troca de estado entre os serviços antigo e o novo e o registro das funções da nova interface de serviço. O RETOS utiliza relocação dinâmica de memória e ligação dos módulos em tempo de execução. Os meta-dados que contém informações sobre o módulo são transmitidos juntamente com o código objeto do módulo em um formato de arquivo RETOS. Os meta-dados adicionados no arquivo aumentam a quantidade de dados enviados pela rede, tornando o processo de reconfiguração ainda mais lento.

Na atualização de um componente com tamanho de código maior que o antigo no ELUS, são necessários apenas 449 ciclos de processamento. Comparando com os sistemas operacionais analisados, esse desempenho é superior. O ELUS não necessita o registro e nem a remoção de componentes em estruturas e tabelas internas em tempo de execução, se limitando apenas na adição e remoção do objeto em um tabela *hash*, diferentemente do que é realizado no SOS e Contiki. O ELUS também reduz o envio de dados de controle através do `EPOS Transport Protocol`, necessitando apenas de 2 a 6 bytes de informação por mensagem.

Tabela 5.5: Resumo da comparação dos três tipos de sistemas analisados levando em consideração o consumo de memória, desempenho na invocação de métodos e tempo de reconfiguração em sistemas profundamente embarcados. Consumo baixo de 1 a 5kb, moderado de 5 a 10kb e alto mais de 10kb.

Tipo de Sistema	**Consumo de Memória**	**Invocação de Métodos**	**Tempo de Reconfiguração**
Infra-estruturas	Moderado/Alto	Baixo	Alto
Máquinas Virtuais	Moderado/Alto	Alto	Baixo
Sistemas Operacionais	**Baixo**/Moderado	Moderado	Moderado

A Tabela 5.5 mostra o resumo das características avaliadas nos três tipos de sistema que suportam reconfiguração dinâmica de software: infra-estruturas, máquinas virtuais e sistemas operacionais. As máquinas virtuais apresentam um alto consumo de memória e um alto sobrecusto na interpretação das instruções, sendo que a grande

vantagem está no tempo de reconfiguração. Infra-estruturas têm a vantagem do tempo de invocação de métodos/funções não ser alterado. Já os sistemas operacionais, de uma maneira geral, apresentam tempo de reconfiguração e invocação de métodos moderados. O consumo de memória era o principal ponto fraco dos SOs, mas o ELUS mostrou-se ser eficiente também neste aspecto.

A Tabela 5.6 apresenta um quadro comparativo do processo de reconfiguração de software nos sistema operacionais para sistemas embarcados analisados nesta dissertação. O grande diferencial do ELUS com relação aos outros SOs está no fato de que os componentes do sistema podem ser selecionados como reconfiguráveis ou não em tempo de compilação e, para todos os componentes não selecionados, nenhum sobrecusto de memória e processamento é adicionado ao sistema.

Tabela 5.6: Comparação do processo de reconfiguração nos sistemas operacionais analisados e no ELUS.

Sistema Operacional	**Processo de Reconfiguração**
TINYOS	Sem suporte direto
MantisOS	Não tem suporte
Nano-Kernel	Módulos reconfiguráveis
RETOS	Relocação dinâmica e ligação em tempo de execução
Contiki	Módulos reconfiguráveis
SOS	Módulos reconfiguráveis
Think	Modelo reflexivo (Fractal)
ELUS	**Componentes reconfiguráveis e selecionados em tempo de compilação**

5.5.4 Alocação e Liberação de Memória

O ELUS utiliza alocação dinâmica de memória para a criação de novos objetos dentro do framework metaprogramado e liberação de memória para a destruição dos objetos antigos quando é recebida uma mensagem de adição de atributo(s). Os objetos

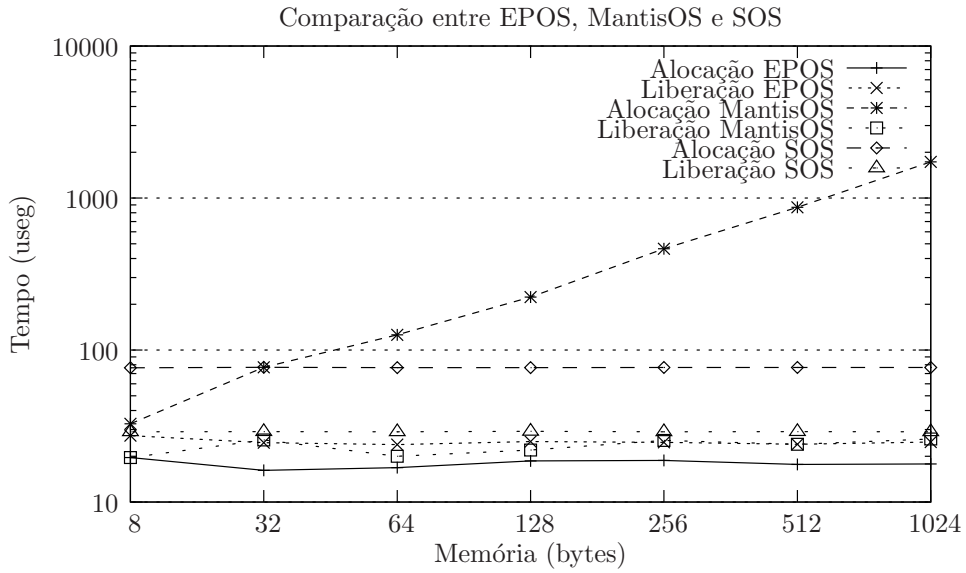

Figura 5.2: Comparação entre alocação e liberação de memória nos sistemas EPOS, MantisOS e SOS.

são importantes para realizar as invocações de métodos através da `vtable` e também para atualizar os endereços dos métodos quando uma reconfiguração ocorre. Para avaliar o impacto de uma alocação e liberação de memória dentro do ELUS, foi realizada uma comparação de desempenho entre os sistemas operacionais embarcados mais utilizados que possuem alocação e liberação dinâmica de memória.

A Figura 5.2 apresenta a comparação entre as funções de alocação e liberação de memória nos sistemas operacionais EPOS, MantisOS 1.0 [ABC+03] e SOS 2.0.1 [HKS+05]. O número de bytes alocados/liberados foram 8, 32, 64, 128, 256, 512 e 1024. Os testes foram realizados na arquitetura AVR utilizando o microcontrolador ATMega128 de 8-bits. O sistema operacional Contiki não foi avaliado pois sua versão para o microcontrolador ATMega128 no momento da comparação mostrou-se instável. O sistema SOS foi escolhido por implementar suporte à reconfiguração dinâmica e por ser amplamente conhecido na área de RSSF. MantisOS foi escolhido pois é amplamente conhecido na área de sistemas embarcados. TINYOS não foi usado nas comparações pois não apresenta suporte a alocação/liberação dinâmica de memória [HSW+00].

Os testes foram mensurados usando as funções de alocação e libera-

ção de memória e de medida de tempo em cada sistema operacional, como mostra a Tabela 5.7. Os resultados são a representação da média de valores obtidos em 1000 execuções. A Figura 5.2 mostra que o EPOS apresentou o melhor desempenho em termos de alocação de memória. Em termos de liberação de memória os três sistemas apresentaram comportamento semelhante. A alocação de memória no MantisOS mostrou-se ser dependente do número de bytes a serem alocados. O gerenciamento de memória no EPOS não depende do tamanho de memória. O seu bom desempenho é devido ao uso de metaprogramação estática na lista responsável pelo gerenciamento dos espaços livres de memória. Portanto, a alocação e/ou liberação dinâmica de memória não é um fator de perda de desempenho para o ELUS.

Tabela 5.7: Principais características dos sistemas operacionais EPOS, SOS e MantisOS.

SO	**Arquiteturas Suportadas**	**Arquitetura do SO**	**API gerência de memória**	**API temporização**
EPOS	AVR, IA32, PowerPC, MIPS	multithreading	malloc(), free()	Chronometer.read()
MantisOS	AVR, MSP430, IA32, XScale	multithreading	mos_mem_alloc(), mos_mem_free()	mos_get_realtime()
SOS	AVR, MSP430, IA32, XScale	Event-driven	sys_malloc(), sys_free()	sys_time32()

Capítulo 6

Considerações Finais

Reconfiguração dinâmica de software é uma característica extremamente importante para uma ampla variedade de sistemas, pois permite que novas atualizações a nível de software sejam adicionadas ao sistema em tempo de execução (e.g. "em campo" sem a necessidade de coletar ou alcançar os nodos fisicamente e programá-los), corrigindo eventuais erros ou agregando novas funcionalidades ao sistema. Reconfiguração dinâmica de software em sistemas profundamente embarcados é um desafio pela própria natureza desses sistemas, onde o poder de processamento, a memória disponível e o consumo de energia são muito limitados. Neste caso, o próprio método de reconfiguração dinâmica de software deve ser eficiente ao ponto de causar o mínimo de influência possível para o sistema.

Esta dissertação apresentou o projeto, implementação e avaliação de uma estrutura de sistema operacional que permite reconfiguração dinâmica de software em sistemas profundamente embarcados. A estrutura é chamada de EPOS LIVE UPDATE SYSTEM (ELUS) e foi construída em torno do aspecto de invocação remota presente no sistema operacional EPOS. O ELUS é composto por um framework metaprogramado responsável por criar um nível de indireção entre as invocações de métodos dos componentes reconfiguráveis e pela atualização do código dos componentes e por uma thread responsável por receber um pedido de reconfiguração vindo do ambiente externo através de uma tecnologia de comunicação (e.g. rádio, ethernet, etc).

Os principais diferenciais do ELUS em relação aos sistemas existentes são:

- **Configurabilidade:** o ELUS permite a seleção dos componentes reconfiguráveis em tempo de compilação. Para todos os componentes não selecionados, nenhum sobrecusto, em termos de memória e desempenho, é adicionado ao sistema.

- **Consumo de memória:** o ELUS tem um consumo de memória bastante reduzido quando comparado às outras técnicas avaliadas neste trabalho. São necessários pouco mais 1.5kb de memória de código e 72 bytes de dados para cada componente reconfigurável do sistema.

- **Reconfiguração:** através do nível de indireção criado pelo framework, os componentes se tornam independentes de posição na memória do sistema, fazendo com que não haja a necessidade de ligação em tempo de execução ou a presença de um *bootloader* para efetuar a reconfiguração.

- **Estrutura de mensagem:** o framework através da estrutura de mensagem cria uma forma eficiente para a passagem de argumentos e/ou valor de retorno entre os métodos dos componentes reconfiguráveis.

- **Simplicidade:** o sistema não tem a necessidade de registrar e desregistrar os componentes em tempo de execução. A reconfiguração é realizada de forma simples através do ELUS TRANSPORT PROTOCOL.

- **Transparência:** o sistema é totalmente transparente para as aplicações e seus usuários. Para a aplicação, uma invocação de um método de um componente reconfigurável é realizada sem nenhuma modificação. Da mesma forma, uma reconfiguração é efetuada transparentemente para a aplicação.

6.1 Trabalhos Futuros

A pesquisa realiza indica os seguintes trabalhos futuros:

- Construção de um framework para criação de protocolos de disseminação de dados reconfiguráveis levando em consideração as características semelhantes dos protocolos estudados e apresentadas neste trabalho. Neste framework, o desenvolvedor poderá escolher quais características são interessantes para a sua aplicação e o framework gerará o protocolo conforme as suas escolhas.

- Integração de um protocolo de disseminação de dados com o ELUS e consequentemente a sua avaliação em termos de consumo de energia e latência.

- Investigar a possibilidade de reconfiguração de software em sistemas de tempo-real. Estudar e propor melhorias nas garantias de tempo-real em nível de protocolos MAC e protocolos de disseminação de dados.

- Implementação de um gerador automático do framework do ELUS baseado nas descrições dos componentes reconfiguráveis em arquivos xml.

- Implementação de um analisador de código capaz de verificar as mudanças entre duas versões de um componente e criar mensagens de reconfiguração no formato do ELUS TRANSPORT PROTOCOL.

- As reconfigurações de software existentes não levam em consideração as mudanças no hardware, assim como um hardware reconfigurável ignora as mudanças no software. O que pode ser explorado é exatamente a integração do software e hardware no domínio de reconfiguração dinâmica.

Referências Bibliográficas

[ABC+03] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. 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, 2003. ACM.

[Aru04] Mahesh Arumugam. 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, 2004. ACM.

[Bag08] Susmit Bagchi. Nano-kernel: a dynamically reconfigurable kernel for wsn. In *MOBILWARE '08: Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[BD93] T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[BHA+05] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic

update in an operating system. In *ATEC'05: Proc. of the annual conf. on USENIX Annual Technical Conference*, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[BHR+06] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 112–121, New York, NY, USA, 2006. ACM.

[BHS03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. 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, 2003. ACM Press.

[CCJ+07] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. 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, 2007. ACM.

[CCVV96] Adam M. Costello, Adam M. Costello, George Varghese, and George Varghese. Self-stabilization by window washing. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 35–44, 1996.

[DAW+08] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session*, Raleigh, North Carolina, USA, November 2008. Best poster award.

[Dev02] Advanced Micro Devices. *AMD Athlon Processor x86 Code Optimization Guide*. Sunnyvale, California, publication 22007, Feb. 2002.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 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, November 2004.

[EN03] Friedrich-Alexander-Universität Erlangen-Nürnberg. Technologie roadmap: Software-verwaltung im farzeug. Technical report, DaimlerChrysler, Erlangen, Germany, sep 2003.

[FKKP07] Meik Felser, Rüdiger Kapitza, Jörgen Kleinöder, and Wolfgang Schröder Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In *International Embedded Systems Symposium 2007 (IESS '07)*, 2007.

[Frö01] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[FSLM02] Jean P. Fassino, Jean B. Stefani, Julia Lawall, and Gilles Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA, June 2002. USENIX Association.

[FSP00] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[Gon07] Raphael Tucunduva Gonçalves. Monitoramento da carga de baterias em sistemas embarcados, 2007.

[HC04] Jonathan W. Hui and David Culler. 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, 2004. ACM.

[HG98] Gilsi Hjalmtysson and Robert Gray. Dynamic C++ classes—A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conf.*, pages 65–76, June 1998.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

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

[HKSS05] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. Sensor network software update management: a survey. *Int. J. Netw. Manag.*, 15(4):283–294, 2005.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[HW04] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[Jun07] Arliones Stevert Hoeller Junior. Gerenciamento do consumo de energia dirigido pela aplicação em sistemas embarcados. Master's thesis, Universidade Federal de Santa Catarina, 2007.

[JWF06] Arliones Stevert Hoeller Junior, Lucas Francisco Wanner, and Antônio Augusto Fröhlich. A hierarchical approach for power management on mobile embedded systems. In *5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 265–274, Braga, Portugal, 2006.

[KJP05] Minseong Kim, Jaemin Jeong, and Sooyong Park. From product lines to self-managed systems: an architecture-based runtime reconfiguration framework. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering*, 16:1293–1306, 1990.

[KP05a] J. Koshy and R. Pandey. 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, January 2005.

[KP05b] Joel Koshy and Raju Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *In Proc. SenSys?05*, pages 243–254. ACM Press, 2005.

[LC02] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[LGN05] Patrick E. Lanigan, Rajeev Gandhi, and Priya Narasimhan. Disseminating code updates in sensor networks: Survey of protocols and security issues. Technical Report CMU-ISRI-05-122, Carnegie Mellon University, October 2005.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[LM03] Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *In PPoPP ?03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[LSL+94] J.W.S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. IEEE*, 82, Jan 1994.

[MCSF09] Hugo Marcondes, Rafael Cancian, Marcelo Stemmer, and Antônio Augusto Fröhlich. Modelagem e implementação de escalonadores de tempo real para sistemas embarcados. In *VI Workshop de Sistemas Operacionais*, Bento Gonçalves, RS, Brasil, Julho 2009.

[MGL+06] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. 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, February 2006.

[MHWF06] H. Marcondes, A.S. Hoeller, L.F. Wanner, and A.A.M. Frohlich. Operating systems portability: 8 bits and beyond. *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 124–130, 20-22 Sept. 2006.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. 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, January 2005.

[NASZ05] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices, 2005.

[NASZ07] Vinayak Naik, Anish Arora, Prasun Sinha, and Hongwei Zhang. Sprinkler: A reliable and energy efficient data dissemination service for extreme scale wireless networks of embedded devices. *IEEE Transactions on Mobile Computing*, 6(7):777–789, 2007.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[PF04] Fauze Valério Polpeta and Antônio A. Fröhlich. 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, 2004. Springer.

[PMSD07] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *CBSE*, pages 242–257, 2007.

[PS08] Juraj Polakovic and Jean-Bernard Stefani. Architecting reconfigurable component-based operating systems. *Journal of Systems Architecture*, 54(6):562–575, 2008.

[RL03] Niels Reijers and Koen Langendoen. 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, 2003. ACM Press.

[SAH+03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[SHE03] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, Los Angeles, CA, USA, 2003.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[Tan88] Andrew S. Tanenbaum. *Computer networks: 2nd edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[TDV08] Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. Efficient sensor network reprogramming through compression of executable modules. In *Proceedings of Fifth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks – SECON 2008*, San Francisco, California, USA, 2008.

[Wan04] Limin Wang. 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, 2004. ACM.

[Wan06] Lucas Francisco Wanner. Suporte de sistema operacional para redes de sensores sem fios. Master's thesis, Universidade Federal de Santa Catarina, 2006.

[WdOF07] Lucas Francisco Wanner, Augusto Born de Oliveira, and Antônio Augusto Fröhlich. Configurable medium access control for wireless sensor networks. In *International Embedded System Symposium*, pages 401–410, Irvine, CA, USA, 2007.

[Wie08] Geovani Ricardo Wiedenhoft. Gestão de energia para sistemas embarcados de tempo real usando técnicas da computação imprecisa. Master's thesis, Universidade Federal de Santa Catarina, 2008.

[WWGF08] Geovani Ricardo Wiedenhoft, Lucas Francisco Wanner, Giovani Gracioli, and Antônio Augusto Fröhlich. Power management in the epos system. *SIGOPS Operating Systems Review*, 42(6):71–80, 2008.

[XLC06] Q. Xie, J. Liu, and P.H. Chou. 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, 2006.

[YMCH08] Sangho Yi, Hong Min, Yookun Cho, and Jiman Hong. Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems. *Comput. Commun.*, 31(4):699–707, 2008.