RITA DE CÁSSIA CAZU SOLDI

ESTUDO SOBRE MÉTODOS PARA MELHORIA DE DESEMPENHO EM MECANISMOS DE RECONFIGURAÇÃO DINÂMICA DE SOFTWARE DE SISTEMAS EMBARCADOS

Florianópolis

2011

RITA DE CÁSSIA CAZU SOLDI

ESTUDO SOBRE MÉTODOS PARA MELHORIA DE DESEMPENHO EM MECANISMOS DE RECONFIGURAÇÃO DINÂMICA DE SOFTWARE DE SISTEMAS EMBARCADOS

Relatório final do desenvolvimento do Trabalho de Conlusão de Curso, parte dos requisitos para aprovação na disciplina Projeto 2

Orientador:

Antônio Augusto Medeiros Fröhlich

Coorientador :

Giovani Gracioli

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis

2011

Trabalho de conclusão de curso o sob título "Estudo Sobre Métodos para Melhoria de Desempenho em Mecanismos de Reconfiguração Dinâmica de *Software* de Sistemas Embarcados", defendido por Rita de Cássia Cazu Soldi e aprovado dia ____ de _____ de _____ , em Florianópolis, Estado de Santa Catarina, pela banca examinadora constituída por:

Banca Examinadora:

Profº Dr. Antônio Augusto Medeiros Fröhlich
Orientador

M. Sc. Giovani Gracioli
Coorientador

M. Sc. Arliones Stevert Hoeller Junior

B. Sc. Rodrigo Steiner

RESUMO

A reconfiguração dinâmica é a capacidade de atualizar um sistema sem que se perca o estado de execução do mesmo. Este tipo de reprogramação é extremamente importante, pois molda o sistema às mudanças no seu ambiente, seja através de mudanças de parâmetros, corrigindo *bugs* ou até acrescentar ou reduzir funcionalidades. Porém, realizar este processo em sistemas embarcados com severas restrições de recursos se torna mais desafiador, visto que os mecanismos de atualização acabam competindo por estes recursos.

O EPOS Live Update System (ELUS) é uma infraestrutura de sistema operacional para reconfiguração dinâmica que possui resultados comprovadamente favoráveis no que diz respeito ao tempo e ao uso de recursos se comparado às outras infraestruturas que realizam a mesma função. Porém, apesar destes bons resultados, ainda é passível de melhoria, e por esta razão o ELUS foi escolhido para ser a base do estudo de melhorias proposto neste trabalho.

O objetivo deste trabalho é realizar um estudo sobre métodos para a melhora de desempenho em mecanismos de reconfiguração dinâmica de *software* para sistemas embarcados. Para tanto, foram revisado os conceitos envolvidos no processo e também foram apresentas duas propostas: (i) utilizar o modelo FRACTAL de componentes como para a melhoria de desempenho do modelo de reconfiguração utilizado pelo ELUS e (ii) realizar alterações na implementação do framework do ELUS.

O resultado apresentado para a primeira proposta é uma modelagem híbrida, a partir da qual é possível observar que apesar de todos sistemas operacionais apresentados nos trabalhos relacionados serem descritos de maneira diferenciada, eles compartilham os mesmos conceitos, premissas e, consequentemente, o mesmo processo de reconfiguração.

Para a segunda proposta houve uma remodelagem do ELUS e a implementação das modificações necessárias. Como resultado, foi alcançada a melhoria de desempenho no mecanismo de reconfiguração dinâmica do ELUS.

Palavras-chave: Sistema embarcado, reconfiguração dinâmica, ELUS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADL	*Architecture Description Language*,	p. 30
ASVM	Application Specific Virtual Machine,	p. 13
DVM	Dynamic Virtual Machine,	p. 13
ELUS	Epos Live Update System,	p. 8
EPOS	Embedded Parallel Operating System,	p. 19
ETP	ELUS Transport Protocol,	p. 20
IDL	*Interface Description Language*,	p. 31
RSSF	Rede de Sensores Sem Fio,	p. 11
SO	Sistema Operacional,	p. 15
vtable	Tabela de Métodos Virtuais,	p. 26

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A reconfiguração dinâmica é a capacidade de atualizar o *software* de um determinado sistema durante sua execução e pode ser usada para diversas finalidades como: realizar atualizações, implementar algoritmos para substituir componentes do sistema sem desligá-lo, monitoramento dinâmico, apoio à otimizações específicas da aplicação, ajuda para que componentes do sistema possam ser recebidos da rede e instalados em tempo de execução, entre outros.

Este tipo de reprogramação é extremamente importante para que possamos adaptar os dispositivos quanto às mudanças que ocorrem no ambiente, modificar o seu conjunto de tarefas e até aprimorar o uso de seus recursos. Entretanto, aplicar a reconfiguração dinâmica a sistemas embarcados com severas restrições (ex. processamento, memória, energia) é um desafio, pois apesar dos mecanismos de atualização competirem pelos já limitados recursos do sistema, eles não devem influenciar o funcionamento do mesmo.

Existem três abordagens diferentes para realizar a reconfiguração dinâmica: atualização do código binário (REIJERS; LANGENDOEN, 2003) (FELSER et al., 2007) (MARRÓN et al., 2006) (YI et al., 2008), máquinas virtuais (BALANI et al., 2006) (LEVIS; CULLER, 2002) (BOULIS et al., 2007) (KOSHY; PANDEY, 2005b) e sistemas operacionais (KYLE; BRUSTOLONI, 2007) (DUNKELS et al., 2004) (CHA et al., 2007) (HAN et al., 2005). Na primeira, são criados métodos para a atualização do código binário do sistema e para que a recepção e a substituição aconteçam utiliza-se ou um gerenciador de inicialização ou um ligador. Já as máquinas virtuais oferecem um ambiente para a execução de instruções da sua linguagem de *script*, assim são criadas aplicações e enviadas para a plataforma alvo que, em tempo de execução, interpreta as instruções. Na abordagem com sistemas operacionais normalmente é realizada uma organização de módulos reconfiguráveis e são criados níveis de indireção entre a aplicação e os módulos através de tabelas e ponteiros. Os novos módulos são substituídos em tempo de execução alterando-se o endereço dos ponteiros ou das tabelas.

Cada um dos tipos de reconfiguração têm suas vantagens e desvantagens. Um estudo comparativo entre as soluções (GRACIOLI; FRÖHLICH, 2009) demonstrou que a abordagem utilizada pelos sistemas operacionais possui melhor balanceamento entre consumo de energia,

tempo de reconfiguração e tempo de invocação dos métodos.

O *Epos Live Update System (ELUS)* é uma infraestrutura de sistema operacional para a reconfiguração dinâmica e que difere das outras infraestruturas pelo baixo consumo de memória, alta configurabilidade, simplicidade e transparência para as aplicações (GRACIOLI; FRÖHLICH, 2009). Comparado-se o ELUS com os trabalhos relacionados pôde-se observar que ele possui um menor tempo de atualização e uma melhor utilização dos recursos. Apesar dos resultados favoráveis, ainda é possível melhorar o consumo de memória, o tempo de invocação de métodos e o tempo da reconfiguração e por isso o ELUS foi escolhido para o estudo proposto neste trabalho.

1.1 OBJETIVOS

O objetivo principal deste trabalho é realizar um estudo sobre métodos para melhorar os mecanismos de reconfiguração de *software* em sistemas embarcados. Para que ele seja atingido, são definidos os seguintes objetivos específicos:

1. Estudar os principais modelos de reconfiguração dinâmica de *software* para sistemas embarcados, a fim de levantar o estado da arte;
2. Estudar conceitos e limitações da infraestrutura ELUS. Importante para compreensão do ambiente no qual a proposta de melhoria será realizada;
3. Modelar e implementar uma proposta que consiga reduzir as limitações encontradas no ELUS;
4. Testar as propostas e analisar a relevância deste trabalho.

1.2 ORGANIZAÇÃO DO TRABALHO

O texto que segue apresenta-se na seguinte estrutura:

O capítulo 2 “RECONFIGURAÇÃO DINÂMICA DE *SOFTWARE* EM SISTEMAS EMBARCADOS” apresenta os trabalhos relacionados referentes à reconfiguração dinâmica de *software* em sistemas embarcados.

O capítulo 3 “*EPOS LIVE UPDATE SYSTEM* (ELUS)” detalha a o ELUS quanto à sua organização e conceitos que serão importantes para o entendimento das propostas de melhoria.

No capítulo 4 “PROPOSTAS DE MELHORIA NO ELUS” serão apresentados detalhes das propostas de melhoria de desempenho, explicando qual limitação elas pretendem reduzir e como serão implementadas.

O capítulo 5 “RESULTADOS OBTIDOS” apresenta os resultados obtidos através das modelagens apresentadas no capítulo 4.

Finalmente, no capítulo 6 “CONCLUSÕES” serão apresentadas as contribuições deste trabalho, desafios encontrados e trabalhos futuros.

2 *RECONFIGURAÇÃO DINÂMICA DE SOFTWARE EM SISTEMAS EMBARCADOS*

Os mecanismos de reconfiguração de *software* em sistemas embarcados podem ser separados em três grupos: os sistemas operacionais, as máquinas virtuais e os que se baseiam na atualização do código binário. Os principais trabalhos relacionados serão apresentados nas próximas seções.

2.1 RECONFIGURAÇÕES BASEADAS EM CÓDIGO BINÁRIO

Na reconfiguração baseada em código binário, as atualizações são enviadas para os nodos através de diferentes métodos e a partir deste novo código o sistema consegue se atualizar. A seguir são apresentados alguns trabalhos na área.

Diff-like é um esquema para atualizar o código de uma rede de sensores, levando em consideração a economia de energia (REIJERS; LANGENDOEN, 2003). Baseado na hipótese de que mesmo quando existe uma atualização grande de uma aplicação, na maioria das vezes as mudanças do código binário são pequenas, foi desenvolvida uma solução similar ao comando *diff* presente nos sistemas operacionais baseados em UNIX.

Sendo assim, Diff-like recebe somente a diferença entre a nova e a velha imagem do sistema, diminuindo assim a quantidade de código transferido pela rede, a utilização de recursos e consequentemente a energia consumida pelos nodos. Para tanto, foi produzido um algoritmo que consegue transformar as diferenças das imagens em um conjunto de comandos baseados em linguagem de *script* como por exemplo *copy, repair* e *insert*.

O processo de reconfiguração é dividido em quatro partes: a inicialização, a construção da imagem, a verificação e o carregamento. Na inicialização, os nodos são informados que a reprogramação se iniciou, então começam a preparar a memória em que o novo código será organizado. Na segunda fase, uma série de mensagens é enviada para que a nova imagem seja montada na área da memória preparada anteriormente. Uma vez montada, é feita uma verificação para conferir os dados recebidos e garantir que cada nodo construiu a imagem corretamente.

Na fase de carregamento as atualizações são realizadas através de comandos baseados em linguagem de *script* e o sistema é reiniciado.

Ainda são abordadas no trabalho soluções para que o sistema seja tolerante à perda de pacotes com dados para a atualização e também sugestões para uma atualização do código em tempo de execução.

Felser et al. propuseram uma infraestrutura para atualizar dinamicamente os nodos de uma Rede de Sensores Sem Fio (RSSF) baseada no código binário da aplicação. A maior parte das informações e do processamento estão no nodo com o maior número de recursos (normalmente a estação base). Quando se deseja atualizar o sistema, o primeiro passo é verificar se a atualização pode ser realizada com segurança. Como base para esta decisão, são analisadas informações como a tabela de símbolos, a tabela de alocação de dados e informações de *debug*, entre outras(FELSER et al., 2007).

Caso seja verificado algum problema ou se as informações obtidas não são suficientes para tomar esta decisão automaticamente, a atualização é considerada insegura, então o sistema pergunta ao administrador se ela pode ou não ser realizada. Este questionamento pretende preservar o estado do sistema, mas só é realmente válida quando o administrador apresenta grande conhecimento sobre o sistema e também sobre o processo de atualização.

Já no caso da atualização ser considerada segura, as atividades dos nodos são suspensas e um ligador incremental remoto (KOSHY; PANDEY, 2005a) liga a nova imagem para a aplicação com a imagem antiga que estava rodando nos nodos. Assim o ligador incremental pode controlar as funções que serão modificadas, bem como deslocar ou realocar o espaço para que as novas funções/modificações sejam acrescidas/reduzidas. Lembrando que somente as diferenças entre as imagens é repassada para os demais nodos da rede.

Após o término das atualizações o sistema não precisa ser reiniciado. A vantagem de suspender as atividades dos nodos está na melhor recuperação do sistema, porque o tempo em que os nodos estarão *offline* é compensado pelo tempo que não será gasto para que o sistema seja reiniciado.

FlexCup (MARRÓN et al., 2006) é um sistema de atualização que permite a reconfiguração dinâmica de nodos que possuem seu *software* baseado no sistema TinyOS (HILL et al., 2000). Foi desenvolvido como parte do TinyCubus (MARRÓN et al., 2005) para fornecer a capacidade de atualização de código dos componentes do *Tiny Data Management Framework*, mas também pode ser utilizado independentemente.

Seu sistema de reconfiguração é dividido em duas etapas : uma de geração de código e outra

de ligação do código. Na primeira etapa, os componentes são compilados na estação base, há uma geração de metadados que os descrevem e também são inseridas informações como tabela de símbolos e de realocação. A fase de ligação do código inicia-se ao receber a atualização dos dados, bem como o novo código e os metadados dos componentes, armazenando-os na memória *flash*. Em seguida a tabela de símbolos da nova imagem é associada à tabela de símbolos da imagem antiga.

A realocação de endereços e a atualização das referências são tarefas de ligador, instalado em cada nodo da rede, que precisa percorrer as entradas da tabela de realocação de todos os componentes e verificar se alguma referência precisa ser atualizada. Lembrando que todas as referências de um novo componente ou de componentes que foram atualizados devem ser obrigatoriamente atualizados. Depois que todas as referências foram corrigidas, a imagem é transferida da memória *flash* para a memória de programa e os nodos são reinicializados utilizando um *bootloader*. O passo de reinicialização dos nodos é importante para eliminar problemas com ponteiros inválidos.

Molecule é um mecanismo adaptativo de reconfiguração dinâmica para RSSF que possuem recursos limitados (YI et al., 2008). Com exceção do *kernel* e das estruturas de dados, elementos como aplicações do usuário, *drivers* dos dispositivos e até os próprios recursos do *kernel* são considerados como módulos carregáveis. A atualização dos módulos é realizada através de ligação direta ou indireta entre os componentes e, sendo assim, podemos considerar que Molecule realiza a reconfiguração tanto no nível do usuário quanto no nível do *kernel*.

Para escolher a melhor combinação, é realizada uma análise de custo para o tempo de execução esperado em cada módulo. Se o melhor custo for o da ligação direta, ela não aumenta o custo na chamada de funções ou no acesso de dados entre os módulos, mas na reconfiguração é mais custoso em termos de energia e processamento, pois cada endereço do módulo que está sendo chamado por outros deve ser atualizado.

Já na ligação indireta, apesar do tempo de execução e o tamanho do código serem maiores do que na ligação direta, a reconfiguração é muito mais rápida, uma vez que toda chamada de função é realizada através de uma tabela com os endereços e somente os endereços nessa tabela seriam atualizados.

Como todas as ligações são efetivadas nos nodos e há um consumo desnecessário dos limitados recursos, foi proposto um terceiro tipo de ligação, na qual todo o processo de atualização é feito em uma estação base (que possui um maior poder de processamento) e apenas as modificações seriam enviadas para os nodos.

2.2 MÁQUINAS VIRTUAIS

A reconfiguração realizada no ambiente das máquinas virtuais normalmente é simples. Por meio de instruções da linguagem de *script* própria, as instruções são interpretadas e executadas em tempo de execução.

Dynamic Virtual Machine (DVM) é uma máquina virtual que executa sob o sistema operacional SOS (BALANI et al., 2006). Seu projeto foi inspirado na estrutura da Maté (LEVIS; CULLER, 2002) e da Application Specific Virtual Machine (ASVM) (LEVIS; GAY; CULLER, 2005), sendo sua arquitetura dividida em um núcleo responsável por interpretar e executar os *scripts* e na linguagem de *scripts* que é compilada para o conjunto de comandos específicos da arquitetura alvo. A DVM utiliza a estrutura de módulos do SOS para carregar novas extensões em tempo de execução.

Maté (LEVIS; CULLER, 2002) é uma máquina virtual que executa sobre o sistema operacional TinyOS (HILL et al., 2000). Possui uma *interface* de alto nível, permitindo que programas complexos fiquem mais curtos (menos de 100 *bytes*), reduzindo assim o uso de bateria na transmissão de novos programas.

O código é dividido em pequenas cápsulas de 24 instruções, que podem se replicar pela rede. Entretanto, quando uma aplicação executa por um longo período, a energia gasta para interpretar o código supera essa vantagem.

SensorWare fornece uma máquina virtual cuja linguagem de *script* é de alto nível e baseada em TCL. Ela permite uma abstração versátil dos nodos sensores e consegue instalar novos serviços dinamicamente. Através de comandos para replicar ou migrar código, o mesmo é seletivamente transferido apenas para os nodos necessários de forma autônoma, tornando a implantação e a execução simultâneas (BOULIS et al., 2007).

Tapper é uma máquina virtual que possui uma linguagem de *scripts* própria e não necessita de um ambiente específico (ex. sistema operacional) para a sua execução, podendo invocar rotinas já compiladas que vão desde rotinas simples como um *timer* até as mais complexas, como configurar interrupções (XIE; LIU; CHOU, 2006).

Ela também fornece uma camada de *software* que consegue interpretar comandos legíveis por seres humanos, como por exemplo, o comando Ax que executa uma leitura no canal “x” do ADC.

No processo de reconfiguração, são usadas as instruções do tipo “+” e “-”, que respectivamente acrescentam ou removem a função desejada do sistema. Estas instruções são traduzidas

em um comando de memória por uma máquina que conhece a memória de dados e o código de cada um dos nodos.

Para que ocorra a tradução, a máquina passa o endereço inicial da nova posição de memória para a nova função adicionada e assim que a memória é alocada, um gerenciador de *buffer* grava este dado em uma tabela, na qual também serão gravados o nome da função e seu tamanho. A máquina virtual ainda não possui suporte para distribuição do código para todos os nodos da rede.

VM* é um *framework* para a construção de ambientes de execução para RSSFs (KOSHY; PANDEY, 2005b). Ele é formado por um interpretador que executa *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.

Já que a maioria das atualizações feitas nos aplicativos na RSSF são incrementais (mesmo em situações extremas os trechos significativos podem permanecer iguais), a reconfiguração dinâmica aproveita esta característica utilizando a técnica de ligação incremental (KOSHY; PANDEY, 2005a), que também ajuda a diminuir o consumo de energia utilizado.

A reconfiguração conta também com um *bootloader* e algumas instruções de *diff-script* como *copy*, *run* e *add*. Lembrando que estes elementos somente são introduzidos no sistema se a atualização de código for necessária.

A fim de reduzir o custo de realocação de memória quando há aumento do código (ex. código novo de uma função maior do que o antigo) são inseridos espaços em branco entre as funções, porém esta abordagem consome mais memória e no caso de sistemas embarcados é um fator crítico.

2.3 SISTEMAS OPERACIONAIS

Na abordagem utilizada pelos sistemas operacionais os módulos são carregados em tempo de execução e alguns deles ainda permitem que uma instância de um objeto seja trocada de modo transparente, também em tempo de execução. A seguir serão identificados os principais sistemas operacionais que suportam a reconfiguração de *software* em sistemas embarcados:

μCLinux é um porte do *kernel* do Linux voltado para sistemas embarcados que não possuem uma unidade de gerenciamento de memória. Apesar de seu código ser remodelado para reduzir o uso da memória, do ponto de vista da aplicação, a interface oferecida ainda é bem parecida com a do Linux, contando com comandos *shell*, suporte à biblioteca C (modificada) e

também possui as chamadas de sistema UNIX (KYLE; BRUSTOLONI, 2007).

Na reconfiguração é utilizado o carregamento de módulo dinâmico que acontece a partir de *bitstreams*, que normalmente são utilizados para descrever quais dados devem ser carregados e são compostos de sequências de comandos de reconfiguração e de pacotes de dados. (WILLIAMS; BERGMANN, 2004)

Os *bitstreams* parciais são oriundos de programas escritos na linguagem C, de *scripts shell* automatizados, de um servidor remoto através da rede ou até de abordagens que utilizam a interface do sistema embarcado (ex. a interface ICAP do FPGA). De maneira simplista podemos dizer que o processo de atualização pode ser descrito como a execução dos comandos de reconfiguração.

Contiki é um sistema operacional leve que possui suporte ao carregamento dinâmico e substituição dos serviços disponibilizados. Foi desenvolvido para atuar em redes de sensores sem fio e possui um *kernel* baseado em eventos que possibilita a utilização de *multithread* preemptiva para cada um dos processos. Esta abordagem de *kernel* o torna muito sensível à solicitações e acontecimentos e por isso esse Sistema Operacional (SO) pode ser considerado como um sistema de tempo real (DUNKELS et al., 2004).

Em tempo de compilação o sistema é particionado em dois grupos para que cada um possa ser implantado de uma maneira. No grupo do núcleo permanecem o *kernel*, o carregador de programas, o suporte da linguagem em tempo de execução e o suporte à comunicação das aplicações com os dispositivos de hardware. Este grupo é compilado em uma única imagem e armazenado nos dispositivos antes da implantação. Já o outro grupo é composto apenas pelas aplicações do sistema e ele é carregado no sistema através do armazenamento de um carregador de aplicações.

Este sistema operacional também implementa os serviços, que são processos e que podem ser substituídos em tempo de execução. Para cada serviço existe uma interface *stub* responsável por redirecionar as chamadas de funções para uma interface de serviço que possui ponteiros para as implementações das funções relativas ao serviço oferecido.

Quando recebe uma mensagem de reconfiguração, o *kernel* envia um evento no qual o serviço se remove automaticamente do sistema. Para transferir o estado do serviço antigo para o atual, o *kernel* disponibiliza uma interface para a passagem de ponteiros entre as duas versões.

RETOS é um sistema operacional que possui quatro objetivos principais : fornecer uma *interface* para programação *multithread*, ter uma abstração orientada à RSSF, ser um sistema resiliente e possuir um *kernel* extensível através de reconfiguração dinâmica (CHA et al., 2007).

Houve uma grande preocupação por parte dos desenvolvedores do SO para conseguir contornar todas as limitações de recursos de forma eficiente e ainda alcançar todos os objetivos principais. No caso do modelo *multithread*, por exemplo, é oferecida uma alta concorrência com preempção e bloqueios de E/S de sistemas subjacentes, ajudando o programador a concentrar-se mais na semântica e no desenvolvimento dos programas ao invés de preocupar-se com a execução ideal dos mesmos.

Para ser um sistema resiliente, o RETOS trabalha com a técnica de operação dual e com a técnica de verificação do código da aplicação. Na primeira o sistema é separado logicamente entre *kernel* e área de execução do usuário. Já na segunda, tanto o código compilado das aplicações quanto seu comportamento em tempo de execução são avaliados. Utilizando as duas abordagens o SO é capaz de detectar tentativas que podem ser prejudiciais ao sistema e permite assim que os nodos executem de forma eficiente e segura.

No caso da reconfiguração dinâmica dos módulos, o SO fornece um mecanismo que utiliza relocação dinâmica de memória e ligação em tempo de execução. Nele são criados, em tempo de compilação, metadados a partir de variáveis e funções e depois armazenados em um arquivo em um formato especial do sistema operacional. Este arquivo posteriormente serve para substituir os endereços utilizados pelo módulo quando o mesmo é carregado no sistema. Também há o armazenamento de uma tabela que dá suporte de acesso entre módulos e aplicações, onde um módulo pode fazer/desfazer registros e também acessar funções a partir desta tabela.

SOS é um sistema operacional baseado em eventos e composto por módulos atualizáveis em tempo de execução. A modularização permite separar o sistema das aplicações, deixando mais fácil manutenção do sistema e possibilitando um melhor desempenho (HAN et al., 2005).

Os módulos são programados de forma cooperativa, de tal modo que conseguem enviar mensagens e se comunicar com o *kernel* através de uma tabela com *jumps*. Então assim que carregados, cada módulo publica-se em duas listas : uma de funções que podem ser utilizadas por outros módulos e uma outra com funções que ele utiliza de outros módulos. Assim, estas funções podem ser armazenadas na tabela junto com as referências dos dados e funções externas.

Na atualização dos módulos o protocolo de distribuição recebe os dados e verifica os metadados no *header* do pacote, feito isto, retira informações importantes para alocar a memória da melhor maneira possível e conseguir receber este novo módulo.

Se durante este processo o SOS descobrir que é incapaz de alocar memória para o novo módulo, sua inserção é imediatamente abortada. Caso contrário, é realizado o download do módulo, são ajustados os ponteiros na tabela e é enviada uma mensagem "init"para que o mó-

dulo seja iniciado.

Quando a atualização consiste em uma remoção de módulo, o *kernel* envia uma mensagem "final"permitindo que o módulo libere todos os recursos que está utilizando e informe sua remoção aos módulos que o utilizam. Após este processo o *kernel* executa a coleta de lixo para liberar a memória e quaisquer recursos pertencentes ao módulo removido.

3 *ELUS - EPOS LIVE UPDATE SYSTEM*

O ELUS é uma infraestrutura de reconfiguração dinâmica que pode ser utilizada em sistemas embarcados com recursos limitados. Esta infraestrutura é construída dentro do *framework* de componentes do EPOS (*Embedded Parallel Operating System*) (FRÖHLICH, 2002) em torno do aspecto de invocação remota, o que permite que um componente seja selecionado como reconfigurável ou não em tempo de compilação. As principais características que a difere dos outros trabalhos relacionados são: configurabilidade, o reduzido consumo de memória, a simplicidade e a transparência para as aplicações e reconfiguração.

- Configurabilidade: componentes podem ou não ser marcados como reconfiguráveis em tempo de compilação, isto permite que não haja um custo adicional para os componentes que não forem marcados como reconfiguráveis.
- Reduzido consumo de memória: comparado com os trabalhos relacionados, o consumo de memória é pequeno. Cerca de 1.6 Kb de código e 26 *bytes* de dados por componente reconfigurável.
- Transparência: A infraestrutura de reconfiguração e a atualização de software são totalmente transparentes para as aplicações.
- Simplicidade: Não há necessidade de registrar a inicialização da interface ou cancelar o registro no término.
- Reconfiguração: Permite que seja atualizado tanto o aplicativo quanto o sistema operacional.

3.1 REQUISITOS PARA UMA RECONFIGURAÇÃO SEGURA

Para que a reconfiguração seja concluída de maneira correta, segura e sem comprometer o sistema, três requisitos principais devem ser satisfeitos (POLAKOVIC; STEFANI, 2008).

- Estado quiescente - É um estado de execução considerado consistente, o qual é alcançado quando nenhuma atividade está sendo realizada no componente atualizado no momento da reconfiguração. No caso do Embedded Parallel Operating System (EPOS), que possui um ambiente multitarefa, o estado quiescente de um componente é atingido quando nenhuma *thread* está invocando métodos deste componente.

- Transferência de estado: A partir do estado quiescente, o estado do componente antigo deve ser transferido para o novo componente. Isto pode ser feito através de métodos `gets` e `sets` ou pela criação de um novo objeto, passando os dados do objeto antigo para o construtor do componente e finalmente deletando o objeto antigo.

- Ajuste de referências: Após a transferência de estados, o sistema deve atualizar as referências para que apontem para o novo objeto, assim a aplicação pode continuar sua execução corretamente.

3.2 VISÃO GERAL DA ARQUITETURA

A arquitetura do ELUS é apresentada na Figura 1. O *framework* de componentes original do EPOS foi estendido para suportar também reconfiguração dinâmica. A invocação de um método de um componente passa pelo `Proxy` que envia uma mensagem ao `Agent`. O valor de retorno depois da execução do método é enviado de volta ao cliente através de uma mensagem. Um nível de indireção é criado entre o cliente e o método real, tornando o `Agent` o único membro da estrutura ciente da posição do componente na memória do sistema. A OS BOX no `Agent` controla o acesso aos métodos do componente através de um semáforo para cada componente, chamando os métodos do `Agent` através de uma tabela. O `Agent` quando recebe uma invocação de método, envia o pedido para o `Adapter` que chama o método real do componente através da tabela de métodos virtuais (`vtable`) do componente.

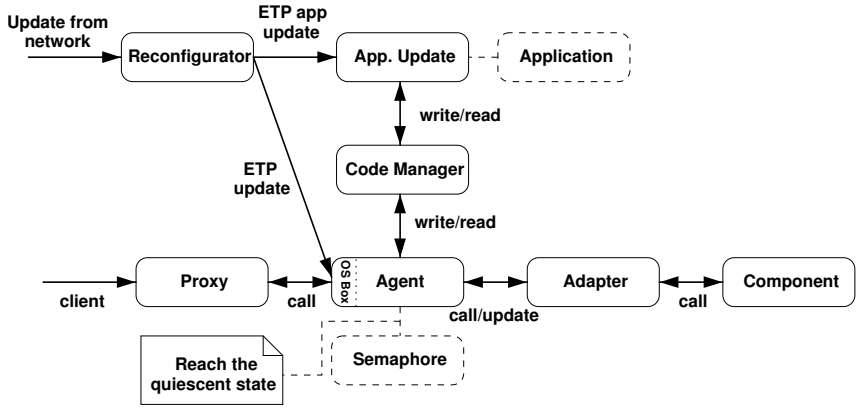

Figura 1: Visão geral da arquitetura do ELUS (GRACIOLI; FRÖHLICH, 2010).

O ELUS Transport Protocol (ETP) é um protocolo para o recebimento de mensagens de reconfiguração e toda atualização ocorre a partir de uma requisição neste formato. O *framework* permite a atualização de componentes, da aplicação e do próprio *framework* de reconfiguração, sendo que o protocolo consegue diferenciar os tipos de atualização através de um código, o `control_byte`.

3.3 ELEMENTOS DO *FRAMEWORK*

Nesta sessão serão apresentados os elementos do *framework* que são utilizados no processo de reconfiguração que a infraestrutura ELUS provê. Também serão apresentados outros componentes que não fazem parte do framework, mas que são fundamentais para o bom funcionamento do processo. A interação entre os componentes no *framework* metaprogramado podem ser observada na Figura 2

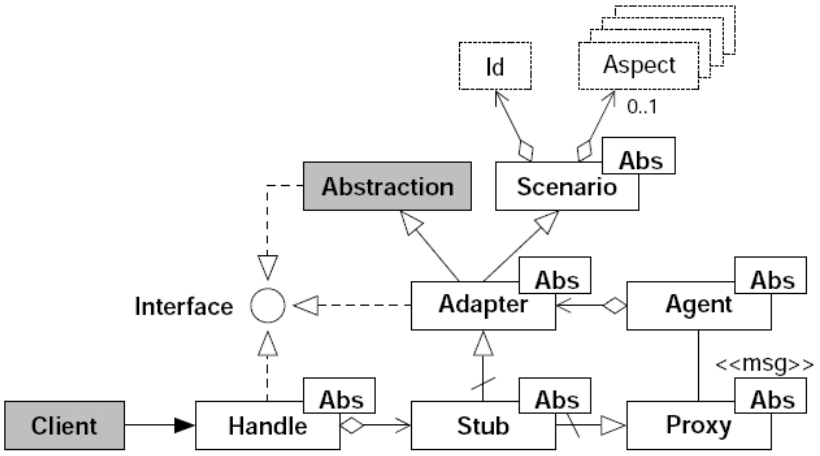

Figura 2: `Framework` metaporgramado.(GRACIOLI; FRÖHLICH, 2009)

Os elementos compõem o `framework` e são utilizados no processo de reconfiguração que a infraestrutura provê são: `Handle, Stub, Proxy, Agent, Scenario e ScenarioHash`. Segue abaixo uma pequena descrição das suas respectivas funções.

- `Handle` - Quando o suporte ao *framework* e à reconfiguração dinâmica estão habilitados, o componente é exportado como um parâmetro para o `Handle`, que possui a mesma interface (operações) que o componente, mas com a invocação dos métodos encaminhadas para o `Stub`. O `Handle` é responsável por verificar se um componente está ou não selecionado como reconfigurável.

- `Stub` - Responsável por criar um `Proxy` ou um `Adapter` conforme os parâmetros recebidos na sua criação via Handle. Então se a reconfiguração do componente está habilitada,

Handle irá criar um Stub que contém todos os métodos do Proxy, permitindo assim a comunicação do Proxy com o Agent, surgindo aqui um nível de indireção entre as chamadas de métodos da aplicação e os componentes reais do sistema.

- Proxy - Responsável por efetuar os métodos do componente. Quando recebe um pedido do Handle, ele utiliza a classe Message para passar o ID do componente e a solicitação desejada para o Agent.

- Agent - Possui duas funções : a invocação de métodos e a reconfiguração dos componentes e do próprio *framework*. Caso seja uma invocação de métodos, a Message será enviada pelo Proxy então ele chama o Adapter e reconfigura o componente através do método do update. Senão a Message foi enviada pela *thread* Reconfigurator, e o Agent deve fazer a reconfiguração do próprio *framework*.

- Scenario - responsável por aplicar os aspectos selecionados para cada componente recebido pelo metaprograma. Também armazena os objetos criados pelo Agent em tempo de execução.

- ScenarioHash - Funciona como uma abstração das operações de controle, armazenamento, busca e recuperação de objetos através de uma tabela *hash*.

Os elementos auxiliares são: Message, Adapter, Application_Update, Reconfigurator e Code_Manager.

- Message - Ponto comum entre o Proxy e o Agent. Oferece uma interface para anexar e recuprar informações úteis para a reconfiguração, como por exemplo tamanho do códito, tipo de reconfiguração, retorno de um método, entre outros.

- Adapter - Funciona como um *wrapper* e realiza a adaptação do componente que foi recebido como parâmetro através dos métodos leave e enter do Scenario antes e depois da invocação do método real.

- ApplicationUpdate - É um componente criado exclusivamente para dar suporte à atualização da aplicação. Sua tarefa principal é desabilitar interrupções do sistema para conseguir atingir o estado quiescente.

- Reconfigurator - É uma thread criada na inicialização do sistema e é responsável por receber o pedido de reconfiguração, construir Message e enviar um pedido de reconfiguração para o Agent

- `Code_Manager` - Responsável por utilizar a estrutura fornecida pelo mediador de *hardware* para gerenciar alocação e liberação de espaços na memória.

Como a reconfiguração será feita em *software* de sistemas embarcados com severas limitações, vale lembrar que o sistema de reconfiguração deve utilizar o mínimo de recursos possível, pois eles serão compartilhados com as aplicações.

3.4 PROCESSOS DE RECONFIGURAÇÃO NO ELUS

O ELUS permite três diferentes tipos de reconfiguração : a de componentes, da aplicação e do próprio *framework* de reconfiguração. Cada uma possui suas próprias características, alvo e fluxo de tarefas.

3.4.1 RECONFIGURAÇÃO DE COMPONENTES

A atualização de componentes é solicitada quando se deseja modificar um componente que foi previamente marcado como reconfigurável. Conforme pode ser visto na Figura 3, o processo começa quando são enviados os dados do novo código e também um pedido de atualização para o `Reconfigurator`, que constrói uma mensagem com o código recebido, dados do componente e informações sobre a atualização solicitada. Esta mensagem é enviada para o `Agent`, que através do `Semaphore` consegue alcançar o estado quiescente do componente que será atualizado. O componente correto é recuperado pelo `Adapter` e repassado para o `Agent`, que faz duas operações: utiliza o `Code_Manager` para alocar um espaço para o novo código caso seja maior do que o antigo e atualiza a tabela de métodos virtuais. A última tarefa a ser realizada antes de liberar o `Semaphore` e retornar o fluxo para o `Reconfigurator`, é verificar a necessidade de atualização do componente no *framework*.

3.4.2 RECONFIGURAÇÃO DO *FRAMEWORK*

Para a atualização do *framework* é necessário um pouco mais de cautela, pois conforme pode ser visto na Figura 3, o `Agent` - mais especificamente o método `trapAgent` - é o ponto de entrada para o serviço de reconfiguração e o seu endereço deve ser conhecido previamente para que o novo código do componente possa ser compilado corretamente. Então para ter certeza de que o endereço do `trapAgent` não será modificado ele não é uma função passível de reconfiguração.

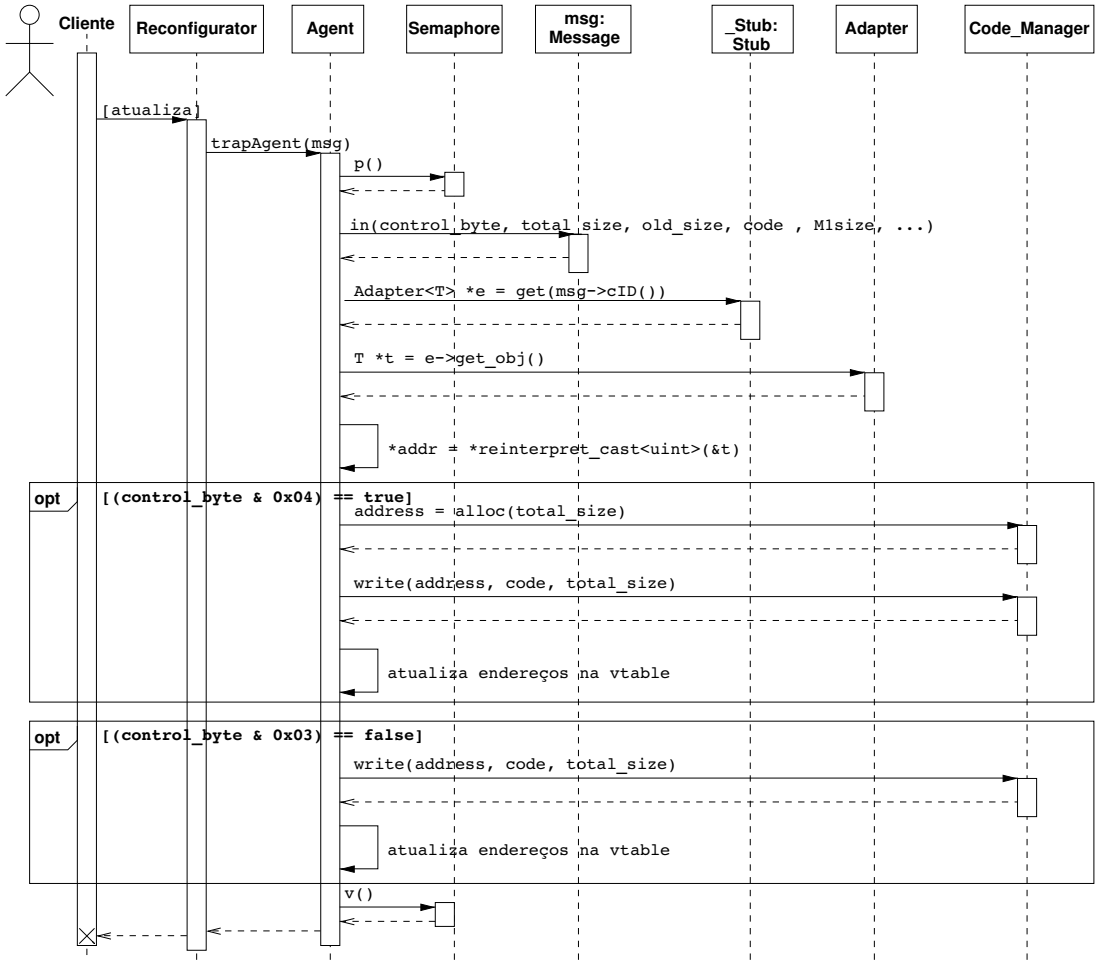

Figura 3: Resumo da sequência de atividades realizadas pelo ELUS em uma reconfiguração de componentes(GRACIOLI; FRÖHLICH, 2009).
1

Para ocorrer a atualização do componente no *framework* é necessário copiar o código recebido e verificar se o tamanho deste novo código é maior que o atual. Se o novo código for menor ou igual ao antigo, ele é simplesmente recebido, copiado para a posição correta e os seus endereços são atualizados no `Dispatcher`. Caso contrário é necessário alocar um novo espaço na memória, atualizar os endereços no `Dispatcher` e liberar a memória antiga.

O `Dispatcher` é um vetor declarado estaticamente que contém os ponteiros para todos os métodos reconfiguráveis do componente dentro do *framework*. Portanto, para adicionar um novo método do componente é necessário verificar se ainda existe alguma posição livre no `Dispatcher`. Se houver, o `Agent` aloca memória para o novo método e insere o endereço no `Dispatcher`. Caso contrário, o vetor precisa ser realocado para uma área maior antes de incluir o novo método do componente. Já a remoção de um método implica na liberação da memória e a atualização dos ponteiros.

3.4.3 RECONFIGURAÇÃO DA APLICAÇÃO

Esta reconfiguração também se inicia com o pedido de atualização para o `Reconfigurator`, que envia uma mensagem para o `ApplicationUpdate`, assim ele desabilita as interrupções para atingir o estado quiescente. O tamanho do código recebido é comparado com o código antigo para saber que tipo de atualização deve ser realizada. Se o novo código não for maior do que o atual ocorre uma simples atualização do endereço, caso contrário, é necessário alocar um novo espaço de memória antes de atualizar o endereço. Após a atualização da aplicação o sistema deve ser reiniciado para que suas novas atividades possam ser realizadas. A Figura 4 mostra o diagrama de sequência para este processo.

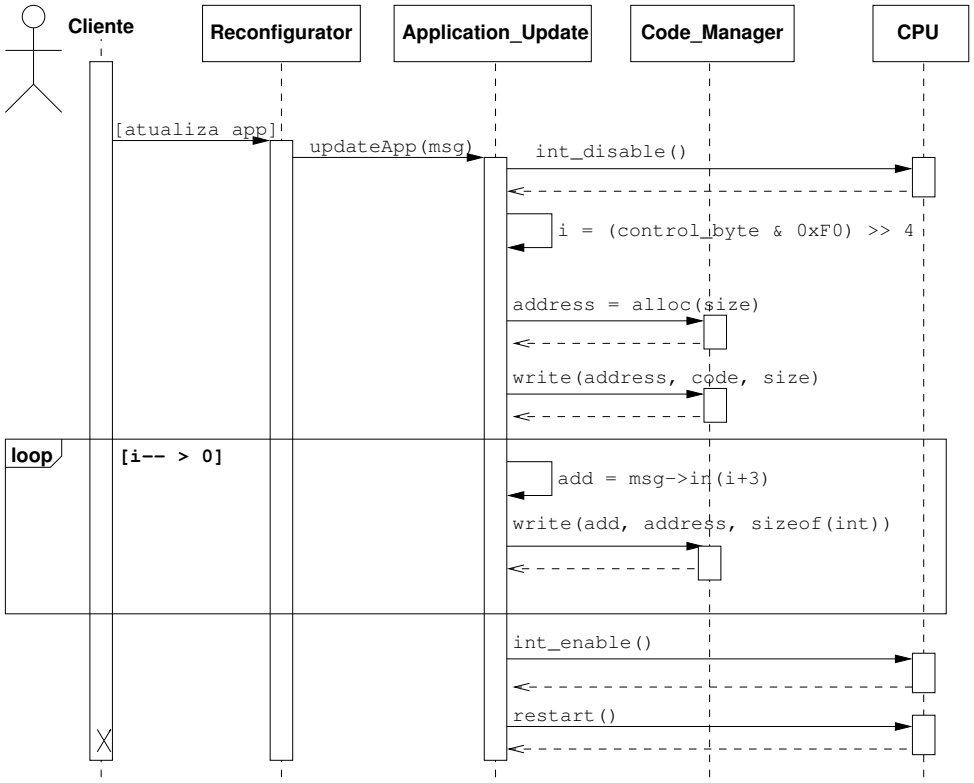

Figura 4: Diagrama de sequência de atividades realizadas pelo ELUS em uma reconfiguração da aplicação(GRACIOLI; FRÖHLICH, 2009).

3.5 LIMITAÇÕES DA INFRAESTRUTURA

3.5.1 CONSUMO EXTRA DE MEMÓRIA

Devido à estrutura do ELUS, cada componente selecionado como configurável gera um sobrecusto em termos de memória, porque além da implementação real do método pelo próprio componente, ainda se faz necessário incluir os métodos do componente no *framework*.

A configuração de consumo mínimo de memória para adicionar o componente no *framework* seria contendo apenas os métodos criar, destruir, atualizar e um método sem parâmetros e sem retorno. Utilizando a Equação 3.1 e a Tabela 1, percebe-se que mesmo com a configuração mínima haveria o sobrecusto de cerca de 1.6Kb de código e 26 bytes de dados.

$$Tamanho_{componente} = \sum_{i=1}^{n}(Método_i) + Create + Destroy + Update \quad (3.1)$$

Tabela 1: Consumo de memória na adição de métodos de um componente genérico no *framework* do ELUS (GRACIOLI; FRÖHLICH, 2009)

Métodos do Framework	**Tamanho da Seção(bytes)**	
	.text	**.data**
Create	180	0
Destroy	138	0
Método sem parâmetro e sem retorno	94	0
Método com parâmetro	98	0
Método com valor de retorno	112	0
Método com parâmetro e com retorno	126	0
Update	1250	0
Dispatcher	0	2X (métodos)
Semaphore	0	18
Total Mínimo	**1662**	**26**

Apesar de ser um consumo bem alto para os limitados recursos dos sistemas embarcados, se comparado a outros trabalhos o custo imposto pelo *framework* ainda é menor. Uma possível alternativa seria estender o *framework* utilizando alguma técnica para reduzir/eliminar o código replicado.

3.5.2 TEMPO DE INVOCAÇÃO DOS MÉTODOS

Outro ponto que pode ser aprimorado é o tempo de invocação de métodos. Atualmente o framework cria uma indireção entre a aplicação e a invocação do método real. Devido a este fato, ele é responsável pelo armazenamento dos objetos reconfiguráveis criados pela aplicação, pelo controle de acesso aos métodos e pela passagem de parâmetros e valor de retorno através da estrutura de mensagem.

Todas as atividades que são realizadas para que um método seja invocado causam uma perda de desempenho e, consequentemente, um aumento no tempo de invocação. Os testes de

performance da invocação dos métodos foram realizados na plataforma Mica2 [2]. Um comparativo de desempenho entre a invocação de métodos normais, a invocação através da Tabela de Métodos Virtuais (vtable) e através do ELUS apresentou o resultado apontado pela Tabela 2.

Tabela 2: Comparação do desempenho da invocação de métodos normal, utilizando a vtable e através do ELUS.(GRACIOLI; FRÖHLICH, 2010)

Tipo do Método	**Tipo de Invocação**		
	Normal	**vtable**	**ELUS**
Sem parâmetro e sem retorno	4 ciclos	9 ciclos	112 ciclos
Sem parâmetro e com retorno	5 ciclos	10 ciclos	132 ciclos
Com parâmetro e sem retorno	8 ciclos	12 ciclos	123 ciclos
Com parâmetro e com retorno	17 ciclos	22 ciclos	143 ciclos

Para reduzir o tempo de invocação dos métodos, a proposta é tentar retirar os níveis de indireção criados pelo *framework*. Como o estado quiescente é um requisito para o bom funcionamento do processo de reconfiguração serão examinadas outras alternativas para se encontrar o estado quiescente. Também será analisada alguma forma de não precisar armazenar o objeto ou substituir o modelo de armazenamento atual (tabela *hash*).

[2] A plataforma Mica2 possui um microcontrolador Atmel Atmega128 de 8Mhz, 4KB de memória RAM, 128 KB de memória flash, 4KB de EEPROM. Também conta com um conjunto de periféricos, incluindo comunicação via rádio.

4 PROPOSTAS DE MELHORIA NO ELUS

No capítulo anterior foi apresentado o *framework* ELUS, bem como sua arquitetura, processo de reconfiguração e limitações da infraestrutura. Com base neste estudo foi possível gerar duas propostas para a melhoria de desempenho. A primeira se refere a uma visão global do processo de reconfiguração e pretende, através de uma modelagem híbrida, incorporar as otimizações realizadas por outros autores para uma melhora global do processo. A segunda proposta pretende examinar o processo de reconfiguração sob o ponto de vista de cada uma das limitações a fim de melhorar o desempenho individual de cada parte do processo.

Neste capítulo serão apresentadas as propostas deste trabalho, que pretendem melhorar o desempenho do ELUS quanto ao consumo de memória, o tempo de invocação de métodos e o tempo da reconfiguração.

4.1 MODELAGEM HÍBRIDA DE THINK E ELUS

A proposta para melhoria global visa incorporar otimizações do modelo FRACTAL (COUPAYE; STEFANI, 2006) de componentes hierárquicos para melhorar o processo de reconfiguração como um todo.

4.1.1 FRACTAL

FRACTAL é um modelo de componentes extensível e hierárquico. Ele foi desenvolvido para ser um modelo independente de linguagem que conseguisse reduzir o tempo gasto com desenvolvimento, implantação e manutenção de sistemas complexos. Para tanto, a sua estrutura deve fornecer características como reflexividade, recursividade, compartilhamento de componentes, ligação inter componentes entre outras.

Conforme pode ser visto no diagrama da Figura 5, a comunicação entre os componente se dá através de ligações (*bindings*) entre as interfaces cliente/servidor, ou seja, cada componente que deseja se comunicar com outro deve possuir pelo menos uma interface com todos os serviços

fornecidos (`IProvided`) e requeridos (`IRequired`).

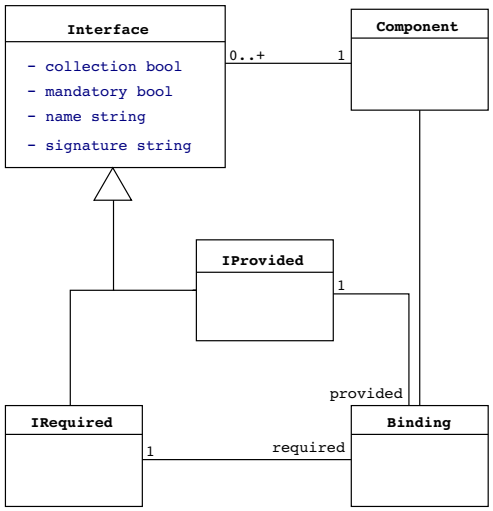

Figura 5: Diagrama de classes - *Binding*

Também há uma divisão lógica dos componentes entre membrana e conteúdo que pode ser observada na Figura 6. A membrana é considerada como proprietária do conteúdo e implementa as interfaces de controle para guiar o comportamento do mesmo. No conteúdo encontram-se os subcomponentes além da implementação das funcionalidades oferecidas pela interface servidor.

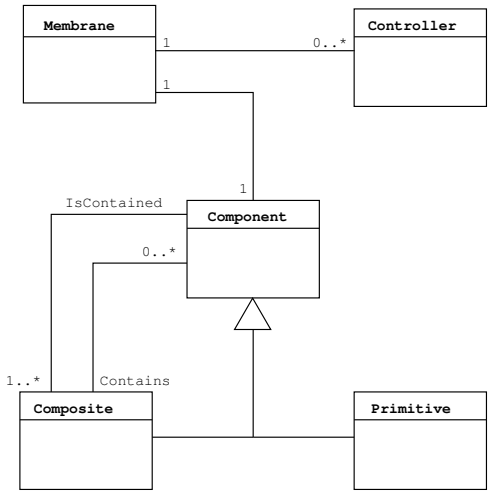

Figura 6: Diagrama de classes - Componente

Abaixo estão algumas vantagens e desvantagens do modelo:

- Vantagens:
  - Maior facilidade para implementar e manter sistemas complexos : Para facilitar a implementação e a manutenção de sistemas com um grande número de componentes, o modelo fornece políticas de boas práticas (Ex. separação lógica entre aspectos

funcionais e não-funcionais), ferramentas de monitoramento, modelos de reconfiguração dinâmica, entre outros.

  - Extensibilidade: É possível adicionar ou remover componentes da arquitetura do sistema em tempo de execução.
  - Adaptabilidade: Um componente pode possuir mais de um controle de interface, podendo adaptar-se conforme o estímulo recebido.
  - Componentes compostos: Fornece uma visão homogênea de todas as camadas de implementação, independente do nível de abstração em que ele se encontra.
  - Compartilhamento de Componentes: A reutilização de uma instância de um componente e a capacidade de compartilhar recursos reduzem a quantidade de memória utilizada pelo software.

- Desvantagens :

  - Não é garantida a compatibilidade de componentes: Alguns componentes podem não conseguir trabalhar junto pois a compatibilidade depende de muitas variáveis, como o tipo de opções selecionadas, extensão do modelo utilizado, nível de exigência aplicado, entre outras.
  - Implementação do modelo: Para implementar um software utilizando este modelo pode ser necessário também criar uma abstração da arquitetura alvo, o que aumenta a quantidade de código que deve ser gerado.
  - Compartilhamento de Componentes: O compartilhamento de recursos pode quebrar o encapsulamento do conceito de caixa-preta do componente.

4.1.2 THINK

THINK é uma implementação do modelo FRACTAL voltada para a construção de sistemas operacionais para plataformas embarcadas. Nos SOs criados por este *framework* os componentes são chamados módulos, porém as ligações - *bindings* - são também um conjunto de módulos especiais que implementam o canal de comunicação entre um ou mais módulos.

Para auxiliar a construção de sistemas operacionais foi criada a biblioteca KORTEX que possui implementação padrão para os componentes de abstração de hardware (responsáveis por incluir exceções, interrupções, *drivers*, entre outros), para os componentes de serviços do sistema operacional (ex. gerenciamento de memória, de sistema de arquivos, comunicação remota, suporte a *threads*) e também com diferentes formas de *bindings* (incluindo *syscall, up-calls*, RPC e IPC).

No modelo FRACTAL todas as funcionalidades e definições são opcionais mas, a título de exemplo, a Figura 7 apresenta uma modelagem para um melhor entendimento do modelo. Ela possui algumas definições que serão utilizadas nos próximos capítulos.

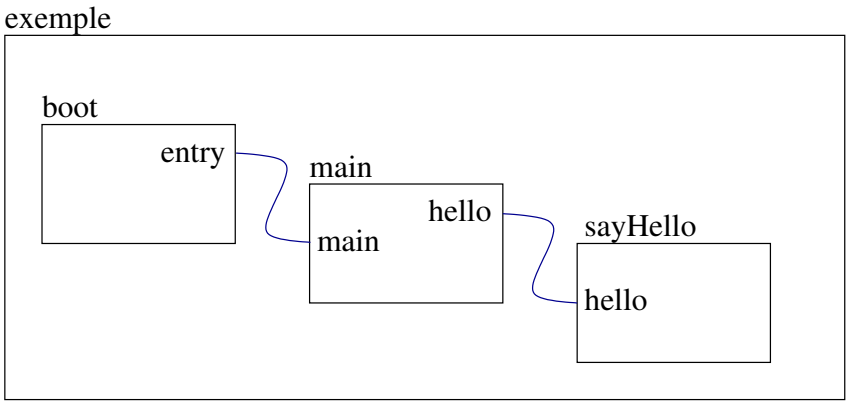

Figura 7: Modelagem do componente `example`

A modelagem do componente `example` mostra três subcomponentes : o `bootloader`, o `main` e o `sayHello`. O `bootloader` é um componente da biblioteca KORTEX e serve como uma ponte entre o *hardware* e o *software*. Sua interface fornece o serviço `entry` que instancia o componente que requisita este serviço. O componente `main` além de consumir o serviço do `bootloader` possui uma interface cliente que está ligada ao componente `sayHello`. Por fim, o `sayHello` possui uma interface servidor para fornecer seu método.

Cada componente deve possuir um arquivo que descreve a arquitetura e um outro que descreve as interfaces. O código presente nos Algoritmos 4.1, 4.2 e 4.3 mostram os arquivos *Architecture Description Language* (ADL) que descrevem, respectivamente, os componentes `example`, `main` e `sayHello`. Pode-se perceber que neles são descritos os componentes em termos de membrana, conteúdo, ligações e subcomponentes.

```
1  component example {
2      contains boot = avr.atmega128.boot.lib.bootloader
3      contains main = main
4      contains sayHello = sayHello
5
6      binds boot.entry to main.main
7      binds main.hello to sayHello.hello
8  }
```

Algoritmo 4.1: Arquivo ADL do componente `example`

```
1  component main {
```

```
2       provides Main as main
3       requires sayHello as hello
4       content main
5   }
```

Algoritmo 4.2: Arquivo ADL do componente `main`

```
1   component sayHello {
2       provides sayHello as hello
3       content sayHello
4   }
```

Algoritmo 4.3: Arquivo ADL do componente `sayHello`

Os arquivos *Interface Description Language* (IDL) descrevem e indicam todos os métodos fornecidos e requeridos por cada componente. Os Algoritmos 4.4, 4.5 e 4.6 mostram o arquivo IDL dos componentes `example`, `main` e `sayHello`.

```
1   public interface example {
2       void MAIN_main(int argc, string argv[]);
3   }
```

Algoritmo 4.4: Arquivo IDL do componente `example`

```
1   public interface main {
2       void MAIN_main(int argc, char** argv);
3       void HELLO_printHelloThink();
4   }
```

Algoritmo 4.5: Arquivo IDL do componente `main`

```
1   public interface MultiLanguageHello {
2       void printHelloWorld(void);
3   }
```

Algoritmo 4.6: Arquivo IDL do componente `sayHello`

Por fim, os Algoritmos 4.7 e 4.8 apresentam o conteúdo dos arquivos que contém a implementação dos componentes. Junto com a implementação são colocadas, como comentário em bloco, algumas anotações que são utilizadas posteriormente pelo compilador THINK. Note

que não há implementação do componente `example`, pois ele será ligado ao `boot` por um outro componente do *framework*, o `LifeCycleController`, que não será abordado neste exemplo.

```
1  /*
2   * @@ ClientInterfacePrefix(hello, HELLO_) @@
3   * @@ ServerInterfacePrefix(main, MAIN_) @@
4   */
5
6  int MAIN_main(int argc, char** argv) {
7      int i = 10;
8      while (i>0){
9          HELLO_printHelloWorld();
10         i--;
11     }
12 }
```

Algoritmo 4.7: Arquivo de implementação do componente `main`

```
1  /*
2   * @@ ServerInterfacePrefix(hello, HELLO_) @@
3   */
4
5  void HELLO_printHelloWorld(){
6    printf ("Hello from THINK! \n");
7  }
```

Algoritmo 4.8: Arquivo de implementação do componente `sayHello`

Apesar de existirem vários níveis de exigências dentro do modelo FRACTAL, para se criar um *software* que permita reconfiguração dinâmica com THINK precisamos, a partir de um modelo já definido, gerar alguns arquivos e inserir anotações especiais. As principais etapas são:

- Gerar arquivos ADL que representam os atributos, as composições, as interfaces, a ligação entre os componentes e outros aspectos relacionados à arquitetura;
- Gerar arquivos IDL descrevendo a interface de cada componente e habilitando a colaboração entre eles.
- Definir interfaces de controle para os componentes;
- Definir controles de ciclo de vida dos componentes;

- Definir tipos de *bindings* que vão conectar um componente a outro;

Em THINK a reconfiguração dinâmica consiste em modificar o sistema em tempo de execução e é realizada através dos seguintes passos: identificar qual parte do sistema deve ser reconfigurada, suspender sua execução, modificar seu estado (incluir/excluir/modificar componentes), transferir o estado do componente antigo para o novo e prosseguir a execução.

Para substituir um componente com segurança e sem interromper o serviço, existe uma série de mecanismos que são implementados como interfaces de controle e monitoram os componentes. Além de fornecer mais segurança e uma boa política de encapsulamento, estes mecanismos provêm uma flexibilidade muito grande no processo de reconfiguração, pois cada componente pode possuir mais de uma interface de controle e consequentemente atualizar-se de várias maneiras.

É importante ressaltar que os arquivos de descrição de arquitetura são indispensáveis para este processo, pois neles que estão atribuídas as propriedades necessárias para que o compilador de THINK possa adicionar os controles, e assim modificar o *software*. Pode-se otimizar o processo definindo explicitamente o componente como reconfigurável através de seu ADL.

4.1.3 CENÁRIO DA MODELAGEM GLOBAL

A maioria dos modelos e metodologias apresentados no Capítulo 2 proporcionam um bom suporte à reconfiguração dinâmica, porém, pecam quanto à utilização dos recursos limitados dos sistemas embarcados. O *framework* ELUS foi comparado com os trabalhos relacionados e possui resultados promissores, mas que ainda podem ser melhorados. Por outro lado, o modelo FRACTAL possui características desejáveis a um bom mecanismo de reconfiguração dinâmica de software em sistemas embarcados, mas as suas implementações - em especial o THINK - ainda não são abrangentes o suficiente. Neste cenário será realizada uma proposta de modelagem do ELUS utilizando o modelo FRACTAL de componentes.

Antes de propor a modelagem é necessário relembrar as propostas de melhoria no ELUS, realizadas no Capítulo 3, e entender como o Modelo FRACTAL pode ajudar no processo.

Pode-se citar como uma restrição da infraestrutura o consumo extra de memória, que ocorre porque cada componente selecionado como configurável precisa ter seu código replicado no *framework*, gerando um sobrecusto para o recurso memória. Este problema poderia ser resolvido com a implementação THINK do modelo FRACTAL, que fornece uma estrutura de reconfiguração em que a maior parte do processamento é realizada em um servidor e na memória de programa do dispositivo acrescenta-se apenas um componente de reconfiguração e os novos

componentes.

A reconfiguração com THINK ocorre a partir do algoritmo de troca de referências das *bindings* que retira informações dos componentes que estão na memória *flash*. Se substituirmos o *framework* do ELUS pela estrutura de THINK a replicação de código irá diminuir, e assim podemos diminuir o consumo de memória.

Outra limitação do ELUS é o tempo de invocação dos métodos, pois ocorre uma indireção entre a aplicação e a invocação do método real, atribuindo ao *framework* uma série de responsabilidades que acabam aumentando o tempo de invocação. Dentre as atribuições do *framework* estão o armazenamento e controle sobre os objetos reconfiguráveis criados pela aplicação, incluindo alcançar o estado quiescente do componente.

A estrutura THINK utiliza um mecanismo diferente de reconfiguração, através das interfaces de controle implementadas pela membrana, assim cada membrana fica responsável pelo controle do conteúdo do seu componente, deixando para o *framework* apenas a tarefa de armazenamento. Com esta abordagem torna-se possível reduzir a responsabilidade do *framework* e consequentemente diminuir o tempo gasto para invocar um método.

O modelo FRACTAL possui uma série de vantagens, mas não podem ser deixadas de lado as suas deficiências. Substituir a infraestrutura toda por uma implementação de FRACTAL não é a solução ideal para alcançar todas as melhorias de performance. Mesmo THINK que é uma implementação de FRACTAL voltada para construção de sistemas operacionais também possui as suas limitações que, convenientemente, podem ser remediadas através dos componentes do ELUS. Exemplos dessa interação são as estruturas auxiliares que fornecem suporte à reconfiguração como, por exemplo, *Code_Manager, Application_Update, Scenario* entre outros). Uma outra vantagem de se utilizar toda a estrutura pela qual o ELUS é envolvido é não precisar implementar nenhuma abstração da arquitetura alvo, já que o EPOS fornece toda a estrutura de mediadores de hardware.

4.1.4 O MODELO HÍBRIDO

A primeira modelagem proposta para este trabalho é uma abordagem híbrida do modelo FRACTAL e a infraestrutura ELUS. A ideia principal é pegar o ponto forte de cada um deles para que um complete o outro. Para tanto, a proposta pretende utilizar, do THINK, a parte da introspeção inerente, as interfaces de controle (`BindingController`,`ComponentIdentity`, `ContentController`, etc), o processo de reconfiguração, os interceptadores e o escalonador de eventos. Já as contribuições do ELUS serão o processo de gerenciar alocação e liberação de espaços na memória, o método `update` e o esquema de troca de mensagens. Será utilizado

como base o sistema operacional EPOS, junto com o seu conjunto de abstrações. A Figura 8 mostra a hierarquia dos componentes no modelo proposto.

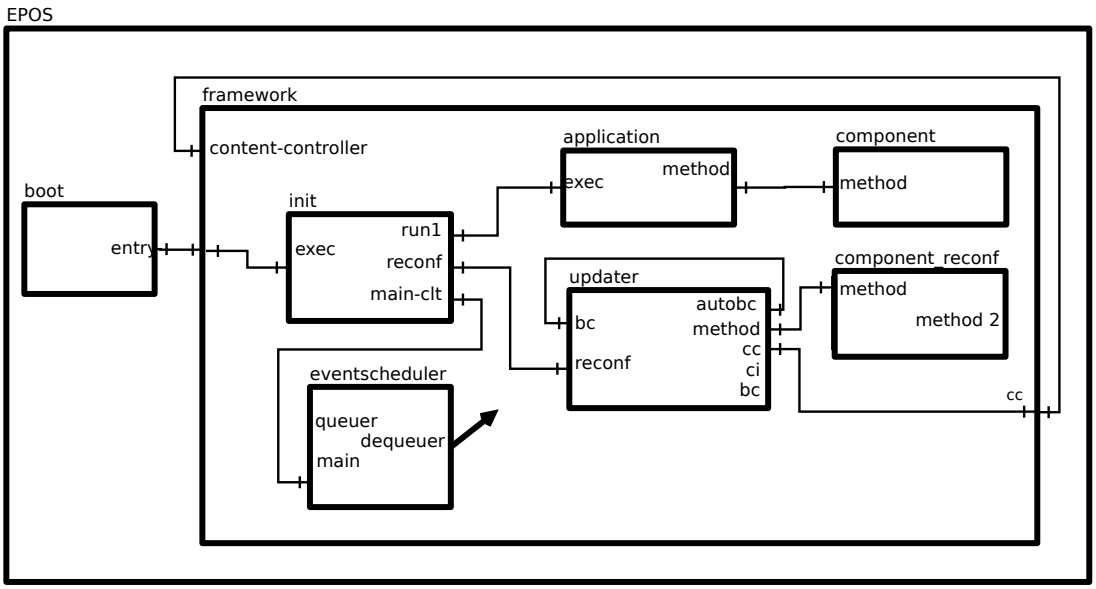

Figura 8: Modelagem da proposta híbrida do modelo FRACTAL e da infraestrutura ELUS para a reconfiguração dinâmica.

O processo de reconfiguração da modelagem proposta ocorre em duas partes, a primeira é realizada em um computador (servidor) com maior capacidade de operação. Nele são desenvolvidos o componente que deseja atualizar, bem como um código de reconfiguração `updater` para este novo componente. O `updater` fornece o serviço `SRV_reconf_execute()` para ligar o componente no código que já está rodando na plataforma alvo. Para isso, ele conta com as seguintes estruturas de controle:

- `ContentController` - É uma interface que permite controlar subcomponentes, com ela podemos pesquisar a interface `ComponentIdentity` de cada um dos subcomponentes, assim, é possível utilizar os serviços de `BindingController` para fazer a ligação inter-componentes.
- `ComponentIdentity` - Interface que permite achar as interfaces cliente, servidor e de controle de cada componente. Ela pode também permitir que um componente use uma interface implementada no mesmo componente, mas com um conteúdo (*content*) diferente.
- `BindingController` - Também é uma interface de controle, utilizada para fazer a ligação (*bindings*) entre os componentes para que a comunicação cliente/servidor tenha efeito.

- `Interceptors` - São estruturas criadas em tempo de compilação com o intuito de formar pontes entre a aplicação e o componente que será reconfigurado, rebatendo qualquer tipo de requisição feita ao componente.

A sequência básica que deve ser seguida para que a reconfiguração seja feita é simples e deve seguir os seguintes passos:

1. A partir do `ContentController` do componente pai, buscar a `ComponentIdentity` do componente que vai substituir o antigo componente no código em execução.
2. Pelo `ComponentIdentity`, encontrar a interface servidor do componente que será substituído.
3. Fazer para cada *binding* do antigo componente :
   (a) Partindo do `ContentController` do componente pai, buscar a interface servidor dos `Interceptors` usando o `ComponentIdentity`.
   (b) Também através do `ComponentIdentity`, encontrar a interface servidora `BindingController`.
   (c) Utilizando o `BindingController`, ligar a interface cliente que estava no antigo componente na interface servidor equivalente do novo componente.

No caso do novo componente possuir interfaces cliente, o processo deve ser repetido para atualizar as referências destas interfaces também.

A segunda parte do processo é realizada na plataforma alvo. Depois de produzido tanto o código do `updater` quando o do componente, a nova imagem do sistema é gerada e enviada para a plataforma alvo utilizando-se *scripts*. Na inicialização do sistema, o `init` executa a aplicação e verifica se existe alguma reconfiguração a caminho. Se houver, o serviço cliente `reconf` do `init` (equivalente ao Code_Manager deste modelo) recebe todos os dados da reconfiguração, escreve-os na memória *flash* e fica na fila de um escalonador de eventos.

Quando for oportuno, o `updater` é retirado da fila e o seu serviço `SRV_reconf_execute()` é acionado. Antes de iniciar a reconfiguração é necessário desabilitar as interrupções e utilizar os `Interceptors`, impedindo o acesso ao componente. Após executar o processo de reconfiguração é muito importante não esquecer de habilitar novamente as interrupções e liberar o espaço de memória em que estavam o `updater` e o componente que foi substituido.

4.2 ESPECIALIZAÇÃO DE TEMPLATES PARA PONTEIROS VOID

A proposta de melhoria individual se baseia em verificar quais partes do processo podem ser modificadas para minimizar cada uma das limitações.

Analisando o código do *framework* da infraestrutura ELUS foi possível observar uma estrutura metaprogramada utilizando *templates*. Segundo Stroustrup (STROUSTRUP, 1997) o comportamento padrão da maioria das implementações de *templates* utilizando C++ é replicar o código para cada função recebida como argumento e uma boa alternativa para evitar este tipo de replicação é extrair os techos comuns à todas as funções *template* para uma especialização para ponteiros *void*.

4.2.1 ESPECIALIZAÇÕES DE *TEMPLATES* EM C++

Mecanismos de *templates* providos pelo C++ são uma maneira de adicionar à classes e funções conceitos genéricos, isto é, permitem criar um código reutilizável onde os tipos sejam passados como parâmetro.

Por padrão, as classes *template* possuem uma única implementação para qualquer argumento recebido e quando deseja-se dar um tratamento mais refinado à um determinado tipo de argumento, é necessário realizar descrições alternativas do *template*. Estas descrições, também conhecidas como especializações, são escolhidas pelo compilador com base nos argumentos fornecidos para o `template`.

O ponteiro *void* é um ponteiro que pode apontar para qualquer tipo de objeto e pode ser explicitamente convertido para qualquer outro tipo. Sendo assim, se for realizada uma especialização de *template* para ponteiros *void*, ela poderia implementar as funções comuns a todos os tipos de argumentos sem que haja replicação de código.

Como exemplo desta ideia são apresentados o algoritmo 4.9 (classe `Vetor` genérica) e o algoritmo 4.10 (utilizando uma base para os demais argumentos).

```
1  template <class T> class Vector{
2      T* v;
3      int sz;
4  public:
5      Vector();
6      Vector(int);
7
```

```
8       T& elem (int i) {return v[i]; }
9       T& operator [] (int i);
10
11      void swap(Vector &);
12
13      // ...
14  };
```

Algoritmo 4.9: Classe `Vector` generalizada (STROUSTRUP, 1997).

```
1   template <class T> class Vector <T*> : private Vector <void*>{
2   public:
3       typedef Vector <void*> Base;
4
5       Vector() : Base() {}
6       explicit Vector(int) : Base(i) {}
7
8       T& elem (int i) {return static_cast<T*&>(Base::elem(i)); }
9       T& operator [] (int i) {return static_cast<T*&>(Base::operator[](i)
            ); }
10
11      // ...
12  };
```

Algoritmo 4.10: Classe `Vector` com base especializada para para ponteiros *void* (STROUSTRUP, 1997).

Podemos observar que no algoritmo 4.10 não há mais a implementação de alguns métodos dentro da classe genérica de `Vector` e, consequentemente, ao invés de uma replicação haverá apenas uma chamada de método.

4.2.2 CENÁRIO DA MODELAGEM INDIVIDUAL

Existem vários trechos do *framework* que podem ser modificados de acordo com a proposta de especializações de *template* para ponteiros *void*. O lugar em que há mais replicação de código é o método `update` da classe `Agent`. A Figura 9 mostra um trecho do método `update` no qual podemos observar que em grande parte do código não há nenhuma referência ao *template*.

Considerando as modificações que devem ser realizadas no modelo atual do ELUS, a figura 10 apresenta o novo diagrama de classes, que agora inclui as especializações para ponteiros *void* das classes `Agent` e `Scenario`.

Figura 9: MTrecho do método `update` da classe `Agent`.

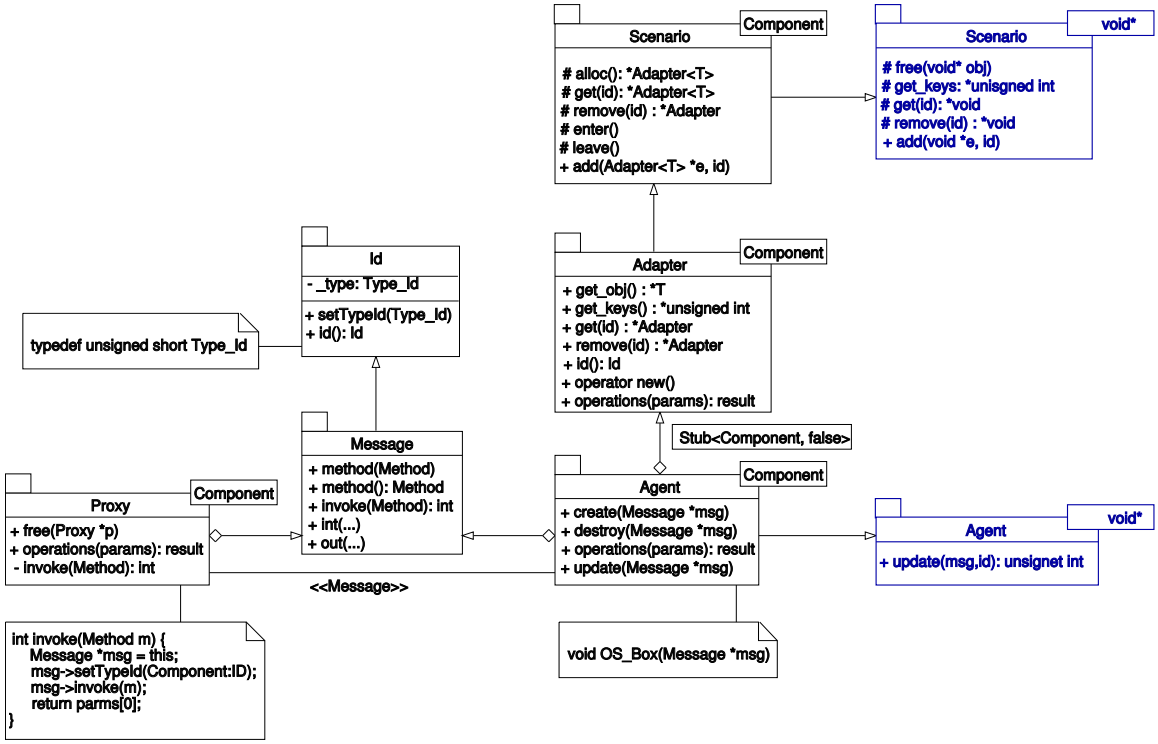

Figura 10: Diagrama da classes envolvidas no processo de reconfiguração de componentes.

Tanto o `Agent<void*>` quanto o `Scenario<void*>` são especializações utilizadas como base para diminuir a replicação de código existente e devido ao seu uso também foram necessárias alterações no código das classes `Agent<T>` e `Scenário<T>`, conforme pode ser observado nos algoritmos 4.11 e 4.12.

```
1
2   template<class T>
3   class Agent : private Agent<void*> {
4       typedef Agent<void*> Base;
5       typedef Stub<T, false> _Stub;
6
7   public:
8
9       static void update(Message* msg){
10          unsigned int *addr = 0;
11          unsigned int n_addr = 0;
12          unsigned int *keys = _Stub::get_keys();
13          Adapter<T> *e;
```

```
14          T *t;
15
16          for(unsigned int i = 0; i < Adapter<T>::NUMBER_OBJS; i++) {
17              if(keys[i] == 0) {
18                  continue;
19              } else {
20                  e = _Stub::get(keys[i]);
21                  t = e->get_obj();
22                  if(T::TYPE == msg->id().type()) {
23                      addr = *reinterpret_cast<unsigned int**>(&t);
24                  }
25                  break;
26              }
27
28          n_addr = Base::update(msg, addr);
29
30          if(n_addr != 0){
31              for(unsigned int i = 0; i < Adapter<T>::NUMBER_OBJS; i++)
                    {
32                  if(keys[i] == 0) {
33                      continue;
34                  } else {
35                      e = _Stub::get(keys[i]);
36                      t = e->get_obj();
37                      if(T::TYPE == msg->id().type()) {
38                          addr = *reinterpret_cast<unsigned int**>(&t);
39                          addr[0] = n_addr;
40                      }
41                  }
42              }
43          }
44      }
45          // ...
46  };
```

Algoritmo 4.11: Trecho do código modificado na classe `Agent<T>`

```
1
2   template<class T>
3   class Scenario : public Id, private __IMP(Shared)<__IMP(Traits)<T>::
        shared>, private Scenario<void*>
4   {
5   public:
```

```
6       static const unsigned int NUMBER_OBJS = ScenarioHash::NUMBER_OBJS;
7       typedef Scenario<void*> Base;
8
9   protected:
10
11      static Adapter<T>* get(unsigned int id){
12          Adapter<T> *e = reinterpret_cast<Adapter<T> *>(Base::get(id));
13          return e;
14      }
15
16      static Adapter<T>* remove(const Id & id){
17          Adapter<T> *e = reinterpret_cast<Adapter<T> *>(Base::remove((
                unsigned int)&(id.id())));
18          return e;
19      }
20
21      // ...
22
23  public:
24      static void add(const Adapter<T> *e, const Id & id) {
25          Base::add(e, (unsigned int)&(id.id()));
26      }
27
28  };
```

Algoritmo 4.12: Trecho do código modificado na classe `Scenario<T>`

Para um entendimento mais completo de como serão realizados os processos de reconfiguração neste ELUS modificado, a figura 11 apresenta o novo diagrama de sequência da reconfiguração de componentes para o *framework*.

O processo de reconfiguração de um componente ainda inicia-se da mesma maneira, através do pedido de atualização para o `Reconfigurator`, envio da mensagem para o `Agent` e a utilização do `Semaphore` para alcançar o estado quiescente do componente. As alterações no código começam no método `update` do `Agent`, que agora pega o endereço da *vtable* antes de utilizar a `Base` (especialização de `Agent` para ponteiro *void*) para recuperar dos dados da mensagem recebida e alocar um espaço para o novo código caso seja maior do que o antigo. As tarefas de atualizar a tabela de métodos virtuais e liberar o `Semaphore` permanecem igual ao código original.

Conforme exposto no capítulo 3, o *framework* é responsável por uma série de atividades que devem ocorrer antes ou depois da invocação do método, o que também ajuda na perda de

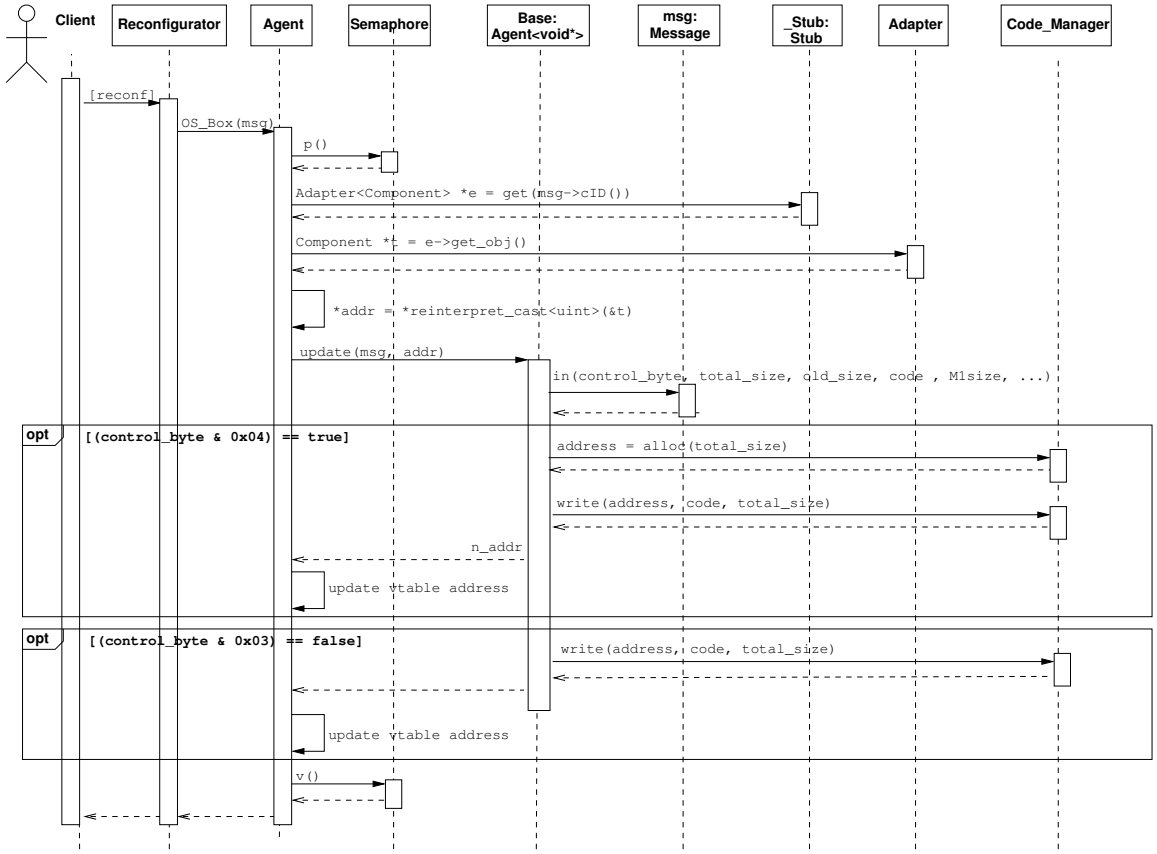

Figura 11: Resumo da sequência de atividades realizadas pelo ELUS modificado em uma reconfiguração de componentes.

1

desempenho. Esta indireção entre a aplicação e o método real criada pelo *framework* prejudica o tempo de invocação.

Neste contexto, verificando a parte do código que recupera este objeto, percebe-se que esta tarefa também é realizada através de classes metaprogramadas e que este caminho poderia ser reduzido caso houvesse uma menor quantidade de indireções até a tabela *hash* em que os objetos estão armazenados.

Embora todos os objetos sejam armazenados pela mesma tabela *hash*, uma grande desvantagem para a invocação de métodos é que o objeto é criado internamente ao *framework* e só pode ser acessado através do `Adapter`. Estes `Adapters` além de realizarem a adaptação do componente são centralizadores dos objetos, ou seja, cada *Adapter<T>* possui vários ponteiros do tipo T. O algoritmo 4.13 mostra um trecho da classe `Adapter`, em que podemos verificar a criação do objeto dentro do *framework*.

```
1 template<class T>
2 class Adapter : public Scenario<T>, public T {
```

```
3   private:
4       T *t;
5
6   public:
7       Adapter(){ t = new T(); }
8
9       template<typename T1>
10      Adapter(T1 a1) { t = new T(a1); }
11      template<class T1, class T2>
12      Adapter(T1 a1, T2 a2) { t = new T(a1, a2); }
13      template<class T1, class T2, class T3>
14      Adapter(T1 a1, T2 a2, T3 a3) { t = new T(a1, a2, a3); }
15      template<class T1, class T2, class T3, class T4>
16      Adapter(T1 a1, T2 a2, T3 a3, T4 a4) { t = new T(a1, a2, a3, a4); }
17
18      T * get_obj() { return t; }
19      void set_obj(T *obj) { t = obj; }
20      // ...
21  };
```

Algoritmo 4.13: Trecho da classe `Adapter`

Inicialmente a alteração da classe `Scenario` era parte da proposta de redução de consumo de memória, mas ao extrapolar esta solução para o tempo de invocação dos métodos foi possível armazenar todos os objetos como se fossem de apenas um tipo (ponteiros *void*), sendo assim, na hora da recuperação deste objeto deve ser realizado um `reinterpret_cast` para que o objeto seja corretamente reinterpretado para o tipo `T` originalmente criado.

5 RESULTADOS OBTIDOS

Neste capítulo serão apresentados os resultados obtidos da implementação das propostas realizadas no Capítulo 4.

5.1 MODELAGEM HÍBRIDA

A modelagem híbrida da maneira como proposta anteriormente não foi possível de se implementar devido à incompatibilidade entre os padrões. Pois apesar do modelo FRACTAL ser independente de linguagem, THINK exige que todos os componentes (e inclusive suas dependências) sejam descritos utilizando o padrão (IDL, ADL e C) e compilados pelo compilador de THINK, capaz de entender as anotações encontradas no código.

Para utilizar THINK como base da implementação do modelo híbrido seria necessário padronizar a descrição de cada componente do sistema operacional EPOS. Esta tarefa é extremamente trabalhosa e exige muito conhecimento do sistema operacional, pois seria necessário descrever inclusive os componentes mais básicos, como listas e filas.

No caso de utilizar o EPOS e ELUS como base, seria necessário estudar o comportamento do compilador de THINK e reproduzir o processo no *framework*. Acontece que os arquivos de descrição possuem papel fundamental no processo de reconfiguração de THINK e sem estas informações não é possível realizar o processo.

Estudando o EPOS/ELUS, THINK e os sistemas operacionais apresentados no capítulo 2 foi possível notar que apesar de cada um possuir uma maneira diferente de expressar o seu processo de reconfiguração, eles compartilham os mesmos conceitos, como:

- Alcançar o estado quiescente: estado no qual nenhuma atividade está sendo realizada no componente atualizado no momento da reconfiguração.
  - EPOS/ELUS - existe um semáforo para cada componente atualizável que garante que nenhuma *thread* invoque métodos deste componente.

  - THINK - utiliza os `intecerptors` para rebater/desviar qualquer tipo de acesso às interfaces do componente.
  - μCLinux - o *kernel* remove o módulo antigo e retira as referências a este módulo.
  - Contiki - ao receber uma mensagem de atualização, o *kernel* envia um evento no qual o serviço que será atualizado deve remover-se do sistema.
  - RETOS - para que ninguém consiga utilizar um módulo, basta retirar seu registro na tabela de funções mantida pelo sistema operacional.
  - SOS - o *kernel* envia uma mensagem `final` para que o módulo libere seus recursos e retire as suas publicações das listas de funções.

- Transferência de estado: transferir o estado do componente antigo para o novo componente.
  - EPOS/ELUS - realizado através de métodos `get` e `set` ou pela criação de um novo objeto com os mesmos argumentos passados no construtor do antigo objeto (e deletando o antigo objeto).
  - THINK - esta transferência ocorre à partir da interface `LifeCycleController` que possui um autômato para controlar o estado do componente ao qual ela está ligada.
  - μCLinux - usando a configuração para que o carregamento do novo módulo seja realizado diretamente no *kernel* não há transferência de estado. Caso esta configuração não seja utilizada, ocorre uma cópia na memória do módulo a ser carregado e a transferência de estados é realizada internamente ao *kernel*.
  - Contiki - o *kernel* disponibiliza uma interface para a passagem de ponteiros entre as duas versões do serviço.
  - RETOS - o *kernel* e seus módulos são ligados dinamicamente e funcionam como uma única imagem, sendo que a transferência de estado é realizada pelo *kernel*.
  - SOS - cada módulo ao ser carregado deve publicar uma lista de funções oferecidas e uma de funções requeridas de outros módulos.

- Ajuste de referências: fazer com que os objetos corretos sejam chamados após a reconfiguração.
  - EPOS/ELUS - o `framework` atualiza as referências para que apontem para o endereço do novo componente.

  - THINK - a interface `BindingController` é a responsável por fazer a ligação entre a interface cliente/servido do componente atualizado e as interfaces cliente/servidor dos componentes que utilizam ou são utilizados por ele.
  - μCLinux - se o módulo for carregado diretamente no *kernel* não é necessário o ajuste de referências, mas se for realizada a cópia do módulo na memória é necessário utilizar os `bitstreams` para ajustar as referências antes que o módulo seja carregado.
  - Contiki - as referências são ajustadas na tabela de funções da interface do serviço.
  - RETOS - assim que o novo módulo é carregado no sistema, RETOS utiliza metadados criados pelo mecanismo de realocação em tempo de compilação para substituir os endereços.
  - SOS - os ponteiros na tabela dentro do *kernel* são atualizados antes da inicialização do novo módulo.

- Nível de indireção: a aplicação não possui acesso direto aos componentes reais.
  - EPOS/ELUS - os elementos `Proxy` e `Adapter` isolam os componentes ao criar um nível de indireção entre as chamadas de métodos da aplicação e os componentes reais do sistema.
  - THINK - cada componente possui uma separação lógica entre membrana e conteúdo. A membrana é considerada dona do conteúdo e ela implementa as interfaces de controle de acesso à esse conteúdo, gerando o nível de indireção.
  - μCLinux - a indireção se encontra no acesso aos módulos através de uma tabela com os endereços de todos os módulos carregados.
  - Contiki - cada serviço consiste em `service interface` e processos que implementam esta interface. Somente a `service interface` possui os ponteiros para as implementações das funções relativas aos processos do serviço. Para chamar um serviço é necessário passar por uma interface `Stub` que redireciona as chamadas de funções para a `service interface` do serviço desejado.
  - RETOS - O acesso à funções pela aplicação é realizado através de uma tabela, na qual um módulo pode registrar-se e cancelar seu registro.
  - SOS - As funções devem ser chamadas à partir de uma tabela em que estão publicadas todas as funções disponíveis por cada módulo, criando aqui um nível de indireção.

5.2 ESPECIALIZAÇÃO PARA PONTEIROS *VOID*

A fim de validar este trabalho, serão realizados testes comparativos entre a versão modificada e o ELUS original apresentado por Gracioli (GRACIOLI; FRÖHLICH, 2009).

5.2.1 CONSUMO DE MEMÓRIA

O ELUS original precisa de um total de 3.5 KB de memória de código, 43 *bytes* para dados, 124 *bytes* para dados não inicializados e 375 *bytes* para *bootloader*.

Conforme pode ser observado na Tabela 3, o resultado alcançado pela modificação do *framework* de ELUS necessita cerca de 2.2 KB de memória de código, 26 *bytes* de dados (para armazenar o `Dispatcher` e outros atributos), e ainda mais 140 *bytes* para armazenar a tabela *hash* e o *Semaphore*.

Tabela 3: Consumo de memória do ELUS modificado

Elementos do ELUS	**ELUS original Tamanho Seção (bytes)**				**ELUS modificado Tamanho Seção (bytes)**			
	.text	**.data**	**.bss**	**.bootloader**				
Reconfigurator	410	0	70	0	410	0	70	0
Code Manager	16	0	2	375	16	0	2	375
App. Update	210	0	0	0	210	0	0	0
OS Box	76	0	0	0	76	0	0	0
Framework	2620	43	52	0	1648	26	68	0
Total	3452	43	124	375	2284	26	140	375

Como o objetivo desta proposta era modificar apenas a maneira como o *framework* realizava o processo de reconfiguração, não foi realizada nenhuma modificação nas outras estruturas (`Reconfigurator, Code Manager, App. Update, OS Box`), que permanecem com os mesmos valores de consumo de memória. Porém se forem comparados os resultados do *framework* entre as duas versões, é possível verificar que houve um ganho de memória, já que o framework original consumia cerca de 1kb a mais de memória do que a nova versão.

Para verificar qual parte do *framework* foi a responsável por este ganho foram realizadas medições mais específicas, desta vez considerando os métodos `create, destroy, update` e mais os quatro tipos possíveis de métodos (com parâmetro, sem parâmetro, com valor de retorno e sem valor de retorno), conforme pode ser observado na Tabela 4.

O resultado observado não foi uma surpresa, pois a classe `Agent` é aquela que contém o maior número de métodos dependentes do tipo do componente e métodos como `Create`,

Tabela 4: Consumo de memória dos métodos individuais em ambas versões do ELUS.

Método do Framework	**ELUS original**		**ELUS modificado**	
	Tamanho da seção (bytes)		**Tamanho da seção (bytes)**	
	.text	**.data**	**.text**	**.data**
Create	180	0	178	0
Destory	138	0	136	0
Método sem parâmetro e valor de retorno	94	0	90	0
Método com um parâmetro e sem valor de retorno	98	0	94	0
Método sem parâmetro e com valor de retorno	112	0	104	0
Método com um parâmetro e valor de retorno	126	0	122	0
Update	1250	0	260	0
Dispatcher	0	2*(# métodos)	0	2*(# métodos)
Semaphore	0	18	0	18
Tamanho mínimo	1662	26	664	26

`Destroy` e `Update` eram replicados para cada componente marcado como reconfigurável. Com a implementação da proposta de especialização para ponteiros *void* como base para os métodos de cada classe específica foi possível reduzir replicação de código, ou seja, a parte mais volumosa da implementação desta classe agora não entra como um consumo individual do componente, mas como uma base comum a todos os componentes reconfiguráveis. Assim, o tamanho mínimo de um novo componente do sistema passa a ser 664KB, um ganho de cerca de 1KB em comparação com o ELUS original.

5.2.2 TEMPOS DE INVOCAÇÃO DE MÉTODO

O tempo de invocação de métodos é avaliado em termos de quantidade de ciclos para realizar uma chamada de método e analisar se houve algum sobre custo neste processo. É uma avaliação importante porque o *framework* deve realizar uma série de atividades antes e depois da invocação de métodos. A Tabela 5 apresenta o tempo de invocação de métodos do ELUS original, modificado e os compara com a chamada de um método normal e através da tabela de métodos virtuais. Percebe-se claramente que há uma perda de desempenho causada pelo nível de indireção criado entre a aplicação e a invocação do método real através do *framework* metaprogramado.

Apesar de ser uma das partes substituídas para uma tentativa de melhora de desempenho,

Tabela 5: Comparação dos tempos de invocação de método entre uma invocação normal, através da vtable, do ELUS e do ELUS modificado

Invocação	Normal	Vtable	ELUS	ELUS Modificado
Método sem parâmetro e sem valor de retorno	4	13	135	135
Método com parâmetro e sem valor de retorno	7	15	139	139
Método sem parâmetro e com valor de retorno	6	14	152	152
Método com parâmetro e com valor de retorno	8	16	161	161

o resultado numérico das duas versões foi igual. Estudando o código gerado (e.g, *assembly*) para ambas versões pode-se concluir que houve diferença das instruções, mas após a recontagem do número de ciclos utilizados para a invocação de método, este número permaneceu sem alterações.

5.2.3 TEMPO DE RECONFIGURAÇÃO

A comparação dos tempos de reconfiguração foi medida em dois cenários: (i) o novo código do componente é menor ou igual ao antigo, sendo assim a atualização pode utilizar a mesma posição do componente e (ii) o novo código de um componente é maior que o antigo, em que se faz necessário a alocação de uma nova posição para o componente. Para ser uma comparação justa em relação aos trabalhos relacionados, esse teste não considerou o tempo de recebimento de dados pela rede e nem o tempo de escrita dos dados na memória de programa. O resultado do teste pode ser observado na Tabela 6.

Tabela 6: Comparação dos tempos de reconfiguração para um componente no ELUS e do ELUS modificado, medido em ciclos

Atualização	ELUS original	ELUS modificado
Mesma posição	362	368
Nova posição	414	397

Este resultado foi obtido mensurando as instruções obtidas pelo *assembly* e verificando no manual do Mica2 sua equivalência em ciclos. Examinando o tempo de reconfiguração em relação ao ELUS original e os resultados obtidos pelas modificações é possível observar um aumento no tempo da reconfiguração quando o componente é colocado na mesma posição e uma diminuição do tempo em relação à atualização em uma nova posição.

Embora este resultado pareça controverso, pode ser facilmente entendido se compararmos o código *assembly* gerado pelo ELUS original e pelo ELUS modificado, pois, apesar do mesmo número de ciclos, tanto as intruções quanto à quantidade de instruções sofreram modificações. Devido à utilização da especialização para ponteiros *void* o código modificado inclui instruções de cálculo e retorno de um valor pela classe `Base`, que servirá para verificar se é necessário ou não a atualização do endereço da tabela virtual em cada objeto.

Já a diminuição do tempo da reconfiguração para uma nova posição pode ser associada à quantidade e à ordem das instruções do código modificado, pois a classe `Base` é responsável por todas as operações, com exceção da atualização do endereço da tabela virtual em cada objeto. O mesmo não ocorre com o ELUS antigo, que realiza este processo através da classe dependente de *template* e precisa realizar verificações relativas ao tipo.

6 CONCLUSÕES

O presente trabalho teve como objetivo realizar um estudo sobre métodos para melhoria de desempenho da reconfiguração dinâmica de *software* de sistemas embarcados. Nele é apresentada uma resenha sobre os mais relevantes mecanismos de reconfiguração dinâmica de *software* para sistemas embarcados e também uma modelagem para melhoria no processo de reconfiguração da infraestrutura ELUS.

São realizadas duas propostas de melhoria de desempenho para reconfiguração dinâmica: Uma modelagem híbrida e uma reestruturação do *framework* utilizando especialização para ponteiros *void*.

O modelo de composição híbrida pretendia reduzir as limitações encontradas nos trabalhos relacionados. A utilização do modelo FRACTAL juntamente com a estrutura fornecida pelo ELUS produziu uma modelagem que agrega características como consumo de memória reduzido, transparência para a aplicação, reflexividade, portabilidade, modularidade, adaptabilidade e extensibilidade. Este modelo alcançou a redução das limitações tanto do *framework* ELUS quanto na implementação THINK. Pode-se dizer que as duas arquiteturas quase complementam-se, ou seja, algumas das limitações podem ser totalmente eximidas - como o caso da dependência de arquitetura na implementação com THINK - e outras parcialmente melhoradas. É importante ressaltar que como o modelo não foi implementado, não houve uma sincronização entre os dois modelos e os resultados obtidos são teóricos.

Outra contribuição do modelo híbrido surgiu da dificuldade de implementá-lo, pois não foi possível separar os conceitos relativos à reconfiguração, como, alcançar o estado quiescente, transferência de estado, ajuste de referências e níveis de indireção entre aplicação e sistema operacional. Além disso, para realizar a implementação conforme o modelo seria necessário criar em volta de cada um dos componentes dos EPOS um artifício para que a fatia que implementa o modelo cliente/servidor conseguisse visualizar e comunicar-se com os demais componentes presentes no ELUS. Examinando THINK, ELUS e os sistemas operacionais descritos no capítulo 2 foi possível verificar que, apesar de cada um deles possuir nomenclaturas diferentes para as fases do processo de reconfiguração, todos eles utilizam o mesmo modelo e

passam pelas mesmas operações para dar suporte à uma reconfiguração segura.

A proposta de especialização através de ponteiros *void* apresentou resultado positivo para o consumo de memória, pois foi possível reduzir cerca de 1.2KB de consumo e atualmente o tamanho mínimo para um novo componente do sistema passa a ser 664KB ao invés dos 1.6KB do ELUS original. O tempo de invocação de métodos permaneceu o mesmo, mas conforme discutido por Gracioli (GRACIOLI; FRÖHLICH, 2009) este resultado ainda é um resultado melhor se comparado aos trabalhos relacionados. Por fim, o tempo de reconfiguração apresentou um resultado positivo para a reconfiguração de componentes em uma nova posição (17 ciclos a menos), porém, as modificações também foram responsáveis pelo aumento em 6 ciclos o tempo de atualização de um componente na mesma posição.

Apesar de possuir um resultado adverso no tempo de reconfiguração, é possível concluir que à partir da abordagem para melhoria individual aplicada em todas as limitações culminou em uma melhoria global, ficando em concordância com os objetivos iniciais deste trabalho.

Dentre os desafios encontrados para a realização dos modelos propostos, é possível ressaltar que, apesar da arquitetura de THINK ser uma excelente alternativa para a construção de componentes para sistemas operacionais, houve complicações durante o seu estudo, em virtude da organização e da disponibilidade da documentação.

Apesar das dificuldades para concretizar este trabalho, o objetivo foi alcançado. Os modelos aqui sugeridos podem ser aplicados em sua essência ou até servir de base de estudos para novas arquiteturas de reconfiguração dinâmica.

6.1 TRABALHOS FUTUROS

- Redução do consumo de energia: estudar possibilidades para a redução do consumo de energia, que também é um recurso muito importante no domínio de sistema embarcado.

- Aumentar a reusabilidade do código: melhorar a arquitetura aplicando diferentes padrões de projeto para uma maior portabilidade do código. Um exemplo seria extrair do *framework* as operações específicas dos componentes marcados como reconfiguráveis, assim não seria necessária a modificação dos elementos do *framework* toda vez que houvesse uma alteração na lista de componentes marcados como reconfigurável.

- Criar documentação detalhada do modelo proposto: uma boa documentação é essencial para que outras pessoas possam dar continuidade ao presente trabalho.

APÊNDICE A – ARTIGO

Melhorando o desempenho do ELUS através de especialização de *templates*

Rita de Cássia Cazu Soldi, Giovani Gracioli e Antônio Augusto Fröhlich

[1]Laboratório de Integração de Software e Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
88040900 – Florianópolis, SC – Brasil

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

Abstract. *Dynamic reconfiguration of software in embedded systems with severe resource constraints is a task that requires a performance optimization, since the reconfiguration process competes with the software for the limited system resources. ELUS is an operating system infrastructure that performs this activity effectively and safely, however, the use of resources can still be improved. This article presents a study of ELUS and proposes a performance improvement using templates specialization.*

Resumo. *Reconfiguração dinâmica de software em sistemas embarcados com severas restrições de recursos é uma tarefa que exige otimização de desempenho, visto que o processo de reconfiguração compete com o software pelos limitados recursos do sistema. O ELUS é uma infraestrutura de sistema operacional que realiza esta atividade de maneira eficaz e segura, porém, a utilização dos recursos ainda pode ser aprimorada. Este artigo apresenta um estudo do ELUS e propõe uma melhoria de desempenho por meio do uso de especialização de templates.*

1. Introdução

Reconfiguração dinâmica é a capacidade de atualizar o *software* de um determinado sistema sem perder o seu estado de execução e pode ser usada para diversas finalidades como realizar atualizações, implementar algoritmos para substituir componentes do sistema, monitoramento dinâmico, apoio a otimizações específicas da aplicação, ajuda para que componentes do sistema possam ser recebidos da rede e instalados em tempo de execução, entre outros.

Este tipo de reprogramação é extremamente importante no contexto de sistemas embarcados para que se possa adaptar os dispositivos às mudanças que ocorrem no ambiente, modificar o seu conjunto de tarefas e até aprimorar o uso de seus recursos. Quando a necessidade é aplicar a reconfiguração dinâmica em sistemas com severas restrições (ex. processamento, memória, energia) torna-se um desafio, pois, apesar dos mecanismos de atualização competirem pelos já limitados recursos do sistema, eles não devem influenciar no funcionamento do mesmo.

O ELUS (*Epos Live Update System*) [Gracioli and Fröhlich 2009] é uma infraestrutura de sistema operacional para a reconfiguração dinâmica que difere das outras infraestruturas pelo baixo consumo de memória, alta configurabilidade, simplicidade e transparência para as aplicações. Se comparado a trabalhos equivalentes, o ELUS consome

menos memória, possui um menor tempo de invocação de métodos e menor tempo de reconfiguração. Apesar dos resultados favoráveis, foi identificado que é possível melhorá-los. O ELUS foi implementado em C++ utilizando o *framework* de componentes meta-programado do EPOS [Fröhlich 2001]. O framework utiliza templates e com isso há uma replicação de código sempre que um novo componente é marcado como reconfigurável. Para contonar esse problema, este artigo propõe o uso da técnica de especialização de templates para diminuir a replicação de código dentro do *framework*. Com o uso dessa técnica, foi possível melhorar o consumo de memória sem perder desempenho em outras caracterícias, como o tempo de invocação de métodos e tempo de reconfiguração dos componentes.

A estrutura deste artigo apresenta-se da seguinte forma. A seção 2 detalha o ELUS quanto à sua organização e algumas das suas limitações. Na seção 3 é detalhada a modelagem para a melhoria de desempenho do processo de reconfiguração dinâmica para a infraestrutura ELUS. Os resultados obtidos com as modificações são apresentados na seção 4. A seção 5 apresenta os trabalhos relacionados e, finalmente, a seção 6 conclui o trabalho.

2. Epos Live Update System (ELUS)

O ELUS é uma infraestrutura de reconfiguração dinâmica que pode ser utilizada em sistemas embarcados com recursos limitados. Construída dentro do *framework* de componentes do EPOS (*Embedded Parallel Operating System*) [Fröhlich 2001] em torno do aspecto de invocação remota, esta arquitetura permite que um componente seja selecionado como reconfigurável ou não em tempo de compilação. As principais características que a difere dos outros trabalhos relacionados são a configurabilidade, o consumo de memória reduzido, a simplicidade e a transparência para as aplicações e reconfiguração.

2.1. Arquitetura

A arquitetura do ELUS é apresentada na Figura 1. O *framework* de componentes original do EPOS foi estendido para suportar também reconfiguração dinâmica. A invocação de um método de um componente passa pelo PROXY que envia uma mensagem ao AGENT. O valor de retorno depois da execução do método é enviado de volta ao cliente através de uma mensagem. Um nível de indireção é criado entre o cliente e o método real, tornando o AGENT o único membro da estrutura ciente da posição do componente na memória do sistema. A OS BOX no AGENT controla o acesso aos métodos do componente através de um semáforo para cada componente, chamando os métodos do AGENT através de uma tabela. O AGENT quando recebe uma invocação de método, envia o pedido para o ADAPTER que chama o método real do componente através da tabela de métodos virtuais (VTABLE) do componente.

O ELUS TRANSPORT PROTOCOL (ETP) é um protocolo para o recebimento de mensagens de reconfiguração e toda atualização ocorre a partir de uma requisição neste formato. O *framework* permite a atualização de componentes, da aplicação e do próprio *framework* de reconfiguração. Conforme pode ser observado na Figura 2, para atualizações referentes a um componente, o ELUS tenta alcançar um estado em que nenhuma *thread* esteja invocando métodos deste componente, ou seja, o estado quiescente. Em seguida é verificada a necessidade de alocar um espaço novo para o código, necessário

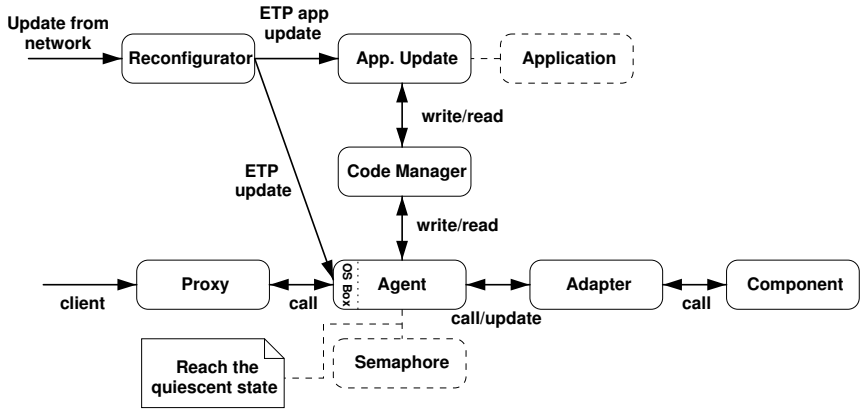

Figura 1. Visão geral da arquitetura do ELUS [Gracioli and Fröhlich 2010].

apenas quando o código novo é maior que o antigo. Na sequência é feita a atualização da tabela de métodos virtuais e é verificado se há necessidade da atualização do componente no *framework*.

Para a atualização de componentes do *framework* é necessário um pouco mais de cautela, pois o `Agent` - mais especificamente o método `trapAgent` - é o ponto de entrada para o serviço de reconfiguração e o seu endereço deve ser conhecido previamente para que o novo código do componente possa ser compilado corretamente. Então para ter certeza de que o endereço do `trapAgent` não será modificado ele não é uma função passível de reconfiguração.

Na reconfiguração da aplicação, a partir do pedido de atualização se inicia o processo de desabilitar as interrupções para atingir o estado quiescente da aplicação. O tamanho do código recebido é comparado com o código antigo para saber se ocorre uma simples atualização do endereço (se o novo código for menor ou igual ao antigo), caso contrário, é necessário alocar um novo espaço de memória antes de atualizar o endereço. Após a atualização da aplicação o sistema deve ser reiniciado para que suas as novas atividades possam ser realizadas.

2.2. Limitações

Devido à estrutura do ELUS, cada componente selecionado como configurável gera um sobre custo em termos de memória, porque além da implementação real do método pelo próprio componente, ainda se faz necessário incluir os métodos do componente no *framework*. Após a geração do sistema final, o código dos métodos do *framework* é replicado para cada componente. Por exemplo, cada componente selecionado como reconfigurável irá ter o código do método *update* replicado. A configuração de consumo mínimo de memória para adicionar o componente no *framework* seria conter apenas os métodos criar, destruir, atualizar e um método sem parâmetros e retornar o valor e mesmo assim haveria o sobre custo de cerca de 1.6KB de código e 26 *bytes* de dados.

Outra limitação é o tempo de invocação dos métodos, pois ocorre uma indireção entre a aplicação e a invocação do método real, atribuindo ao *framework* uma série de responsabilidades que acabam aumentando o tempo de invocação. Dentre as atribuições do *framework* estão o armazenamento e controle sobre os objetos reconfiguráveis criados pela aplicação, incluindo alcançar o estado quiescente do componente. Todas estas

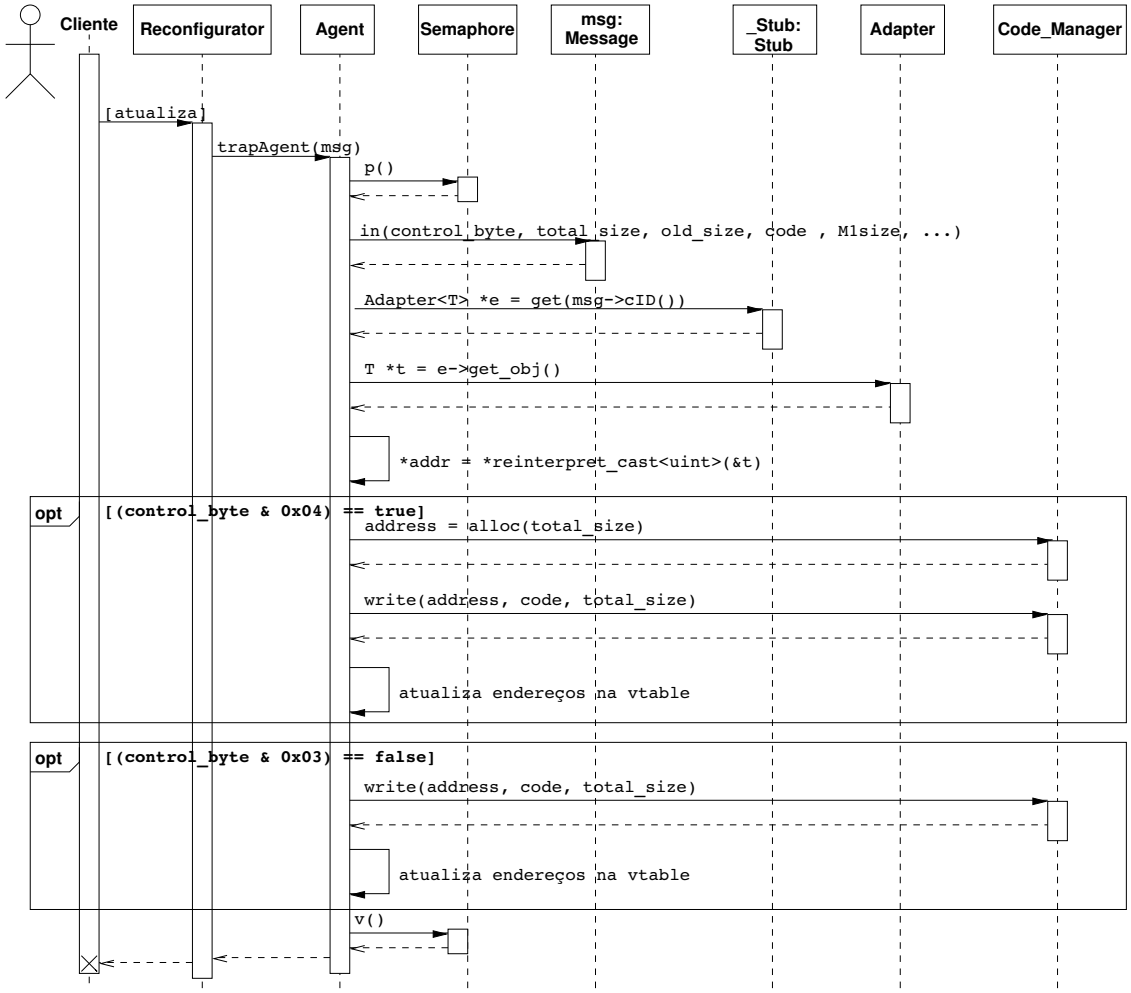

Figura 2. Resumo da sequência de atividades realizadas pelo ELUS em uma reconfiguração de componentes.[Gracioli and Fröhlich 2009]

responsabilidades fazem com que o tempo de invocação de métodos utilizando o *framework* seja cerca de 10 vezes mais lento do que a invocação normal ou através de uma VTABLE. Mesmo com esse sobrecusto, o ELUS apresenta um melhor desempenho que os trabalhos relacionados, possuindo o tempo de invocação pelo menos 7 vezes mais rápido, o tempo de reconfiguração com cerca de 197 ciclos menor e consumindo menos de 33% da memória utilizada por qualquer um dos trabalhos relacionados [Gracioli and Fröhlich 2010].

3. Proposta de modelagem

Analisando o código do *framework* da infraestrutura ELUS foi possível observar uma estrutura metaprogramada utilizando *templates*, conforme pode ser observado na Figura 3.

Os elementos compõem o `framework` e são utilizados no processo de reconfiguração que a infraestrutura provê são: `Handle`, `Stub`, `Proxy`, `Agent`, `Scenario` e `ScenarioHash`. Segue abaixo uma pequena descrição das suas respectivas funções.

- `Handle` - Quando o suporte ao *framework* e à reconfiguração dinâmica estão

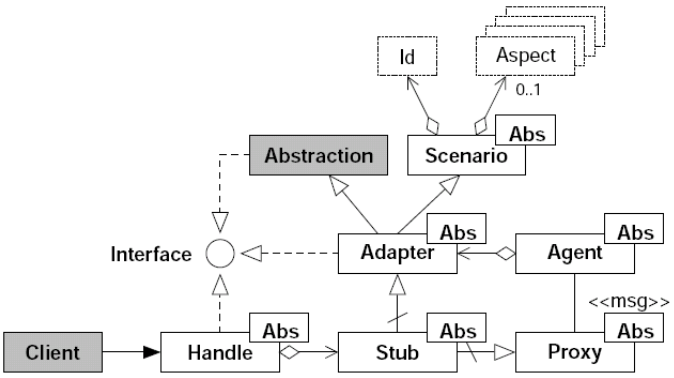

Figura 3. Interação entre os elementos do *framework* metaprogramado [Gracioli and Fröhlich 2009].

habilitados, o componente é exportado como um parâmetro para o `Handle`, que possui a mesma interface (operações) que o componente, mas com a invocação dos métodos encaminhadas para o `Stub`. O `Handle` é responsável por verificar se um componente está ou não selecionado como reconfigurável.

- `Stub` - Responsável por criar um `Proxy` ou um `Adapter` conforme os parâmetros recebidos na sua criação via Handle. Então se a reconfiguração do componente está habilitada, Handle irá criar um `Stub` que contém todos os métodos do `Proxy`, permitindo assim a comunicação do `Proxy` com o `Agent`, surgindo aqui um nível de indireção entre as chamadas de métodos da aplicação e os componentes reais do sistema.
- `Proxy` - Responsável por efetuar os métodos do componente. Quando recebe um pedido do `Handle`, ele utiliza a classe `Message` para passar o `ID` do componente e a solicitação desejada para o `Agent`.
- `Agent` - Possui duas funções : a invocação de métodos e a reconfiguração dos componentes e do próprio *framework*. Caso seja uma invocação de métodos, a `Message` será enviada pelo `Proxy` então ele chama o `Adapter` e reconfigura o componente através do método do `update`. Senão a `Message` foi enviada pela *thread* `Reconfigurator`, e o `Agent` deve fazer a reconfiguração do próprio *framework*.
- `Scenario` - responsável por aplicar os aspectos selecionados para cada componente recebido pelo metaprograma. Também armazena os objetos criados pelo `Agent` em tempo de execução.
- `ScenarioHash` - Funciona como uma abstração das operações de controle, armazenamento, busca e recuperação de objetos através de uma tabela *hash*.

Os elementos auxiliares são: `Message`, `Adapter`, `Application_Update`, `Reconfigurator` e `Code_Manager`.

- `Message` - Ponto comum entre o `Proxy` e o `Agent`. Oferece uma interface para anexar e recuprar informações úteis para a reconfiguração, como por exemplo tamanho do códito, tipo de reconfiguração, retorno de um método, entre outros.
- `Adapter` - Funciona como um *wrapper* e realiza a adaptação do componente que foi recebido como parâmetro através dos métodos `leave` e `enter` do `Scenario` antes e depois da invocação do método real.

- `ApplicationUpdate` - É um componente criado exclusivamente para dar suporte à atualização da aplicação. Sua tarefa principal é desabilitar interrupções do sistema para conseguir atingir o estado quiescente.
- `Reconfigurator` - É uma `thread` criada na inicialização do sistema e é responsável por receber o pedido de reconfiguração, construir `Message` e enviar um pedido de reconfiguração para o `Agent`
- `Code_Manager` - Responsável por utilizar a estrutura fornecida pelo mediador de *hardware* para gerenciar alocação e liberação de espaços na memória.

Apesar de todos os elementos serem importantes para o processo de reconfiguração, o comportamento padrão para implementação de *templates* utilizando C++ é replicar o código para cada função recebida como argumento. Desta forma, os elementos que interagem diretamente com o componente atualizável são replicados para cada argumento de tipo de componente recebido. Uma boa alternativa para evitar a replicação de código é extrair os techos comuns a todas as funções *template* para uma especialização para ponteiros *void*.

3.1. Especializações de *Templates* em C++

Mecanismos de *templates* providos pelo C++ são uma maneira de adicionar à classes e funções conceitos genéricos, isto é, permitem criar um código reutilizável onde os tipos sejam passados como parâmetro.

Por padrão, as classes *template* possuem uma única implementação para qualquer argumento recebido e quando deseja-se dar um tratamento mais refinado à um determinado tipo de argumento, é necessário realizar descrições alternativas do *template*. Estas descrições, também conhecidas como especializações, são escolhidas pelo compilador com base nos argumentos fornecidos para o `template`.

O ponteiro *void* é um ponteiro que pode apontar para qualquer tipo de objeto e pode ser explicitamente convertido para qualquer outro tipo. Sendo assim, se for realizada uma especialização de *template* para ponteiros *void*, ela poderia implementar as funções comuns a todos os tipos de argumentos sem que haja replicação de código.

Como exemplo desta ideia são apresentados na Figura 4 (classe `Vetor` genérica) e na Figura 5 (utilizando uma base para os demais argumentos).

```
1  template <class T> class Vector{
2     T* v;
3     int sz;
4  public:
5     Vector();
6     Vector(int);
7
8     T& elem (int i) {return v[i]; }
9     T& operator [] (int i);
10
11    void swap(Vector &);
12
13    // ...
14 };
```

Figura 4. Classe `Vector` generalizada [Stroustrup 1997].

```
1  template <class T> class Vector <T*> : private
       Vector <void*>{
2  public:
3    typedef Vector <void*> Base;
4
5    Vector() : Base() {}
6    explicit Vector(int) : Base(i) {}
7
8    T& elem (int i) {return static_cast<T*&>(Base::
         elem(i)); }
9    T& operator [] (int i) {return static_cast<T*&>(
         Base::operator[](i)); }
10
11   // ...
12 };
```

Figura 5. Classe `Vector` com base especializada para para ponteiros *void* [Stroustrup 1997].

Podemos observar que na Figura 5 não há mais a implementação de alguns métodos dentro da classe genérica de `Vector` e, consequentemente, ao invés de uma replicação haverá apenas uma chamada de método.

Existem vários trechos do *framework* que podem ser modificados de acordo com esta proposta. A Figura 6 mostra um trecho do método `update` da classe `Agent`. Nela podemos observar que em grande parte do código não há nenhuma referência ao argumento `T` do *template* e por isso é uma das classes em que há mais replicação de código.

```
static void update (Message* msg) {
  unsigned char mID, ctr_byte;
  unsigned int size, t_size, code, old_size;
  unsigned int n_addr = 0;
  unsigned int *addr = 0;
  Message *msg_extra;
  msg->in(ctr_byte, mID, t_size, code, msg_extra);
  unsigned char *new_code = (unsigned char *) code;
            Código que utiliza o argumento "class T"
  unsigned int *p = (unsigned int *)(*addr);
  char zero = 0;
  unsigned int displacement = 0;
  unsigned int dispatcher_size = 0;
  unsigned int total_msgs = (ctr_byte & 0xF0) >> 4;
  switch(ctr_byte) {
      ...
  }
}
```

```
unsigned int *keys = _Stub::get_keys();
Adapter<T> *e;
T * t;
for (unsigned int i = 0; i < Adapter<T>::N_OBJS; i++) {
    if(keys[i] == 0) {
        continue;
    } else {
        e = _Stub::get(keys[i]);
        t = e->get_obj();
        if(T::TYPE == msg->id().type()) {
            addr = *reinterpret_cast<unsigned int**>(&t);
        }
        break;
    }
 }
```

Figura 6. Trecho do método `update` da classe `Agent`.

Considerando as modificações que devem ser realizadas no modelo atual do ELUS, a Figura 7 apresenta o novo diagrama de classes. Tanto o `Agent<void*>` quanto o `Scenario<void*>` são especializações utilizadas como base para diminuir a replicação código existente. Estas classes realizam os mesmos métodos, porém o retorno agora é um ponteiro `void` e deve ser reinterpretado pelo tipo `T` do *template* através das classes `Agent<T>` e `Scenário<T>`.

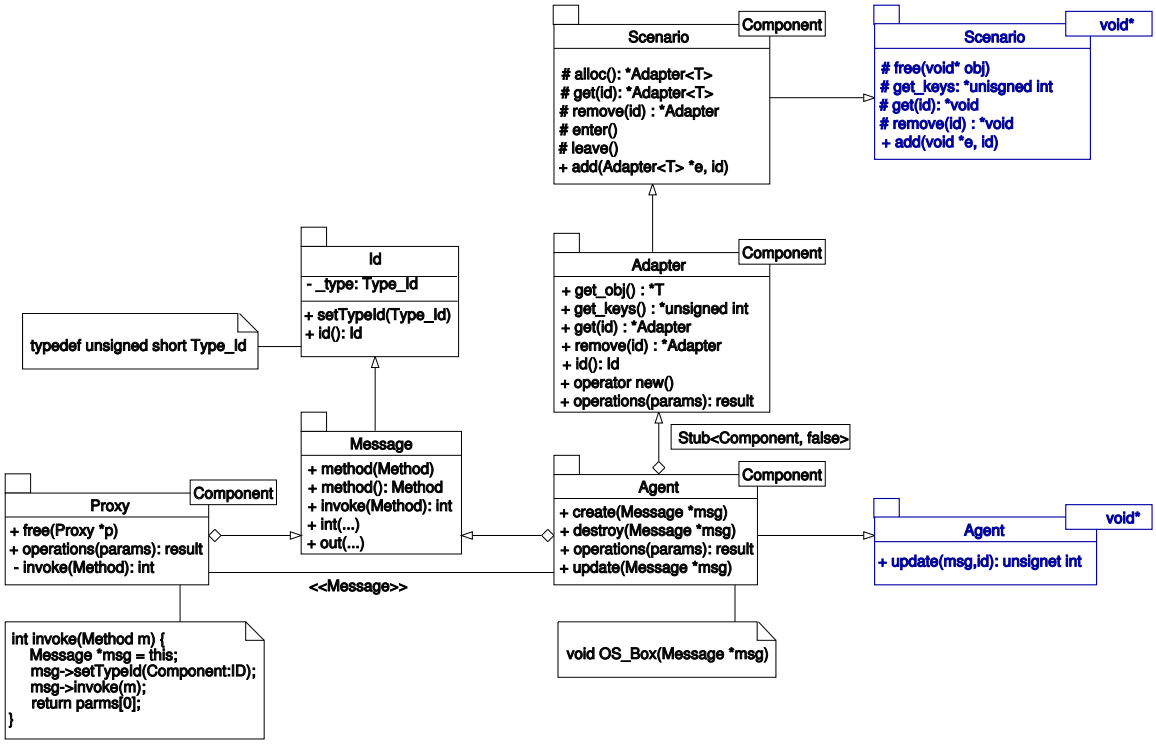

Figura 7. Diagrama da classes envolvidas no processo de reconfiguração.

4. Resultados

A avaliação do ELUS modificado foi realizada na plataforma Mica2 [1] e o código gerado pelo compilador GNU g++ versão 4.0.2. Os testes realizados visam medir o consumo de memória, desempenho de invocação de métodos e tempo de reconfiguração. Ainda, na medição de consumo de memória foi utilizada a ferramenta GNU `objdump` na versão 2.16.1 para analisar as imagens do sistema geradas pelo compilador e o tempo gasto na reconfiguração foi medido em ciclos do processador segundo o manual do microcontrolador.

4.1. Consumo de memória

Esta avaliação mensurou o sobrecusto adicionado ao sistema quando um componente é marcado como reconfigurável. A Tabela 1 apresenta os resultados comparando o ELUS original com a versão modificada. A configuração mínima para o ELUS original é de cerca de 1.6 KB de memória, sendo que com a modificação realizada por este trabalho são necessários apenas 664 bytes para representar o mesmo componente.

Para verificar qual parte do *framework* foi a responsável por este ganho, foram realizadas medições mais específicas, desta vez considerando os métodos `create`, `destroy`, `update` e mais os quatro tipos possíveis de métodos (com parâmetro, sem parâmetro, com valor de retorno e sem valor de retorno), conforme pode também ser observado na Tabela 1.

[1] Microcontrolador Atmel Atmega128 de 8Mhz, 4KB de memória RAM, 128KB de memória flash, 4KB de EEPROM

Tabela 1. Consumo de memória dos métodos individuais em ambas versões do ELUS.

Método do Framework	ELUS original Tam. seção (bytes)		ELUS modificado Tam. seção (bytes)		Direfença
	.text	.data	.text	.data	(%)
Create	180	0	178	0	1,11
Destory	138	0	136	0	1,45
Método sem parâmetro e valor de retorno	94	0	90	0	4,25
Método com um parâmetro e sem valor de retorno	98	0	94	0	4,08
Método sem parâmetro e com valor de retorno	112	0	104	0	7,14
Método com um parâmetro e valor de retorno	126	0	122	0	3,17
Update	1250	0	260	0	79,20
Dispatcher	0	2 *(# métodos)	0	2*(# métodos)	0
Semaphore	0	18	0	18	0
Tamanho mínimo	1662	26	664	26	59,12

A Tabela mostra que o método `Update` é o maior responsável por esta diminuição. Este método se encontra na classe `Agent` que é a classe que continha uma grande parte de código que não utilizava informação de tipo, mas mesmo assim era replicada para cada componente marcado como reconfigurável.

4.2. Tempos de invocação de método

O tempo de invocação de métodos é avaliado em termos de quantidade de ciclos para realizar uma chamada de método e analisar se houve algum sobre custo neste processo. É uma avaliação importante porque o *framework* deve realizar uma série de atividades antes e depois da invocação de métodos. A Tabela 2 apresenta o tempo de invocação de métodos do ELUS original, modificado e os compara com a chamada de um método normal e através da tabela de métodos virtuais. Percebe-se claramente que há uma perda de desempenho causada pelo nível de indireção criado entre a aplicação e a invocação do método real através do *framework* metaprogramado.

Apesar de ser uma das partes substituídas para uma tentativa de melhora de desempenho, o resultado numérico das duas versões foi igual. Estudando o código gerado (e.g, *assembly*) para ambas versões pode-se concluir que houve diferença das instruções, mas após a recontagem do número de ciclos utilizados para a invocação de método, este número permaneceu sem alterações.

4.3. Tempo de Reconfiguração

A comparação dos tempos de reconfiguração foi medida em dois cenários: (i) o novo código do componente é menor ou igual ao antigo, sendo assim a atualização pode utilizar a mesma posição do componente e (ii) o novo código de um componente é maior que o

Tabela 2. Comparação dos tempos de invocação de método entre uma invocação normal, através da vtable, do ELUS e do ELUS modificado

Invocação	Normal	Vtable	ELUS	ELUS Modificado
Método sem parâmetro e sem valor de retorno	4	13	135	135
Método com parâmetro e sem valor de retorno	7	15	139	139
Método sem parâmetro e com valor de retorno	6	14	152	152
Método com parâmetro e com valor de retorno	8	16	161	161

antigo, em que se faz necessário a alocação de uma nova posição para o componente. Para ser uma comparação justa em relação aos trabalhos relacionados, esse teste não considerou o tempo de recebimento de dados pela rede e nem o tempo de escrita dos dados na memória de programa. O resultado do teste pode ser observado na Tabela 3.

Tabela 3. Comparação dos tempos de reconfiguração para um componente no ELUS e do ELUS modificado, medido em ciclos

Atualização	ELUS original	ELUS modificado
Mesma posição	362	368
Nova posição	414	397

Examinando o tempo de reconfiguração em relação ao ELUS original e os resultados obtidos pelas modificações é possível observar um aumento no tempo da reconfiguração quando o componente é colocado na mesma posição e uma diminuição do tempo em relação à atualização em uma nova posição. Embora este resultado pareça controverso, pode ser facilmente entendido se o código *assembly* gerado tanto pelo ELUS original quanto pelo ELUS modificado forem analisados. Devido à utilização da especialização para ponteiros *void* o código modificado inclui instruções de cálculo e retorno de um valor pela classe `Base`, que servirá para verificar se é necessário ou não a atualização do endereço da tabela virtual em cada objeto.

Já a diminuição do tempo da reconfiguração para uma nova posição pode ser associada à quantidade e à ordem das instruções do código modificado, pois a classe `Base` é responsável por todas as operações, com exceção da atualização do endereço da tabela virtual em cada objeto. O mesmo não ocorre com o ELUS antigo, que realiza este processo através da classe dependente de *template* e precisa realizar verificações relativas ao tipo.

5. Trabalhos relacionados

Contiki [Dunkels et al. 2004] tem seu *kernel* baseado em eventos com *multithreads* preemptivas para cada um dos processos, sendo que cada um deles possui uma interface *stub*, responsável por redirecionar as chamadas para a interface que possui ponteiros para as implementações das funções relativas ao serviço oferecido. Quando recebe uma mensagem de reconfiguração, o *kernel* disponibiliza uma interface para a passagem de ponteiros

entre as duas versões e, no caso de uma remoção de serviço, é enviado um evento no qual o serviço remove-se automaticamente do sistema. Contiki não possui o melhor desempenho para a utilização dos recursos, pois apenas para fornecer à reconfiguração dinâmica já é necessário cerca de 6KB de memória, além disso sua chamada de métodos possui vários ciclos de verificações que devem ser realizadas antes e depois da invocação.

SOS [Han et al. 2005] é composto por módulos atualizáveis em tempo de execução e programados de forma cooperativa, de tal modo que conseguem enviar mensagens e se comunicar com o *kernel* através de uma tabela com *jumps*. Para a atualização dos módulos existe um protocolo de distribuição que faz o *download* dos dados, também aloca memória para o novo módulo, além de ajustar os ponteiros na tabela e dar inicio ao processo através da mensagem `init`. Quando a atualização consiste em uma remoção de módulo, o *kernel* envia uma mensagem `final` para que o módulo libere todos os recursos que está utilizando e informe sua remoção aos módulos que o utilizam. Observando a utilização de recursos este sistema operacional utiliza cerca de 23KB e ainda possui uma grande perda de desempenho para a invocação de métodos, acrescentando mais de 850 ciclos no processo.

O sistema operacional RETOS pretende contornar de forma eficiente limitações de recursos e ainda atingir os seguintes objetivos: fornecer uma interface para programação *multithread*, ter uma abstração orientada à rede de sensores sem fio (RSSF), ser um sistema resiliente e possuir um *kernel* extensível através de reconfiguração dinâmica [Cha et al. 2007]. O RETOS fornece um mecanismo que utiliza relocação dinâmica de memória e ligação em tempo de execução para a reconfiguração dos módulos. Desta maneira, é capaz de gerar metadados a partir de variáveis e funções em tempo de compilação que são armazenados para posteriormente substituir os endereços utilizados pelo módulo quando o mesmo é carregado pelo sistema. Embora utilizar metadados seja uma boa alternativa para a atualização, existe um sobre custo associado à quantidade extra de dados enviados pela rede, o que resulta em uma maior utilização dos recursos energia e memória.

Já MOS (*MANTIS sensor OS*) [Bhatti et al. 2005] é um sistema *multithread* que possui suporte à reprogramação dinâmica e acesso remoto via *shell*. MOS consegue economizar energia através de uma implementação diferenciada do escalonador de eventos, em que é possível identificar quando as *threads* estão ativas e controlar o consumo de energia do microcontrolador. A reconfiguração dinâmica é implementada através de uma chamada para uma biblioteca do *kernel* do MOS e ela pode ser realizada em diversas granularidades, variando desde uma variável dentro de *thread* até o sistema operacional inteiro. O baixo consumo de recursos é uma característica forte no MOS, um exemplo é o tamanho total necessário para armazenar suas estruturas, pois, somando o tamanho do *kernel*, do escalonador de enventos e da rede são utilizados menos de 500 *bytes* de RAM e 14KB de memória de programa.

6. Considerações finais

Neste artigo apresentou-se um estudo do *framework* do ELUS e uma proposta de melhoria da desempenho da reconfiguração dinâmica realizada pela infraestrutura. A nova abordagem foi avaliada em termos de consumo de memória, tempo de invocação de métodos e tempo de reconfiguração. A especialização através de ponteiros *void* apresentou resultado positivo para o consumo de memória, pois foi possível reduzir cerca de 60% do

consumo e atualmente o tamanho mínimo para um novo componente do sistema passa a ser 664 KB. Já o tempo de invocação de métodos não sofreu alterações, mas ainda é positivo se comparado aos trabalhos relacionados. Por fim, o tempo de reconfiguração de componentes diminuiu cerca de 17 ciclos para um componente em uma nova posição, entretanto, as modificações também foram responsáveis pelo aumento em 6 ciclos no tempo de atualização de um componente na mesma posição.

Dentre as implementações futuras estão a medição do consumo de energia, que também é um recurso muito importante no domínio de sistemas embarcados e o aumento da reusabilidade do código.

Referências

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. (2005). Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. (2007). RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, page 157. ACM.

Dunkels, A. et al. (2004). Contiki-a lightweight and flexible operating system for tiny networked sensors.

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

Gracioli, G. and Fröhlich, A. (2009). ELUS: a Dynamic Software Reconfiguration Infrastructure for Embedded Systems.

Gracioli, G. and Fröhlich, A. (2010). ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In *Telecommunications (ICT), 2010 IEEE 17th International Conference on*, pages 981–988. IEEE.

Han, C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176. ACM.

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

APÊNDICE B – CÓDIGO FONTE

B.1 EPOS–UPDATE

B.1.1 INCLUDE/FRAMEWORK/INDIVIDUAL

```
1  #ifndef __adapter_h
2  #define __adapter_h
3
4  #include "scenario.h"
5  #include <aspects/id/global.h>
6  #include "scenariohash.h"
7
8  __BEGIN_SYS
9
10 template<class T>
11 class Adapter : public Scenario<T>, public T
12 {
13
14 private:
15     T *t;
16
17 public:
18     //Adapter(){ }
19     Adapter(){ t = new T(); }
20
21     template<typename T1>
22     Adapter(T1 a1) { t = new T(a1); }
23     template<class T1, class T2>
24     Adapter(T1 a1, T2 a2) { t = new T(a1, a2); }
25     template<class T1, class T2, class T3>
26     Adapter(T1 a1, T2 a2, T3 a3) { t = new T(a1, a2, a3); }
27     template<class T1, class T2, class T3, class T4>
28     Adapter(T1 a1, T2 a2, T3 a3, T4 a4) { t = new T(a1, a2, a3, a4); }
29
```

```
30      T * get_obj() { return t; }
31      void set_obj(T *obj) { t = obj; }
32
33      static void free(Adapter * adapter) {
34          Scenario<T>::free(adapter);
35      }
36
37      void * operator new(unsigned int size) {
38          return Scenario<T>::alloc();
39      }
40
41      static Adapter *get(const Id & id) {
42          return Scenario<T>::get((unsigned int)&(id.id()));
43      }
44
45      static Adapter *get(unsigned int id) {
46          return Scenario<T>::get(id);
47      }
48
49      static unsigned int *get_keys() {
50          return Scenario<T>::get_keys();
51      }
52
53      static Adapter *remove(const Id & id) {
54          Adapter *obj = Scenario<T>::remove(id);
55          return obj;
56      }
57
58      const Id & id() { return *this; }
59
60      // Test_Component
61      int sum() { return t->sum(); }
62      void abc() { t->abc(); }
63      void set_a(int v) { t->set_a(v); }
64      int setGet(int v) { return t->setGet(v); }
65
66  };
67
68  __END_SYS
69
70  #endif
```

code/adapter.h

```
1  #ifndef __agent_h
2  #define __agent_h
3
4  #include <cpu.h>
5  #include <machine.h>
6  #include <condition.h>
7  #include <mutex.h>
8  #include <clock.h>
9  #include <chronometer.h>
10 #include <test_component.h>
11 #include <utility/malloc.h>
12 #include <utility/string.h>
13 #include "message.h"
14 #include "stub.h"
15 #include <code_manager.h>
16 #include <system/framework_defs.h>
17
18 __BEGIN_SYS
19
20 #define MAX_METHODS 11
21 typedef void (Dispatcher)(Message *);
22 typedef Imp::Code_Manager Code_Manager;
23
24 extern void trapAgent(Message* msg);
25 extern Dispatcher *services[][MAX_METHODS];
26
27 template<>
28 class Agent<void*>{
29
30 public:
31     static unsigned int update(Message* msg, unsigned int *addr){
32         unsigned int *p = (unsigned int *)(*addr);
33         unsigned char mID, control_byte;
34         unsigned int size, total_size, code, old_size;
35         Message *msg_extra;
36         msg->in(control_byte, mID, total_size, code, msg_extra);
37
38         unsigned char *new_code = (unsigned char *) code;
39         unsigned int total_msgs = (control_byte & 0xF0) >> 4;
40         unsigned int n_addr = 0;
41
42         char zero = 0;
```

```
43          unsigned int displacement = 0;
44          unsigned int dispatcher_size = 0;
45
46
47          switch(control_byte) {
48          //Adding method without relocation - ETP Code 0x00
49          case 0x00:
50              //method's address inside the received code because it
                    also contains the framework code
51              msg_extra->in(displacement);
52              //alloc memory for the new method
53              n_addr = Code_Manager::alloc(size);
54              //write the new code into memory
55              Code_Manager::write(n_addr, new_code, size);
56              //add the new method into dispatcher
57              services[msg->id().type()][msg->method()] = (Dispatcher*)(
                    n_addr);
58
59              goto vtable_realloc;
60
61              break;
62              //Adding method with relocation - ETP Code 0x01
63          case 0x01:
64              //method's address inside the received code because it
                    also contains the framework code
65              msg_extra->in(displacement, dispatcher_size);
66              //alloc memory for the new method
67              n_addr = Code_Manager::alloc(size);
68              //write the new code into memory
69              Code_Manager::write(n_addr, new_code, size);
70
71              //TODO alocao de nova memoria para o novo dispatcher,
                    copia dos dados do velho para o novo e liberacao da
                    memoria antiga
72
73              goto vtable_realloc;
74              break;
75
76              //remove a method - ETP Code 0x02
77          case 0x02:
78              //release memory
79              Code_Manager::free((unsigned int*)(*(p+(mID*sizeof(
                    unsigned int)))), size);
```

```
80              break;
81              //put 0 on the vtable position
82              Code_Manager::write((*p+(mID*sizeof(unsigned int))), (
                    unsigned char *)&zero, sizeof(unsigned int));
83
84              //put 0 on the dispatcher position
85              services[msg->id().type()][mID] = (Dispatcher*) 0;
86
87              //no memory allocation needed for the new component's code
                    - ETP Code 0x03
88          case 0x03:
89              //put the new component's code in the same memory position
90              p += mID - 2;
91              Code_Manager::write((*p * 2), new_code, total_size);
92
93              //update the methods' address in vtable
94              for(unsigned int i = 1; i <= total_msgs; i++) {
95                  msg_extra->in(size, msg_extra);
96                  Code_Manager::write(*(p + i * sizeof(unsigned int)), (
                        unsigned char *)*(p - (i * sizeof(unsigned int))) +
                         size, sizeof(unsigned int));
97              }
98
99              break;
100             //memory allocation for the new component's code - ETP
                    Code 0x04
101         case 0x04:
102             msg_extra->in(old_size);
103             n_addr = Code_Manager::alloc(total_size);
104
105             //write new code
106             Code_Manager::write(n_addr, new_code, total_size);
107
108             //release the old memory location
109             Code_Manager::free((unsigned int *)(*(p + mID)), old_size)
                    ;
110
111             //update the methods' address in vtable
112             Code_Manager::write((*p), (unsigned char *) n_addr, sizeof
                    (unsigned int));
113             for(unsigned int i = 0; i <= total_msgs; i++) {
114                 msg_extra->in(size, msg_extra);
115                 Code_Manager::write(*(p + i * sizeof(unsigned int)), (
```

```
                    unsigned char *)*(p - (i * sizeof(unsigned int))) +
                     size, sizeof(unsigned int));
116             }
117             break;
118             //update the component inside the framework, the new code
                    is less or equal than the old
119         case 0x05:
120             Code_Manager::write((unsigned int)services[msg->id().type
                    ()][0], new_code, total_size);
121             break;
122
123             //update the methods' address in dispatcher
124             for(unsigned int i = 1; i <= total_msgs; i++) {
125                 msg_extra->in(size, msg_extra);
126                 unsigned int *a = (unsigned int *)(services[msg->id().
                        type()][i-1]);
127                 services[msg->id().type()][i] = (Dispatcher *) (a +
                        size);
128             }
129
130             //update the component inside the framework, the new code
                    is bigger than the old
131         case 0x06:
132             //alloc memory for the new component
133             n_addr = Code_Manager::alloc(total_size);
134
135             Code_Manager::write(n_addr, new_code, total_size);
136
137             //update the methods' address in dispatcher
138             services[msg->id().type()][0] = (Dispatcher *) n_addr;
139             for(unsigned int i = 1; i <= total_msgs; i++) {
140                 msg_extra->in(size, msg_extra);
141                 unsigned int *a = (unsigned int *)(services[msg->id().
                        type()][i-1]);
142                 services[msg->id().type()][i] = (Dispatcher *) (a +
                        size);
143             }
144
145             break;
146         }
147         goto exit;
148
149         vtable_realloc:
```

```
150             //alloc memory for the new vtable. msg->method() will have the
                    last position inside the new vtable.
151             n_addr = Code_Manager::alloc(msg->method() * sizeof(unsigned
                    int));
152
153             //copy from old to new vtable
154             for(unsigned int i = 0; i < msg->method()-1; i++) {
155                 Code_Manager::write((unsigned int )(addr + i * sizeof(
                        unsigned int)), (unsigned char *)(*(p + i)), sizeof(
                        unsigned int));
156             }
157
158             //put the new address on the new vtable
159             Code_Manager::write((unsigned int)(addr + msg->method() *
                    sizeof(unsigned int)), (unsigned char *) services[msg->id()
                    .type()][msg->method()] + displacement, sizeof(unsigned int
                    ));
160
161             //free the old vtable
162             Code_Manager::free(p, old_size);
163
164             return n_addr;
165
166             exit:
167             return 0;
168         }
169     };
170
171     template<class T>
172     class Agent : private Agent<void*> {
173         typedef Agent<void*> Base;
174         typedef Stub<T, false> _Stub;
175
176     public:
177
178         static void create(Message* msg) {
179             const _Stub* stub = new _Stub;
180             _Stub::add(stub, msg->id());
181         }
182
183         static void destroy(Message* msg) {
184             delete _Stub::remove(msg->id());
185         }
```

```
186
187
188         static void update(Message* msg){
189             unsigned int *addr = 0;
190             unsigned int n_addr = 0;
191             unsigned int *keys = _Stub::get_keys();
192             Adapter<T> *e;
193             T *t;
194
195             for(unsigned int i = 0; i < Adapter<T>::NUMBER_OBJS; i++) {
196                 if(keys[i] == 0) {
197                     continue;
198                 } else {
199                     e = _Stub::get(keys[i]);
200                     t = e->get_obj();
201                     if(T::TYPE == msg->id().type()) {
202                         addr = *reinterpret_cast<unsigned int**>(&t);
203                     }
204                     break;
205                 }
206             }
207
208             //get the vtable's address
209             n_addr = Base::update(msg, addr);
210             if(n_addr != 0){
211                 //update the vtable's address in each object
212                 for(unsigned int i = 0; i < Adapter<T>::NUMBER_OBJS; i++)
                        {
213                     if(keys[i] == 0) {
214                         continue;
215                     } else {
216                         e = _Stub::get(keys[i]);
217                         t = e->get_obj();
218                         if(T::TYPE == msg->id().type()) {
219                             addr = *reinterpret_cast<unsigned int**>(&t);
220                             addr[0] = n_addr;
221                         }
222                     }
223                 }
224             }
225         }
226
227             // Test_Component
```

```
228             static void sum(Message* msg) {
229                 msg->out(_Stub::get(msg->id())->sum());
230             }
231 
232             static void abc(Message* msg) {
233                 _Stub::get(msg->id())->abc();
234             }
235 
236             static void set_a(Message* msg) {
237                 int e; msg->in(e);
238                 _Stub::get(msg->id())->set_a(e);
239             }
240             static void setGet(Message* msg) {
241                 int e; msg->in(e);
242                 msg->out(_Stub::get(msg->id())->setGet(e));
243             }
244         };
245 
246 
247         __END_SYS
248 
249     #endif
```

code/agent.h

```
1   #ifndef __scenario_h
2   #define __scenario_h
3   
4   #include <traits.h>
5   #include <aspects/shared/shared.h>
6   #include <aspects/id/global.h>
7   #include <utility/hash.h>
8   #include "scenariohash.h"
9   
10  __BEGIN_SYS
11  
12  template<>
13  class Scenario<void*>{
14  public:
15          static void add(const void *e, unsigned int id) { hash.insert(
                e, id); }
16  
17  protected:
```

```
18          static void free(void* *obj){ delete obj; }
19
20          static void *get(unsigned int id){return hash.get(id); }
21
22          static unsigned int *get_keys() {return hash.get_keys();}
23
24          static void *remove(unsigned int id){
25              void* a = hash.get(id);
26              hash.remove(id);
27              return a;
28          }
29
30  private:
31      static ScenarioHash hash;
32
33  };
34
35  template<class T>
36  class Scenario : public Id, private __IMP(Shared)<__IMP(Traits)<T>::
        shared>, private Scenario<void*>
37  {
38  public:
39      static const unsigned int NUMBER_OBJS = ScenarioHash::NUMBER_OBJS;
40      typedef Scenario<void*> Base;
41
42  protected:
43      Scenario() {}
44
45      static Adapter<T>* alloc()
46      { return (Adapter<T>*)::operator new(sizeof(Adapter<T>)); }
47
48      static Adapter<T>* get(unsigned int id){
49          Adapter<T> *e = reinterpret_cast<Adapter<T> *>(Base::get(id));
50          return e;
51      }
52
53      static Adapter<T>* remove(const Id & id){
54          Adapter<T> *e = reinterpret_cast<Adapter<T> *>(Base::remove((
                unsigned int)&(id.id())));
55          return e;
56      }
57
58      static unsigned int *get_keys() {return Base::get_keys();}
```

```
59
60      void enter() {
61          __IMP(Shared)<__IMP(Traits)<T>::shared>::enter();
62      }
63
64      void leave() {
65          __IMP(Shared)<__IMP(Traits)<T>::shared>::leave();
66      }
67
68  public:
69      static void add(const Adapter<T> *e, const Id & id) {Base::add(e,
            (unsigned int)&(id.id())); }
70
71  };
72
73  __END_SYS
74
75  #endif
```

code/scenario.h

```
1   #ifndef __scenariohash_h
2   #define __scenariohash_h
3
4   #include <utility/hash.h>
5
6   __BEGIN_SYS
7
8   class ScenarioHash {
9   public:
10      typedef void * Hash;
11      static const unsigned int NUMBER_OBJS = 12;
12  public:
13      ScenarioHash() {
14          for(unsigned int i = 0; i < NUMBER_OBJS; i++) {
15              objects[i] = 0;
16              keys[i] = 0;
17          }
18      }
19
20      static void *get(unsigned int id) {
21          for(unsigned int i = 0; i < NUMBER_OBJS; i++) {
22              if(keys[i] == id) {
```

```
23                  return objects[i];
24              }
25          }
26          return 0;
27      }
28
29      static unsigned int *get_keys() { return keys; }
30
31      template <typename E>
32      static void insert(E *e, unsigned int id) {
33          for(unsigned int i = 0; i < NUMBER_OBJS; i++) {
34              if(objects[i] == 0) {
35                  objects[i] = (void *) e;
36                  keys[i] = id;
37                  size++;
38                  break;
39              }
40          }
41      }
42
43      static void remove(unsigned int id) {
44          for(unsigned int i = 0; i < NUMBER_OBJS; i++) {
45              if(keys[i] == id) {
46                  objects[i] = 0;
47                  keys[i] = 0;
48                  size--;
49                  break;
50              }
51          }
52      }
53
54  private:
55      static Hash objects[NUMBER_OBJS];
56      static unsigned int keys[NUMBER_OBJS];
57      static unsigned int size;
58  };
59
60  __END_SYS
61
62  #endif
```

code/scenariohash.h

B.1.2 SRC/FRAMEWORK

```
#include <system/config.h>
#include <framework/individual/message.h>
#include <framework/individual/agent.h>
#include <semaphore.h>

__BEGIN_SYS

static Imp::Semaphore quiescent_state[LAST_TYPE_ID + 1];

Dispatcher* services[LAST_TYPE_ID + 1][MAX_METHODS] = {
        { &Agent<__IMP(Test_Component)>::create,
          &Agent<__IMP(Test_Component)>::destroy,
          &Agent<__IMP(Test_Component)>::sum,
          &Agent<__IMP(Test_Component)>::abc,
          &Agent<__IMP(Test_Component)>::set_a,
          &Agent<__IMP(Test_Component)>::setGet,
          &Agent<__IMP(Test_Component)>::update },

};

void trapAgent(Message* msg) {
    Dispatcher *tmp;
    tmp = *services[msg->id().type()][msg->method()];
    quiescent_state[msg->id().type()].p();
    tmp(msg);
    quiescent_state[msg->id().type()].v();
}

__END_SYS
```

code/agent.cc

B.1.3 APP

```
#include <utility/ostream.h>
#include <framework/framework.h>
#include <network.h>
#include <channel.h>
#include <alarm.h>

```

```
7  #include <remote_thread.h>
8  #include <system/config.h>
9  #include <framework/individual/message.h>
10 #include <framework/individual/agent.h>
11 #include <application_update.h>
12
13 __USING_SYS
14
15 OStream cout;
16
17
18 int main()
19 {
20     cout << "\n TESTE \n";
21
22     Test_Component *t = new Test_Component();
23     int x = t->sum();
24     t->abc();
25     t->set_a(10);
26     int y = t->setGet(0);
27
28     int z = x + y; // 21
29     cout << "z = " << z << "\n";
30
31    Test_Component *t1 = new Test_Component();
32     int x1 = t1->sum(); // _a = 10; x1 = 11
33     t1->abc(); // _b = 10
34     t1->set_a(15); // _a = 15
35     int x2 = t1->setGet(0); // y = 15 ; _a = 0
36
37     int x3 = x1+x2; //x3 = 11+ 15 = 26
38     cout << "x3 = " << x3 << "\n";
39
40     Chronometer *c = new Chronometer();
41     c->start();
42         for(int i = 0; i < 7; i++ ) {
43             Alarm::delay(1000000);
44             cout << " i = " << i << "\n";
45         }
46      cout << "tempo = " << c->read() << "\n";
47
48
49
```

```
50
51      cout << ":::test2 \n";
52      char data[6];
53      data[0]  = 0x81;
54      data[1]  = 0xe0;
55      data[2]  = 0x90;
56      data[3]  = 0xe0;
57      data[4]  = 0x08;
58      data[5]  = 0x95;
59
60      char method = 0x02;
61      int size = 0x06;
62
63      Message msg(__SYS(TEST_COMPONENT), 0);
64      msg.method(__SYS(TEST_COMPONENT_UPDATE));
65      msg.out((char) 0x03, (char) method, (unsigned int) size, (unsigned
            char *) data);
66      trapAgent(&msg);
67
68      x = t->sum();
69      t->abc();
70      t->set_a(10);
71      y = t->setGet(0);
72
73      z = x + y; //11
74
75      cout << "z = " << (int)z << "\n";
76
77
78  }
```

code/teste.cc

REFERÊNCIAS

BALANI, R. et al. Multi-level software reconfiguration for sensor networks. In: ACM. *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. [S.l.], 2006. p. 121.

BOULIS, A. et al. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive and Mobile Computing*, Elsevier, v. 3, n. 4, p. 386–412, 2007.

CHA, H. et al. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In: ACM. *Proceedings of the 6th international conference on Information processing in sensor networks*. [S.l.], 2007. p. 157.

COUPAYE, T.; STEFANI, J. Fractal component-based software engineering. In: SPRINGER-VERLAG. *Proceedings of the 2006 conference on Object-oriented technology: ECOOP 2006 workshop reader*. [S.l.], 2006. p. 117–129.

DUNKELS, A. et al. Contiki-a lightweight and flexible operating system for tiny networked sensors. Published by the IEEE Computer Society, 2004.

FELSER, M. et al. Dynamic software update of resource-constrained distributed embedded systems. *Embedded System Design: Topics, Techniques and Trends*, Springer, p. 387–400, 2007.

FRÖHLICH, A. EPOS: Embedded Parallel Operating System. Citeseer, 2002.

GRACIOLI, G.; FRÖHLICH, A. ELUS: a Dynamic Software Reconfiguration Infrastructure for Embedded Systems. 2009.

GRACIOLI, G.; FRÖHLICH, A. ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In: IEEE. *Telecommunications (ICT), 2010 IEEE 17th International Conference on*. [S.l.], 2010. p. 981–988.

HAN, C. et al. A dynamic operating system for sensor nodes. In: ACM. *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. [S.l.], 2005. p. 163–176.

HILL, J. et al. System architecture directions for networked sensors. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 11, p. 93–104, 2000. ISSN 0362-1340.

KOSHY, J.; PANDEY, R. Remote incremental linking for energy-efficient reprogramming of sensor networks. In: CITESEER. *European Workshop on Wireless Sensor Networks*. [S.l.], 2005. p. 354–365.

KOSHY, J.; PANDEY, R. VMSTAR: synthesizing scalable runtime environments for sensor networks. In: ACM. [S.l.], 2005. p. 254.

KYLE, D.; BRUSTOLONI, J. UClinux: a linux security module for trusted-computing-based usage controls enforcement. In: ACM. *Proceedings of the 2007 ACM workshop on Scalable trusted computing*. [S.l.], 2007. p. 70.

LEVIS, P.; CULLER, D. Mate: A tiny virtual machine for sensor networks. *ACM SIGARCH Computer Architecture News*, ACM, v. 30, n. 5, p. 95, 2002.

LEVIS, P.; GAY, D.; CULLER, D. Active sensor networks. In: USENIX ASSOCIATION. *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. [S.l.], 2005. p. 356.

MARRÓN, P. et al. Flexcup: A flexible and efficient code update mechanism for sensor networks. *Wireless Sensor Networks*, Springer, p. 212–227, 2006.

MARRÓN, P. et al. TinyCubus: a flexible and adaptive framework sensor networks. In: *Proceedings of the 2nd European Workshop on Wireless Sensor Networks*. [S.l.: s.n.], 2005. p. 278–289.

POLAKOVIC, J.; STEFANI, J. Architecting reconfigurable component-based operating systems. *Journal of Systems Architecture*, Elsevier, v. 54, n. 6, p. 562–575, 2008.

REIJERS, N.; LANGENDOEN, K. Efficient code distribution in wireless sensor networks. In: ACM. *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. [S.l.], 2003. p. 60–67.

STROUSTRUP, B. *C++ Programming Language, The (3rd Edition)*. [S.l.]: Addison-Wesley Professional, 1997. ISBN 0201327554.

WILLIAMS, J.; BERGMANN, N. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In: CITESEER. *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*. [S.l.], 2004.

XIE, Q.; LIU, J.; CHOU, P. Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In: *The Fifth International Conference on Information Processing in Sensor Networks, 2006. IPSN 2006*. [S.l.: s.n.], 2006. p. 342–349.

YI, S. et al. Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems. *Computer Communications*, Elsevier, v. 31, n. 4, p. 699–707, 2008.