UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA/ INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MECATRÔNICA

NEIMA PRADO DOS SANTOS

RECONFIGURAÇÃO DINÂMICA EM SISTEMAS DISTRIBUÍDOS DE TEMPO REAL BASEADOS EM COMPONENTES

Salvador
2011

NEIMA PRADO DOS SANTOS

RECONFIGURAÇÃO DINÂMICA EM SISTEMAS DISTRIBUÍDOS DE TEMPO REAL BASEADOS EM COMPONENTES

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Escola Politécnica e do Instituto de Matemática da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo
Co-orientador: Prof. Dr. Luciano Porto Barreto

Salvador
2011

Sistema de Bibliotecas da UFBA

Santos, Neima Prado dos.
Reconfiguração dinâmica em sistemas distribuídos de tempo real baseados em componentes / Neima Prado dos Santos. - 2011.
167 f. : il.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.
Co-orientador: Prof. Dr. Luciano Porto Barreto.
Dissertação (mestrado) - Universidade Federal da Bahia, Escola Politécnica, Instituto de Matemática, Salvador, 2011.

1. Sistemas operacionais distribuídos (Computadores). 2. Controle em tempo real. I. Macêdo, Raimundo José de Araújo. II. Barreto, Luciano Porto. III. Universidade Federal da Bahia. Escola Politécnica. IV. Universidade Federal da Bahia. Instituto de Matemática. V. Título.

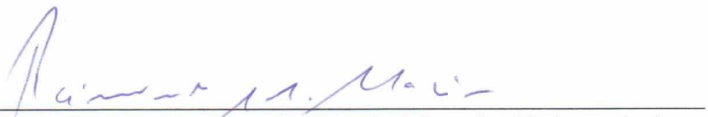
CDD - 004.33
CDU - 681.3.014

TERMO DE APROVAÇÃO

NEIMA PRADO DOS SANTOS

RECONFIGURAÇÃO DINÂMICA EM SISTEMAS DISTRIBUÍDOS DE TEMPO REAL BASEADOS EM COMPONENTES

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia, pela seguinte banca examinadora:



Prof. Dr. Raimundo Jose de Araujo Macedo (Orientador)
Ph.D. em Ciência da Computação, University of Newcastle Upon Tyne, Inglaterra
Professor Titular do Departamento de Ciência da Computação da Universidade Federal da Bahia

Prof. Dr. Renato Fontoura de Gusmão Cerqueira
Doutor em Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brasil
Professor Assistente do Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro

Prof. Dr. Flávio Morais de Assis Silva
Doutor em Informática, Technische Universitat Berlin, Alemanha.
Professor Adjunto do Departamento de Ciência da Computação da Universidade Federal da Bahia

Prof. Dr. Claudio Sant'Anna
Doutor em Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brasil
Professor Adjunto do Departamento de Ciência da Computação da Universidade Federal da Bahia

Salvador, 12 de maio de 2011

Às minhas queridas mãe e irmã, de quem incentivo e presença fazem tudo valer a pena.

AGRADECIMENTOS

Primeiramente a Deus, por prover as condições para a realização deste trabalho, na companhia de familiares, professores e amigos tão especiais.

À minha mãe, Márcia, e minha irmã, Vanessa, pelo amor e o apoio incondicionais.

A meus orientadores Raimundo José de Araújo Macêdo e Luciano Porto Barreto, pela compreensão, o incentivo, a atenção e a generosidade em compartilhar seus conhecimentos e experiências.

A meu chefe Ed Rey Carneiro Brito, por entender as necessidades especiais advindas da execução desta tarefa.

Aos colegas Sandro Santos Andrade e Alírio Santos de Sá pelo companheirismo e a disposição em ajudar, Paulo César Rocha Chaves, pela amizade desinteressada e o ânimo contagiante, e Frederico Jorge Ribeiro Barboza, pelo encorajamento e o estímulo a continuar investindo na vida acadêmica.

Aos professores do Programa de Pós-Graduação em Mecatrônica (PPGM) da Universidade Federal da Bahia, cujos ensinamentos e apoio foram primordiais à realização deste trabalho, especialmente a Flávio de Assis Silva e Herman Lepikson.

Aos colegas e aos funcionários do PPGM, em especial às secretarias Lúcia Lago e Maria do Socorro de Oliveira, pela competência e o tratamento prestativo.

Comemorar somente as maiores vitórias é deixar de valorizar o caminho para alcançá-las.

(Autor Desconhecido)

RESUMO

Sistemas modernos de controle e supervisão são caracterizados pela utilização de componentes de prateleira (*Commercial off-the-shelf* - COTS), como componentes de rede, sistemas operacionais e *middlewares*, visando à otimização de custos e reutilização. Um dos aspectos cruciais em tais sistemas é sua capacidade de se adaptar a mudanças no ambiente computacional ou novos requisitos da aplicação.

A plataforma ARCOS - *Architecture for Control and Supervision*, baseada em componentes de tempo-real, foi desenvolvida no Laboratório de Sistemas Distribuídos (LaSiD), no âmbito do Programa de Pós-Graduaçào em Mecatrônica da Universidade Federal da Bahia, para atender a essas novas demandas de sistemas industriais de controle e supervisão.

Na presente dissertação, é apresentado o serviço de reconfiguração dinâmica desenvolvido para sistemas baseados em componentes, que permite modificar a composição de uma aplicação em tempo de execução, procurando minimizar o impacto em partes do sistema não envolvidas na reconfiguração em questão. É descrita a utilização desse serviço na reconfiguração de duas variações de uma aplicação de controle baseada em componentes: uma delas desenvolvida sem o emprego do ARCOS e uma outra em que esse *framework* é empregado. Adicionalmente, são apresentados experimentos realizados com o objetivo de avaliar o impacto da reconfiguração nas aplicações alvo.

Palavras-chave: Sistemas de tempo real; Componentes distribuídos; Sistemas de controle.

ABSTRACT

Modern supervisory and control systems are characterized by the use of commercial off-the-shelf components, such as networks, operational systems and middlewares, aiming at cost reducing and reuse. One of the most important aspects of such systems is its capacity to adapt to computing environmental changes or new application requirements. The platform ARCOS – Architecture for Control and Supervision, real-time component based, was developed on the Distributed Systems Laboratory (LaSiD) to deal with these new demands of industrial supervisory and control systems.

In this dissertation, the dynamic reconfiguration service for component-based systems is presented. This service allows to modify the composition of an application at runtime, aiming at reducing the impact of the changes on parts of the system that are not involved in the reconfiguration. The use of this service is also described for the reconfiguration of two variants of a component-based control application, one of them developed with ARCOS and the other without the employment of that framework. The results of the experiments performed to evaluate the influence of the reconfiguration on the target applications are presented too.

Keywords: Realtime systems; Distributed components; Control Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

Introdução

1. 1 Motivação e Objetivos

À medida que os sistemas computacionais se tornam cada vez mais abrangentes e complexos, torna-se fundamental a utilização de tecnologias e padrões que contribuam para diminuição do seu tempo de desenvolvimento e promovam redução de custos, sem para tanto negligenciar seus requisitos de confiabilidade. Nos sistemas mecatrônicos, em que o *software* exerce papel importante, integrado a dispositivos eletromecânicos no controle de aplicações críticas, tais como controle de tráfego e sistemas de energia e telecomunicação, evidencia-se a necessidade de mecanismos que permitam alterar a sua configuração em tempo de execução, garantindo o funcionamento correto desses sistemas durante o maior período de tempo possível, mesmo que de forma degradada, sob pena de acarretar graves perdas financeiras ou humanas.

Tipicamente, sistemas de controle são constituídos por sensores, atuadores, controlador, planta, canal de comunicação e possivelmente uma interface de operação. Uma planta é uma denominação genérica de um sistema físico, que consiste em um conjunto de objetos físicos ou inorgânicos que formam um determinado aparelho, e cujo comportamento depende do comportamento de cada um dos objetos do sistema, como é o caso das máquinas e

processos industriais (PAZOS, 2003). O controlador monitora o estado real do processo de uma planta através de um número de sensores. Com base nas leituras dos sensores, o controlador utiliza um algoritmo de controle para calcular suas saídas, que são enviadas aos atuadores. O operador pode interagir com o controlador através de uma interface de operação. As informações trocadas entre os diversos componentes fluem através do canal de comunicação.

Os mecanismos empregados para desempenhar as funções de controle nos sistemas têm evoluído de dispositivos mecânicos, relés, contatos e circuitos integrados (CI) a computadores. O fato de a funcionalidade dos sistemas ser implementada inicialmente através de *hardware* restringia sua flexibilidade, tornando sua alteração difícil. Em contrapartida, computadores digitais estão cada vez mais robustos, baratos, rápidos e compactos, podem ser programados e re-programados para executar tarefas variadas e com alto grau de complexidade e podem ser associados a outros computadores, possibilitando a distribuição da complexidade, melhor aproveitamento de recursos, especialização e tolerância a falhas. Tais características os têm projetado como o principal mecanismo de controle nos sistemas físicos (AUSLANDER, 1996).

Em plantas industriais de larga escala é necessário distribuir a função de controle entre vários componentes. A principal motivação para essa distribuição é o fato de os computadores precisarem estar próximos ao processo de forma a assegurar pronta reação às suas mudanças (SANS e outros, 2003). Dentre outras razões encontram-se: disponibilidade de processadores embutidos adequados, requisitos temporais de comunicação, necessidade de aumento de desempenho através de paralelismo, simplificação das tarefas de construção e manutenção através de modularidade, redução de custo e tempo de desenvolvimento através de reutilização de componentes, integração de sistemas legados, disponibilidade de plataformas de *software* específicas.

Em sistemas distribuídos tais como sistemas militares, veículos automotivos e plantas industriais, as informações podem ser originadas de várias fontes e consumidas em vários destinos. Por exemplo, na indústria automotiva, alguns sistemas modernos de ABS apresentam um computador dedicado em cada roda que troca informações de frenagem com os demais. Além disso, os sistemas de suspensão, gerenciamento de energia e controle do motor trabalham de forma integrada com o ABS (*Anti-lock Braking System*) para manter o automóvel sob controle (SANS e outros, 2003).

Além de demandar garantia de atendimento aos requisitos temporais, muitas das aplicações de controle precisam operar ininterruptamente, não suportando mecanismos convencionais de atualização e manutenção. Como exemplo de tais aplicações pode-se citar: sistemas militares de defesa, veículos não tripulados, sistemas de energia e telecomunicação. A crescente dependência da sociedade em sistemas controlados por *software* tem levado à necessidade de sistemas confiáveis, robustos e continuamente disponíveis. A capacidade de adaptar um sistema em tempo de execução é um aspecto crítico para garantir seu funcionamento ininterrupto (OREIZY, 1998). Mudanças evolutivas são difíceis de serem previstas em tempo de projeto, conseqüentemente, os sistemas precisam ser suficientemente flexíveis para permitir mudanças incrementais arbitrárias (KRAMER; MAGEE, 1985). Para tanto, é primordial a existência de um mecanismo para alteração de sua configuração em tempo de execução que permita ao sistema continuar funcionando de forma consistente, durante e após a adaptação. Essa reconfiguração efetuada durante o tempo de execução do sistema é dita dinâmica, e pode ser empregada para permitir que sistema se recupere de uma falha, adequar seu desempenho às mudanças percebidas no ambiente ou gerenciar os recursos disponíveis, possivelmente restritos, de acordo com as condições do ambiente. Além disso, possibilita manutenção corretiva e evolutiva minimizando o tempo em que o sistema permanece indisponível. No outro tipo de reconfiguração, denominada estática, componentes

são combinados durante a compilação para produzir uma aplicação, de acordo com critérios predefinidos.

As facilidades providas pela tecnologia de componentes tornam seu emprego promissor nas demandas de disponibilidade, flexibilidade, confiabilidade e extensibilidade dos sistemas. Sistemas distribuídos oferecem potencialmente um ambiente flexível para sua modificação e extensão (KRAMER; MAGEE, 1985). Componentes de *software* possuem baixo acoplamento e suas dependências são explícitas e visíveis, o que favorece o processo de mudança de configuração em tempo de execução (HILMAN; WARREN, 2004). Entretanto, faz-se necessária a disponibilização de mecanismos que permitam gerenciar e coordenar as alterações do sistema em tempo de execução, que se beneficiem das funcionalidades providas pela tecnologia de componentes, ao mesmo tempo em que preservem sua integridade, tanto na dimensão funcional quanto não funcional. Tal necessidade evidencia-se através da iniciativa do *Object Management Group* (OMG) que, reconhecendo que sistemas computacionais em áreas como finanças, negócios, controle de processos, dentre outras, devem estar disponíveis ininterruptamente aos seus usuários, elaborou um padrão de Atualização Online (*Online Upgrades*) para permitir que servidores CORBA sejam atualizados de forma ordenada e segura, sem que precisem ser desligados para tanto (OMG, 2003).

No presente trabalho, é proposta uma arquitetura para um serviço de reconfiguração a ser utilizado por aplicações baseadas em componentes, particularmente àquelas voltadas aos sistemas de controle e supervisão industriais. Há uma separação entre os mecanismos de reconfiguração e consistência, havendo possibilidade de utilização de diferentes abordagens para preservação de consistência, quando necessário. Além disso, a estrutura do próprio serviço é baseada em componentes, favorecendo sua integração com outros serviços e até mesmo sua própria reconfiguração.

A arquitetura descrita foi implementada utilizando o CIAO, uma implementação do modelo de componentes CORBA, com suporte a *tempo real*, para ser integrada ao ARCOS (ARquitetura de Controle e Supervisão) (ANDRADE; MACÊDO, 2007), um *framework* baseado em componentes, também desenvolvido sobre o CIAO, que objetiva prover uma plataforma extensível para sistemas de supervisão e controle, abordando aspectos como interoperabilidade e mecanismos para garantias temporais.

Com a finalidade de estudar o impacto da reconfiguração na aplicação, através da comparação de seu desempenho fim-a-fim, com e sem a ocorrência de reconfiguração, foram criadas duas variações de uma aplicação de controle veicular, assim como definidas métricas a serem empregadas na análise do desempenho obtido, de modo a investigar a viabilidade da reconfiguração dinâmica em aplicações do domínio em foco.

1.2 Estrutura da dissertação

O conteúdo desse documento está distribuído em seis capítulos, sendo o primeiro deles a introdução.

O capítulo dois discorre sobre *Middleware* para Sistemas Distribuídos de *Tempo Real*, tecnologia que provê aos sistemas a infra-estrutura necessária para a comunicação em ambientes distribuídos e mecanismos para garantia dos requisitos temporais das aplicações. Os *middlewares* TAO e CIAO, utilizados na implementação do serviço apresentado neste trabalho, são descritos, bem como o CORBA e seu modelo de componentes, já que os primeiros se baseiam na sua especificação. No capítulo também são apresentadas algumas abordagens para escalonamento de tarefas em modelos de sistema variados.

A reconfiguração dinâmica em sistemas distribuídos de *tempo real* é discutida no capítulo três. São descritas algumas abordagens de preservação de consistência, dentre elas a

adotada na implementação do serviço aqui detalhado. Além disso, são apresentadas tecnologias relacionadas à reconfiguração dinâmica: separação de concernência, reflexão computacional, desenvolvimento baseado em componentes, padrões de projeto e gerenciamento de recursos.

O capítulo quatro é sobre a Arquitetura de Controle e Supervisão (ARCOS) e versa sobre sua estrutura e funcionamento.

No capítulo cinco, é detalhado o serviço de reconfiguração dinâmica, objeto deste trabalho. São abordadas a arquitetura do serviço e sua implementação.

O capítulo seis é dedicado à apresentação dos testes realizados a partir da execução de duas variações de uma aplicação de controle veicular, uma delas utilizando o ARCOS e a outra implementada diretamente sobre o CIAO (*Component-Integrated* ACE ORB).

No capítulo sete discorre-se sobre os trabalhos relacionados e, finalmente, o capítulo oito é reservado à conclusão e sugestão de trabalhos a serem realizados no futuro.

Capítulo 2

Middleware para Sistemas Distribuídos de *Tempo Real*

Considerando que a preocupação com os requisitos temporais nos sistemas de *tempo real* é um fator crucial, e que na implementação do serviço de reconfiguração dinâmica e das aplicações deste trabalho foi empregado um *middleware* com foco nesse tipo de sistema (CIAO), no presente capítulo são apresentados alguns conceitos e tecnologias intrínsecos a esse domínio.

2.1 Sistemas Distribuídos de *Tempo Real*

Sistemas de *tempo real* são um caso especial de sistemas computacionais nos quais o computador interage diretamente com o ambiente, não somente por meio de uma interface homem-máquina, mas, sobretudo, através de dispositivos que captem informações do ambiente e nele interfiram, denominados sensores e atuadores, respectivamente (MACÊDO e outros, 2004). Nesses sistemas, um objeto controlado e um sistema de controle interagem através da utilização de sensores e atuadores (KOPETZ, 1999).

Um sistema de *tempo real* que controla um dispositivo físico ou um processo precisa interagir com um ambiente que tem propriedades temporais variadas, exibir comportamento previsível e executar em um sistema com recursos limitados (BURNS, 2002).

Além disso, os sistemas de tempo-real da atualidade precisam ainda lidar com a distribuição e heterogeneidade dos seus componentes, o que torna o seu desenvolvimento uma tarefa ainda mais complexa.

Uma vez que é objetivo de tais sistemas controlar um determinado dispositivo ou processo, seus requisitos são ditados pelo objeto controlado. Dessa forma, não basta que o sistema atenda aos requisitos funcionais da aplicação fornecendo resultados computacionais corretos, é preciso também que os requisitos temporais sejam atendidos. Em sistemas de *tempo real* a resposta só pode ser considerada correta se, além de logicamente exata, esta for gerada e processada em um intervalo de tempo determinado.

Sistemas de *tempo real* devem produzir resultados em um intervalo de tempo determinado pelo ambiente denominado *deadline*. Se o resultado não tem utilidade depois de passado o *deadline*, este é denominado firme (*firm*), caso contrário é dito suave (*soft*). Se o não cumprimento de um *deadline* firme pode resultar em catástrofe, o *deadline* é classificado como rígido (*hard*) (KOPETZ, 2000). Aqueles sistemas que precisam cumprir pelo menos um *deadline* rígido são classificados de críticos (*hard*), como é o caso dos sistemas de controle de plantas nucleares, controle de vôo e monitoramento médico. Caso contrário, os sistemas são classificados como suaves (*soft*).

Sistemas de *tempo real* podem ainda ser classificados de acordo com seu modelo de ativação de tarefas. Na abordagem acionada por tempo (*time-triggered*) as tarefas são ativadas pela passagem do tempo, sendo que o sistema computacional as inicia em um instante pré-definido. No modelo acionado por eventos (*event-triggered*) a ativação das tarefas é efetuada com a percepção de um evento pelo sistema (KOPETZ, 1993).

Grande parte das aplicações de *tempo real* atuais é composta por componentes heterogêneos, distribuídos através de plataformas diversas e sistemas de comunicação. Por exemplo: sistemas de controle geralmente são constituídos por vários componentes, cada qual

executando funções específicas, que precisam ser integradas. Tais componentes incluem processadores, sensores, atuadores, redes de comunicação, entre outros (HECK; WILLS; VACHTSEVANOS, 2001).

Dentre as vantagens identificadas com essa distribuição pode-se citar: compartilhamento de recursos, suporte à tolerância a falhas através de replicação, melhor aproveitamento de recursos, concorrência, desempenho aumentado e extensibilidade. Entretanto, apesar dos benefícios apontados, o desenvolvimento de tais aplicações apresenta complicações não existentes em sistemas centralizados. Em aplicações distribuídas é preciso lidar com a heterogeneidade de *software* e *hardware*, distribuição de componentes, independência de falhas, concorrência, segurança e comunicação. Além disso, em sistemas de *tempo real*, os requisitos temporais precisam ser atendidos. Tais fatores tornam o desenvolvimento de sistemas distribuídos de *tempo real* uma atividade bastante complexa e suscetível a erros.

Tradicionalmente, essa complexidade tem sido abordada através da utilização de tecnologias proprietárias de *hardware* e *software*, muito específicas de cada aplicação, tornando-as muito dependentes de *hardware*, prejudicando a flexibilidade e possibilidade de reutilização, além de dificultar a extensibilidade. A tecnologia de *middleware* se propõe a amenizar essas limitações, na medida em que fornece uma infra-estrutura padronizada e interoperável para o desenvolvimento de aplicações distribuídas, contribuindo para liberar os desenvolvedores de aplicações dos detalhes de baixo nível das plataformas, provendo interfaces padronizadas de alto nível para gerenciamento de recursos do sistema e diminuindo custos de desenvolvimento através de *frameworks* reutilizáveis (WANG e outros, 2003).

2.2 Escalonamento em Sistemas de *Tempo Real*

Uma vez que a previsibilidade é um requisito importante em sistemas de *tempo real*, é preciso garantir que os recursos do sistema sejam alocados de forma a atender à suas demandas, sob pena de inviabilizar o atendimento dos seus requisitos temporais. Para tanto, faz-se necessário organizar e ordenar as tarefas do sistema, de forma a conseguir o máximo aproveitamento dos recursos, sem, no entanto, pôr em risco o seu funcionamento correto, seja no aspecto funcional ou temporal. Processos ou tarefas formam a unidade lógica de computação no processador. Um programa tipicamente consiste de vários processos, cada qual com sua *thread* de controle (AUDSLEY e outros, 1995). Naturalmente, se existem tarefas competindo por recursos do sistema, é necessário haver uma forma de organizar o acesso a eles.

O papel de gerenciar a execução das tarefas do sistema cabe ao escalonador, que é responsável por distribuir os recursos computacionais necessários ao funcionamento do sistema respeitando suas restrições temporais (MACÊDO e outros, 2004). Ele é o componente do sistema que gerencia o processador, em tempo de execução, e implementa uma política de escalonamento ao ordenar para execução um conjunto de tarefas (FARINES; FRAGA; OLIVEIRA, 2000). Os escalonadores produzem uma escala (*scheduling*) para um conjunto de processos, sendo que, se o conjunto de processos pode ser escalonado para atender a determinadas precondições então o conjunto é dito praticável (*feasible*). Uma condição típica para processos periódicos de *tempo real* críticos é a de que sempre cumpram seus *deadlines* (SHAW, 2003; AUDSLEY e outros,1995).

Tarefas podem ser periódicas, aperiódicas ou esporádicas. Tarefas periódicas consistem de invocações idênticas que acontecem em intervalos fixos. Tarefas aperiódicas consistem em uma seqüência de invocações que chegam aleatoriamente, geralmente em resposta a algum evento externo. Tarefas esporádicas são um caso especial de tarefas

aperiódicas em que é possível determinar um intervalo mínimo entre duas ativações consecutivas (FIDGE, 2002).

Algoritmos de escalonamento podem atribuir prioridades às tarefas de forma estática ou dinâmica. Na modalidade estática, há necessidade de conhecer todas as tarefas do sistema, uma vez que suas prioridades são atribuídas previamente à execução do mesmo. No método dinâmico, as prioridades são determinadas em tempo de execução, o que incute mais flexibilidade ao sistema, possibilitando lidar com eventos imprevistos, embora incorrendo em maior sobrecarga.

Os escalonadores podem ser categorizados em *on-line* e *off-line*, sendo que ambos objetivam satisfazer as restrições temporais das tarefas (LAPLANTE, 2004). Um escalonador *on-line* efetua decisões de escalonamento durante o tempo de execução do sistema, que podem depender das características dos processos existentes e do seu estado. Já o escalonador *off-line* efetua todas as decisões antes da execução do sistema (MACÊDO e outros, 2004; AUDSLEY e outros, 1995).

De modo geral, escalonamento estático *off-line* é empregado em sistemas de *tempo real* críticos. Entretanto, o método dinâmico pode ser adequado para utilização em sistemas de *tempo real* suaves (*soft*), suportando um procedimento de recuperação de erro para *deadlines* firmes perdidos e permitindo lidar com situações em que a carga de pior caso seja excedida (AUDSLEY e outros, 1995). Através do escalonamento dinâmico pode-se, por exemplo, mover temporariamente para um outro processador os processos prestes a perderem seu *deadline* devido a uma sobrecarga temporária, e fazer com que sejam executados antes dos processos locais, caso estes tenham menor importância.

Escalonadores podem ser diferenciados ainda por suportar ou não preempção. Caso preempção seja suportada, o escalonador pode suspender a execução do processo e reiniciá-la posteriormente sem afetar o seu comportamento. Caso contrário, tal

comportamento não é possível. Uma vez que em sistemas de *tempo real* orientados a prioridades o processo de mais alta prioridade deve ter sua execução garantida no momento apropriado, é preciso que os processos de prioridade mais baixa sejam *preemptíveis* (SHAW, 2003).

Um modelo de sistema comumente utilizado para estudo do escalonamento apresenta um conjunto de processos T com elementos $\{\tau_1 ... \tau_i ... \tau_n\}$ onde n é a cardinalidade de T. Cada τ_i é caracterizado pelo deu *deadline* (D_i), período (P_i) e tempo de computação (C_i). Restrições simplificadas para esse conjunto de processos são (AUDSLEY e outros, 1995):

(i) Os processos possuem tempo de computação menor que seu *deadline*, e o *deadline* é igual ao período.

(ii) Os tempos de computação para um determinado processo são constantes.

(iii) Todos os processos são periódicos.

(iv) Não existem relações de precedência entre processos.

(v) Não é permitida comunicação ou sincronização entre processos.

(vi) Requisitos de recursos dos processos não são considerados.

(vii) Mudanças de contexto possuem custo zero.

(viii) Todos os processos são alocados a um único processador.

Um modelo mais realista requer o afrouxamento dessas restrições, por exemplo, com a inclusão de:

(i) Tarefas periódicas e aperiódicas.

(ii) Relações de precedência arbitrárias entre processos.

(iii) Proteção de recursos compartilhados entre processos.

(iv) Variação de tempo de computação entre o mínimo (melhor caso) e máximo (pior caso).

(v) Sistemas compostos de vários nós com alocação de processos estática e dinâmica.

A seguir serão apresentadas algumas abordagens para o escalonamento de tarefas em sistemas de *tempo real*.

2.2.1 Escalonamento Estático de Tarefas Periódicas

O escalonamento Taxa Monotônica (*Rate Monotonic*/ RM) é estático e adequado a sistemas com um único processador e com tarefas periódicas e independentes em que seu *deadline* seja igual ao período. O RM requer um escalonador *preemptível*, sendo que a atribuição das prioridades depende do período das tarefas (LIU; LAYLAND, 1973). Nessa abordagem, quanto mais curto o período de ativação da tarefa, maior a sua prioridade associada (SHAW, 2003). O algoritmo é dividido em duas fases: na primeira é avaliado se um conjunto de processos é realizável e na segunda suas prioridades são atribuídas. Para que o escalonamento seja realizável o conjunto de tarefas deve obedecer à seguinte restrição, onde *n* é o número de processos e U é a utilização total do processador pelos processos:

$$U \leq n(\sqrt[n]{2} - 1), \text{ onde U} = \sum_{i=1}^{n} C_i / P_i$$

À medida que *n* cresce, a utilização do processador converge para 0,69. Dessa forma, caso um conjunto de processos tenha uma taxa de utilização inferior a 69%, eles podem ser escalonados através do algoritmo de taxa monotônica. Entretanto, essa condição suficiente pode ser relaxada quando as tarefas do conjunto apresentam períodos múltiplos da tarefa mais prioritária (FARINES; FRAGA; OLIVEIRA, 2000).

No escalonamento *Deadline* Monotônico (*Deadline Monotonic*/ DM) (LEUNG; MERRILL, 1980; LEUNG; WHITEHEAD, 1982), a necessidade de que o período da tarefa seja igual ao *deadline* é relaxada, bastando que o *deadline* da tarefa seja menor que seu

período. No DM a tarefa com menor deadline é que obtém a maior prioridade, sendo que a restrição abaixo deve ser obedecida para que o escalonamento seja realizável:

$$U \leq n\left(\sqrt[n]{2} - 1 \right), \text{ onde U} = \sum_{i=1}^{n} C_i / D_i$$

2.2.2 Escalonamento Dinâmico de Tarefas Periódicas

Conforme mencionado, no escalonamento dinâmico as prioridades são atribuídas às tarefas de acordo com parâmetros que podem variar em tempo de execução. Nas abordagens citadas a seguir, o modelo do sistema é o mesmo do descrito anteriormente, mas as formas de atribuição das prioridades são diferentes.

No algoritmo Menor Deadline Primeiro (*Earliest Deadline First*/ EDF), o processo pronto com deadline mais próximo possui a mais alta prioridade em qualquer momento (LIU; LAYLAND, 1973).

Na abordagem Menor Tempo para o Deadline (*Minimum (Least) Laxity First*/ MLF), o tempo de folga (*laxity*) de um processo é definido como o seu *deadline* menos o tempo remanescente de computação. Esse valor é usado para conferir maior prioridade às tarefas com menor tempo de folga, sendo que, enquanto um processo executa, ele pode sofrer preempção por outro cujo tempo de folga tenha decrescido abaixo do seu (AUDSLEY e outros, 1995).

No algoritmo Maior Urgência Primeiro (Maximim Urgency First/ MUF), é atribuída uma propriedade de urgência a cada tarefa, que é definida como uma combinação de duas prioridades fixas e uma dinâmica. Uma propriedade fixa de criticalidade é associada às tarefas, sendo que o escalonador atribui às tarefas mais críticas os maiores níveis de prioridade (STEWART; KHOSLA, 1991). A sub-prioridade dinâmica é inversamente proporcional ao tempo de folga (*laxity*). Se duas ou mais tarefas compartilham a criticalidade mais alta, então a tarefa de menor tempo de folga é selecionada para execução. Caso uma sub-

prioridade estática, denominada prioridade de usuário, seja especificada, ela é levada em consideração se as tarefas tiverem a mesma criticalidade e sub-prioridade dinâmica.

2.2.3 Escalonamento de Tarefas Aperiódicas Independentes

Para lidar com tarefas aperiódicas em sistemas com um único processador utiliza-se o conceito de servidores, que são processos periódicos cuja função é atender aos processos aperiódicos. Esses servidores são caracterizados pelo seu período e tempo de computação (capacidade) (COTET e outros, 2002).

Na abordagem de Troca de Prioridades (*Priority Exchange*), caso não exista processo aperiódico para ser atendido durante o tempo de execução do servidor, a prioridade do servidor é atribuída a um processo de prioridade mais baixa. Dessa forma, a prioridade do servidor diminui, mas o tempo reservado aos processos aperiódicos é mantido. Com o Servidor Postergável (*Deferrable Server*/ DS), quando não existe processo aperiódico a ser atendido, o servidor empresta seu tempo de computação para o atendimento de tarefas periódicas, mas mantém a sua prioridade alta. Dessa forma, quando chega um processo aperiódico logo após o servidor ter se suspendido, ele pode ser atendido imediatamente. O tempo de computação do servidor é reabastecido no inicio de seu período.

Diferentemente do DS, o Servidor Esporádico (*Sporadic Server*/ SS) permite a variação do ponto no qual o tempo de computação do servidor é reabastecido, pois o tempo de reabastecimento da capacidade depende do momento em que o servidor foi ativado. Caso o tempo de computação do servidor não seja utilizado, ele é registrado para utilização posterior. Quando um processo aperiódico é ativado, ele pode executar imediatamente caso haja tempo disponível.

2.2.4 Escalonamento de Tarefas Dependentes

Em sistemas de *tempo real* mais realistas, os processos interagem para satisfazer os requisitos globais do sistema, sendo que tais interações variam de sincronização simples a proteção de exclusão mútua de um recurso compartilhado, além de relações de precedência (AUDSLEY e outros, 1995). O compartilhamento de recursos e suas relações de exclusão decorrentes podem ocasionar bloqueios em tarefas mais prioritárias de maneira ilimitada, no que se convencionou chamar de inversão de prioridade. A inversão de prioridade ocorre quando um processo τ_i de mais alta prioridade fica bloqueado por um processo τ_j de menor prioridade executando uma região crítica, mas que sofre preempção de processos com prioridade maior que a sua e menor que a de τ_i.

Quando as tarefas se apresentam como dependentes, a inversão de prioridades é inevitável em um escalonamento dirigido a prioridades, entretanto, existem alguns métodos que visam limitá-la através da imposição de certas regras no compartilhamento de recursos de modo que o pior caso de bloqueio experimentado por uma tarefa possa ser conhecido antecipadamente (FARINES; FRAGA; OLIVEIRA, 2000). Duas dessas técnicas são o Protocolo de Herança de Prioridade (*Priority Inheritance Protocol*) e Protocolo de Prioridade Teto (*Priority Ceiling Protocol*).

2.2.4.1 Herança de Prioridade

Nesta abordagem, inversão de prioridade é resolvida dinamicamente através da mudança da prioridade do processo que está causando o bloqueio. Dessa forma, o processo de prioridade baixa bloqueando a região crítica tem sua prioridade aumentada para a mesma prioridade do processo de maior prioridade que aguarda para executar a região crítica, o que faz com que tenha preferência sobre os processos de prioridade intermediária e com que o

número de vezes em que um processo pode ser bloqueado por processos de menor prioridade seja limitado.

2.2.4.2 Prioridade Teto

Embora o Protocolo de Herança de Prioridade limite o número de bloqueios que um processo de prioridade mais alta possa sofrer, este limite é considerado excessivo, além do fato de que não há previsão de mecanismos para evitar *deadlock* (AUDSLEY e outros, 1995). Tais dificuldades são contornadas pelo Protocolo de Prioridade Teto de forma que um processo de prioridade mais alta só possa ser bloqueado no máximo uma vez por ativação e *deadlocks* e bloqueios em cadeia sejam evitados. O funcionamento do protocolo é o seguinte:

- A todos os processos é atribuída uma prioridade estática;
- Semáforos binários protegem o acesso a regiões críticas e possuem um valor de teto associado que corresponde à prioridade do processo mais prioritário que o utiliza;
- Um processo possui uma prioridade dinâmica que é o máximo entre sua própria prioridade e a prioridade que herda devido ao bloqueio de processos de maior prioridade;
- Um processo só pode bloquear um semáforo se sua prioridade dinâmica for maior do que o valor de teto associado a qualquer semáforo correntemente bloqueado (*locked*), com exceção daqueles que o próprio processo já tenha bloqueado.

2.2.5 Escalonamento em Sistemas Distribuídos

Em sistemas distribuídos, as tarefas geralmente possuem demandas de processamento em vários nós diferentes, sendo que uma tarefa distribuída pode ser dividida

em múltiplas sub-tarefas, cada uma delas a ser executada em determinada ordem em um nó específico. Em ambientes distribuídos, os escalonadores locais escalonam sub-tarefas ou segmentos de tarefas globais, ao invés de tarefas completas e integradas (KAO; GARCIA-MOLINA, 1993).

Em (KAO; GARCIA-MOLINA, 1993) é abordada a atribuição de *deadlines* a sub-tarefas com base no *deadline* da tarefa principal. Uma tarefa global consiste de várias sub-tarefas a serem executadas em série ou em paralelo. Para cada tarefa ou sub-tarefa são definidos os seguintes atributos: tempo de chegada, *deadline*, folga (*slack*), tempo real de execução e tempo estimado de execução. Também é definida a propriedade de flexibilidade de uma tarefa como sendo a razão entre sua folga e seu tempo de execução real. O sistema é composto de nós que servem tanto tarefas locais quanto sub-tarefas de tarefas globais. A ordem em que as tarefas são atendidas é escalonada por um escalonador de *tempo real* independente e residente em cada nó. Tarefas globais são processadas primeiro por um gerenciador de processos que atribui *deadlines* a sub-tarefas, as submete ao nó apropriado para execução e impõe as restrições de precedência entre as sub-tarefas de uma tarefa global.

No caso de tarefas globais compostas de sub-tarefas seriais, são apresentadas quatro estratégias possíveis:

- *Ultimate Deadline* (UD) : Nessa modalidade, o *deadline* atribuído à sub-tarefa é o mesmo da tarefa global.
- *Effective Deadline* (ED) : O *deadline* atribuído à sub-tarefa é o *deadline* da tarefa global subtraído do somatório dos tempos de execução estimados para as tarefas subseqüentes.
- *Equal Slack* (EQS) : Nessa estratégia, o tempo de folga restante para execução da tarefa global é calculado e divido igualmente entre as tarefas subseqüentes,

de modo a evitar que a prioridade das tarefas seja demasiadamente afetada pelo tempo de folga existente.

- *Equal Flexibility* (EQF): Nesse caso, a divisão do tempo de folga é feita proporcionalmente ao tempo de execução de cada tarefa.

Nos experimentos descritos foi demonstrado que, embora o EQF diminua a diferença de *deadlines* perdidos entre tarefas locais e globais, essas últimas ainda perdem seus *deadlines* com maior freqüência que tarefas locais.

Para tarefas compostas por sub-tarefas paralelas as seguintes estratégias são descritas:

- *Ultimate Deadline*: Também para sub-tarefas paralelas, o *deadline* atribuído à sub-tarefa é o mesmo da tarefa global.
- DIV-x: Define-se um parâmetro *x* e atribui-se a cada sub-tarefa um valor de *deadline* calculado a partir do valor do *deadline* da tarefa global dividido por *x* vezes o número de sub-tarefas.
- *Globals First* (GF): Nessa estratégia, sub-tarefas são sempre atendidas antes de tarefas locais, embora a ordem de atendimento de tarefas de menor *deadline* primeiro seja mantida dentro das classes de tarefas locais e globais, mas sub-tarefas globais são sempre escalonadas antes das tarefas locais.

Nos testes descritos no trabalho, as estratégias DIV-x e GF foram consideradas as mais aceitáveis.

No trabalho descrito em (WANG e outros, 2005) a aplicação é composta de clientes e servidores de aplicação que hospedam um ou mais componentes, sendo que cada invocação dos clientes aciona a execução de um ou mais métodos em um ou vários componentes. Uma tarefa é definida como uma seqüência de execuções de métodos em componentes e é associada a um descritor de chegada na forma de uma função e um requisito

temporal na forma de um *deadline*. São declaradas classes de serviço em cada componente onde são especificados a seqüência de métodos de cada classe, sua função de invocação e o *deadline* para cada invocação da seqüência de métodos. Dessa forma, se uma seqüência de invocações de métodos em determinada classe obedece a uma função de chegada estabelecida, o componente garante um atraso de pior caso limitado e previamente determinado.

Para a provisão de garantias temporais, um controle de admissão é definido no qual uma tarefa só é admitida caso o somatório do *deadline* de todas as invocações dos métodos se mantenha inferior ou igual ao *deadline* desejado. Além disso, os recursos consumidos são computados com base na função de chegada das tarefas, de modo que não seja admitida uma nova tarefa caso não haja recursos disponíveis.

A determinação da utilização do processador que deve ser atribuída a cada componente para uma determinada classe é feita através do cálculo da carga de trabalho (*workload*) do componente, que é caracterizada pelo tempo de computação da tarefa multiplicado pela sua função de chegada. Para satisfazer todas as classes de serviço, a utilização do processador alocada para os componentes precisa ser de pelo menos o somatório das cargas de trabalho de suas classes. Entretanto, existe a possibilidade de se tomar emprestado recursos subutilizados por outros componentes.

No trabalho apresentado em (ZHANG e outros, 2005) é apresentado um serviço de escalonamento distribuído baseado na especificação de *threads* distribuídas (*Distributable Threads*/ DT) do *Real Time* CORBA. É definido um Serviço de Escalonamento Distribuído (DSS) que obtém os parâmetros de escalonamento especificados pela aplicação e interage com mecanismos de escalonamento local para efetuar decisões de escalonamento de impacto global. Quando uma DT é acionada por uma aplicação, ela se comunica com o DSS para determinar se é escalonável, considerando as DT já existentes, e recebe do DSS seus

parâmetros globais de escalonamento que, por sua vez, são informados também ao escalonador local que pode implementar várias estratégias de escalonamento tais como menor *deadline* primeiro e taxa monotônica.

Se o DSS determina que a execução da DT pode tornar o sistema não escalonável ele aciona o mecanismo de gerenciamento de sobrecarga que pode comandar a negação da aceitação da DT, o cancelamento de uma DT preexistente ou um ajuste de QoS (*Quality of Service*/ Qualidade de Serviço) do sistema.

Em 2009, Zhang, Gill e Lu propuseram um serviço mais abrangente orientado a componentes constituído por mecanismos de controle de admissão, balanceamento de carga e reinicialização de tarefas compostas de subtarefas localizadas em diferentes processadores. A técnica de escalonamento adotada para validação do serviço foi a de Limite de Utilização Aperiódica (*Aperiodic Utilization Bound* – AUB), na qual são consideradas como pertencentes ao conjunto corrente de tarefas aquelas que submeteram trabalhos (*jobs*) mas cujos *deadlines* não expiraram. O algoritmo de escalonamento utilizado foi o Escalonamento Monotônico de Deadline Fim a Fim (*End-to-End Deadline Monotonic Scheduling* – EDMS), no qual uma subtarefa tem maior prioridade se pertencer a uma tarefa com *deadline* menor. O controle de admissão pode ser configurado para avaliar a admissão de toda a tarefa destinada a determinado processador ou dos subtrabalhos separadamente. A reinicialização de tarefas permite que tarefas que já foram completadas, mas que ainda não tiveram seu *deadline* expirado, sejam excluídas do conjunto de tarefas correntes. Através de balanceamento de carga as tarefas podem ser distribuídas para os processadores com a menor taxa de utilização calculada.

Em (DINATALE; STANKOVIC, 1994) é descrita uma estratégia de escalonamento distribuído através de uma combinação de processamento *on-line* e *off-line*

(pré-processamento). O sistema é composto por processos, sendo que cada um deles pode ser constituído de partes indivisíveis denominadas tarefas.

A fase de pré-processamento identifica os grupos de comunicação e os categoriza em tarefas, mensagens enviadas através da rede e restrições de precedência entre elas. Uma vez que o grafo de precedência tenha sido construído, o pré-processador atribui a cada tarefa e a cada mensagem uma janela de tempo ou fatia de tempo, levando em consideração as relações de precedência. O conjunto de fatias de tempo atribuídas a cada tarefa pertencente ao mesmo processo e às mensagens trocadas pelos processos através da rede é chamado de *template* e armazenado no nó onde o processo é carregado (*loaded*). Esses *templates* são utilizados em tempo de execução para escalonar e garantir a execução síncrona do conjunto de processos que se comunicam. Para cada grupo, um processo é designado como mestre, sendo que o escalonador local responsável pela ativação do processo mestre procura ativar as tarefas locais e mensagens que precisam ser trocadas de acordo com o *template* e, ao mesmo tempo, solicita aos demais nós que escalonem as tarefas de comunicação no tempo apropriado.

2.3 Padrões CORBA para *Tempo Real*

Para os sistemas distribuídos de *tempo real*, além de automatizar aspectos de programação em rede, prover uma interface padrão que possibilite integração entre os *software*s que a utilizam e suportar interoperabilidade, o *middleware* deve prover previsibilidade na interação dos componentes (WILLS, 2001), de modo a garantir o atendimento de seus requisitos temporais. Dentre os *middlewares* com foco em *tempo real* existentes encontram-se: (i) ACE (*Adaptive Communication Environment*), um *framework* orientado a objetos que provê serviços de comunicação em *tempo real*, (ii) TAO (*The ACE Orb*) (SCHMIDT, LEVINE, MUNGEE, 1998), uma extensão do CORBA (OMG 2001) para

tempo real construído no topo do ACE, (iii) CIAO (*Component-Integrated ACE ORB*) (WANG, 2004), uma implementação do CCM (*Corba Component Model*) com suporte a *tempo real*, tendo como base o TAO, (iv) TMOES (*TMO Execution Support*) (KIM e outros, 2001), um *middleware* desenvolvido para facilitar a construção de aplicações distribuídas de *tempo real* que utilizam TMOs (*Time-Triggered Message-Triggered Objects*) compatíveis com CORBA e (v) OSA+ (SCHNEIDER; PICIOROGA; BRINKSCHULTE, 2004), um *middleware* para *tempo real* baseado em Java. Nas subseções seguintes serão detalhados o TAO e o CIAO, *middlewares* utilizados na elaboração do presente trabalho. Entretanto, antes de abordar esses *middlewares mais* detalhadamente, será apresentada uma visão geral do CORBA com seu respectivo modelo de componentes, já que essa especificação é base do TAO e do CIAO.

2.3.1 TAO

O TAO (*The ACE ORB*) (SCHMIDT; LEVINE; MUNGEE, 1999) é uma implementação do CORBA para sistemas de *tempo real* baseada na especificação RT-CORBA (*Real-Time* CORBA) (OMG, 2003), que define recursos para suportar previsibilidade fim-a-fim para operações em aplicações CORBA de prioridade fixa e dinâmica. A seguir será apresentada uma introdução sobre o CORBA e o RT-CORBA e, em seguida, o TAO será abordado.

2.3.1.1 CORBA

Middlewares para computação distribuída residem entre as aplicações e os sistemas operacionais, pilhas de protocolos e *hardware*. Provêem uma abstração de programação de alto-nível, tais como objetos remotos, possibilitando aos desenvolvedores construir aplicações distribuídas de forma similar à da programação centralizada

(MCKINLEY, 2004). Dentre os *middlewares* para computação distribuída existentes encontram-se o *Common Object Request Broker Architecture* (CORBA), o *Distributed Component Object Model* (DCOM) (MICROSOFT, 2000), o Java RMI (*Remote Method Invocation*) (SUN MICROSYSTEMS, 2004) e o *Enterprise Java Beans* (EJB) (SUN MICROSYSTEMS, 2006).

No âmbito dos sistemas distribuídos, o *CORBA (Common Object Request Broker Architecture*) tem se destacado, dentre outros motivos, pelo fato de ser uma arquitetura aberta e se propor a tornar o desenvolvimento desses sistemas mais fácil e produtivo. A especificação CORBA define um modelo de objetos distribuídos e uma arquitetura de serviços de suporte a este modelo, com ênfase aos aspectos de interoperabilidade e portabilidade em ambientes heterogêneos (COSTA; KON, 2002). A especificação CORBA 2.x concentra-se em interfaces que são contratos entre clientes e servidores que definem a forma como clientes podem enxergar e acessar os serviços providos por um servidor como se eles fossem locais ao cliente. Tarefas comuns de programação em rede tais como localização e ativação de objetos, *marshaling* e *unmarshaling* de parâmetros, demultiplexação de requisições e recuperação de falhas de segurança são automatizadas pelos *Object Request Brokers* (ORB*),* o que facilita o desenvolvimento de aplicações distribuídas flexíveis e serviços reutilizáveis em ambientes distribuídos heterogêneos (SCHMIDT; LEVINE; MUNGEE, 1998).

Objetos, na terminologia CORBA, dizem respeito a entidades identificáveis e encapsuladas que provêm um ou mais serviços que podem ser requisitados por clientes. O ORB permite que objetos efetuem e recebam requisições em um ambiente distribuído, sendo responsável por todos os mecanismos necessários para encontrar a implementação do objeto para atendimento da requisição, preparar a implementação do objeto para receber a requisição e transportar os dados constituintes da mesma. A interface apresentada ao cliente é

independente da localização do objeto, linguagem de programação utilizada ou quaisquer outros aspectos não refletidos na interface do objeto. Os principais elementos da estrutura do ORB podem ser vistos na Figura 1.

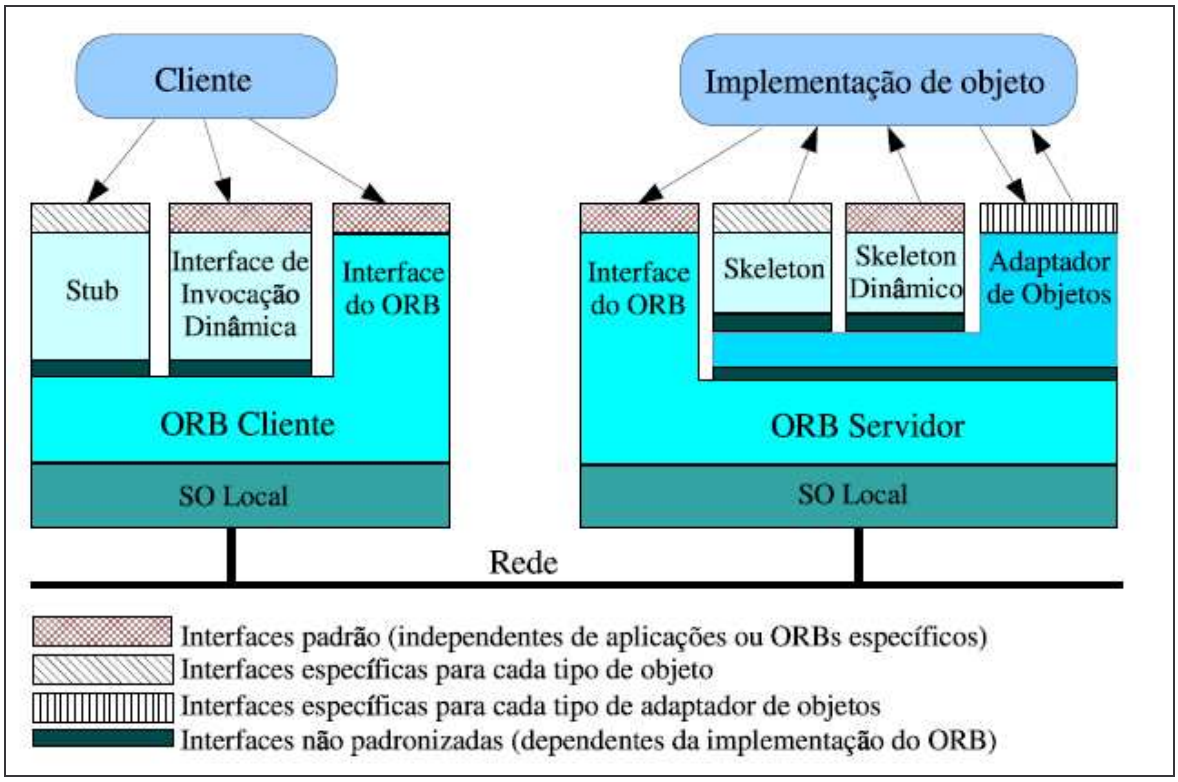


Figura 1: Estrutura da Interface de Requisição de Objetos
(COSTA; KON, 2002)

Para fazer uma requisição, o cliente pode utilizar a Interface de Invocação Dinâmica (*Dynamic Invocation Interface*/ DII) ou um *stub*. O código do *stub* é gerado a partir da Linguagem de Definição de Interfaces (*Interface Definition Language*/ IDL). A IDL define os tipos de objetos através da especificação de suas interfaces, que consistem em um conjunto de operações nomeadas e seus respectivos parâmetros. Compiladores IDL convertem as definições IDL em código de programação de acordo com o mapeamento da linguagem sendo empregada. Além disso, para cada interface IDL, o compilador gera código *stub* e *skeleton*. O *stub* é compilado com o código do cliente e é responsável por converter as requisições enviadas e retornadas pelo cliente em um formato apropriado a ser transmitido através da rede. Ao contrário dos *stubs*, a DII pode ser utilizada nos casos em que a interface

do objeto a ser chamado não é conhecida antecipadamente. Para tanto, o cliente deve suprir informações sobre a operação a ser executada e os tipos de parâmetros a serem passados. O padrão CORBA provê mapeamentos da IDL para as linguagens de programação C, C++, Java, Smaltalk, COBOL, PL/I, LISP, Python e IDLScript.

Adicionalmente à IDL, interfaces podem ser adicionadas a um serviço de Repositório de Interfaces (*Interface Repository*/ IR) que representa a estrutura de uma interface como objetos, permitindo acesso a ela em tempo de execução.

Para que um cliente possa fazer requisições a um objeto é preciso que conheça sua referência, pois é através das informações nela armazenadas que é possível identificar, localizar e acessar o objeto desejado. O Protocolo Geral Inter-ORB (GIOP) especifica a sintaxe de transmissão e um conjunto padrão de formatos de mensagens para permitir que ORBs desenvolvidos independentemente se comuniquem através de qualquer tipo de transporte orientado à conexão. O Protocolo Internet Inter-ORB (IIOP) especifica como o GIOP é implementado no *Transmission Contro Protocol/ Internet Protocol* (TCP/IP) (HENNING; VINOSKI; VINOSKI, 1999).

Um ORB localiza a implementação apropriada, transmite os parâmetros e transfere o controle para a Implementação do Objeto através de um *skeleton* estático ou dinâmico. O *skeleton* é compilado junto ao servidor e sua função é a de receber as requisições que chegam e encaminhá-las aos objetos da aplicação. De forma análoga à DII, a Interface de Esqueleto Dinâmico (*Dynamic Skeleton Interface*/ DSI) permite a servidores receber e tratar requisições sem conhecimento de suas operações e assinaturas em tempo de execução. Quando a requisição é completada, o controle e valores de retorno são retornados ao cliente. *Skeletons* são específicos à interface e ao adaptador de objetos. Adaptadores de objetos compatibilizam o modelo de programação no qual os objetos são implementados às realidades da linguagem de programação e processo servidor, criando e ativando objetos CORBA,

gerando referências e despachando requisições aos objetos apropriados através de um *skeleton*. É possível existirem vários adaptadores de objetos disponíveis com interfaces apropriadas para tipos específicos de objetos. Entretanto, a especificação define um tipo padrão denominado Adaptador Portável de Objetos (POA). Em uma aplicação servidora, o POA é responsável por criar referências de objetos, ativação de objetos e despacho das requisições feitas a objetos aos seus respectivos serventes (*servants*). Um servente é uma entidade em linguagem de programação que implementa o comportamento das requisições em um ou mais objetos. Requisições feitas em referências de objetos são mediadas pelo ORB e transformadas em invocações em um servente particular. Durante a sua existência, um objeto pode estar associado a múltiplos serventes. Uma aplicação pode possuir vários POA, cada um deles representa um grupo de objetos com características similares que são controladas através de políticas do POA, especificadas quando ele é criado. Através da configuração de tais políticas é possível determinar, entre outros, se aplicações utilizarão múltiplas *threads*, se o identificador do objeto será determinado pela aplicação ou pelo POA, se objetos são transitórios ou persistentes ou se um mesmo servente ou serventes distintos serão criados para cada objeto.

A especificação CORBA também inclui os Serviços CORBA (CORBA *Service*), que agrupam serviços básicos geralmente demandados por uma variedade de aplicações, incluindo segurança, suporte a transações e comunicação assíncrona (BOLTON; WALSHE, 2001). De especial interesse para esse trabalho encontra-se o Serviço de Eventos.

Serviço de Eventos

O serviço de eventos CORBA permite o desacoplamento da transferência de mensagem entre os objetos, provendo os modelos *pull* e *push* de comunicação entre consumidores e fornecedores.

No modelo *push*, um fornecedor se conecta ao canal de eventos e inicia o envio de eventos. É de responsabilidade do canal de eventos armazenar as mensagens até que sejam entregues a um ou mais consumidores interessados. Ao conectar-se ao canal, o fornecedor recebe um objeto *proxy* que representa o consumidor, para o qual envia os eventos. Por sua vez, o consumidor se conecta a um objeto *proxy* que representa o fornecedor. Quando o canal de eventos possui uma mensagem disponível, o *proxy* do fornecedor a envia ao consumidor.

No modelo *pull*, o canal de eventos extrai os dados do consumidor, sendo o consumidor que dirige o envio de mensagens. Um fornecedor se conecta a um *proxy* do consumidor, e, por sua vez, um consumidor interessado se conecta ao canal de eventos através do *proxy* do fornecedor. Quando um consumidor está interessado em receber um evento, ele o solicita através dos métodos *pull* ou *try_pull* do *proxy* do fornecedor. Feito isso, *proxy* do fornecedor aciona o *proxy* do consumidor para que solicite ao fornecedor conectado que envie um novo evento.

Através da utilização do canal de eventos, é possível ainda utilizar ambos os modelos simultaneamente, caso a aplicação assim o requeira.

2.3.1.2 Real-Time CORBA

Embora a especificação CORBA possibilite o desenvolvimento de aplicações distribuídas de forma mais produtiva e menos propensa a erros, ela não é apropriada para satisfazer as necessidades de previsibilidade das aplicações de *tempo real*. Além disso, não existem mecanismos para especificar e garantir os requisitos de QoS fim-a-fim (SCHMIDT; LEVINE; MUNGEE, 1999). Para reparar essa deficiência, foi elaborada a especificação RT-CORBA, que define interfaces padrão e políticas de QoS que permitem às aplicações configurar e controlar: (i) recursos do processador através de *pools* de *threads*, mecanismos de atribuição de prioridade a operações, exclusão mútua e serviço de escalonamento global,

(ii) recursos de comunicação através de protocolos e ligações explícitas e (iii) recursos de memória através da alocação de requisições em filas e limitação do tamanho do *pool* de *threads* (SCHMIDT; KUHNS, 2000).

O gerenciamento dos recursos do processador é facilitado através da disponibilidade de mecanismos para controlar a prioridade das invocações das operações. São definidos dois tipos de prioridade: CORBA e nativa. As operações podem receber uma prioridade CORBA entre 0 e 32.767, que é mapeada pelo ORB para uma prioridade nativa da plataforma operacional em uso. É possível também utilizar prioridades declaradas pelo servidor ou propagadas pelo cliente. No primeiro tipo, o servidor determina a prioridade na qual a invocação a um objeto específico ocorrerá, que é embutida na referência do objeto a que o ORB do cliente tem acesso. No modelo de prioridade propagada pelo cliente, este acrescenta a prioridade desejada à lista de contexto do serviço que é embutida na requisição enviada através do GIOP, que deve ser atendida pelo servidor. Adicionalmente, é possível definir transformações de prioridade que permitem alterar a prioridade em que operações são executadas em dois momentos: antes da invocação do servente ou no momento em que é feita uma invocação a um objeto pelo servente. É possível ainda alocar previamente *pools* de *threads*, permitindo às aplicações se beneficiarem da capacidade *multithreading* com limitação dos recursos consumidos, e definir valores para atributos tais como níveis de prioridade. *Pools* de *threads* podem ser associados ao POA, sendo que cada um deles só pode possuir um *pool* de *threads*, que pode ser compartilhado por outros POA. Pode-se configurar o *pool* de *threads* de modo que cresça dinamicamente até um limite máximo e de forma a possuir *threads* com prioridades distintas. Um conjunto de *mutexes* padrão é definido para permitir a sincronização entre as *threads*. É especificado também um serviço de escalonamento global através do qual as aplicações podem informar seus requisitos de processamento em termos de parâmetros tais como tempo de execução de pior caso e período.

O gerenciamento da comunicação Inter-ORB é facilitado através da existência de interfaces para selecionar e configurar determinadas propriedades do protocolo de comunicação. Além disso, aplicações cliente podem se associar explicitamente a um objeto servidor utilizando bandas de prioridade e conexões privadas. Desse modo, evita-se a ocorrência de atrasos causados pela postergação da ativação do servidor e alocação de recursos até o tempo de execução, e permite-se que o cliente especifique prioridades para cada conexão de rede e utilize conexões privadas para o atendimento de suas demandas de QoS, preservando as prioridades fim-a-fim e eliminando as principais fontes de inversão de prioridade.

A versão 1 do Real-Time CORBA (RTC1) tinha seu foco em aplicações de *tempo real* com prioridades fixas de escalonamento. A versão 2 (RTC2) foi elaborada com o intuito de superar essa limitação. Tal versão estende o RTC1 através da provisão de interfaces e mecanismos que aplicações podem utilizar para agregar serviços de escalonamento dinâmico e interagir com eles através do sistema distribuído, permitindo especificar e utilizar disciplinas de escalonamento e parâmetros que descrevem e definem com mais exatidão seus requisitos de execução e recursos (KRISHNAMURTHY e outros, 2004). Para tanto, é introduzido o conceito de *thread* distribuída (DT), que carrega seus parâmetros de escalonamento através de limites de sistemas, e pode abranger uma ou mais *threads*. Uma *thread* distribuída é uma abstração de uma cadeia de chamadas de métodos por múltiplas threads em vários processadores (OMG, 2003). Em cada nó, o escalonador local escalona a DT de acordo com sua informação de escalonamento. DTs interagem com o escalonador em pontos de escalonamento específicos que incluem chamadas da aplicação, aquisição e liberação de recursos e em pontos pré-definidos nas invocações CORBA. A interface da DT para o escalonador local é bem definida. A operação *begin_scheduling_segment* inicia uma porção do programa de *tempo real* e passa os parâmetros de escalonamento para o escalonador local.

A operação *update_scheduling_segment* atualiza os parâmetros existentes para a DT e a operação *end_scheduling_segment* finaliza o segmento, indicando que a região de código de *tempo* real acabou. O padrão de escalonamento dinâmico CORBA permite ao sistema modificar parâmetros de escalonamento durante a operação, imprimindo maior facilidade de adaptação às aplicações.

Embora a especificação aborde somente o escalonamento local, não impede o desenvolvimento de escalonadores que efetuem escalonamento global. De fato, em (ZHANG e outros, 2005; BRYAN e outros, 2005) são apresentadas propostas de serviços de escalonamento global baseados nos mecanismos de escalonamento dinâmico do CORBA.

2.3.1.3 The ACE ORB (TAO)

O TAO é uma implementação de desempenho otimizado para *tempo real* do padrão CORBA, tendo servido de orientação para alguns aspectos da especificação *Real Time* CORBA. Para atender ao domínio para o qual foi projetado, foram feitos os seguintes aprimoramentos para otimizar seu desempenho: (i) definição de políticas e mecanismos para especificação de requisitos de QoS, (ii) desenvolvimento de subsistema de entrada e saída de *tempo real* para mediar o acesso do ORB e das aplicações aos recursos de rede de baixo nível e do sistema operacional, para reforçar atendimento das especificações de QoS, (iii) otimizações no CORBA *General Inter-ORB Protocol* (GIOP) que minimizam inversão de prioridade e permitem uma comunicação previsível e eficiente entre sistemas com ORBs heterogêneos, (iv) utilização de compiladores de IDL otimizados, (v) otimização das políticas de demultiplexação do *Portable Object Adapter* (POA), (vi) adoção de um modelo integrado de gerenciamento de memória que minimiza cópia de dados e contenção por bloqueio através do ORB.

De especial interesse para este trabalho são os serviços de eventos e escalonamento previstos pelo TAO, uma vez que sua utilização possibilita a entrega de eventos assíncrona e com fraco acoplamento, a partir de prioridades previamente definidas, o que contribui para a previsibilidade do sistema.

Aplicações de *tempo real* que utilizam o TAO podem especificar os requisitos de escalonamento de recursos ao Serviço de Escalonamento de Tempo-Real para suas operações através da definição da estrutura *RT_Info* de uma *RT_Operation*. Os atributos de uma *RT_Info* incluem tempo de execução no pior caso, período, importância e dependências. Utilizando técnicas de escalonamento tais como Taxa Monotônica (RMS/ *Rate Monotonic Scheduling*), *Earliest Deadline First* (EDF), *Minimum Laxity First* (MLF) e *Maximum Urgency First* (MUF) e abordagens de análise tais como *Rate Monotonic Analysis* (RMA) (SCHMIDT; LEVINE; MUNGEE, 1999). A depender da disciplina de escalonamento selecionada, os atributos pertinentes devem ser informados através da *RT_Info*. RMA é utilizada para validar a escalonabilidade das operações antes do início da execução da aplicação. Caso as operações possam ser escalonadas, o Serviço de Escalonamento atribui uma prioridade a cada requisição, com base nas propriedades das operações e estratégia de escalonamento empregada. Feito isso, o Módulo de Despacho é configurado com o número e tipos de filas de despacho necessárias. Este módulo pode estar localizado no subsistema de entrada e saída, no núcleo do ORB ou no canal de eventos. Em tempo de execução, o módulo de despacho identifica a fila para a qual a requisição deve ser enviada, calculando as porções dinâmicas das prioridades, caso haja.

O Serviço de Eventos de *Tempo Real* do TAO foi concebido com o intuito de acrescentar funcionalidades ao Serviço de Eventos CORBA para satisfazer as necessidades das aplicações distribuídas de *tempo real*, possibilitando o uso eficiente da rede e dos recursos computacionais, filtragem e correlação de eventos, diminuição dos atrasos no despacho de

eventos e suportando processamento periódico (SCHMIDT, O'RYAN, 2002). É possível aos consumidores especificar semânticas de conjunção ou disjunção quando registram seus requisitos de filtragem. Consumidores podem ainda se subscrever para receber subconjuntos de eventos, sendo que o canal as utiliza para filtrar eventos enviados pelos fornecedores, encaminhando-os apenas aos consumidores interessados.

Para garantir que os consumidores executem a tempo de atender seus *deadlines*, o módulo de despacho do serviço de eventos colabora com Serviço de Escalonamento de *Tempo Real*. Caso chegue um novo evento com uma prioridade superior a da *thread* correntemente em execução, o módulo de despacho faz com que a *thread* em execução sofra preempção e o novo evento seja despachado. O módulo de despacho sempre obtém eventos no topo da fila, que é ordenada de acordo com a prioridade do evento e, opcionalmente, pela sua subprioridade. Outra funcionalidade provida pelo serviço é a especificação de temporizadores, para os quais os consumidores podem se registrar, com garantia de que os *timeouts* serão despachados de acordo com a prioridade de seus respectivos consumidores.

2.3.2 CIAO

O CIAO é uma implementação do CCM que visa permitir a configuração declarativa das funcionalidades do RTCORBA, o que torna sua utilização adequada em aplicações com requisitos diferenciados de QoS, tais como, previsão e limitação do atraso fim-a-fim das requisições, disponibilidade de CPU para atendimento de *deadlines* e manutenção da taxa de invocação de componentes. A seguir o CCM será descrito, sucedido pelo detalhamento do CIAO.

2.3.2.1 CORBA Component Model

Embora a utilização do CORBA tenha simplificado o desenvolvimento do lado cliente das aplicações, a implementação de aplicações servidoras através dele é dificultada pela ausência de abstrações de alto nível para implementá-las (WANG, 2004). Além disso, seu modelo de objetos não prevê um mecanismo que permita programar conexões e dependências entre serviços, nem existe uma forma padronizada de distribuir e iniciar implementações de objetos remotamente.

Como forma de sanar essas deficiências foi especificado o Modelo de Componentes do CORBA (CCM) no CORBA 3. A especificação do CCM estende o modelo de objetos do CORBA para suportar o conceito de componentes e estabelece padrões para implementar, empacotar (*packaging*), montar (*assembling*) e implantar (*deploying*) componentes.

As novas características introduzidas no CCM são construídas com base nos conceitos convencionais de CORBA, através de uma extensão da IDL (conhecida como IDL3) para a declaração de componentes e através do mapeamento dos conceitos introduzidos nessas extensões em termos da IDL convencional (COSTA; KON, 2002).

O Modelo de Componentes

Componentes são extensões de objetos CORBA, especificados em IDL3 e denotados por uma referência de componente, que é representada por uma referência de objeto (OMG, 2002). A IDL3 é um super conjunto da IDL CORBA original com novas construções para expressar a colaboração de componentes dos lados cliente e servidor.

Um tipo de componente é uma coleção nomeada de elementos que podem ser descritos por uma definição de componente em IDL ou uma estrutura correspondente no

Repositório de Interfaces. Por ser abstrato, o tipo de componente é instanciado para criar entidades concretas com estado e identidade.

As interações entre componentes são especificadas por meio de portas, que podem ser dos seguintes tipos:

- Facetas: interfaces providas pelo componente, em número de zero ou mais, cada uma apresentando uma visão diferente do componente para seus clientes.
- Receptáculos: um receptáculo armazena as referências de objetos para os quais o componente pode invocar operações. Um receptáculo pode possuir uma ou múltiplas conexões a depender de sua especificação.
- Emissores de eventos: São pontos de conexão que emitem eventos de um tipo específico para um ou mais consumidores interessados ou para um canal de eventos.
- Receptores de eventos: Um receptor de evento permite a entrega de mensagens ao componente através de uma operação *push.*
- Atributos: São valores nomeados que podem ser consultados e modificados através de operações disponíveis para esse fim, podendo ser empregados na configuração do componente ou um outro propósito.

O acesso a um componente pode também ser feito através de uma interface equivalente que permite aos clientes que não aderirem ao modelo continuar a acessar os componentes CCM. Foi definido um elemento *Home* para cada tipo de componente que permite gerenciar o ciclo de vida de suas instâncias com operações para criar, localizar e destruí-las. A Figura 2 representa o componente com suas respectivas interfaces e portas.

O Modelo de Programação do Contêiner

O CCM define dois mecanismos padrão para a implantação dos componentes: Servidores de Componentes (*Component Servers*) e Contêineres.

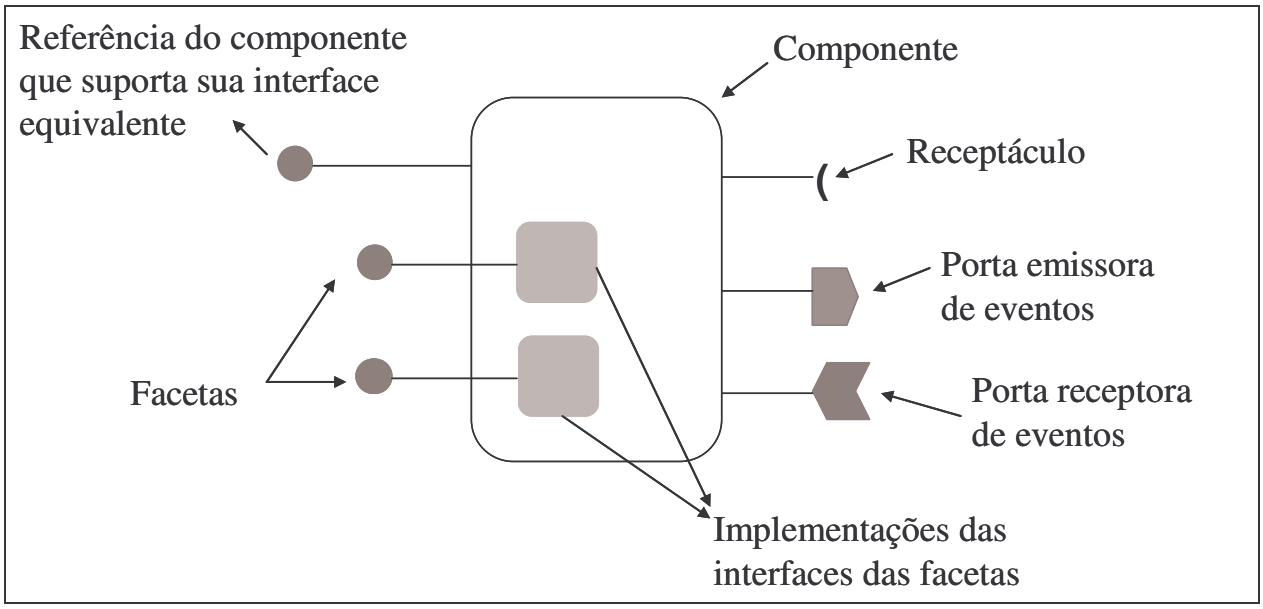


Figura 2: Interfaces do componente CCM e respectivas portas

Servidores de componentes são implementações genéricas de servidores que o *framework* CCM utiliza para hospedar componentes. O Servidor de Componente é responsável por prover o ambiente de execução adequado ao componente, configurando seu ORB para atendimento de suas necessidades. Os Contêineres têm a função de configurar e gerenciar o POA. Um servidor de componentes contém um ORB e vários contêineres que, por sua vez, possuem um POA. Implementações de componentes são instaladas e ativadas em contêineres, e são responsáveis pela intermediação de cada conjunto de serventes que implementam um componente com os serviços dos quais ele depende. Dessa forma, o suporte de execução é provido aos componentes, separando os aspectos relacionados à programação e configuração de servidores, ORB e POA, dos aspectos concernentes ao desenvolvimento do componente e da aplicação (WANG, 2004).

Uma vez que todas as interações do componente são intermediadas pelo contêiner, incluindo aquelas com os clientes e o ambiente de execução, são disponibilizadas interfaces

para padronizar interações entre as implementações de componentes e seu contêiner. Tais interfaces são categorizadas em:

- Interfaces externas: São interfaces disponíveis aos clientes do componente e podem ser do tipo *home* ou aplicação.
- Interfaces internas: Através da interface interna o componente interage com seu contêiner para acessar o contexto e serviços gerenciados pelo contêiner, tais como, gerenciamento do ciclo de vida e operações transacionais.
- Interfaces de *callback*: São interfaces que as implementações de componentes devem possuir para que seu contêiner possa informá-las de eventos pertinentes, tais como a ativação ou desativação do componente.

O Modelo de Implementação de Componentes

O CCM especifica um *Framework* de Implementação de Componentes (*Component Implementation Framework*/ CIF) para padronizar e automatizar a geração de código para implementação dos componentes. Para tanto, é definida a Linguagem de Implementação de Componentes (*Component Implementation Language*/ CIDL) que descreve a estrutura e estado de implementações de componente (OMG, 2002). O compilador CIDL deve ser utilizado para gerar automaticamente a implementação dos serventes do componente, que realizam a maioria das suas funcionalidades de gerenciamento. Além do código dos serventes, o compilador também gera o esqueleto para o código dos executores, artefatos que implementam o comportamento do componente.

Uma implementação de um componente compreende um conjunto de artefatos que devem possuir comportamentos e relacionamentos específicos para prover uma implementação apropriada (WANG, 2004). À definição dos componentes e os artefatos que o compõem denomina-se composição. Além dos executores, a composição deve especificar

obrigatoriamente *home* do componente e o nome do executor da *home*. A especificação do *home* do componente identifica implicitamente o tipo de componente para o qual a composição provê implementação. O nome do executor da *home* é utilizado como o nome do artefato de programação gerado pela CIDL como o esqueleto do executor da home.

Implantação e Configuração

Com o propósito de definir mecanismos através dos quais aplicações baseadas em componentes possam ser implantadas, a OMG publicou a especificação *Deployment and Configuration of Component-based Applications* (OMG, 2003).

De acordo com a especificação, componentes de *software* podem ser compostos de código compilado ou por uma associação de outros componentes através de portas, denominada montagem (*assembly*). Implementações de montagens de componentes são agrupadas em pacotes (*packages*) que podem conter diversas implementações do mesmo componente, além de metadados que descrevem seu conteúdo.

Os metadados dos componentes e aplicações são armazenados em arquivos XML denominados descritores. Além de definir os descritores de componentes, montagens e pacotes separadamente, é possível criar um descritor único, chamado Plano de Implantação, que deve conter: os artefatos que compõem a implantação, informação sobre como criar instâncias a partir desses artefatos, onde instanciar os componentes e as conexões entre eles, bem como quais atributos devem ser inicializados e a interface disponibilizada pela aplicação.

A execução da aplicação é realizada através da cooperação das seguintes entidades:

- Gerenciador de Execução (*Execution Manager*): responsável por gerenciar o processo de implantação para um ou mais domínios, sendo um domínio definido como um conjunto de nós independentes conectados. O Gerenciador

de Execução separa em planos diferentes os componentes a serem implantados em nós distintos e passa esses sub-planos ao Gerenciador do Nó. Além disso, ele cria uma instância do Gerenciador de Aplicação do Domínio que pode ser utilizada posteriormente para iniciar a aplicação.

- Gerenciador de Nó (*Node Manager*): gerencia a implantação de componentes que residem em um nó determinado.
- Gerenciador de Aplicação (*Application Manager*): é utilizado para iniciar e terminar a aplicação. Existem dois tipos de gerenciador de aplicação: Gerenciador de Aplicação do Domínio (*Domain Application Manager*) e Gerenciador de Aplicação do Nó (*Node Application Manager*). O Gerenciador de Aplicação do Nó é responsável por executar e finalizar instâncias de componentes no nó em que está localizado. Tais atividades são solicitadas e gerenciadas pelo Gerenciador de Aplicação do Domínio.
- Aplicação (*Application*): Os tipos de Aplicação são Aplicação de Domínio e Aplicação de Nó. Essas entidades representam a aplicação global e a aplicação em determinado nó, respectivamente. Aplicações de Domínio são criadas por Gerenciadores de Aplicação de Domínio, enquanto Aplicações de Nó são criadas por Gerenciadores de Nó. Uma Aplicação de Domínio mantém referências a Aplicações de Nó, que criam os contêineres onde são instanciados os componentes da aplicação e concluem a inicialização da aplicação finalizando a conexão dos componentes e ativando-os quando solicitado.
- Gerenciador de Repositório (*Repository Manager*): mantém uma coleção de pacotes de componentes. Através dele podem ser armazenadas implementações de componentes para serem buscadas posteriormente, sob demanda

Para implantar uma aplicação conforme a especificação, é necessário chamar o método *preparePlan* do Gerenciador de Execução, passando como parâmetro o plano de implantação. Para acionar a aplicação, o método *startLaunch* do Gerenciador de Aplicação de Domínio, retornado pelo passo anterior, deve ser invocado, informando as propriedades de configuração, se desejado. Essa operação retorna uma à Aplicação de Domínio e as conexões que são providas pela aplicação nas portas externas. Para concluir a implantação da aplicação, deve ser chamado o método *finishLaunch* da Aplicação de Domínio, informando as conexões para as portas externas da aplicação, se desejado. Observa-se que as atividades relativas a cada nó são delegadas ao Gerenciador de Nó, Gerenciador de Aplicação de Nó e Aplicação de Nó.

2.3.2.2 Component Integrated ACE ORB - CIAO

O CIAO foi concebido como uma forma de aprimorar o CCM no que diz respeito à garantia dos requisitos temporais de determinados tipos de aplicação e, nesse intuito, permitir a especificação, validação, empacotamento, configuração e implantação de componentes e integrar tais mecanismos com os recursos do RTCORBA existentes no TAO, tais como *pools* de *threads*, *lanes* e políticas de atribuição de prioridade determinada pelo cliente ou servidor. Além disso, é prevista uma extensão dos descritores do CCM para comportar a provisão de QoS tanto do lado cliente quanto servidor (DENG e outros, 2007).

Para permitir a configuração dos aspectos de QoS mencionados, o CIAO define um novo formato de arquivo com extensão CSR (CIAO *server resource descriptor*). Através deste tipo de arquivo é possível especificar um conjunto de políticas que determinam o comportamento da aplicação no que diz respeito a suas necessidades de QoS, tais como as prioridades das *threads* de *tempo real*, e configurar recursos para garanti-las.

A criação dos componentes obedece ao especificado no CCM, sendo disponibilizado um compilador CIDL que processa o arquivo contendo a descrição da composição e gera os serventes, um contexto específico que implementa conexões externas e subscrições de consumidores e gerencia interações com o contêiner, além da estrutura dos executores do componente. São gerados serventes para o componente, suas facetas e sua *home*.

Os serventes gerados contêm e gerenciam tanto executores de componente, que são responsáveis pela funcionalidade da aplicação, como contextos específicos do componente, que gerenciam conexões do componente utilizando código gerado pelo compilador CIDL. O servente gerado encaminha as requisições feitas às operações da interface do componente aos executores. Operações de gerenciamento de conexão são encaminhadas ao contexto específico do componente (WANG, 2004).

Deployment and Configuration Engine (DAnCE)

O CIAO dispõe do DAnCE para efetuar a implantação da aplicação, cuja implementação é baseada na especificação *Deployment and Configuration of Component-based Distributed Applications* (OMG, 2003). As várias entidades definidas pela especificação são implementadas como objetos CORBA. Para assegurar o atendimento dos requisitos de QoS das aplicações, o DAnCE estende a especificação através da configuração dos recursos do servidor de componentes como um conjunto de políticas e como componentes se relacionam com essas políticas, que influenciam a QoS fim-a-fim. Com tais aprimoramentos, o DAnCE o permite a configuração e controle de recursos do processador através de *pools* de *thread*, mecanismos de atribuição de prioridade, *mutexes* intra-processos; recursos de comunicação via propriedades do protocolo e associação explícita; e recursos de

memória através do armazenamento de requisições em filas e limitando o tamanho dos *pools* de *threads* (DENG e outros, 2007).

Para atender aos requisitos de QoS, o DAnCE estende o padrão da OMG acrescentando um Tratador de Configuração de XML (XML *Configuration Handler*) que percorre o plano de implantação e armazena a informação lida como uma estrutura de dados IDL que pode transferir informação entre processos de forma eficiente e evita processar o arquivo XML múltiplas vezes. Para gerenciar a implantação da aplicação, o Gerenciador de Execução recebe como entrada essas estruturas IDL e as disponibiliza aos gerenciadores de aplicação de Domínio e de Nó. O último utiliza as opções de configuração dos recursos do servidor existentes no plano para configurar os contêineres na Aplicação de Nó que ele cria. Os contêineres utilizam as especificações no plano para configurar o suporte de *tempo real*, incluindo *pools* de *threads*, modelos de propagação de prioridades e de conexões.

Capítulo 3

Reconfiguração Dinâmica em Sistemas Distribuídos

3.1 Preservação de Consistência

De forma a garantir que o sistema mantenha seu funcionamento correto durante e após a reconfiguração dinâmica, devem ser providos mecanismos para preservação de consistência. Uma vez que o estado da aplicação pode ser modificado através das interações entre os componentes, durante tais interações pode existir estado transitório distribuído entre os componentes, deixando o estado dos componentes mutualmente inconsistente. Para evitar perda de informações e alcançar um estado consistente após a reconfiguração, se faz necessário atingir um estado consistente da aplicação, na parte afetada do sistema, antes da mudança (KRAMER; MAGEE, 1990).

Além da consistência mútua, pode-se identificar dois outros tipos: integridade estrutural e preservação de invariantes (MOAZAMI-GOUDARZI 1999). Para que os requisitos de integridade estrutural do sistema sejam alcançados, deve-se obedecer às especificações de como os componentes devem ser conectados e o modo como as aplicações devem ser construídas. Invariantes do estado da aplicação consistem em uma lista de predicados, cada um expresso sobre o estado de um subconjunto dos componentes do sistema.

Para que a correta operação do sistema seja garantida, os predicados especificados devem ser avaliados para verdadeiro.

A seguir serão apresentadas algumas abordagens encontradas na literatura para tratar desse tema, sendo que a orientada a conexão, descrita na subseção 3.1.3, foi adotada neste trabalho para conduzir o sistema ao estado apropriado à realização da reconfiguração.

3.1.1 Conic – Kramer e outros

Nos trabalhos descritos em Kramer e Magee (1990) e em Magee e Sloman (1989), a reconfiguração é efetuada em um sistema distribuído, que consiste de um conjunto de nós de processamento com conexões direcionadas indicando os caminhos de comunicação entre eles. Um nó é definido como uma unidade de processamento que pode iniciar e atender transações. Uma conexão é um canal de comunicação direto do iniciador da comunicação para o receptor. A troca de mensagens entre dois nós conectados é feita através de transações. Assume-se que transações são completadas em tempo limitado e que o iniciador da transação tem conhecimento de sua completude. Transações são a única forma pela qual o estado de um nó é afetado por outro nó conectado do sistema e subdividem-se entre dependentes e independentes. Transações dependentes precisam de outras transações para que se completem e, caso isso não ocorra, a transação é dita independente. Mudanças são descritas em termos de modificações na estrutura do sistema e compreendem a criação de nó, remoção de nó, remoção de conexão e criação de conexão. A condição estabelecida para que o estado do nó permaneça consistente durante e após a reconfiguração é a de que nele não existam transações incompletas.

A partir da descrição das mudanças a serem efetuadas e da configuração atual do sistema são determinados os componentes que terão sua atividade restringida para que a reconfiguração possa ser efetuada sem tornar inconsistente o estado dos componentes. Esses componentes são

instruídos pelo Gerenciador de Mudanças a alternarem para um estado passivo para que o estado de segurança do sistema (*safe state*), denominado quiescência, possa ser alcançado pelos nós afetados.

Os autores demonstram que estado de segurança pode ser alcançado em tempo finito. São identificados dois tipos de sistemas: com e sem dependência entre transações.

Para sistemas com transações independentes um componente encontra-se no estado passivo se:

(a) continua a aceitar e atender transações, mas

(b) não inicia novas transações, e

(c) qualquer transação que tenha iniciado já se completou.

Um nó é dito passivo quando se encontra no estado passivo. Entretanto, o estado passivo não é suficiente para a reconfiguração já que o nó pode continuar processando transações iniciadas por outros nós. Para consistência durante a mudança é necessário que o nó não esteja participando de uma transação e nem receba nem inicie uma nova transação. Tal propriedade é denominada quiescência. Dessa forma, um nó é dito quiescente se:

(a) encontra-se no estado passivo, e

(b) não está envolvido no atendimento de alguma transação, e

(c) nenhuma transação foi ou será iniciada por outros nós que requeira atendimento por este nó.

Um nó N se torna quiescente se todos os nós em seu conjunto passivo (*Passive Set*), denotado por PS(N) estão no estado passivo. Para sistemas com transações independentes, fazem parte deste conjunto:

(a) o nó N, e

(b) todos os nós que podem iniciar diretamente transações em N.

Para sistemas com transações dependentes a definição do conjunto passivo é modificada. É definido um conjunto passivo estendido (*Enlarged Passive Set*) denotado por EPS(N) formado por:

(a) todos os nós constantes em PS(N),

(b) todos os nós que podem iniciar transações dependentes que resultem em transações conseqüentes em N.

São definidas algumas regras para a reconfiguração a depender da operação a ser executada:

- Remoção de nó: O nó que será removido deve estar em estado quiescente e sem conexões de e para outros nós.
- Criação e remoção de conexão: O nó do qual a conexão se origina deve estar em estado quiescente.
- Criação de nó: O nó deve estar em estado quiescente, o que é trivial já que um novo nó está inicialmente isolado.

No processo de gerenciamento, as regras de reconfiguração são aplicadas aos comandos de reconfiguração e a lista dos componentes que precisam estar no estado passivo é definida, sendo que tais componentes formam o conjunto passivo de modificação (*Change Passive Set*), denominado CPS. O CPS é a união dos conjuntos passivos PS de cada componente em QS (conjunto quiescente), sendo tais conjuntos definidos por:

CS = { conexões c | c é uma conexão de/ para um nó a ser removido}

LS = { conexões l | l é uma conexão em uma diretiva de criação/ remoção de conexão}

QS = { nós n | n é o iniciante em uma conexão constante em (CS $\cup$ LS)}

CPS = $\cup$ PS(i), para todo i em QS

A ordem adotada para a efetivação dos comandos de reconfiguração á a seguinte:

1. Levar os componentes necessários ao estado passivo;
2. Remover as conexões;
3. Remover os componentes;
4. Criar os novos componentes (pode ser feita a qualquer tempo antes da conexão);
5. Criar as conexões entre os novos componentes;
6. Ativar os novos componentes.

Caso seja permitido que várias reconfigurações sejam realizadas paralelamente, o conjunto de nós que deve ser bloqueado para uma mudança é um super-conjunto do conjunto passivo de modificação. O conjunto de nós a serem bloqueados inclui os nós para os quais conexões são direcionadas, de forma que um comando de reconfiguração não tente criar uma conexão para um nó que tenha sido excluído por outra transação concorrente, conforme representado abaixo.

LockSet = { CPS $\cup$ { nós n | n é um nó receptor em uma conexão constante em (CS $\cup$ LS)}

Os princípios descritos no modelo foram aplicados em um problema exemplo, o Jantar dos Filósofos, implementado e testado no ambiente Conic para sistemas distribuídos.

3.1.2 Framework para Serviço de Reconfiguração Dinâmica – Moazami-Goudarzi

Moazami-Goudarzi (1999) propôs um *framework* para a provisão de um serviço de reconfiguração dinâmica com preservação de consistência, em que são identificados os elementos básicos necessários à provisão de tal serviço, a saber: *Reconfiguration Manager* (RM), *Configuration Database* (CDB) e *Consistency Manager* (CM). Além desses, é requerido um *Event Composition Service* (ECS) do qual o serviço de reconfiguração é cliente. Uma nova abordagem para preservação de consistência entre componentes é apresentada na

qual, dada a descrição das mudanças, identifica-se automaticamente e se direciona os componentes necessários para o estado de segurança.

Foi feito um protótipo da solução suportada no ambiente de programação distribuída Regis tendo o Java como linguagem de reconfiguração.

O RM recebe as mensagens enviadas pelo ECS para selecionar e executar *scripts* de reconfiguração. Para tanto, ele se comunica com o CDB para verificar se a execução do *script* interferirá com outros *scripts* executando. Caso isso ocorra, ele instrui o CM a efetuar as ações necessárias para garantir que componentes não sejam deixados em estado inconsistente. Feito isso, o RM executa o *script* e ao término informa ao CM, para que reverta quaisquer ações tomadas como parte do mecanismo de consistência mútua, retornando o sistema a seu estado normal.

O CDB é a interface utilizada para alterar e consultar a configuração do sistema. São aceitas requisições de reconfiguração básicas tais como criar e remover conexões e componentes. Atualizações no CDB são refletidas na configuração da aplicação através de *hooks* para o ambiente de execução distribuído. O CDB também suporta uma interface de geração de eventos para os quais clientes podem se inscrever e receber notificações de eventos de reconfiguração de seu interesse.

O CM é a entidade responsável por assegurar que componentes não sejam deixados em estado de inconsistência mútua como resultado da execução das mudanças de configuração. Dessa forma, a reconfiguração só é feita após o CM sinalizar que pode ser feita com segurança.

No trabalho é apresentada uma abordagem alternativa à de Kramer e Magee (KRAMER; MAGEE, 1990; MAGEE; SLOMAN, 1989) para alcançar o estado de segurança do sistema. No modelo assumido, transações se completam em tempo limitado e não podem se intercalar, ou seja, o componente não participa de uma nova transação enquanto outra está

em progresso. O estado de quiescência do componente é alcançado bloqueando sua execução quando não existe transação em andamento. Dessa forma, de acordo com o algoritmo apresentado, é enviada uma mensagem *block* para nós no conjunto quiescente, denominado *blocking set* (BSet). Caso o componente N que recebeu a mensagem esteja ocioso ele passa ao estado bloqueado. Uma vez que componentes que dependem de N também podem precisar ser bloqueados, N pode desbloquear temporariamente para atender algumas requisições. Entretanto, para garantir que em algum ponto N permaneça bloqueado, ele só desbloqueia para atender transações de outro membro do BSet, as demais são enfileiradas e só são atendidas após a reconfiguração. O BSet cresce dinamicamente, uma vez que quando um componente recebe uma requisição de um membro do BSet ele próprio de torna membro do BSet e integra o BSet estendido. Quando todos os membros do BSet original são bloqueados, os componentes do BSet estendido podem ser desbloqueados e a reconfiguração ter prosseguimento.

3.1.3 Abordagem orientada à conexão – Wermelinger

No modelo de sistema apresentado por Wermelinger (1999), o sistema é definido como um conjunto conectado de nós, onde uma conexão é dada por uma porta de iniciação e uma de recepção. Cada nó tem uma interface de nó na qual dependências entre portas são especificadas. Uma dependência entre portas é definida por uma porta iniciadora e outra receptora. A porta P1 é considerada dependente da porta P2, se ao ser recebida uma requisição em P1, uma transação é iniciada em conexões partindo de P2, sendo que transações são combinações entre portas de iniciação e recepção existentes nas interfaces dos nós dos sistemas. Assume-se que transações se completam em tempo finito e que o iniciador sabe quando terminam. Para evitar *deadlock*, as conexões e dependências entre portas não podem formar ciclos. Uma transação T1 é dita dependente de uma transação T2 se a sua porta

receptora e a porta iniciadora de T2 participam de uma relação de dependência ou se T1 é dependente de uma transação que depende de T2.

Na proposta apresentada, somente as conexões que serão removidas são bloqueadas. Para efetuar o bloqueio de uma conexão, se aguarda a transação em curso ser finalizada e não se inicia uma nova. Quando uma conexão está bloqueada o nó não atende transações que dela dependam. Para assegurar que o bloqueio de uma conexão não impedirá que outras transações pendentes sejam bloqueadas, o gerente de configuração ordena o bloqueio de acordo com a dependência entre transações: se uma transação T1 depende de outra transação T2 então a mensagem de bloqueio é enviada ao iniciador de T2 somente depois que T1 esteja bloqueada.

Os comandos de reconfiguração possíveis são criação e remoção de nós e conexões e solicitação de bloqueio de uma conexão. Para minimizar o tempo de interrupção do sistema, a execução dos comandos empregados pelo gerente de configuração é dada por uma ordem temporal <: se c1 < c2 então o comando c2 só pode ser executado depois que o comando c tenha completado. Comandos não relacionados podem ser executados em paralelo. Os seguintes relacionamentos são identificados:

1) Se uma transação T1 depende de uma transação T2, então T1 deve ser bloqueado antes de T2;
2) Uma conexão deve ser bloqueada antes de ser removida;
3) Um nó só pode ser removido depois que suas conexões tiverem sido removidas;
4) Um nó só pode ser conectado após sua criação;
5) Uma transação iniciada por um nó só pode ser estabelecida se não existirem outras transações iniciadas pelo nó a serem removidas.

3.1.4 Serviço de Reconfiguração Dinâmica para CORBA – Bidan e outros

Em Bidan (1998) é proposto um algoritmo para reconfiguração dinâmica baseado no Serviço de Clico de Vida do CORBA. Como objetivos, são identificados a minimização do tempo de interrupção da aplicação e a preservação de consistência, sendo definido que o estado de um objeto é consistente se não existem RPCs (*Remote Procedure Calls*) pendentes iniciados por aquele objeto. São tratados RPCs independentes, já que os autores argumentam que em CORBA a noção de RPCs aninhados não é especificada.

São descritas as seguintes ações de reconfiguração: *create*, *remove*, *link*, *unlink*, *transferLink* e *transferState*, algumas delas com precondições para serem realizadas:

1. Para que uma conexão seja removida (*unlink*) ela deve estar em estado passivo. Uma conexão encontra-se em estado passivo quando não existem requisições pendentes feitas através da conexão;
2. Para que um objeto seja removido (*remove*) as conexões que chegam a ele ou partem dele devem estar em estado passivo;
3. Para que as requisições de uma conexão c1 em estado passivo sejam transferidas para outra conexão c2 (*transferLink*) é necessário que c1 esteja em estado passivo e c2 exista;
4. Para que o estado de um objeto o1 seja transferido para outro objeto o2 (*transferState*) são considerados os mesmos pré-requisitos para a remoção de o1, além de ser necessário que o2 exista.

Para que sejam reconfiguráveis, os objetos da aplicação devem implementar a interface *RO_Object* que possibilita sua iteração com o *Dynamic Reconfiguration Manager* (DRM) que provê as primitivas de reconfiguração. Além disso, qualquer objeto reconfigurável redefine a operação *invoke*, provida para efetuar RPCs ao ORB, de forma que a requisição de RPC é efetuada somente quando a conexão alvo está válida e não está passiva.

Caso a conexão esteja inválida, é lançada uma exceção e, caso a conexão esteja em estado passivo, a *thread* de execução é bloqueada.

3.2 Tecnologias Relacionadas

Na construção de sistemas reconfiguráveis, algumas tecnologias podem contribuir para reduzir sua complexidade provendo soluções já validadas tanto para o desenvolvimento do sistema em si como para a infraestrutura de suporte, tais como *middleware* e sistemas operacionais. A seguir serão apresentadas algumas dessas tecnologias, tendo sido sua disposição inspirada parcialmente em artigo de McKinley (2004).

3.2.1 Separação de Interesses (*Separation of concerns*)

Embora orientação a objetos tenha introduzido o conceito de classes de forma a representar objetos similares, muitas vezes os requisitos não funcionais das aplicações continuam a ser codificados em diferentes classes, dificultando seu desenvolvimento e manutenção.

Separação de interesses permite a separação entre o desenvolvimento do comportamento funcional particular da aplicação e o código para implementação de suas responsabilidades transversais (*crosscutting concerns*), de interesse comum a vários módulos, tais como QoS, tolerância a falhas e segurança. Essa abordagem é utilizada na programação orientada a aspectos (AOP), na qual o código que implementa os aspectos transversais é desenvolvido separadamente das outras partes do sistema.

Kiczales e outros (1997) apresentam a distinção entre componentes e aspectos a depender da propriedade a ser implementada no sistema. A propriedade é classificada como um componente se pode ser encapsulada em um procedimento de forma bem localizada, com fácil acesso e composição, como é o caso de objetos, métodos, procedimentos e API

(*Application Programming Interface*). Caso contrário, é qualificada como aspecto, sendo que este representa propriedades que tendem a afetar o desempenho e semântica dos componentes de forma sistemática, tais como padrões de acesso à memória e sincronização de objetos concorrentes. A implementação baseada em AOP de uma aplicação é constituída por de uma linguagem de programação de componentes, uma ou mais linguagens de programação de aspectos, um combinador de aspectos (*aspect weaver*), um programa implementando os componentes na linguagem de programação de componentes e um ou mais programas implementando os aspectos nas linguagens adequadas. Em tempo de desenvolvimento, são especificados na implementação do componente os pontos em que os aspectos devem ser inseridos. A combinação da implementação dos componentes e dos aspectos pode ser efetuada pelo combinador em tempo de execução ou compilação, a depender de como ele foi projetado.

A separação de interesses simplifica a codificação dos aspectos não funcionais do sistema, além de facilitar a sua manutenção e evolução. Esses benefícios também podem ser úteis no desenvolvimento de *middlewares* com propriedade de adaptação, pois possibilitam separar os aspectos não funcionais da implementação do núcleo do *middleware*. Dessa forma, podem ser geradas versões configuráveis para aplicações de domínios específicos tais como aplicações de *tempo real*.

3.2.2 Reflexão Computacional

Através da reflexão computacional um programa é capaz de descobrir, e possivelmente alterar, seu próprio comportamento. Reflexão compreende duas atividades: introspecção para permitir que uma aplicação observe seu próprio comportamento, e intercessão para permitir que a aplicação modifique seu comportamento com base na observação. Um sistema reflexivo distingue entre objetos *base-level* (relacionados aos

aspectos funcionais dos sistemas) e objetos *meta-level* (relacionados a aspectos como políticas, mecanismos ou estratégias) (ROMÁN; KON; CAMPBELL, 2001). O *meta-level* é uma representação do *base-level* e modificações em um nível são refletidas no outro. Um protocolo *meta-object* (MOP) é uma interface que possibilita introspecção sistemática e intercessão nos objetos *base-level*. MOPs podem suportar reflexão estrutural e comportamental. Reflexão estrutural trata os aspectos relacionados a hierarquias de classes, interconexão de objetos e tipos de dados. Reflexão comportamental enfoca a semântica da aplicação de modo que a mesma possa se adaptar às condições do ambiente. O desenvolvedor pode utilizar os serviços reflexivos nativos de linguagens de programação ou fornecidos por uma plataforma de *middleware*.

No tocante a aplicações distribuídas, reflexão possibilita adaptar o seu comportamento através da modificação da própria aplicação ou do *middleware* que a suporta. *Middlewares* reflexivos se concentram em adaptar aspectos não funcionais de aplicações distribuídas tais como QoS, desempenho, segurança, tolerância a falhas e gerenciamento de energia. Particularmente, quando combinada com AOP, reflexão pode permitir que um MOP adicione código em uma aplicação em tempo de execução.

3.2.3 Desenvolvimento Baseado em Componentes

Componentes de *software* são unidades de *software* que podem ser produzidas independentemente por terceiros, implantadas e compostas (SZYPERSKI, 2002), permitindo dessa forma reutilização e construção de aplicações através da composição de diversos componentes COTS (*Commercial off-the-shelf*). Com a tecnologia de componentes, os sistemas são construídos através da integração de componentes com interfaces bem definidas, o que favorece sua reutilização e evolução, promovendo redução de custos e aumento de produtividade. Além disso, *frameworks* baseados em componentes tais como Microsoft

COM+ (MICROSOFT, 2000), .NET (MICROSOFT, 2000), *Sun Enterprise Java Beans* (SUN MICROSYSTEMS, 2006) e CORBA *Component Model* (OMG, 2006) promovem a separação de aspectos funcionais e não-funcionais dos sistemas, o que favorece a manutenção, flexibilidade e reutilização dos componentes, uma vez que o código para tratar os requisitos não funcionais da aplicação não se encontra embutido neles, a exemplo da separação entre componentes e contêineres, já que os componentes são responsáveis pelos requisitos funcionais e os contêineres, pelos não-funcionais (CONAN e outros, 2001). Outro benefício é o conceito de programação baseada em atributos, que permite descrever propriedades e requisitos de componentes de *software* independentes de sua implementação. Programação baseada em atributos pode ser feita através de: i) arquivos de configuração em XML associados ao componente de *software*, ii) meta-informação diretamente associada ao componente de *software* nos arquivos fonte e integrada no código executável como meta-dado, iii) modelos conceituais que provêm suporte à representação de QoS e outras especificações não funcionais (DE MIGUEL; RUIZ; GARCIA, 2002).

Outro aspecto relevante diz respeito à capacidade de reflexão. Modelos de componentes reflexivos apresentam interfaces que permitem inspecionar e modificar a estrutura do sistema, bem como monitorar e interferir no seu comportamento em tempo de execução.

Tais facilidades tornam o emprego da tecnologia de componentes na construção de sistemas com capacidade de reconfiguração dinâmica bastante promissor, uma vez que, por apresentarem baixo acoplamento e dependências explícitas e visíveis, os componentes de *software* favorecem o processo de mudança de configuração em tempo de execução (KRAMER; MAGEE, 1990).

Dois tipos de composição são possíveis: estática e dinâmica. Na primeira, a aplicação é construída através da combinação dos componentes em tempo de compilação. Na

composição dinâmica é possível adicionar, remover e configurar componentes em tempo de execução. Para tanto, a arquitetura deve suportar ligação tardia (*late binding*) que permite acoplar dois componentes em tempo de execução através de interfaces bem definidas.

Composição estática é útil para permitir que os sistemas se adaptem automaticamente aos recursos e configurações de *hardware* e *software* disponíveis, minimizando a necessidade de configurar as aplicações manualmente. Composição dinâmica vai além da estática no sentido em que permite às aplicações se adequarem às mudanças do ambiente, sem interromper seu funcionamento, razão pela qual é necessária nos sistemas de *tempo real* com demandas consideráveis de disponibilidade.

Quando suportado pelo *middleware*, desenvolvimento baseado em componentes provê um sistema flexível e extensível (SADJADI; MCKINLEY, 2003).

3.2.4 Padrões de Projeto

Desenvolvedores de *middleware* e aplicações se deparam com muitos desafios relacionados a tópicos de projeto e programação complexos, tais como persistência, organização de dados, gerenciamento de conexão, inicialização de serviços, distribuição, controle de concorrência e confiabilidade. Muitos desses desafios ocorrem repetidamente em várias aplicações e domínios. Até meados dos anos 90, o conhecimento necessário para solucioná-los encontrava-se restrito. Durante a última década, desenvolvedores de *software* experientes têm ajudado a superar esses problemas através da criação de um conjunto de documentos contendo conhecimento reutilizável denominados padrões (SCHIMIDT, 2003).

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas (GAMMA e outros, 2000) e ajudam a escolher alternativas de projeto que tornem o sistema reutilizável e a evitar alternativas que comprometam sua reutilização. Através do seu emprego, é possível evitar a adoção de soluções que já se mostraram insatisfatórias em

projetos anteriores e aproveitar as melhores, o que diminui o risco, tempo e custo de desenvolvimento.

Além de beneficiarem as aplicações, padrões de projeto também podem beneficiar o desenvolvimento dos *middlewares*: no TAO, por exemplo, foram empregados diversos padrões, tais como, *Wrapper Facade*, *Reactor*, *Acceptor-Connetor*, *Leader Follower*, entre outros. De acordo com seus desenvolvedores, o uso de padrões ajudou a melhorar a extensibilidade, manutenibilidade e desempenho desse *middleware* distribuído (SCHIMIDT, 2003). Também na arquitetura do serviço de reconfiguração proposta neste trabalho, foi utilizado o padrão *Command*.

3.2.5 Gerenciamento de Recursos

Gerenciadores de recursos mantêm registro dos recursos computacionais disponíveis e alocados. Dessa forma, uma vez conhecidos os recursos utilizados pelos componentes do sistema, podem prover informações ao mecanismo de reconfiguração sobre a viabilidade das mudanças solicitadas, evitando comprometer a estabilidade da aplicação. Em contrapartida, podem acionar a reconfiguração caso seja detectada necessidade de realocar os recursos existentes. Por exemplo, verificado que um recurso está ocioso em um nó e sobrecarregado em outro, o gerenciador de recursos poderia acionar uma reconfiguração que transferisse um componente para o nó com maior disponibilidade de recursos (KRAMER; MAGEE, 1985). A seguir são apresentadas algumas abordagens para o tema.

3.2.5.1 Núcleos de Sistema Operacional Orientados a Recursos (Resource Kernels)

Um núcleo orientado a recursos (NR) é um sistema que provê acesso aos recursos do sistema de forma temporal, garantida e protegida (EIDE e outros, 2004). O NR permite que aplicações especifiquem apenas suas demandas de recursos, sendo responsabilidade do

núcleo do sistema operacional (SO) satisfazer tais demandas utilizando esquemas de gerenciamento de recursos adequados. Uma aplicação informa os recursos de que necessita através da solicitação de reservas. Uma reserva representa a porção de um recurso compartilhado como CPU, disco ou rede. Baseado nas demandas temporais das reservas, o NR as prioriza e executa uma reserva de maior prioridade antes de uma de menor prioridade se ambas forem elegíveis para execução (RAJKUMAR e outros, 1998).

Em (OIKAWA; RAJKUMAR, 1998) é apresentado um NR baseado em Linux denominado Linux/RK. Neste trabalho, recursos compartilhados no espaço temporal são especificados através dos seguintes parâmetros: C, D, T, onde T representa o período de recorrência, C representa o tempo de processamento em T e D é o *deadline* dentro do qual as C unidades de processamento devem estar disponíveis após T. Um conjunto de recursos (*resource set*) representa um conjunto de reservas associado a um ou mais programas. Caso o núcleo de OS ou um gerenciador de QoS encontre algum desequilíbrio na utilização dos recursos, uma aplicação será notificada e poderá alterar seus parâmetros de QoS para equilibrar a utilização. Os seguintes mecanismos garantem que a utilização dos recursos seja baseada nas reservas:

8. Controle de admissão: Um teste de controle de admissão é efetuado sempre que uma nova reserva é solicitada para determinar se pode ser aceita ou não. Se a reserva pode ser admitida, uma reserva baseada nos parâmetros informados é criada.
9. Política de escalonamento: Uma política de escalonamento controla a alocação dinâmica de recursos, de modo que uma aplicação possa receber sua quantidade de recursos reservada.
10. Garantia: A utilização dos recursos por uma aplicação baseada nas reservas alocadas é garantida. Um mecanismo de garantia impede que um recurso seja utilizado em quantidade maior que a reservada.

11. Acompanhamento: A quantidade de recursos utilizada por uma aplicação é acompanhada. Tal informação é utilizada pela política de escalonamento e mecanismo de garantia. Além disso, essa informação pode ser consultada para fins de observação ou controle de alocação de recursos.

3.2.5.2 *Quality Objects* (QuO)

QuO (VANEGAS, 1998; ZINKY; BAKKEN; SCHANTZ, 1995) é uma *framework* baseado em CORBA para criação de aplicações distribuídas que se adaptam a diferentes níveis de QoS (qualidade de serviço) disponíveis. A interação entre o cliente e o objeto servidor é feita através de objetos delegados (*delegate objects*) localizados no lado cliente e gerados a partir de código escrito em uma linguagem de especificação de QoS (QDL), dos padrões de utilização esperados e dos requisitos de QoS entre o cliente e o objeto. Quando um método é chamado ou um resultado retornado, o delegado verifica o estado do sistema, conforme registrado por um conjunto de contratos e seleciona um comportamento baseado nele.

As regiões de operação e os requisitos de serviço de uma aplicação são especificados em contratos descritos através de uma linguagem de descrição de contratos (CDL). Cada contrato consiste em um número de regiões aninhadas, cada uma definida por um predicado sobre os valores de objetos de condições de sistema e um conjunto ações a serem executadas em caso de transições.

Objetos de condições de sistema provêm interfaces que possibilitam medir e controlar os estados de recursos, mecanismos e gerenciadores que são relevantes aos contratos. Alguns objetos de condições de sistema acionam avaliação de contratos quando os valores monitorados mudam. Outros, não acionam avaliação de contratos, mas provêm seus valores quando consultados.

QuO permite a especificação de tratadores, tanto no cliente quanto no objeto, a serem invocados quando ocorrem transições entre regiões.

3.2.5.3 *Quality-based Adaptive Resource Management* (QARMA)

O QARMA (FLEEMAN e outros, 2004) é um framework para gerenciamento de recursos baseado em CORBA que objetiva integrar vários serviços CORBA em uma infraestrutura para gerenciamento de recursos e consiste dos seguintes tipos de elementos:

- Monitores: obtêm informação sobre utilização de recursos, disponibilidade, desempenho da aplicação e condições ambientais;
- Detectores: avaliam subconjuntos particulares de informação no repositório do sistema para decidir pelo acionamento de um tomador de decisões para efetuar uma realocação de recursos;
- Tomador de decisões: utiliza um subconjunto da informação no repositório do sistema para decidir que ações devem ser tomadas para assegurar que os requisitos de desempenho fim-a-fim possam ser satisfeitos e que o desempenho geral seja aprimorado quando possível;
- Efetuadores (*enactors*): Recebem instruções do tomador de decisões sobre que ações efetuar no sistema e as executam.

O repositório do sistema armazena informação estática e dinâmica que descreve os sistemas de *software* e os recursos que existem no ambiente de computação. De acordo com os autores, um dos objetivos principais do QARMA é servir como um ponto de integração para vários serviços e mecanismos CORBA de forma a criar um *framework* de gerenciamento de recursos coerente.

3.2.5.4 *Real-Time Adaptive Resource Management* (RT-ARM)

O *Real-Time Adaptive Resource Management* (PAVAN e outros, 2001) provê mecanismos para negociação de QoS e adaptação. A arquitetura hierárquica do RT-ARM inclui gerenciadores de serviços que são entidades que encapsulam um conjunto de recursos e seus mecanismos de gerenciamento. O gerenciador de serviço de nível mais alto pode receber uma requisição para um serviço integrado que demande recursos de gerenciadores de recurso de níveis mais baixos. Gerenciadores de recursos de baixo-nível controlam diretamente recursos tais como CPU, rede ou memória. O RT-ARM suporta uma representação de QoS multidimensional em que cada dimensão especifica um intervalo aceitável para um parâmetro de qualidade da aplicação. Um conjunto de especificações de intervalos, um por dimensão, define uma região de QoS. Um gerenciador de serviço de nível mais alto traduz uma requisição de QoS para serviços integrados em requisitos individuais de QoS.

Para efetuar controle de admissão e adaptação, o RT-ARM aplica um protocolo de confirmação (*commit*) de duas fases, baseado em transações, recursivamente em cada gerenciador de serviço. A primeira fase lida com a reserva de recursos e executa um teste de disponibilidade originado do gerenciador de serviço que recebeu a requisição de admissão. O processo se move através de uma árvore cujos nós são os gerenciadores de serviço resultantes do processo de tradução da solicitação de QoS. Na segunda fase, o gerenciador de serviço iniciador verifica o sucesso da fase de reserva e confirma ou cancela a reserva de recursos ao longo da árvore. Caso o sistema não consiga alocar recursos suficientes, um gerenciador de recursos adapta uma aplicação de baixa criticalidade à sua QoS mínima contratada e transfere os recursos liberados para a nova aplicação. Posteriormente, quando recursos são liberados, o gerenciador de recursos expande o nível de QoS de aplicações mais críticas.

RT-ARM reconhece três situações em que a QoS de uma aplicação pode mudar depois da admissão:

- Contração de QoS de aplicações de baixa criticalidade quando uma nova aplicação inicia;
- Expansão de QoS quando aplicações finalizam e liberam recursos;
- Adaptação de *feedback* que depende do monitoramento da QoS disponibilizada e utiliza a diferença entre a desejada e obtida para adaptar o comportamento da aplicação.

Capítulo 4

Arquitetura de Controle e Supervisão

A Arquitetura para COntrole e Supervisão (ARCOS) (ANDRADE; MACÊDO, 2005; ANDRADE e outros, 2006; ANDRADE, 2006; ANDRADE; MACÊDO, 2007) é uma adaptação baseada em componentes do padrão DAIS (*Data Acquisition for Industrial Systems)* (OMG, 1999) elaborado pela OMG. Conforme mencionado, o serviço apresentado neste trabalho foi integrado ao ARCOS de modo a possibilitar a criação e remoção de seus componentes em tempo de execução, de forma consistente.

4.1 O padrão DAIS

O propósito da especificação DAIS é suportar a transferência de grande quantidade de dados de um processo industrial para uma variedade ampla de clientes, em *tempo real* e de forma eficiente, sendo dividida em três categorias de funcionalidades: *DAIS Server, DAIS Data Access e DAIS Alarms and Events*. O *DAIS Server* especifica as operações necessárias para a criação de sessões, grupos de aquisição e demais operações de interação com o cliente. O *DAIS Data Access* provê mecanismos para leitura, escrita ou subscrição de itens de dados presentes no servidor. A especificação DAIS *Alarms and Events* permite ao cliente efetuar subscrição ao recebimento de alarmes e eventos gerados por dispositivos e

outros sistemas. Eventos reportam mudanças de estado enquanto que alarmes possuem um estado de falha adicional ao seu próprio e têm a finalidade de chamar a atenção para uma condição que necessita de intervenção do operador. Uma implementação em conformidade com a especificação deve implementar o *DAIS Server*, deve implementar o *DAIS Data Access* ou o *DAIS Alarms and Events* e, finalmente, pode implementar o *DAIS Data Access* e o *DAIS Alarms and Events*.

O *DAISServer* é o ponto inicial de contato dos clientes DAIS, sendo que a organização dos dados a serem expostos por ele é feita sob a forma de árvore que pode conter informações tanto dos dados do ambiente (temperatura, pressão, velocidade, posição, etc) quanto das características dos sensores e atuadores tais como modelo e localização. O cliente solicita ao *DAISServer* a criação de um *DAISDataAccessSession* para gerenciar a sua utilização do servidor. Para que o cliente possa construir e visualizar a árvore DAIS do servidor, o *DAISDataAccessSession* disponibiliza um objeto *DAISNodeHome*. Através dele é possível criar novos nós e localizar nós específicos.

A obtenção periódica dos dados das árvores é possibilitada através do provimento do objeto *DAISGroupHome* ao cliente, também pelo *DAISServer*. O *DAISGroupHome* permite a criação de grupos de aquisição, nos quais são informados os nós da árvore que pertencem ao grupo bem como a freqüência em que os dados devem ser adquiridos. Também é possível criar grupos públicos, cujo acesso não se restringe a um determinado cliente. Para cada grupo criado, o *DAISGroupHome* instancia um objeto do tipo *DAISGroupManager* e retorna sua referência ao cliente, para que este possa incluir folhas indicativas de quais dados devem ser adquiridos através daquele grupo particular. O envio dos dados ao cliente pode ser feito de forma síncrona e assíncrona. Nessa última, os dados são enviados para um objeto de *callback* fornecido previamente pelo cliente através da invocação do método *cllbck* do *DAISGroupManager*. A escrita de dados nos dispositivos também pode ser efetuada de forma

síncrona e assíncrona. No último caso, o método *on_write_complete* é acionado no objeto de *callback* do cliente, sinalizando a atualização efetiva.

4.2 Funcionamento do ARCOS

Na sua implementação atual, o ARCOS é composto por três módulos: aquisição, controle e supervisão.

O módulo de aquisição de dados é responsável por enviar aos clientes DAIS os dados coletados da planta. Além disso, recebe informações para atuar sobre ela.

O módulo de controle é um cliente DAIS que recebe as informações monitoradas da planta obtidas pelo módulo de aquisição, executa os cálculos necessários para seu controle e produz informações de atuação que são encaminhadas ao módulo de aquisição para que a atuação seja efetuada.

O módulo de supervisão recebe mensagens de monitoramento da planta e pode produzir eventos, caso necessário.

Devido ao fato de o DAIS ter sido especificado com base no CORBA 2.x, foi feita uma reestruturação da especificação baseada em componentes sem, entretanto, impactar na implementação dos clientes, de modo a não prejudicar sua interoperabilidade com clientes já existentes.

A implementação DAIS realizada no ARCOS contempla os grupos de funcionalidade *DAIS Server* e *DAIS Data Access*, estando disponíveis a leitura de dados através de subscrição e a escrita de dados bloqueante e não-bloqueante.

Os componentes responsáveis pela configuração da aplicação, aquisição dos dados e controle são os seguintes: *DAISServer*, *DAISDataAccessSession*, *DAISNodeHome*, *DAISNodeIterator*, *DAISGroupHome*, *DAISGroupManager* e *DAISGroupEntryIterator*, *ControlManager* e *Controller*. Existe apenas uma instância do *DAISServer* e do

DAISNodeHome na aplicação. O papel desempenhado pelos componentes listados será detalhado a seguir.

4.2.1 Criação da árvore DAIS

Conforme mencionado, os dados a serem disponibilizados são organizados sob a forma de uma árvore, armazenada pelo *DAISServer*. Na fase de configuração, um objeto do tipo *DAISDataAccessSession* é fornecido pelo *DAISServer* representando a sessão do cliente em particular. O *DAISDataAccessSession* é então utilizado para obter o *DAISNodeHome*, que provê métodos para a construção e visualização da árvore DAIS.

De forma a permitir a comunicação do *DAISServer* com variados tipos de dispositivos, foi concebido o componente *DAISProvider*, que efetua a aquisição efetiva dos dados e deve ser conectado ao *DAISServer*. Sempre que o *DAISServer* recebe uma solicitação para criação ou atualização da árvore, ele delega a operação ao *DAISProvider* associado passando como parâmetro o objeto *DAISNodeHome*. Além disso, a leitura e escrita de dados nos dispositivos é feita pelo *DAISProvider* específico.

Para que o *DAISProvider* possa ser conectado ao *DAISServer* ele deve prover uma faceta derivada da interface *IDAISProviderBaseFacet*, que possui os métodos *build_dais_tree* e *get_value*, responsáveis pela construção efetiva da árvore e obtenção dos dados da planta atualizados, respectivamente. De forma que seja possível ao *DAISProvider* realizar as operações relacionadas à manutenção da árvore, ele recebe do *DAISServer* uma referência ao *DAISNodeHome*. Desse modo, a implementação concreta do *DAISProvider* deve invocar o método *add_node* do DAISNodeHome para construir a árvore desejada.

Caso haja interesse em consultar a árvore DAIS, o *DAISNodeHome* retorna um objeto *DAISNodeIterator* que permite percorrê-la.

4.2.2 Criação dos grupos de aquisição

No DAIS, o cliente deve especificar os dados de interesse a serem obtidos da planta através da criação de grupos. A partir do *DAISDataAccessSession*, o cliente obtém o *DAISGroupHome*, responsável pela criação dos grupos para aquele cliente específico. Para cada grupo criado é associado um *DAISGroupManager* e sua referência é retornada ao cliente, que a utiliza para indicar ao servidor DAIS as entradas da árvore DAIS em que está interessado. As entradas dos grupos também são organizadas em forma de árvore. No momento de criação do grupo, o cliente informa qual a taxa de atualização desejada, bem como o objeto de *callback* disponível para receber a atualização dos dados do grupo criado.

4.2.3 Aquisição dos dados

A sinalização do momento em que deve ser efetuada uma nova coleta de dados para o grupo é feita pelo *DAISGroupClock*, que gera eventos periódicos para o *DAISGroupManager* associado, na taxa especificada pelo grupo. Ao receber esses eventos, o *DAISGroupManager* obtém do *DAISProvider* os dados atualizados e os envia ao canal de eventos de *tempo real*. O próprio *DAISGroupManager* recebe esses dados do canal de eventos e os encaminha ao cliente através do objeto de *callback* previamente fornecido. Essa abordagem foi adotada devido ao fato de o Serviço de Eventos de *Tempo Real* utilizado consistir em um ponto de escalonamento, realizando o despacho de eventos considerando sua prioridade. Na Figura 3, os fluxos de 1 a 6 representam as operações de aquisição de dados.

4.2.4 Controle

O módulo de controle do ARCOS define um *framework* para implementação de malhas fechadas (OGATA, 2001) para o controle discreto de uma variável única, realizado

em um período de amostragem pré-definido. Este módulo define mecanismos flexíveis que permitem o emprego de estratégias de controle variadas.

O recebimento dos dados a serem utilizados no cálculo dos valores de atuação é feito pelo *ControlManage*r*,* desempenhando assim o papel de cliente no ARCOS. Nele deve ser conectado o componente *Controller* responsável por gerar as informações de atuação, de modo a manter a estabilidade da planta. Para que possa ser conectado ao *ControlManager*, o *Controller* deve prover uma faceta derivada da interface *IControllerBaseFacet* que possui o método *control*. Dessa forma, através da especialização do *Controller* podem ser implementadas diversas estratégias de controle. O objeto de *callback* do *ControlManager*, ao receber os dados da planta atualizados, os envia ao canal de eventos. Esse evento é recebido pelo *ControlManager* e encaminhado por este ao componente *Controller* para que ele realize um novo cálculo dos dados para atuação. Feito isso, o *ControlManager* envia ao *DAISGroupManager* as informações de atuação, que são repassadas ao *DAISProvider* para que a atuação seja efetuada no dispositivo adequado. Tal comportamento está representado na Figura 3, através dos fluxos de 7 a 11.

4.2.5 Supervisão

O ARCOS disponibiliza ainda as ferramentas *DAIS Server Browser* e *DAIS Server Manager* para o monitoramento do estado dos sensores a atuadores.

O *DAIS Server Browser* é um cliente DAIS que permite visualizar a árvore DAIS e adquirir dados de qualquer servidor DAIS. Através da ferramenta, é possível criar grupos de aquisição e monitorar as informações disponíveis por intermédio deles.

O *DAIS Server Manager* permite o monitoramento de um servidor DAIS, apresentando sessões já criadas e grupos existentes.

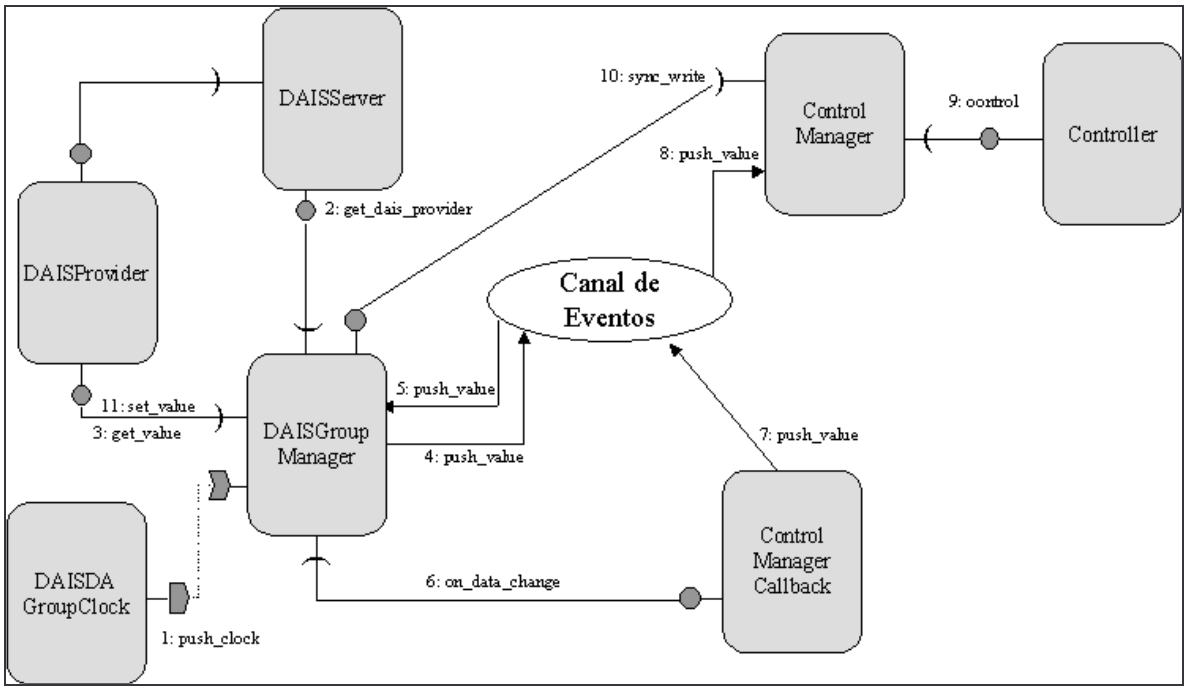


Figura 3. Funcionamento do ARCOS

Capítulo 5

Serviço de Reconfiguração Dinâmica

5.1 Objetivo

O serviço de reconfiguração dinâmica apresentado neste trabalho tem como objetivo dar suporte às mudanças da aplicação em tempo de execução, em especial àquelas desenvolvidas com a utilização do ARCOS, provendo mecanismos para a preservação do seu estado consistente. Neste intuito, algumas diretrizes foram estabelecidas:

a) Utilizar uma abordagem não muito intrusiva à estrutura das aplicações, de modo a facilitar sua utilização por aplicações já existentes;

b) Identificar e separar as funcionalidades necessárias à reconfiguração dinâmica e permitir sua integração e utilização coordenada, em conjunto ou em separado;

c) Prover mecanismos de configuração do serviço de modo a permitir sua adequação a variados tipos de aplicação;

d) Evitar a utilização de mecanismos e estruturas que possam comprometer ou trazer imprevisibilidade ao desempenho da aplicação alvo;

e) Aproveitar as características de distribuição e interoperabilidade das tecnologias de *middleware* disponíveis, o CIAO particularmente, para permitir a sua utilização em ambientes heterogêneos e distribuídos;

f) Contrabalançar usabilidade e complexidade, de forma a prover funcionalidades de suporte à reconfiguração, mas sem dificultar desnecessariamente a sua utilização.

5.2 Visão Geral da Arquitetura

O serviço de reconfiguração dinâmica é composto por três componentes distintos: *Reconfiguration Manager* (RM), *Consistency Manager* (CM) e *Deployment Manager* (DM). Além desses três elementos, os componentes da aplicação têm sua participação através da provisão de portas que lhes permitem interagir com os componentes do serviço, conforme ilustrado na Figura 4. Tais componentes foram definidos com o intuito de separar os papéis necessários à reconfiguração dinâmica e conceder maior flexibilidade de utilização, uma vez que, a depender da implementação de cada elemento e de sua integração com os demais, o serviço de reconfiguração pode assumir comportamentos distintos, adequando-se às aplicações alvo.

De modo geral, o RM se comunica com o CM para conduzir a aplicação a ser reconfigurada ao estado de reconfiguração, no qual as modificações estruturais poderão ser realizadas, sem prejuízo de consistência de estado de seus componentes. Para tanto, podem ser identificadas três fases distintas: fase de pré-reconfiguração, fase de reconfiguração e fase de pós-reconfiguração. Na fase de pré-reconfiguração a aplicação é conduzida ao estado de reconfiguração, de acordo com a abordagem de preservação de consistência implementada pelo CM. A fase de reconfiguração é iniciada com o recebimento pelo RM da notificação de que a fase de pré-reconfiguração foi concluída com sucesso. Nessa fase, o RM interage com o DM acionando as operações de modificação estrutural da aplicação. Concluídas as alterações pelo DM, o RM solicita ao CM a que reconduza a aplicação ao estado normal de execução, o que inicia a fase de pós-reconfiguração.

Na Figura 5 podem ser observadas as classes que compõem o modelo, que será detalhado posteriormente, sendo que apenas as operações relacionadas diretamente à modificação estrutural da aplicação e preservação de consistência foram incluídas, e seus parâmetros foram omitidos. A seguir os componentes do serviço serão descritos conceitualmente, e sua implementação detalhada na subseção seguinte.

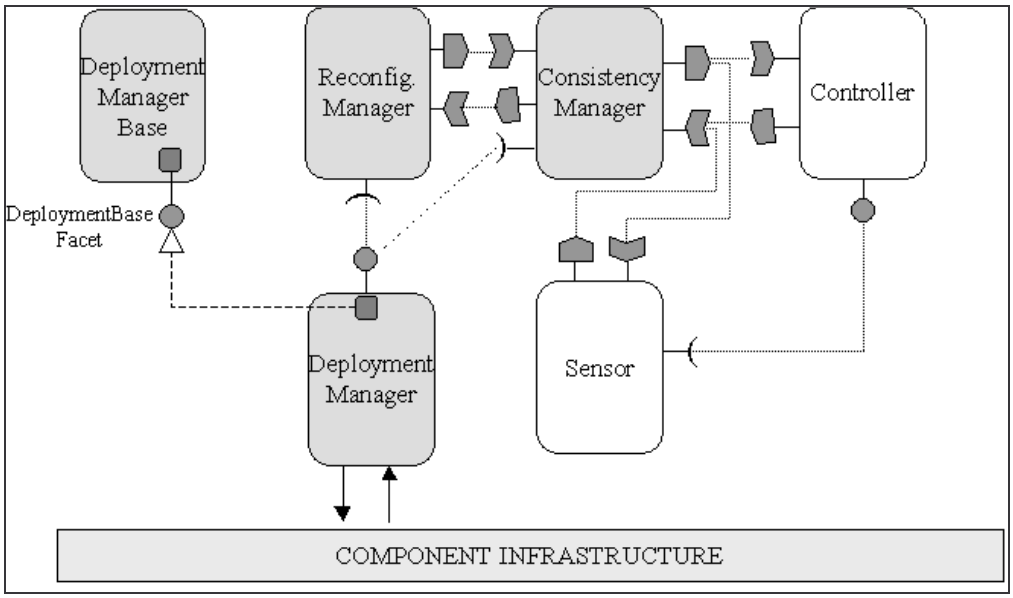


Figura 4: Integração entre serviço de reconfiguração e aplicação

5.2.1 *Reconfiguration Manager*

Conforme mencionado anteriormente, o RM desempenha o papel de gerenciamento da reconfiguração. É ele que recebe as solicitações de modificação do sistema especificadas sob a forma de ações de reconfiguração válidas e, com auxílio do CM e DM, as realiza.

Para tanto, o RM disponibiliza as operações *reconfigure*, *get_status* e *get_error* através de sua interface de provisão de serviços. Por intermédio da primeira, as alterações estruturais da aplicação podem ser comandadas, informando-se as ações de reconfiguração a serem efetuadas e se o CM deve ser acionado para conduzir a aplicação ao estado de reconfiguração. Caso o CM seja empregado, a fase de pré-reconfiguração é iniciada e o RM

aguarda o recebimento da notificação de sua conclusão bem sucedida, emitida pelo CM, para iniciar a fase de reconfiguração propriamente dita. Concluídas as modificações especificadas nas ações de reconfiguração, a fase de pós-reconfiguração começa e a aplicação é conduzida ao seu estado normal pelo CM. Caso não haja solicitação de utilização do CM na reconfiguração, a fase de reconfiguração é iniciada imediatamente procedendo-se as mudanças solicitadas na aplicação, sempre por intermédio do DM.

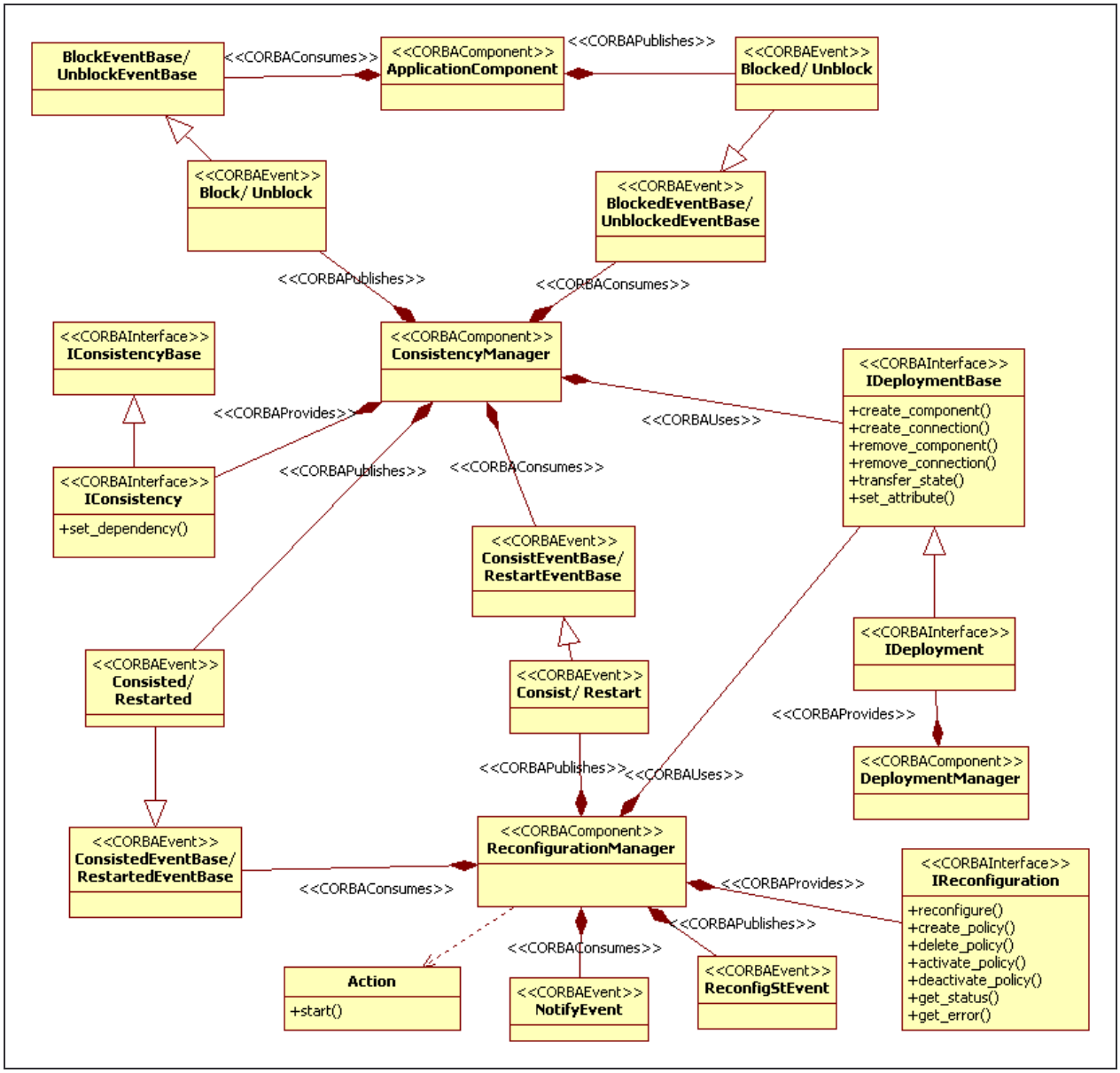


Figura 5. Classes do serviço de reconfiguração

A operação *get_status* permite inspecionar a situação do RM verificando se existe alguma reconfiguração em andamento. Caso o status indique que houve erro na tentativa de

reconfiguração, a operação *get_error* fornece os detalhes do erro ocorrido, informando em que etapa ele ocorreu, se durante a tentativa de alcançar o estado de reconfiguração ou durante as mudanças estruturais propriamente ditas. Caso o erro tenha acontecido durante a condução ou retirada do sistema do estado de reconfiguração, são disponibilizadas também as conexões que não puderam ser bloqueadas ou desbloqueadas, conforme o caso. Se o erro tiver ocorrido durante a reconfiguração, a ação em que ele ocorreu é indicada.

Além de comandar diretamente a reconfiguração através do acionamento da operação *reconfigure*, é possível definir políticas[1] segundo as quais a modificação estrutural da aplicação deve ser realizada. Para tanto, estão disponíveis as operações *create_policy*, *delete_policy*, *activate_policy* e *deactivate_policy* que permitem, respectivamente, a criação, remoção, ativação e desativação de políticas. Tais políticas são acionadas pelo RM quando este é notificado de um evento específico, previamente associado às ações de reconfiguração na ocasião da criação da política.

Vale observar que tanto a solicitação direta de reconfiguração, através da operação *reconfigure*, quanto o cadastramento de políticas, permite a determinação de um tempo máximo em que a reconfiguração deve ser realizada, como também do tempo máximo em que a solicitação feita ao CM de conduzir ou retirar o sistema do estado de reconfiguração deve ser concluída. Caso o tempo especificado seja alcançado e tais atividades não tenham sido finalizadas, a reconfiguração é suspensa e a expiração do tempo é comunicada ao solicitante da reconfiguração.

As ações de reconfiguração passíveis de serem comandadas através do RM, tanto diretamente quanto através da especificação de políticas, são a criação e remoção de componentes e conexões, a transferência de estado entre componentes e a alteração do valor de seus atributos.

---

[1] Entende-se por política neste trabalho a especificação de comportamentos distintos a depender do estímulo recebido pelo sistema.

5.2.2 *Consistency Manager*

O CM é o componente responsável por conduzir o sistema ao estado de reconfiguração, apropriado para a realização das mudanças estruturais em tempo de execução, de modo a garantir a permanência da aplicação em estado consistente durante essas alterações. Para tanto, determina quais componentes terão seu comportamento modificado para que a reconfiguração possa ser realizada, a depender das operações de reconfiguração a serem efetuadas e da abordagem de preservação de consistência adotada. Feito isso, interage com os componentes da aplicação para alcançar o estado necessário à reconfiguração. Essa interação é feita através de portas especificadas para essa finalidade, cujo tipo e comportamento podem variar a depender da abordagem de consistência empregada.

O CM toma conhecimento das ações de reconfiguração a serem realizadas considerando a consistência de estado através do recebimento de uma mensagem. Feito isso, comunica-se com os componentes da aplicação e, uma vez alcançado o estado de reconfiguração, informa o fato ao RM. Outra função exercida pelo CM é a de sinalizar aos componentes o final da reconfiguração, no momento determinado pelo RM. Caso o tempo máximo para a condução do sistema ao estado de reconfiguração tenha sido especificado e seja ultrapassado, o CM suspende as atividades e reporta ao RM o ocorrido.

Embora a definição das portas e do comportamento do CM seja dependente do tipo de abordagem de consistência adotada, no presente trabalho optou-se por utilizar a abordagem de bloqueio de conexões sugerida por Wermelinger (WERMELINGER, 1999), de modo a ilustrar de forma mais concreta a integração e utilização do serviço de reconfiguração. Tal abordagem foi adotada por permitir maior seletividade no bloqueio dos componentes, uma vez que não é necessário suspender a atividade de todo o componente durante a reconfiguração, ao invés disso, apenas as conexões que serão removidas sofrem bloqueio.

De modo a alcançar o estado de reconfiguração, o CM mantém o registro das conexões da aplicação e se comunica com os componentes pertinentes, solicitando o bloqueio de conexões, quando apropriado. Como ilustra a Figura 6, na fase de pré-reconfiguração iniciada após ser ativado pelo RM (2. *consist*), o CM efetua o bloqueio das conexões necessárias (3 e 4. *block*). Por sua vez, o componente informa ao CM quando a solicitação de bloqueio for atendida (5 e 6. *blocked*). Na abordagem de preservação de consistência adotada, as solicitações de bloqueios e desbloqueios consideram a dependência entre as conexões para evitar *deadlock*; ou seja, uma conexão é bloqueada somente quando as conexões que dela dependem já estiverem bloqueadas. Do mesmo modo, uma conexão é desbloqueada somente quando as conexões das quais depende já o tiverem sido.

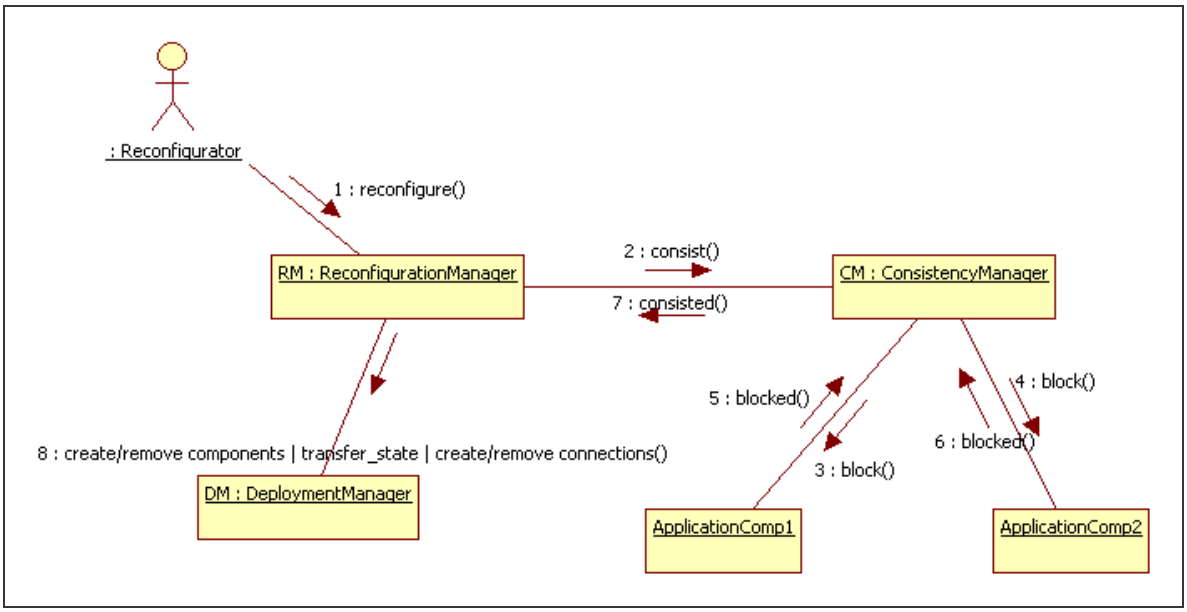


Figura 6. Interação entre os componentes do serviço e da aplicação nas fases de pré-reconfiguração e de reconfiguração

Uma vez que todas as solicitações de bloqueio tenham sido atendidas, o CM informa a situação ao RM (7. *consisted*), iniciando a fase de reconfiguração. Dessa forma, o RM prossegue com a reconfiguração interagindo com o DM (Seção 5.2.3). Conforme ilustrado na Figura 7, finda esta interação, começa a fase de pós-reconfiguração em que o RM solicita ao CM que desbloqueie as conexões (1. *restart*) que, por sua vez, notifica os

componentes a esse respeito (2 e 3. *unblock*), de modo que o sistema possa retornar ao seu estado normal de funcionamento. Após receber confirmação do desbloqueio das conexões pelos componentes notificados (4 e 5. unblocked), o CM informa ao RM o restabelecimento de funcionamento normal da aplicação (6. *restarted*).

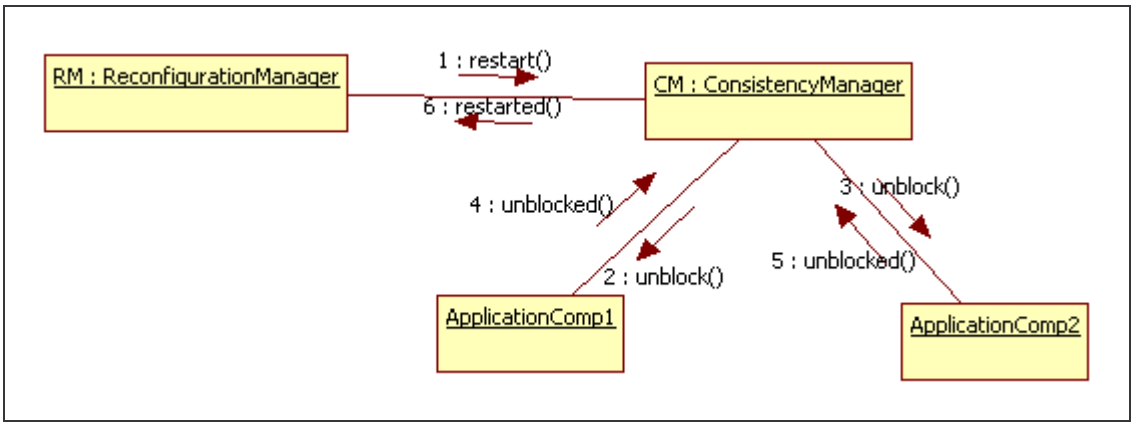


Figura 7. Interação entre os componentes do serviço e da aplicação na fase de pós-reconfiguração

A depender da abordagem de consistência adotada, o CM pode apresentar comportamentos distintos. Por exemplo, utilizando-se a abordagem de preservação de consistência descrita por Kramer e Magee (KRAMER; MAGEE, 1990), todos os nós que participam direta ou indiretamente de conexões a serem removidas devem ser conduzidos ao estado passivo, no qual apenas atendem requisições de cuja execução dependa a completude de outras. Já na abordagem orientada a conexões de Wermelinger (WERMELINGER, 1999), o estado para reconfiguração é alcançado através do bloqueio das conexões que serão removidas, obedecendo-se uma ordem dada pela dependência entre elas. Observa-se, portanto, que a determinação dos elementos a terem seu comportamento alterado e a ordem em que isso é realizado pode divergir a depender da estratégia empregada. Através da arquitetura apresentada no presente trabalho, é possível implementar *Consistency Managers* com comportamentos distintos e conectá-los ao *Reconfiguration Manager*, *Deployment Manager* e componentes da aplicação. Obviamente, estes últimos devem estar preparados

para interagir com o CM adotado através da provisão de portas de recepção e de emissão de eventos adequadas à estratégia adotada. No diagrama de sequência da Figura 8 pode ser visualizada a interação do RM, CM, DM e componentes da aplicação (*Sensor* e *Controller*) para a estratégia de consistência escolhida para este trabalho. Para fim exemplificativo, considera-se que, para que a aplicação alcance o estado de reconfiguração, os seus componentes *Sensor* e *Controller* devem ter conexões bloqueadas. As setas horizontais representam as mensagens trocadas pelos componentes. Conforme o diagrama, o acionamento do RM através da mensagem *reconfigure* (mensagem de número 2) dá início ao procedimento de reconfiguração, com envolvimento do CM, que se comunica com os componentes *Sensor* e *Controller* através das mensagens *block_connection* (4 e 6) e *blocked* (5 e 7) para conduzir a aplicação ao estado de reconfiguração. A operação *consisted* (8), sinaliza ao RM a conclusão bem sucedida da fase de pré-reconfiguração, de modo que o RM possa iniciar a fase de reconfiguração acionando as operações de alteração da estrutura da aplicação (9 a 12). O término da fase de reconfiguração é informado ao CM pelo RM através da mensagem *restart* (13). O CM, por sua vez, inicia a fase de pós-reconfiguração solicitando o desbloqueio dos componentes *Sensor* e *Controller*, de forma que a aplicação retorne ao seu estado normal de execução (mensagens 14 a 17). Finalmente, o CM notifica ao RM a conclusão da fase de pós-reconfiguração através da mensagem *restarted* (18). Adicionalmente, a mensagem *reconfig_status* (19) representa a publicação da situação da reconfiguração, de forma que os interessados possam verificar se houve erro durante os procedimentos.

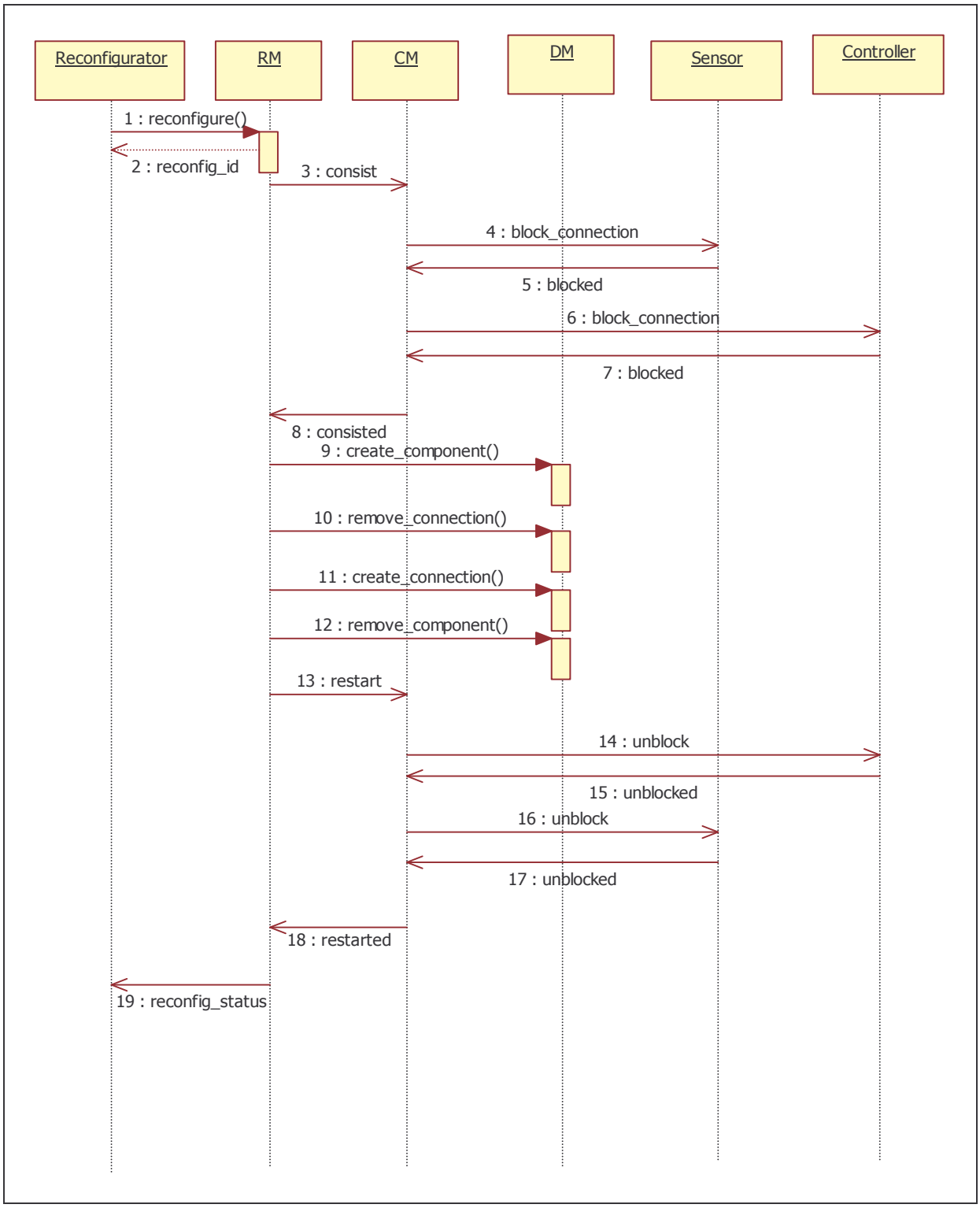


Figura 8. Diagrama de Seqüência do Serviço de Reconfiguração

5.2.3 *Deployment Manager*

O DM foi concebido com o propósito de intermediar a comunicação dos componentes do serviço com sua infraestrutura de execução, encapsulando as operações que

agem sobre a configuração da aplicação implantada, fornecendo uma interface padronizada para atuar sobre sua estrutura e preservando os demais componentes do serviço de reconfiguração de eventuais alterações de implementação sofridas pela infraestrutura adotada. Dessa forma, mesmo que o mecanismo de criação e remoção de componentes e conexões do ambiente de execução seja modificado, o *Deployment Manager* poderá ser adaptado de modo a diminuir o impacto dessas alterações junto a seus clientes. Outra funcionalidade importante diz respeito ao fato de que o DM é responsável por manter internamente, e disponibilizar quando solicitado, a configuração corrente da aplicação, com os componentes instanciados e conexões existentes entre eles.

As seguintes operações são disponibilizadas pela interface de provisão de serviços do componente para permitir a modificação da aplicação implantada:

- *create_component*: Através desta operação são criados novos componentes.
- *remove component*: Utilizada para remover componentes existentes.
- *create_connection*: Esta operação é utilizada para criar conexões entre componentes.
- *remove_connection*: Esta operação é empregada para remover uma conexão entre componentes.
- *transfer_state*: Essa operação deve ser utilizada para acionar a transferência de estados entre componentes da aplicação.
- *set_attribute*: Atualiza o valor de um atributo do componente.

5.3 Implementação

Para a validação da arquitetura proposta, o serviço de reconfiguração aqui apresentado foi construído utilizando-se o CIAO, implementação da especificação CCM, com enfoque em aplicações com necessidades específicas de QoS.

O *Deployment Manager* (DM), o *Consistency Manager* (CM) e o *Reconfiguration Manager* (RM) foram implementados como componentes CCM e, como tal, comunicam-se através das portas especificadas pelo modelo de componentes e ficam hospedados em um contêiner, que provê o seu ambiente de execução e intermedeia o seu acesso a serviços tais como notificação de eventos, persistência e segurança.

Na codificação do serviço projetado, optou-se por utilizar as funcionalidades disponíveis no modelo de componentes adotado, evitando-se alterar suas ferramentas de geração de código, o que poderia restringir a utilização do serviço e prejudicar sua compatibilidade com futuras versões do CIAO.

5.3.1 *Deployment Manager*

Embora o CIAO disponha do DAnCE e ReDaC como ferramentas para implantação (*deployment*) das aplicações, elas não foram consideradas adequadas para efetuar as alterações estruturais neste trabalho. Isso se deve ao fato de que, em ambas as ferramentas mencionadas, a criação e a remoção de componentes da aplicação são feitas a partir da varredura completa do arquivo XML com o descritor de implantação (*deployment descriptor*) e seu conseqüente processamento, o que pode incorrer em um acréscimo desnecessário no tempo necessário à reconfiguração. Além disso, tanto o DAnCE como o ReDaC utilizam estruturas internas para armazenar a configuração da aplicação, não havendo até a ocasião de implementação do serviço de reconfiguração aqui descrito, forma padronizada para inspecionar o conteúdo de tais estruturas, funcionalidade necessária ao gerenciamento da reconfiguração. Entretanto, alguns componentes do DAncE foram utilizados pelo serviço de reconfiguração, e serão identificados oportunamente.

Conforme mencionado na seção 2.3, através do DM é possível efetuar a criação e remoção de componentes e conexões e a transferência de estado entre componentes. Essas

operações estão disponíveis através da faceta *IDeploymentManager*, derivada da interface *IDeploymentManagerBase*, que é provida pelo componente, sendo que a funcionalidade de tais operações está descrita na Tabela 1 e a especificação de tais facetas e do DM encontra-se nas Figuras 9, 10 e 11.

Tabela 1: Descrição das operações da interface *IDeploymentManagerBase*

Operação	Descrição
create_component	Realiza a criação do componente a partir da informação da aplicação na qual ele será instanciado, o nó em que será situado, o nome e o tipo do componente e retorna a referência do mesmo. Para a criação do componente, o *Node Application* do DAnCE, responsável pelo nó informado, é localizado e então solicitada a instanciação do componente. A aplicação a que pertencem e o nome, nó, tipo, referência e situação de funcionamento dos componentes criados pelo DM ficam armazenados em sua base de dados interna que pode ser consultada posteriormente.
remove_component	Remove o componente que possui o identificador informado. Para tanto, a base de dados interna do DM é consultada e solicitado ao *Node Application* correspondente a sua remoção.
create_connection	Efetua a criação da conexão entre as portas dos componentes indicadas. A conexão é feita recuperando-se a referência dos componentes envolvidos na conexão. Feito isso, a faceta é obtida do componente que provê o serviço e conectada ao receptáculo do componente que utiliza o serviço. De forma análoga ao que ocorre com a criação de componentes, as informações das conexões também ficam armazenadas na base de dados interna do DM.
remove_connection	Realiza a remoção da conexão indicada pelo identificador informado. Através do identificador, o DM localiza a referência do componente que terá a faceta desconectada do seu receptáculo e o nome da respectiva porta. Uma vez que a operação de desconexão é invocada no componente que possui o receptáculo, ela só é acionada se o componente não estiver falho. Caso esse não seja o caso, apenas a base de dados do DM é atualizada.
transfer_state	Comanda a transferência de estado entre os componentes especificados como fonte e destino. Para tanto, o componente destino deve possuir uma faceta derivada da interface *IStateDestBase* e o componente do qual o estado será transferido precisa disponibilizar uma faceta derivada da interface *IStateSourceBase*. O DM localiza as referências dos objetos através dos seus identificadores e aciona os métodos *get_state* e *set_state* nos componentes que fornecerão e receberão o estado, respectivamente.
set_attribute	Atualiza o valor do atributo especificado no componente desejado.
register_type	Armazena os dados de implementação de um determinado tipo de componente passível de ser criado dinamicamente pelo DM.

Tabela 1: Descrição das operações da interface *IDeploymentManagerBase* (continuação)

Operação	Descrição
Configure	Acrescenta na base de dados do DM as informações de componentes e conexões implantados da aplicação identificada pelo nome do plano de implantação (*deployment plan*) passado como parâmetro.
get_component_ref	Localiza a referência do componente a partir de seu identificador e a retorna.
get_component_type	Informa o tipo do componente a partir do seu identificador.
get_connection_info	Retorna a referência dos componentes, nomes e tipos de portas dos componentes envolvidos na conexão a partir do seu identificador.
get_component_info	Retorna a referência e tipo do componente instanciado, bem como o nó e a aplicação em que está instanciado.
update_status	Atualiza internamente a situação do componente em caso de sua falha ou recuperação. Desse modo, caso o componente seja detectado como falho é possível notificar o DM através dessa operação. Isso é necessário para que não seja solicitada a desconexão a um componente falho, uma vez que não há garantia de que seja executada corretamente.

Através do método *configure*, o DM obtém do CIAO, e armazena em sua base de dados interna, os componentes e as conexões implantados originalmente da aplicação cujo identificador é passado como parâmetro. Tais informações são atualizadas à medida que são feitas solicitações de alterações estruturais ao DM.

Na implementação corrente, a instalação das bibliotecas dos componentes deve ser feita através do DAnCE ou ReDaC, cabendo ao *Deployment Manager* a criação e remoção de componentes cuja implementação já esteja instalada no ambiente de execução, sendo que o *Deployment Manager* disponibiliza a operação *register_type* para que sejam registradas as informações das bibliotecas de componentes instaladas no ambiente de execução, para cada tipo de componente passível de ser criado dinamicamente. Para a criação do componente, o DM localiza o *Domain Application Manager* da aplicação cujo nome é passado como parâmetro na invocação do método *create_component*. A partir do *Domain Application Manager*, o DM obtém o *Node Application* do nó no qual se deseja instanciar o novo componente. Feito isso, solicita a criação do componente ao *Node Application* informando os

dados das bibliotecas do componente. A remoção do componente é feita solicitando ao *Node Application* que remova o componente determinado pelo identificador informado.

```
1   module ReconfigService
2   {
3      module Deployment
4      {
5        struct CompInfo {
6          string app_name;
7          string id;
8          string node;
9          string type;
10       };
11
12       struct ConnInfo {
13         string id;
14         string type;
15         string port_name_ini;
16         string instance_ini;
17         string port_name_rec;
18         string instance_rec;
19       };
20
21       struct CompTypeInfo {
22         string name;
23         string executor_dll;
24         string executor_entrypt;
25         string servant_dll;
26         string servant_entrypt;
27       };
28
29       struct ConnInfoDetail {
30         string id;
31         string instance_ini;
32         string port_name_ini;
33         string type_ini;
34         string instance_rec;
35         string port_name_rec;
36         string type_rec;
37         string conn_type;
38       };
39
40       struct  CompInfoDetail {
41         string id;
42         ::Components::CCMObject component_ref;
43         string type;
44         string node;
45       };
46
47       interface IDeploymentManagerBase
48       {
49           Components::CCMObject create_component (in CompInfo
    comp_info) raises (ReconfigService::RSException);
50           void remove_component (in string id) raises
    (ReconfigService::RSException);
51           void create_connection (in ConnInfo conn_info) raises
    (ReconfigService::RSException);
```

Figura 9: Definição IDL da faceta base do *Deployment Manager*

```
52          void remove_connection (in string id) raises
    (ReconfigService::RSException);
53              void transfer_state (in string instance_source, in
    string instance_dest) raises (ReconfigService::RSException);
54          void set_attribute (in string instance, in string name,
    in string type, in string value) raises
    (ReconfigService::RSException);
55          void register_type (in CompTypeInfo type_info);
56          void configure (in string info);
57          Components::CCMObject get_component_ref(in string
    comp_id);
58          string get_component_type (in string comp_id);
59          ConnInfoDetail get_connection_info(in string conn_id);
60          CompInfoDetail get_component_info(in string comp_id);
61          void update_status (in string component_id, in unsigned
    short status);
62       };
63    };
64 };
```

Figura 9: Definição IDL da faceta base do *Deployment Manager* (continuação)

```
1  module ReconfigService
2  {
3     Module Deployment
4     {
5        interface IDeploymentManager : IDeploymentManagerBase
6        {
7        };
8     };
9  };
```

Figura 10: Definição IDL da faceta concreta do *Deployment Manager*

```
1   module ReconfigService
2   {
3      module Deployment
4      {
5         component DeploymentManager
6         {
7           provides
    ::ReconfigService::Deployment::IDeploymentManager dm_fac;
8             uses multiple
    ::ReconfigService::Deployment::IStateSourceBase dm_state_source;
9             uses multiple
    ::ReconfigService::Deployment::IStateDestBase dm_state_dest;
10        };
11        home DeploymentManagerHome manages DeploymentManager
12        {
13        };
14     };
15  };
```

Figura 11: Definição IDL do componente *Deployment Manager*

Para criar uma conexão, as referências dos componentes a serem conectados são recuperadas da base de dados com as informações de configuração da aplicação, e a conexão é solicitada ao componente em cujo receptáculo será conectado a faceta. A remoção de

conexões é efetuada com base nos seus identificadores, informados na ocasião de sua criação. Ao ser solicitada a remoção de uma conexão, o DM consulta sua base de dados para obter a referência e a porta do componente que terá sua conexão excluída. Após localizar a referência do componente que participa da conexão através de um receptáculo, o DM solicita a esse componente a desconexão. O tipo de conexão tratado correntemente pelo serviço é aquela efetuada entre receptáculos simples e facetas.

Para a transferência de estado entre componentes, devem ser informados os identificadores do componente que participará como fonte do estado e o componente destino. Também nesse caso, o DM consulta sua base de dados interna para obter as referências dos componentes que participarão da operação. Tais componentes precisam prover facetas derivadas dos tipos *IStateSourceBase* e *IStateDestBase*. A faceta *IStateSourceBase* define o método *get_state* que deve ser implementado pelo componente do qual o estado será obtido. Conforme pode ser visto na Figura 12, esse método pode receber como parâmetro o identificador do componente. Tal parâmetro pode ser informado caso o componente do qual o estado será obtido armazene o estado de mais de um componente. Para que seja possível manipular valores de tipo arbitrário, o retorno do método *get_state* é do tipo *Any*. Dessa forma, para que um componente funcione como fonte de estado ele deve prover uma faceta especializada a partir de *IStateSourceBase*. Em contrapartida, um componente que irá receber o estado deve prover uma faceta derivada do tipo *IStateDestBase*. No Código 4 pode ser verificado que essa faceta declara o método *set_state* que, por sua vez, recebe como parâmetro um valor do tipo *Any*. O DM utiliza essas facetas quando acionado o método *transfer_state*, cujo comportamento é obter o estado do componente fonte do estado através da chamada ao método *get_state* e repassá-lo para o componente destino através do método *set_state*. O relacionamento entre tais facetas e o DM está representado na Figura 13.

```
1   Module ReconfigService
2   {
3      Module Deployment
4      {
5        interface IStateSourceBase {
6          any get_state(in string component_id);
7        };
8        interface IStateDestBase {
9          void set_state (in any state);
10       };
11     };
12  };
```

Figura 12: Arquivo IDL com definição das interfaces para transferência de estados

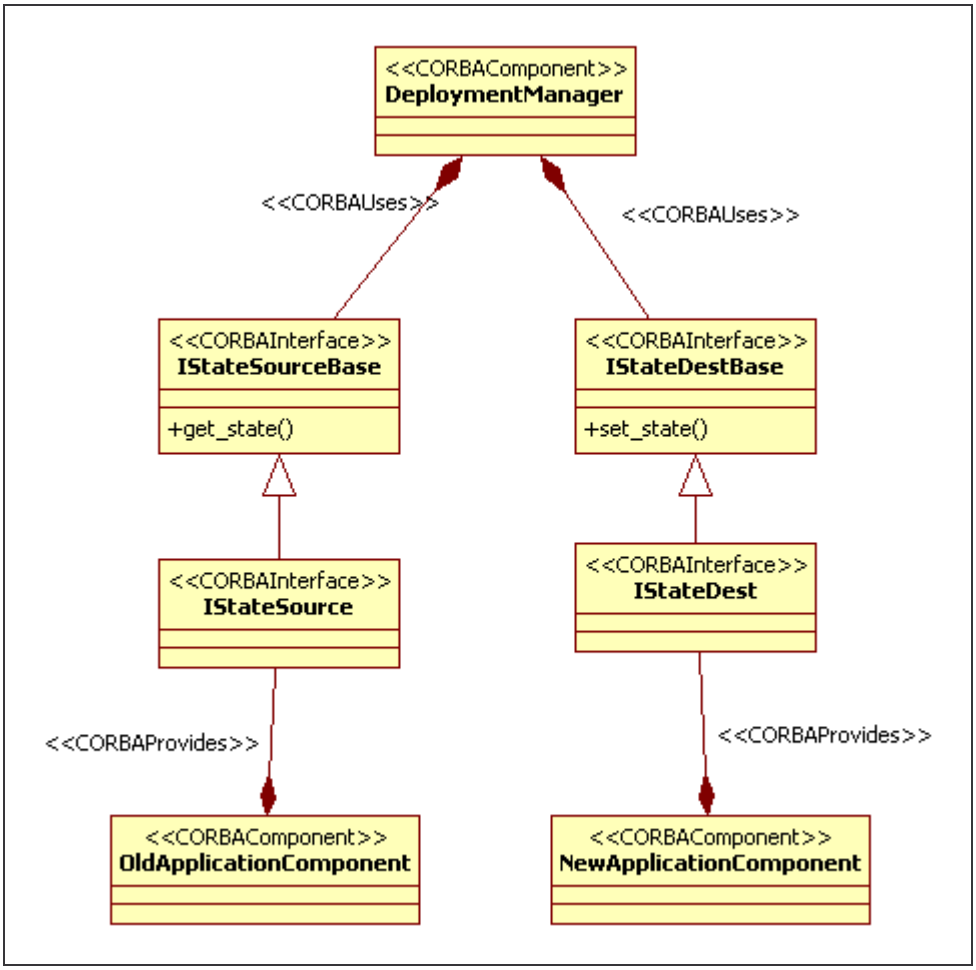


Figura 13. Hierarquia de classes para transferência de estados

Para atualização do atributo do componente, o método *set_attribute* obtém o *StandardConfigurator* a partir de sua referência. A interface *StandardConfigurator* é especificada pelo CCM e implementada pelo CIAO e provê um objeto para especificar as configurações de atributos do componente. Acionando-se a operação *set_attribute* do

configurador passando como parâmetros os nomes dos atributos e respectivos conteúdos convertidos para o tipo *Any*, os valores pertinentes são atualizados no componente.

As operações *get_component_ref, get_component_type*, *get_connection_info* e *get_component_info* são operações auxiliares providas pelo do DM para obter as informações dos componentes e conexões a partir dos seus identificadores.

A ocorrência de falha ou reinício de um componente é notificada ao DM através da operação *update_status*. Tal informação é utilizada para evitar que seja solicitada a um componente falho a desconexão de uma faceta do seu receptáculo.

5.3.2 *Reconfiguration Manager*

O *Reconfiguration Manager* possibilita o acionamento da reconfiguração através da implementação da interface *IReconfigManager*, conforme exibido nas Figuras 14 e 15. Uma vez aceita a solicitação de reconfiguração, o RM interage com o CM e o DM para efetivar as alterações solicitadas constantes no *script* de reconfiguração. Na Tabela 2 consta a descrição das operações providas através da interface *IReconfigManager*.

A reconfiguração pode ser acionada tanto através da operação *reconfigure* quanto pela criação de políticas. Conforme apresentado na Figura 14, o método *reconfigure* recebe como parâmetros o nome do arquivo que contém o *script* de reconfiguração, a indicação se a reconfiguração deve ser consistida e, caso essa indicação seja positiva, o tempo máximo em que a aplicação deve atingir o estado de reconfiguração, quando solicitado. Tal tempo também é considerado como o tempo máximo para a aplicação sair desse estado. Caso a consistência seja solicitada e esse valor seja zero, o tempo não será limitado. Pode ser especificado também o tempo máximo total em que a reconfiguração deve ser realizada. Esse tempo abrange o tempo para condução e retirada do sistema do estado de reconfiguração e a execução das ações propriamente ditas entre essas duas fases. Caso seja atribuído um valor

não nulo ao tempo máximo de reconfiguração, o RM configura um componente auxiliar to tipo *Clock*, que é responsável pela sinalização da passagem do tempo ao RM. Ao receber o evento de *timeout*, o RM suspende a reconfiguração atualizando seu estado e sinalizando o fato aos componentes interessados.

```
1   module ReconfigService
2   {
3      module Reconfiguration
4      {
5        enum ReconfigStatusValue {IDLE, ONGOING, FINISHED, ERROR};
6        enum ReconfigKindValue {NONE, STARTED, POLICY};
7
8        struct ReconfigStatus {
9          ReconfigStatusValue status;
10         ReconfigKindValue kind;
11         string identifier;
12       };
13
14       struct ReconfigError {
15         string identifier;
16         unsigned long action_number;
17         RSErrorReason error_reason;
18         string error_detail;
19         ObjectIds obj_ids;
20       }
21
22       interface IReconfigManager {
23         string reconfigure (in string script, in unsigned short
    consist, unsigned long timeout_reconfig, unsigned long
    timeout_consist);
24         void create_policy (in string name, in string event_name,
    in string script_file, in unsigned short consist, unsigned long
    timeout_reconfig, unsigned long timeout_consist);
25         void delete_policy (in string event_name);
26         void activate_policy (in string event_name);
27         void deactivate_policy (in string event_name);
28         ReconfigError get_error (in string identifier);
29         ReconfigStatus get_status(in string identifier);
30       };
31
32       eventtype NotifyEvent {
33         public string event_name;
34         public string event_type;
35       };
36
37       eventtype ComponentFailedEvent:NotifyEvent {
38         public string component_id;
39       };
40
41       eventtype ReconfigStEvent {
42         public ReconfigStatus rec_status;
43       };
44    };
45  };
```

Figura 14: Arquivo IDL da interface *IReconfigManager*

Na criação das políticas, além dos parâmetros passados à operação *reconfigure*, são informados o nome da política e o evento que a aciona. O RM é notificado da ocorrência de um evento que aciona uma política através de uma porta do tipo *NotifyEvent*. Esse tipo de evento pode ser especializado para comportar eventos específicos. Nas linhas 37 a 39 da Figura 14 pode-se observar a criação de um evento do tipo *ComponentFailedEvent*, utilizado para reportar a falha de um componente ao RM, e que, além dos parâmetros *event_name* e *event_type* herdados da classe base, define o parâmetro *component_id* que identifica o componente falho. Ao receber um evento do tipo *ComponentFailedEvent*, o RM verifica se existe alguma política registrada para ele e, em caso afirmativo, aciona o *script* de reconfiguração adequado.

```
1   module ReconfigService
2   {
3      module Reconfiguration
4      {
5         component ReconfigurationManager
6         {
7            provides IReconfigManager rm_fac;
8            publishes ::ReconfigService::Consistency::Consist
    rm_consist_pub;
9            publishes ::ReconfigService::Consistency::Restart
    rm_restart_pub;
10           Consumes
    ::ReconfigService::Consistency::ConsistedEventBase
    rm_consisted_cons;
11           Consumes
    ::ReconfigService::Consistency::RestartedEventBase
    rm_restarted_cons;
12           Uses
    ::ReconfigService::Deployment::IDeploymentManagerBase rm_dm_rec;
13           consumes NotifyEvent rm_notify_cons;
14           publishes ReconfigStEvent rm_status_pub;
15           Consumes
    ::ReconfigService::Consistency::ConsistencyError
    rm_cons_error_cons;
16        };
17        home ReconfigurationManagerHome manages
    ReconfigurationManager {};
18     };
19  };
```

Figura 15: Definição IDL do *Reconfiguration Manager*

Nas Figuras 16 e 17 são listados trechos do XML *Schema* que define o conteúdo dos arquivos XML válidos a serem submetidos ao RM e de arquivo XML exemplificando cada comando válido em um *script*, respectivamente.

No esquema do arquivo XML constam os elementos que podem ser incluídos no *script*, os quais correspondem às ações reconfiguração possíveis. Dessa forma, para definir uma ação de reconfiguração para a criação de um componente, é necessário acrescentar no arquivo de XML fornecido ao RM, uma entrada rotulada como *createComponent* e informar seus atributos, que são o identificador do componente, o nó em que será instanciado, seu tipo e a aplicação a que pertencerá. Para a criação de conexões deve ser utilizado o rótulo *createConnection* e informados o identificador da conexão e seu tipo, como também o nome da instância e da porta do componente iniciador da conexão – aquele que participa dela através de um receptáculo, e o nome da instância e da porta do componente que tem o papel de receptor na conexão – aquele que participa por meio de uma faceta.

Tabela 2: Descrição das operações da interface IReconfigManager

Operação	Descrição
reconfigure	Comanda a reconfiguração especificada pelo *script* cujo nome é passado como parâmetro. Caso o parâmetro *consist* tenha valor 1, o *Consistency Manager* é acionado. Caso o valor do *timeout_consist* seja diferente de zero, ele especifica tanto o tempo máximo (em microsegundos) em que a aplicação deve alcançar o estado de reconfiguração, quanto o tempo máximo em que deve sair dele, quando solicitado. De modo análogo, o valor de *timeout_reconfig* determina o tempo máximo em que a reconfiguração deve ser realizada. Caso algum dos tempos expire, a reconfiguração é suspensa e a ocorrência é informada ao solicitante. O identificador da nova reconfiguração é retornado se já não houver reconfiguração em andamento.
create_policy	Registra junto ao RM um script de reconfiguração a ser executado na ocorrência de um evento determinado. São informados como parâmetros, o nome da política, o nome do evento que a aciona, o *script* a ser executado quando ela é acionada e se as ações constantes no *script* devem ser consistidas. Também na criação de políticas é possível especificar os tempos máximos para a condução e retirada do sistema do estado de reconfiguração e da reconfiguração como um todo.
delete_policy	Remove a política solicitada.

Tabela 2: Descrição das operações da interface IReconfigManager (continuação)

Operação	Descrição
activate_policy	Modifica a situação da política para ativa, o que faz com que seja considerada na notificação de um evento. As políticas são criadas como ativas por padrão.
deactivate_policy	Desativa a política especificada.
get_status	Retorna a situação da reconfiguração para o identificador informado como parâmetro. O valor ONGOING (em andamento) ou IDLE (ocioso) retornado para o campo status indica a existência ou não de reconfiguração em andamento. Caso o valor do campo status seja diferente de IDLE, o campo *kind* informa se a reconfiguração foi ativada diretamente (STARTED) ou acionada através de uma política (POLICY). O valor ERROR para o campo status sinaliza a existência de erro durante a reconfiguração. Caso o valor desse campo seja FINISHED, a reconfiguração foi completada normalmente. Caso o identificador informado como parâmetro seja uma string vazia, um dos valores IDLE ou ONGOING com a situação corrente de reconfiguração é retornado. Se não houver reconfiguração em andamento, o campo status terá o valor IDLE, o campo *kind* assumirá o valor NONE e o campo *identifier* estará vazio.
get_error	Caso a reconfiguração tenha apresentado erro em alguma das suas fases, essa operação retorna o seu detalhamento. O campo *error_reason* é preenchido com o motivo do erro, de acordo com os tipos de erro previamente especificados no sistema, sendo que uma descrição livre do erro pode ser armazenada no campo *error_detail*. No campo *action_number* fica registrada a ação em que ocorreu o erro, caso ele tenha ocorrido na fase de efetivação da reconfiguração. Se o erro houver ocorrido durante a fase de consistência, as conexões que porventura não puderam ser conduzidas ou retiradas do estado de reconfiguração, a depender do estágio em que se encontra a reconfiguração, são listadas no campo *obj_ids*.

Para a alteração de um atributo através do RM deve-se incluir no *script* um elemento *setAttribute* com o nome da instância que terá o valor do atributo modificado, o nome, tipo e novo valor do atributo. A transferência de estados é acionada através do rótulo *transferState*, que possui como atributos dois identificadores: o da instância que fornecerá e o da que receberá o estado. Para a remoção de componentes e conexões basta que sejam informados seus identificadores por meio dos elementos *removeComponent* e *removeConnection*, respectivamente. Os comandos de reconfiguração são executados na

ordem em que aparecem no *script* e são encaminhados ao DM para que este efetue as modificações.

```
1    <?xml version="1.0" encoding="ISO-8859-1"?>
2    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.ufba.br"
4    xmlns="http://www.ufba.br" elementFormDefault="qualified">
5
6      <xs:element name="createComponent">
7        <xs:complexType>
8          <xs:sequence>
9            <xs:element ref="id" />
10           <xs:element ref="node" />
11           <xs:element ref="type" />
12           <xs:element ref="applicationId" />
13         </xs:sequence>
14       </xs:complexType>
15     </xs:element>
16
17     <xs:element name="createConnection">
18       <xs:complexType>
19         <xs:sequence>
20           <xs:element ref="id" />
21           <xs:element ref="type" />
22           <xs:element ref="initiator" />
23           <xs:element ref="receptor" />
24         </xs:sequence>
25       </xs:complexType>
26     </xs:element>
27   …
130  </xs:schema>
```

Figura 16: Trecho do XML Schema do *script* de reconfiguração recebido pelo RM

Quando a reconfiguração é acionada diretamente através do método *reconfigure*, o RM verifica a inexistência de reconfiguração em andamento e, nesse caso, gera, armazena e retorna o identificador da nova reconfiguração. Se a reconfiguração for solicitada em conseqüência da notificação de um evento registrado em uma política e não houver reconfiguração em andamento, o identificador da nova reconfiguração fica sendo o nome do evento sinalizado. Esse identificador pode ser utilizado pelos clientes do RM para consultar o andamento da reconfiguração através da operação *get_status*. Tal método retorna uma estrutura do tipo *ReconfigStatus*, detalhada na Tabela 3, que contém as informações relativas ao andamento da reconfiguração em andamento. Se a situação da reconfiguração indicar que houve erro, a estrutura *ReconfigError* (Tabela 4) é preenchida com o detalhamento desse erro. Na Tabela 5 são listados os possíveis erros capturados durante a reconfiguração.

```
1    <?xml version="1.0"?>
2    <script xmlns="http://www.ufba.br"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://www.ufba.br script.xsd">
5
6    <command>
7      <createComponent>
8        <id>NewController-idd</id>
9        <node>Node1</node>
10       <type>Controller</type>
11       <applicationId>Cruise-Control</applicationId>
12     </createComponent>
13   </command>
14   <command>
15     <removeConnection>
16       <id>sensor-controller</id>
17     </removeConnection>
18   </command>
19   <command>
20     <transferState>
21       <instanceSource>Controller-idd</instanceSource>
22       <instanceDest>NewController-idd</instanceDest>
23     </transferState>
24   </command>
25   <command>
...  ...
52   </script>
```

Figura 17: Exemplo de arquivo XML com trecho de *script* de entrada para o RM

Além da possibilidade de obter a situação da reconfiguração através da operação *get_status*, o RM disponibiliza um evento cujo conteúdo é a estrutura *ReconfigStatus*. Dessa forma, os componentes interessados no resultado da reconfiguração podem realizar subscrição para receber tais eventos. Esses eventos são enviados logo após a conclusão da reconfiguração com sucesso ou após a ocorrência de erro.

Tabela 3: Descrição dos campos retornados através da estrutura *ReconfigStatus*

Campo	Descrição	Valores assumidos
Status	Indica se existe reconfiguração em andamento	IDLE – Não existe reconfiguração em andamento ONGOING – Existe reconfiguração em andamento FINISHED – A reconfiguração foi terminada normalmente ERROR – Ocorreu erro durante a reconfiguração

Tabela 3: Descrição dos campos retornados através da estrutura *ReconfigStatus* (continuação)

Campo	Descrição	Valores assumidos
Kind	Armazena o tipo de reconfiguração	NONE – Tipo assumido quando não há reconfiguração em andamento STARTED – A reconfiguração foi acionada diretamente através da invocação da operação *reconfigure* POLICY – A reconfiguração foi acionada pela notificação de evento registrado através da uma política
Identifier	Contém o identificador da reconfiguração	Caso o campo *kind* tenha o valor STARTED, seu conteúdo é o identificador gerado no acionamento da reconfiguração Se o valor do campo *kind* for POLICY, o identificador é preenchido com o nome do evento cuja notificação acionou a reconfiguração.

Tabela 4: Descrição dos campos retornados através da estrutura *ReconfigError*

Campo	Descrição	Valores assumidos
action_number	Registra a posição da ação de reconfiguração que ocasionou erro	O valor permanece zerado se não ocorrer erro durante a reconfiguração. Caso contrário, o número de ação para a qual o erro foi detectado é armazenado. Esse número corresponde à posição da ação no script de reconfiguração submetido, sendo que a contagem se inicia em 1.
error_reason	Contém o motivo do erro	Os possíveis motivos de erro são padronizados pelo sistema e preenchidos de acordo pelos módulos componentes do serviço de reconfiguração. Ver Tabela 5.
error_detail	Armazena texto detalhando o erro ocorrido	Campo de preenchimento opcional
obj_ids	Identificadores das conexões não consistidas	Caso o erro ocorrido seja relacionado ao bloqueio e desbloqueio de conexões, os identificadores das conexões que não puderam ser consistidas são listados.

Tabela 5: Catálogo de erros do serviço de reconfiguração

Motivo do erro	Descrição
SYSTEM_ERROR	Erro genérico de sistema
CREATE_COMPONENT_ERROR	Erro na criação do componente
REMOVE_COMPONENT_ERROR	Erro na remoção do componente
CREATE_CONNECTION_ERROR	Erro na criação da conexão
REMOVE_CONNECTION_ERROR	Erro na remoção da conexão
TRANSFER_STATE_ERROR	Erro durante a transferência de estado
SET_ATTRIBUTE_ERROR	Erro durante a atribuição de novo valor ao atributo
BLOCK_FAILED	Solicitação de bloqueio do componente falhou
UNBLOCK_FAILED	Solicitação de desbloqueio do componente falhou
CONSISTENCY_TIMEOUT	Tempo máximo expirado para a condução do sistema ao estado de reconfiguração
RESTART_TIMEOUT	Tempo máximo expirado para a retirada do sistema do estado de reconfiguração
RECONFIGURATION_TIMEOUT	Tempo máximo esgotado para a reconfiguração
UNKNOWN_ERROR	Erro desconhecido
RECONFIG_STATUS_NOT_FOUND	Não foi encontrado reconfiguração com o identificador informado
INVALID_POLICY	Política não encontrada

Conforme mencionado, o método *reconfigure* do RM recebe como parâmetros o nome do arquivo que contém o *script* de reconfiguração, a indicação se o CM deve ser acionado para conduzir o sistema ao estado de reconfiguração, bem como o tempo máximo para que isso ocorra, além do tempo máximo para a reconfiguração como um todo. Para obter as ações de reconfiguração, o RM percorre o arquivo XML com o *script* e converte suas entradas em objetos do tipo *CreateComponentAction*, *RemoveComponentAction*, *CreateConnectionAction*, *RemoveConnectionAction*, *TransferStateAction* e *SetAttributeAction*. Tais objetos são armazenados em um vetor que, na ocasião da efetivação das mudanças estruturais solicitadas, é percorrido para acionamento da operação *start*, definida na classe *Action*, da qual as classes desses objetos são derivadas. Observa-se desse

modo que, caso a forma de acionamento do DM venha a ser alterada, a implementação dessas classes poderá ser modificada, sem que o código do RM seja afetado. A representação de tais classes e relacionamentos pode ser vista na Figura 18.

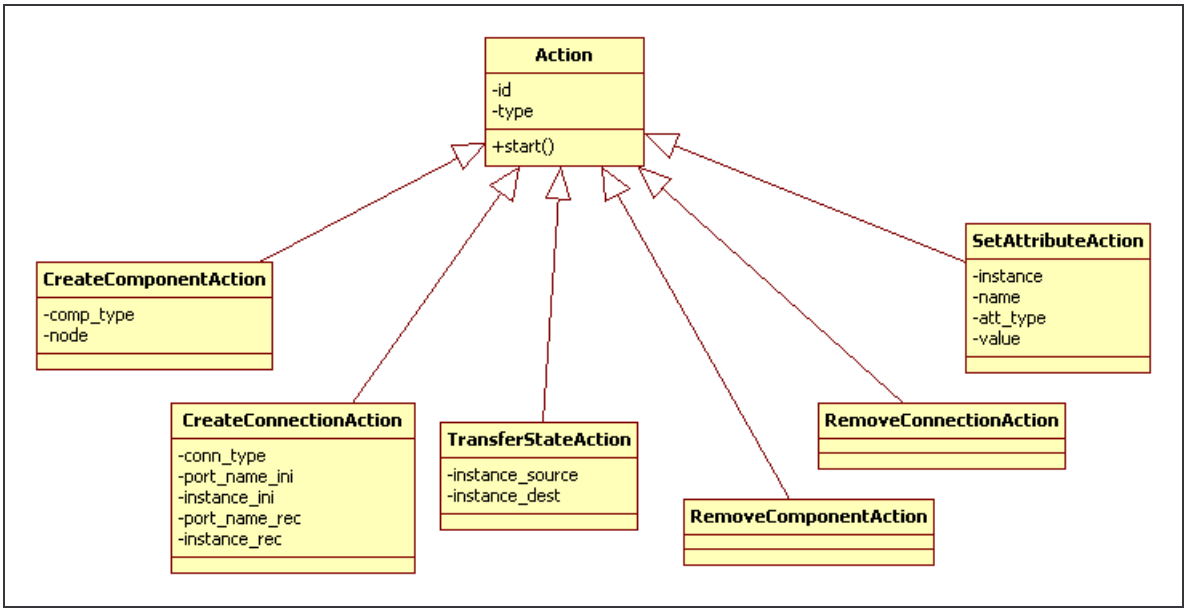


Figura 18. Classes *Action* auxiliares do *Reconfiguration Manager*

Conforme pode ser verificado no XML *Schema*, os comandos de reconfiguração podem assumir qualquer ordem, cabendo ao solicitante da reconfiguração se certificar de que a execução das ações existentes no *script* não incorrerá em inconsistência estrutural na aplicação. Por exemplo, para a criação de uma conexão, é preciso que as instâncias dos componentes informados de fato existam e que as portas dos componentes sejam compatíveis. Entretanto, em sua versão atual, o RM não efetua esse tipo de verificação.

5.3.3 *Consistency Manager*

Uma vez tendo recebido a solicitação do RM para conduzir o sistema ao estado de reconfiguração, o CM se comunica com os componentes da aplicação de forma assíncrona, solicitando o bloqueio das conexões necessárias na ordem determinada pela abordagem adotada. Assim que todas as solicitações de bloqueio tiverem sido atendidas, o CM comunica

a situação ao RM, para que o mesmo possa prosseguir com a reconfiguração. Terminada a reconfiguração, o RM solicita ao CM que comande o desbloqueio das conexões afetadas, de modo que o sistema possa funcionar em seu estado normal. Na solicitação do RM recebida pelo CM também são informadas as conexões a serem removidas e criadas, de modo que o ele possa comandar o seu bloqueio e desbloqueio na ordem apropriada.

Nas Figuras 19 a 21 são apresentadas as interfaces e eventos base especificados para o CM, seus clientes e componentes com os quais interage, e que devem ser especializados para implementar o comportamento de cada componente. Conforme exposto anteriormente, no presente trabalho optou-se pela abordagem orientada a conexões para ilustrar um possível funcionamento do CM. Para tanto, as interfaces e eventos forem especializadas de acordo, conforme apresentado nas Figuras 22 a 24. Na IDL apresentada na Figura 25, é possível observar os tipos de eventos que o CM consome e publica, bem como a faceta através da qual disponibiliza alguns de seus serviços. Conforme pode ser constatado na Figura 22, para a implementação da abordagem de consistência adotada, foi definido o método *set_dependency* que recebe e armazena a informação de dependência entre as conexões. De acordo com essa abordagem, duas portas de um determinado componente são dependentes se ao receber uma requisição em uma de suas portas, é iniciada uma transação em outra porta pertencente a ele. Ainda de acordo com a abordagem, a dependência entre conexões é derivada dessa dependência entre as portas, sendo que se uma porta receptora *r* depende de uma iniciadora *i*, então toda transação recebida por *r* depende de toda conexão que parte de *i*. Neste trabalho, o modelo de Wermelinger foi adaptado para o CCM, considerando que as portas iniciadoras são representadas pelos receptáculos e as portas receptoras pelas facetas. Além disso, a identificação do tipo de conexão é feita através do tipo de componente e da porta participante da conexão. Desse modo, ao se registrar que um tipo de conexão é dependente de outra, esses dados devem ser informados. Essa dependência é

considerada pelo CM na ocasião das solicitações de bloqueio e desbloqueio de componentes da aplicação.

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        valuetype ConsistActionBase {};
6
7        typedef sequence<ConsistActionBase> ConsistActionsBase;
8
9        interface IConsistencyManagerBase {
10       }
11
12       eventtype ConsistEventBase {
13         public ConsistActionsBase consist_actions;
14         public unsigned long timeout;
15       };
16
17       eventtype RestartEventBase {
18       };
19
20       eventtype BlockedEventBase {
21         public string connection_id;
22       };
23
24       eventtype UnblockedEventBase {
25         public string connection_id;
26       };
27     };
28  };
```

Figura 19: Definição IDL da faceta provida e dos eventos base consumidos pelo *Consistency Manager*

Para calcular a ordem em que determinada conexão deve ser bloqueada, o CM considera a cadeia de dependências registradas. Dessa forma, se existe uma conexão do tipo *C1* que dependa de uma conexão do tipo *C2* e uma conexão do tipo *C2* que dependa de uma do tipo *C3*, então a ordem de *C3* deve ser mais alta que a de *C2* e *C1*, e a ordem de *C2* deve ser maior que a de *C1*. O CM utiliza essa ordem para organizar a solicitação de bloqueio de conexões, só comandando o bloqueio de conexões de determinada ordem quando as conexões de ordem menor tiverem sido bloqueadas. Esse comportamento garante que as conexões das quais outras dependem serão bloqueadas depois das dependentes.

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        eventtype ConsistedEventBase {
6          public string reconfig_id;
7        };
8
9        eventtype RestartedEventBase {
10         public string reconfig_id;
11       };
12     };
13  };
```

Figura 20: Definição IDL dos eventos base consumidos pelos clientes do *Consistency Manager*

O tipo de evento que sinaliza ao CM que o sistema deve ser conduzido ao estado de reconfiguração e que informa quais ações de reconfiguração serão consistidas e o tempo máximo em que isso deve ser realizado é o *ConsistEventBase*. A especialização do *valuetype ConsistActionBase* ilustrada na Figura 22 estabelece quais dados são passados ao CM, nessa implementação específica. O campo *id* corresponde ao identificador de conexão, o campo *type* indica o tipo de ação, que pode ser criação ou remoção de conexão. Já os campos *instance*, *type* e *port* sucedidos de *ini* ou *rec* contêm o identificador dos componentes que participam da conexão como iniciadores ou receptores, seu tipo e a porta respectiva. O RM cria essa coleção de dados a partir das ações de reconfiguração que recebe através do *script* ou da definição das políticas. Esses dados permitem ao CM obter, por intermédio do DM, a referência do componente para o qual será solicitado o bloqueio da porta que participa da conexão.

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        Eventtype BlockEventBase {
6          public string connection_id;
7        };
8
9        Eventtype UnblockEventBase {
10         public string connection_id;
11       };
12     };
13  };
```

Figura 21: Definição IDL dos eventos base dos componentes para interação com o Consistency Manager

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        enum ConsistencyStage {BLOCK, UNBLOCK};
6
7        valuetype ConsistAction : ConsistActionBase
8        {
9          public string id;
10         public string type;
11         public string instance_ini;
12         public string type_ini;
13         public string port_ini;
14         public string instance_rec;
15         public string type_rec;
16         public string port_rec;
17       };
18
19       struct ConnDependency {
20         string comp_type_ini;
21         string port_name_ini;
22         string comp_type_rec;
23         string port_name_rec;
24
25       };
26
27       typedef sequence<ConnDependency> ConnsDependency;
28
29       interface IConsistencyManager:IConsistencyManagerBase
30       {
31         void set_dependency (in ConnsDependency conns_dep);
32       };
33
34       eventtype Consist:ConsistEventBase {
35       };
36       eventtype Restart:RestartEventBase {
37       };
38       eventtype Blocked:BlockedEventBase {
39       };
40       eventtype Unblocked:UnblockedEventBase {
41       };
42       eventtype ConsistencyError {
43         public ConsistencyStage stage;
44         public ObjectIds obj_ids;
45       };
46     };
47  };
```

Figura 22: Definição IDL da faceta provida e dos eventos concretos consumidos e publicados pelo *Consistency Manager*

Após conduzir a aplicação ao estado de reconfiguração, o CM comunica essa ocorrência ao RM através do evento *Consisted*. Terminada a reconfiguração, o RM encaminha ao CM um evento do tipo *RestartEventBase* para que o CM possa comandar o desbloqueio dos componentes adequados. Entretanto, caso não seja possível bloquear ou

desbloquear os componentes no tempo limitado, ele envia ao RM, e possivelmente a outros consumidores subscritos, um evento do tipo *ConsistencyError*, contendo o detalhamento do erro encontrado, sendo que o campo *stage* pode assumir o valor *Block* ou *Unblock* indicando se o erro ocorreu na fase de bloqueio ou de desbloqueio. No campo *obj_ids* ficam armazenadas as conexões em que o erro ocorreu. Na Figura 19 pode-se observar que no evento derivado do tipo *ConsistEventBase* enviado ao CM é possível informar o limite de tempo mencionado.

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        eventtype Consisted:ConsistedEventBase {
6        };
7        eventtype Restarted:RestartedEventBase {
8        };
9      };
10  };
```

Figura 23: Definição IDL dos eventos concretos consumidos pelos clientes do *Consistency Manager*

```
1   module ReconfigService
2   {
3      module Consistency
4      {
5        eventtype Block:BlockEventBase {
6        };
7        eventtype Unblock:UnblockEventBase {
8        };
9      };
10  };
```

Figura 24: Definição IDL dos eventos concretos dos componentes para interação com o *Consistency Manager*

5.3.4 Componentes da Aplicação

Para que o CM possa se comunicar com os componentes da aplicação é necessário que eles estejam preparados para receber suas mensagens, adotar o comportamento esperado, e responder a elas. Conforme definido na IDL apresentada nas Figuras 21 e 24, os componentes recebem mensagens do CM através das portas receptoras de evento do tipo *BlockEventBase* e *UnblockEventBase*, que podem conter o identificador da conexão a ser

bloqueada ou desbloqueada. Essa informação pode ser necessária para identificar a conexão a ser afetada, caso o componente possua mais de um receptáculo. Ao receber um evento do tipo *Block*, o componente deverá modificar seu comportamento de modo a não enviar novas requisições através da conexão. A chegada do evento *Unblock* sinaliza ao componente que novas requisições através do receptáculo estão liberadas. Após bloquear ou desbloquear a conexão solicitada, o componente da aplicação responde ao CM com um evento *Blocked* ou *Unblocked*, conforme especificação na IDL das Figuras 19 e 22.

```
    module ReconfigService
1   {
2      module Consistency
3      {
4        component ConsistencyManager {
5          publishes Consisted cm_consisted_pub;
6          publishes Restarted cm_restarted_pub;
7          publishes Block cm_block_pub;
8          publishes Unblock cm_unblock_pub;
9          publishes ConsistencyError cm_cons_error_pub;
10         consumes ConsistEventBase cm_consist_cons;
11         consumes RestartEventBase cm_restart_cons;
12         consumes BlockedEventBase cm_blocked_cons;
13         consumes UnblockedEventBase cm_unblocked_cons;
14         provides IConsistencyManager cm_fac;
15         Uses ::ReconfigService::Deployment::
                IDeploymentManagerBase cm_dm_rec;
16       };
17       home ConsistencyManagerHome manages ConsistencyManager {};
18     };
19  };
```

Figura 25: Definição IDL do componente *Consistency Manager*

O efetivo bloqueio da conexão depende do tipo de comportamento do componente. Caso ele possua sua própria *thread* de execução, será preciso evitar que sejam enviadas requisições através da porta bloqueada. Entretanto, se for um componente que só é ativado ao receber requisições de outros componentes, o bloqueio e desbloqueio dos componentes que efetuam requisições a ele podem ser suficientes.

Além de interagir com o *Consistency Manager*, os componentes da aplicação podem interagir com o *Deployment Manager* com o objetivo de efetuar a transferência de estado. Para tanto, tais componentes devem prover facetas derivadas do tipo *IStateSourceBase* e *IStateDestBase*. Vale observar que os componentes fonte e destino do estado não precisam

ser do mesmo tipo ou possuir a mesma implementação, desde que provejam as facetas determinadas, e que os métodos *get_state* e *set_state* sejam implementados levando em conta essa particularidade. Tal característica permite, por exemplo, que o estado do componente seja armazenado em um repositório e que esse repositório possa ser utilizado para recuperar o estado do componente em caso de falha do mesmo.

Outro aspecto relevante é o fato de que o estado dos componentes precisa ser armazenado em variáveis do tipo CORBA *Dynamic Any*. Tais tipos de dados são criados dinamicamente através de fábricas e podem ser enviados como parâmetro sem que o componente precise ter conhecimento prévio de sua estrutura. Uma vez que o estado dos componentes é dependente de sua implementação, através da utilização do *Dynamic Any* o DM pode comandar sua transferência sem restrição do tipo dos dados transferidos.

5.3.4.1 Implementando um Componente para Uso com o Serviço de Reconfiguração

Conforme mencionado, a comunicação entre o CM e os componentes da aplicação é realizada de modo assíncrono através das suas portas emissoras e receptoras de evento. Dessa forma, aqueles componentes passíveis de participar da reconfiguração por funcionarem como origem ou destino de conexões que podem ser removidas em tempo de execução, ou que precisam ter seu estado preservado durante a reconfiguração, devem possuir as portas para publicação e recepção dos eventos adequados, bem como para a transmissão de estados, conforme ilustrado pelas declarações IDL do componente *Sensor* exibidas nas Figuras 26 e 27. Nas linhas 6 a 9 da Figura 27 pode-se observar a declaração das portas de comunicação com o CM.

```
1   module Control
2   {
3     interface IControllerFacet {
4       void control (in float control_data);
5     };
6
7     interface IActuatorFacet {
8       void actuate (in float actuation_data);
9     };
10
11    interface IPlantActuatorFacet {
12      void update (in float actuation_data);
13    };
14
15    interface IPlantSensorFacet {
16      float get_value ();
17    };
18
19    eventtype TimeOut {};
20  };
```

Figura 26: Definição IDL dos componentes da aplicação de controle

```
1   module Control
2   {
3     component Sensor
4       {
5         consumes TimeOut timeout;
6         consumes ReconfigService::Consistency::UnblockEventBase
            comp_unblock_cons;
7         consumes ReconfigService::Consistency::BlockEventBase
            comp_block_cons;
8         publishes ReconfigService::Consistency::Blocked
            comp_blocked_pub;
9         publishes ReconfigService::Consistency::Unblocked
            comp_unblocked_pub;
10        uses IControllerFacet controller;
11        uses IPlantSensorFacet plant;
12      };
13
14    home SensorHome manages Sensor
15      {
16      };
17  };
```

Figura 27: Definição IDL do componente *Sensor*

Caso seja necessário transferir o estado entre versões novas e antigas do componente, é preciso também que ele proveja as facetas para transferência de estado. O componente do qual o estado será obtido deverá prover uma faceta do tipo *IStateSourceBase*, conforme ilustrado nas linhas 15 e 22 da Figura 28, e o componente destinatário deverá prover uma faceta derivada de IStateSourceBase. O DM se conecta a essas facetas na ocasião em que é acionado pelo DM para realizar a transferência de estado entre os componentes. Nas

linhas 3 a 13 da Figura 28 pode ser vista a declaração da estrutura que corresponde ao estado a ser transferido pelo componente *Controller*. Conforme mencionado anteriormente, essa estrutura será encapsulada em um dado do tipo *Any*, para que possa ser transferida entre os componentes.

```
1   module Control
2   {
3     struct StateDataController {
4       float kp;
5       float ki;
6       float kd;
7       unsigned short sampling_rate;
8       float setpoint;
9       float control_old;
10      float error_sum;
11      float old_error;
12      float control_new;
13    };
14
15    interface IStateSource:
    ::ReconfigService::Deployment::IStateSourceBase {
16    };
17
18    component Controller
19      {
20        provides IControllerFacet controller_facet;
21        uses     IActuatorFacet actuator;
22        provides IStateSource comp_state_source_fac;
23        attribute float kp;
24        attribute float ki;
25        attribute float kd;
26        attribute unsigned short sampling_rate;
27        attribute float setpoint;
28      };
29
30    home ControllerHome manages Controller
31      {
32      };
33  };
```

Figura 28: Definição IDL do componente *Controller*

Capítulo 6

Resultados Empíricos

Com a finalidade de avaliar o impacto no desempenho de uma aplicação que utilize o serviço de reconfiguração dinâmica apresentado neste trabalho foram implementadas duas variações de uma aplicação de controle de velocidade veicular, sendo que uma delas utilizando o ARCOS e a outra não. Os resultados obtidos na execução dessas variações de aplicação serão apresentados nesta seção.

6.1 Controlador PID para Controle da Velocidade Veicular

O comportamento físico da simulação do veículo é caracterizado pelas equações (PONT, 2001):

$$Accel = (Throttle * ENGINE\_POWER - (FRIC * Old\_speed)) / MASS \quad (1)$$

$$Dist = Old\_speed + Accel \times (1/SAMPLE\_RATE) \quad (2)$$

$$Speed = SQRT ((Old\_speed)^2 + 2 \times Accel \times Dist) \quad (3)$$

O veículo apresenta propriedades de potência do motor, coeficiente de fricção e massa que são representadas pelas constantes ENGINE_POWER, FRIC e MASS. Informados a velocidade corrente (*Old_speed*) e o valor da atuação no acelerador (*Throttle*), se obtém a velocidade do veículo (*Speed*). A taxa de amostragem utilizada pelo controlador é representada por SAMPLE_RATE.

Tal modelo serviu de base para a implementação de um controlador PID (Proporcional-Integrativo-Derivativo) (OGATA, 2001) que atua diretamente no acelerador com base no erro mensurado entre a velocidade atual e a velocidade desejada (*setpoint*), conforme ilustrado no esquema da Figura 29. Os controladores PID, ou uma variação deles, são utilizados em mais da metade dos controladores industriais e sua atuação é uma combinação das ações proporcional, integral e derivativa. A calibração do valor calculado na saída do controlador depende da ponderação de três parâmetros essenciais: ganho proporcional (*kp*), ganho integral (*ki*) e ganho derivativo (*kd*). O termo proporcional enfatiza a reação do controlador aos erros momentâneos. O termo integral considera a reação segundo o somatório dos erros (histórico). Por fim, o termo derivativo determina a reação em função da taxa de modificação do erro mensurado (derivada). A sintonia desses parâmetros visa atender aos requisitos da aplicação tais como tempo de subida, tempo de estabilização e *overshoot* (OGATA, 2001).

Em ambas as variações da aplicação, uma delas utilizando o ARCOS e a outra sem empregá-lo, os valores das constantes e dos parâmetros adotados foram os mesmos utilizados em (ANDRADE, 2006), com exceção do valor de SAMPLE_RATE. Tais valores constam na Tabela 6. Os experimentos foram conduzidos em um computador com processador Core 2 Duo, 2.83 GHZ, 2 GB de memória RAM e sistema operacional Debian Linux kernel 2.6. O detalhamento das aplicações mencionadas e os resultados obtidos serão abordados nas subseções seguintes.

6.1.1 Métricas de desempenho

Para a aplicação veicular, o aspecto principal na avaliação do controlador consiste em verificar sua capacidade em manter a velocidade próxima ao valor desejado. Para este fim, algumas métricas de avaliação foram consideradas (OGATA, 2001), dentre elas: tempo de

subida, tempo de estabilização e *overshoot*. O tempo de subida (*raise time*) é o tempo necessário para o valor atual (resposta) alcançar uma faixa percentual do valor desejado. Três faixas de valores são normalmente utilizadas: 10% a 90%, 5% a 95% e 0% a 100%. O tempo de estabilização (*settling time*) é o tempo necessário para a curva de resposta alcançar e permanecer dentro de um intervalo em torno do valor final. Este intervalo é especificado por uma porcentagem absoluta do valor final (geralmente entre 2% a 5%). O percentual máximo de *overshoot* corresponde ao valor máximo obtido pela curva de resposta. O tempo de subida e o *overshoot* são adequados nos casos em que a variável medida precisa atingir um valor maior que o seu valor inicial. Nos casos em que o *setpoint* é modificado para um valor inferior ao valor corrente da variável medida, se torna mais adequado considerar os valores do tempo de descida e *undershoot*. Tal situação pode ocorrer, por exemplo, quando a velocidade do veículo encontra-se estabilizada em 80 km/h e se altera a velocidade alvo para 60 km/h.

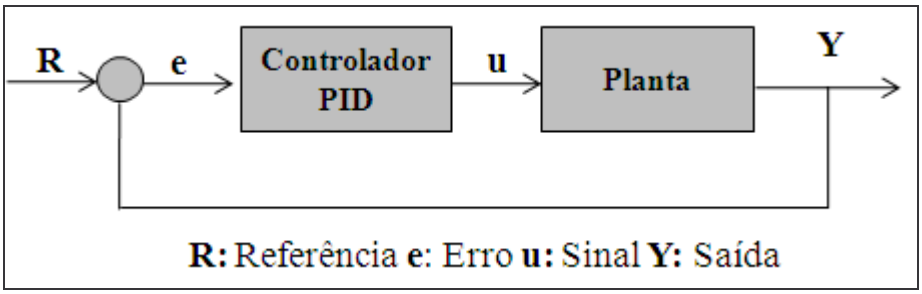


Figura 29. Malha de controle PID

Tabela 6: Valores adotados para as constantes do modelo do veículo

Constante	Valor
ENGINE_POWER	5000
FRIC	50
MASS	1000
SAMPLE_RATE	100 ms (10 Hz)
Kp	0.05
Ki	0
KD	0

Para avaliar o impacto da reconfiguração no desempenho da aplicação durante a reconfiguração foram definidos quatro cenários: reconfiguração simples antes e depois da

estabilização da velocidade; reconfiguração com modificação do *setpoint* e reconfiguração com simulação de perturbação, tendo sido a reconfiguração acionada após a estabilização da velocidade. Na variante com perturbação, foram feitas duas execuções, uma delas considerando um *setpoint* de 80 km/k, e de 200 km/h para a outra. O comportamento da aplicação com a velocidade de 200 km/h foi estudado no intuito de investigar se a reconfiguração causaria impacto mais significativo quando o veículo trafegasse em alta velocidade.

Para cada cenário, foram obtidas as curvas de resposta da velocidade do veículo, com e sem o procedimento de reconfiguração, de modo a avaliar o impacto da reconfiguração no desempenho da aplicação de controle. Para calcular o tempo de estabilização foi adotado o valor de 2% do valor desejado. Para cálculo do tempo de subida ou de descida considerou-se uma variação percentual de 5% do valor do *setpoint*. Estes cenários de avaliação são descritos nas subseções seguintes.

6.2 Estudo de Caso I – Reconfiguração dinâmica de aplicação de controle de velocidade veicular sem a utilização do ARCOS

A implementação dos componentes dessa aplicação está representada na Figura 30. O *Gerador de Pulso* produz pulsos, como referências temporais, para o *Sensor* de velocidade na taxa de amostragem especificada pela aplicação de controle. A cada pulso recebido, o *Sensor* obtém o valor da velocidade atual e o repassa para o *Controlador1*. Este, por sua vez, calcula o novo valor de atuação no acelerador de acordo com o erro entre o valor medido e o valor desejado para a velocidade. A falha do *Controlador1* é tolerada através de uma adaptação da replicação semi-ativa, sendo que o estado da réplica (*Controlador2*) é atualizado a cada vez que um novo valor de atuação é calculado. O *Controlador1* repassa os valores de atuação ao *Atuador*, que atualiza os parâmetros correspondentes no objeto

controlado (veículo), representado pelo componente *Planta*. Uma vez que existe uma conexão entre o *Controlador2* e o *Controlador1* para que seja feita a transferência de estado, é através dela que o *Controlador2* detecta a falha do primeiro. Isso é feito contabilizando o tempo entre duas operações de atualização de estado. Caso esse tempo ultrapasse o valor especificado por um valor de *timeout* configurável no *Controlador2*, ele envia um evento ao *Reconfiguration Manager* notificando a ocorrência da falha. Este último, por sua vez, aciona a reconfiguração do sistema, de acordo com a política especificada previamente. Assume-se que na ocorrência de falhas o *Controlador1* possui um comportamento do tipo *fail-stop*.

As equações (1), (2) e (3) são utilizadas pelo *Sensor* para calcular a velocidade atualizada da *Planta*. Desse modo, ao receber o pulso do *Gerador de Pulso*, o *Sensor* se comunica com a *Planta* obtendo o valor atual da variável *throttle*. Este valor é substituído na equação (1) para calcular a aceleração do veículo, que é dependente da atuação no acelerador e da velocidade anterior. Conhecendo-se a aceleração, a distância que o veículo percorreu durante o tempo transcorrido entre uma medição e outra é obtida através da equação (2). Finalmente, a equação (3) é empregada para o cálculo da velocidade atual do veículo (*Planta*). Este valor é o que é enviado ao *Controlador1* para ser comparado com o *setpoint*, para correção da atuação, caso necessário.

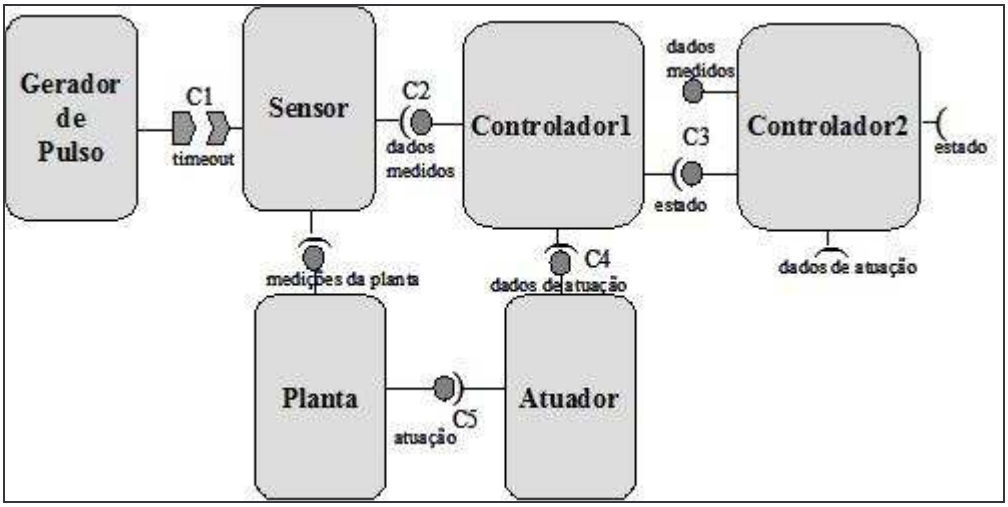


Figura 30. Componentes da aplicação de controle de velocidade veicular

Os valores das velocidades observadas do veículo no decorrer do tempo, para os cenários de teste, foram registrados e utilizados na elaboração dos gráficos de desempenho apresentados a seguir, sendo que o valor da velocidade observado pelo componente *Sensor*, calculado através da utilização das equações (1), (2) e (3), era obtido e registrado a cada ciclo de controle.

Nos cenários de reconfiguração concebidos, é realizada a substituição de um controlador falho através da reorganização das conexões das quais ele participa. A política de reconfiguração é escrita numa linguagem específica que fornece abstrações de alto nível para o desenvolvedor da política de reconfiguração e permite a geração automática do código XML do *script* de reconfiguração (SANTOS; MACÊDO; BARRETO, 2008). A política de reconfiguração de substituição de um controlador falho é apresentada na Figura 31.

```
1 (Controlador1.failed) => {
2    destroy connection C2, C3, C4
3    create connection C6 : Sensor.r1 -> Controlador2.f1
4    create connection C7 : Controlador2.r1 ->  Atuador.f1
5  }
```

Figura 31. Política de reconfiguração para substituição de controlador falho

A notificação da falha do componente *Controlador1* ao *Reconfiguration Manager*, expressa através do predicado `Controlador1.failed`, aciona a remoção das conexões originais e a conexão do novo controlador aos componentes existentes. Em primeiro lugar, o *script* remove as conexões originais (*C2, C3* e *C4*) através do comando *destroy* (linha 2), que atua sobre uma lista de conexões existentes. Nas linhas 03 e 04 são estabelecidas duas conexões (*C6* e *C7*) entre o novo controlador e os componentes pertinentes (*Sensor* e *Atuador*). Os descritores *f1* e *r1* representam as facetas e receptáculos dos componentes envolvidos no procedimento de conexão. Vale observar que, uma vez que na política especificada o *Consistency Manager* é utilizado, ele providencia o bloqueio da conexão entre o *Sensor* e *Controlador* antes que as conexões sejam reorganizadas.A título de exemplo, a

Figura 32 representa um trecho do XML gerado para o comando de criação da conexão *C6* entre o componente *Controlador2* e o *Sensor* (linha 3) da Figura 31.

```
1    <command>
2      <createConnection>
3        <id>C6</id>
4        <type>SIMPLEXRECEPTACLE_FACET</type>
5          <initiator>
6            <portName>r1</portName>
7            <instance>Sensor</instance>
8          </initiator>
9          <receptor>
10           <portName>f1</portName>
11           <instance>Controlador2</instance>
12         </receptor>
13     </createConnection>
14     </command>
```

Figura 32. Trecho de arquivo XML referente à conexão do Controlador2 ao Sensor

A utilização de uma linguagem específica facilita a verificação automática de propriedades importantes no contexto do domínio de reconfiguração. Por exemplo, no cenário de falha exposto, espera-se que o comando *destroy* atue somente nas conexões nas quais haja participação do componente falho (*C2*, *C3* e *C4*, no caso), pois não faz sentido, além de ser extremamente indesejável, afetar as conexões de outros componentes nesse contexto. Tal verificação é facilmente realizável através de análise simples dos comandos do *script* de reconfiguração.

6.2.1 Resultados da Simulação

6.2.1.1 Cenário 1: Reconfiguração simples antes da estabilização da velocidade

Este cenário consiste no acionamento da reconfiguração antes da estabilização do valor da velocidade do veículo em 80 km/h. A Figura 33 mostra as curvas de evolução da velocidade do veículo na execução de reconfiguração após 1s do início da execução, portanto, antes de ter sido atingida a estabilização da velocidade. A proximidade das curvas (a diferença é praticamente imperceptível no gráfico) de evolução da velocidade nas situações com a reconfiguração e sem a reconfiguração atesta a ausência de impacto significativo na aplicação.

Os tempos de subida e de estabilização e o percentual de *overshoot* medidos foram aproximadamente os mesmos para a simulação com e sem reconfiguração, sendo que o tempo de subida foi de 3 s, o tempo de estabilização de 3,8 s e o percentual de *overshoot* 2,3%.

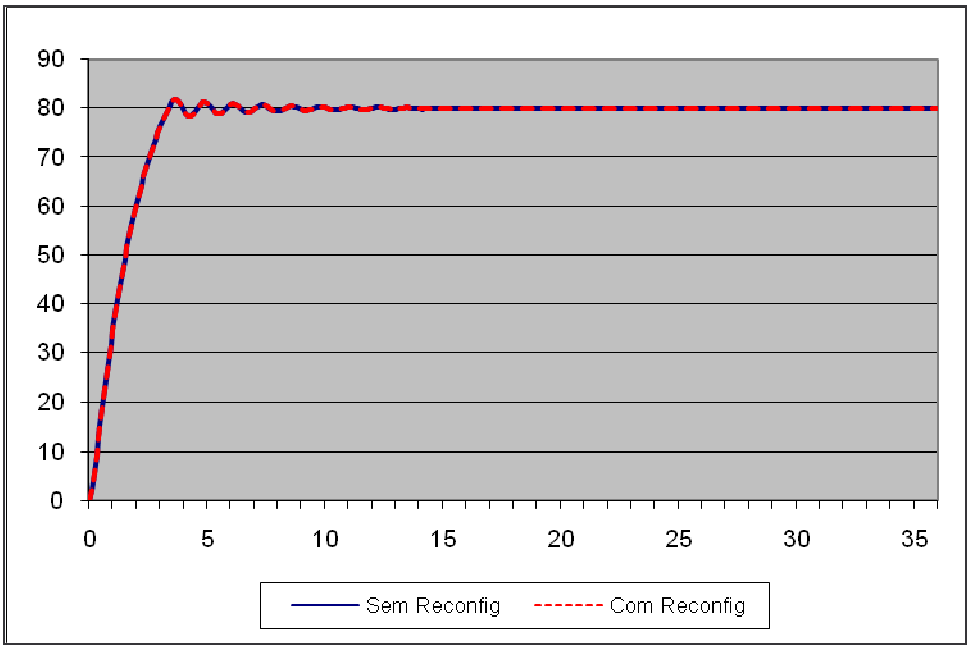


Figura 33. Evolução da velocidade no tempo com reconfiguração antes da estabilização

6.2.1.2 Cenário 2: Reconfiguração simples após a estabilização da velocidade

Neste cenário foi acionada a reconfiguração após a estabilização do valor da velocidade do veículo em 80 km/h. A Figura 34 mostra as curvas de evolução da velocidade do veículo na execução de reconfiguração aos 20s do seu início, aproximadamente, portanto, após atingida a estabilidade da velocidade. A proximidade das curvas de evolução da velocidade nas situações com a reconfiguração e sem a reconfiguração atesta que, também neste cenário, não houve impacto significativo na aplicação.

Foram observados os mesmos valores para as propriedades do sistema controlado, no que diz respeito ao alcance da velocidade de 80 km/h, com e sem a reconfiguração: tempo de subida ≈ 3 s, tempo de estabilização ≈ 3,8 s e *overshoot* ≈ 2,3 %. Além disso, o tempo

medido em que a conexão entre o *Sensor* e *Controlador* permanece bloqueada foi de aproximadamente 93 ms.

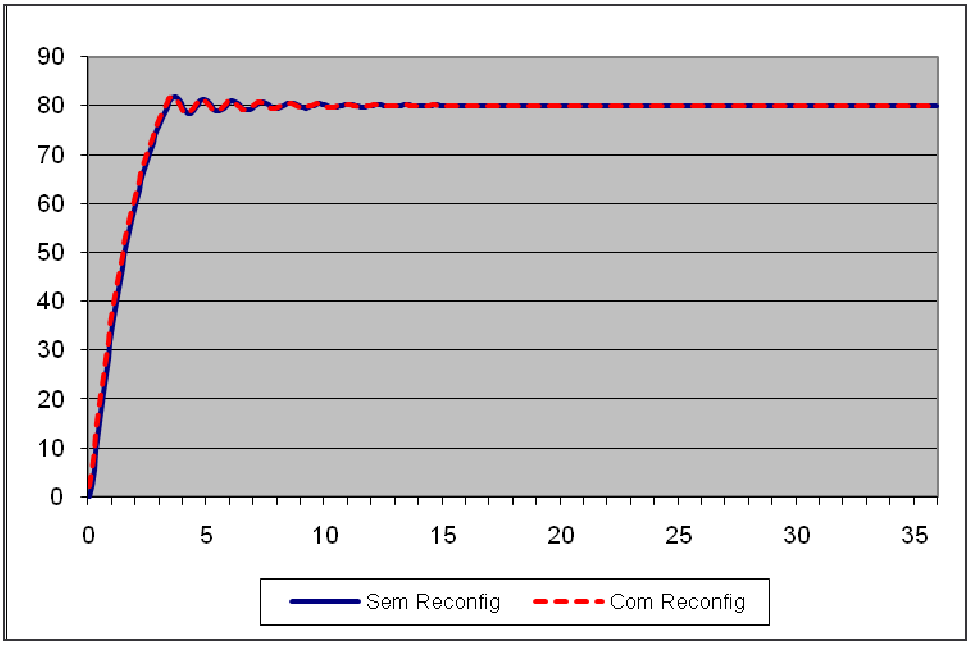


Figura 34. Evolução da velocidade no tempo com reconfiguração após estabilização

6.2.1.3 Cenário 3: Reconfiguração com modificação de *setpoint*

Este cenário consiste na aplicação de reconfiguração após a estabilização do valor da velocidade do veículo em 80 km/h e posterior mudança de *setpoint* para 60 km/h, seguida de reconfiguração aos 20 s. Assim como no cenário anterior, a Figura 35 ilustra diferença desprezível entre os tempos na ausência e presença da reconfiguração.

Uma vez que os parâmetros utilizados para o *setpoint* de 80 km/h são os mesmos do cenário anterior, os valores de tempo de subida, tempo de estabilização e *overshoot* são os mesmos. Dessa forma, neste cenário, importa analisar o tempo de estabilização, tempo de descida e o percentual de *undershoot* obtidos após a mudança do *setpoint* de 80 km/h para 60 km/h. Nessa condição, foram obtidos, tanto para a execução com e quanto para a sem reconfiguração, os valores de 1,8 s para o tempo de estabilização, 0,5 para o tempo de descida e um *undershoot* de 9,2%.

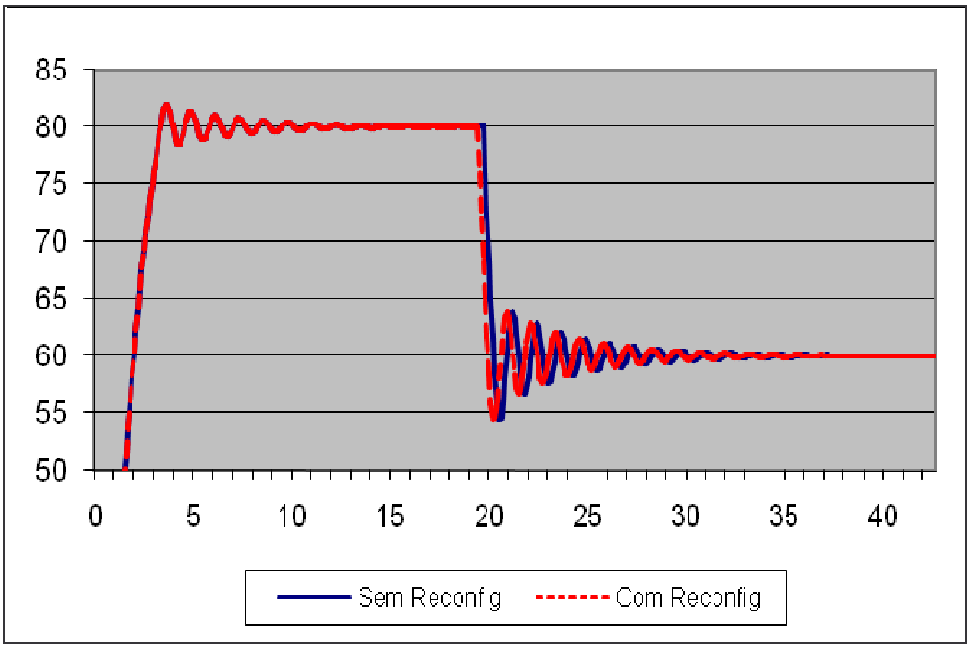


Figura 35. Evolução da velocidade no tempo em cenário com mudança de *setpoint*

6.2.1.4 Cenário 4: Reconfiguração com perturbação simulada

Este cenário consiste na simulação de uma perturbação após a estabilização do valor da velocidade do veículo em duas situações: com *setpoint* de 80 km/h e de 200 km/h. Em ambos os casos, o valor da variável *throttle* foi alterado para 2,5, representando, por exemplo, a incidência de uma rajada de vento na traseira do veículo, ocasionando aumento repentino da sua velocidade. A aplicação foi executada com ausência e ocorrência de reconfiguração, sendo que, para o *setpoint* de 80 km/h, a reconfiguração foi acionada depois de aproximadamente 20 s do início da sua execução. Para o *setpoint* de 200 km/h, a reconfiguração foi iniciada após 30 s.

Para o *setpoint* de 80 km/h, os índices medidos foram relativos ao comportamento da aplicação após a perturbação. Para a modalidade sem reconfiguração o tempo de estabilização foi de 21,2 s, o tempo de descida de 21,4 s e o *undershoot* de 7,18%. Com reconfiguração, os índices obtidos foram: tempo de estabilização de 21,1 s, tempo de descida de 21,3 s e *undershoot* de 7,11%. Observou-se ainda que a perturbação fez com que a

velocidade chegasse a 88 km/h. Tal comportamento pode ser constatado no gráfico da Figura 36.

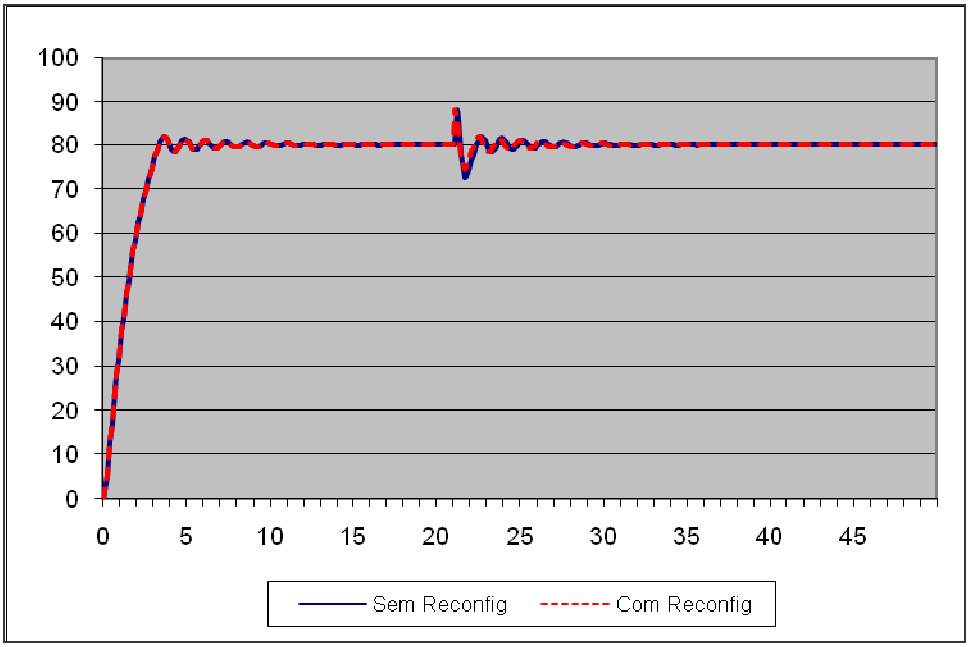


Figura 36. Evolução da velocidade no tempo em cenário com perturbação e *setpoint* em 80 km/h

Conforme ilustrado na Figura 37, com o *setpoint* em 200 km/h, o tempo de subida e tempo de estabilização foram os mesmos para as variações com e sem reconfiguração, ficando o valor do primeiro em 6,1 s e o do segundo em 7,9 s.

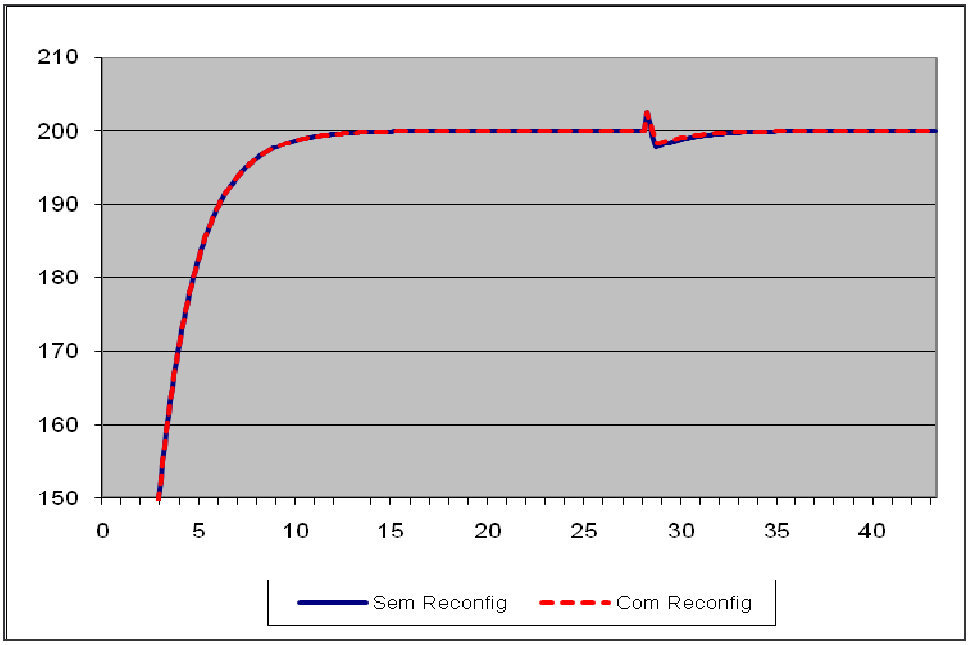


Figura 37. Evolução da velocidade no tempo em cenário com perturbação e *setpoint* em 200 km/h

Em ambos os casos, com e sem reconfiguração, antes da perturbação a velocidade do veículo não ultrapassou os 200 km/h, não tendo sido, portanto, observado *overshooting*.

Após a perturbação, a velocidade do veículo alcançou 202,5 km/h. Para a execução sem reconfiguração o menor valor da velocidade foi de 197,8 km/h. Com reconfiguração, esse mínimo foi de 198,3. De acordo com essas medidas, o valor da velocidade permaneceu dentro do limite de 2% considerado para cálculo do tempo de estabilização.

6.3 Estudo de Caso II – Reconfiguração dinâmica de aplicação de controle de velocidade veicular com a utilização do ARCOS

Para avaliar o desempenho da reconfiguração no ARCOS os componentes da aplicação de controle da velocidade veicular foram adaptados de uma implementação do controle PID para controle veicular disponibilizada como exemplo no ARCOS.

Conforme mencionado no capítulo 4, a arquitetura do ARCOS foi concebida de forma a permitir que através da especialização das interfaces *IDAISProviderBaseFacet* do componente *DAISProvider* e *IControllerBaseFacet* do componente *Controller* seja possível implementar, respectivamente, o tratador do dispositivo de aquisição de dados e o controlador desejado. Dessa forma, a utilização do serviço de reconfiguração possibilita que tais componentes sejam substituídos em tempo de execução, sendo que, neste trabalho, optou-se por substituir o componente *Controller*. Conforme ilustrado na Figura 38, a faceta do *Controller* se conecta ao receptáculo do *Control Manager*.

O modelo matemático apresentado nas equações (1), (2) e (3) também é utilizado nesse cenário pelo *DAISSimulatedCarProvider* para simular o comportamento do veículo. A árvore DAIS criada para utilização com o *DAISSimulatedCarProvider* é apresentada na

Figura 39 e possui duas folhas DAIS: uma delas representando o sensor de velocidade e a outra o acelerador.

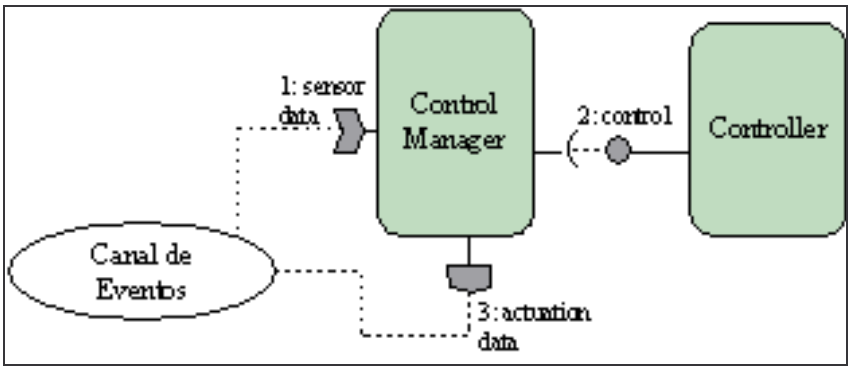


Figura 38. Estrutura de controle do ARCOS

Diferentemente da aplicação anterior, o *script* de reconfiguração efetua a substituição de um controlador antigo por um novo. Tal substituição poderia ter sido motivada por uma atualização de *hardware* ou de *software* do componente. Sendo assim, a reconfiguração é acionada invocando diretamente a operação *reconfigure* do RM. Uma vez acionada a reconfiguração, a conexão entre o componente *ControlManager* e o *PIDController* é bloqueada e o *PIDController* substituído por um novo, conforme especificado no *script* de reconfiguração reproduzido na Figura 40 . Vale observar que, conforme especificado entre as linhas 31 e 36 do *script*, é comandada a transferência de estado entre os controladores, antes que o componente antigo seja destruído (linhas 37 a 41).

Os dados para a geração dos gráficos foram coletados através do registro dos valores da velocidade do veículo calculados pelo componenente *DAISSimulatedCarProvider*. As métricas utilizadas para avaliar o impacto da reconfiguração sobre a aplicação são as mesmas da subseção 6.1.1, sendo que e os valores obtidos são detalhados a seguir.

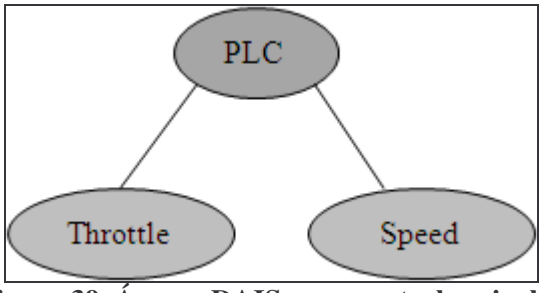


Figura 39. Árvore DAIS para controle veicular

```
1   <?xml version="1.0"?>
2   <script xmlns="http://www.ufba.br"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://www.ufba.br script.xsd">
4
5   <command>
6     <createComponent>
7       <id>ARCOS-NewPIDController-idd</id>
8       <node>MainNode</node>
9       <type>ARCOS-PIDController-mdd</type>
10    </createComponent>
11  </command>
12  <command>
13    <removeConnection>
14      <id>ControlManager_PIDController</id>
15    </removeConnection>
16  </command>
17  <command>
18    <createConnection>
19      <id>ControlManager_NewPIDController</id>
20      <type>SIMPLEXRECEPTACLE_FACET</type>
21        <initiator>
22          <portName>controller</portName>
23          <instance>ARCOS-ControlManager-idd</instance>
24        </initiator>
25        <receptor>
26          <portName>controller</portName>
27          <instance>ARCOS-NewPIDController-idd</instance>
28        </receptor>
29    </createConnection>
30  </command>
31  <command>
32    <transferState>
33      <instanceSource>ARCOS-PIDController-idd</instanceSource>
34      <instanceDest>ARCOS-NewPIDController-idd</instanceDest>
35    </transferState>
36  </command>
37  <command>
38    <removeComponent>
39      <id>ARCOS-PIDController-idd</id>
40    </removeComponent>
41  </command>
42  </script>
```

Figura 40. Arquivo XML com script de reconfiguração utilizado na simulação com o ARCOS

6.3.1 Resultados da Simulação

Os cenários utilizados com ARCOS foram os mesmos utilizados na aplicação anterior, sendo que os gráficos das Figuras 41 a 45 ilustram os resultados obtidos.

6.3.1.1 Cenário 1: Reconfiguração simples antes da estabilização da velocidade

No cenário em que a reconfiguração foi acionada antes da estabilização da velocidade, os valores obtidos para os tempos de subida e estabilização foram os mesmos tanto para a execução com, quanto para a sem reconfiguração, sendo que o primeiro foi de 3s, e o segundo de 3,2 s. O percentual de *overshoot* foi de 0,66% com reconfiguração e de 0,83% sem reconfiguração. O gráfico que ilustra esse comportamento consta da Figura 41.

6.3.1.2 Cenário 2: Reconfiguração simples após a estabilização da velocidade

No segundo cenário (Figura 42), os valores dos índices medidos para o tempo de estabilização e o tempo de subida foram os mesmos e ficaram em 3,2 s e 3 s, respectivamente. O *overshoot* apresentado foi ligeiramente diferente entre as execuções com e sem reconfiguração, tendo sido de 0,83% na primeira e 0,85% na segunda. Adicionalmente, observou-se que o tempo em que a conexão entre o *Control Manager* e o *Controller* permaneceu bloqueada foi de 100 ms.

6.3.1.3 Cenário 3: Reconfiguração com modificação de *setpoint*

Neste cenário, os tempos obtidos após a mudança do *setpoint* de 80 km/h para 60 km/h foram os mesmos tanto para a simulação com reconfiguração quanto para a sem reconfiguração: o tempo de estabilização foi de 0,9 s, o tempo de descida foi de 0,5 s e o *undershoot* por volta de 2,65%. Na Figura 43 o gráfico correspondente a esse cenário é exibido.

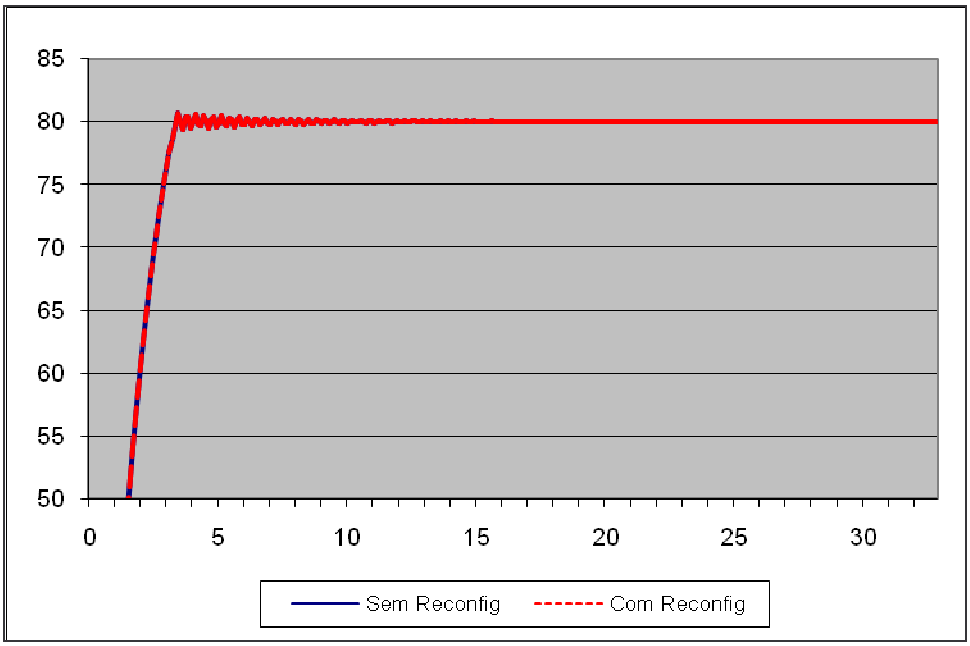


Figura 41. Evolução da velocidade no tempo com reconfiguração antes da estabilização

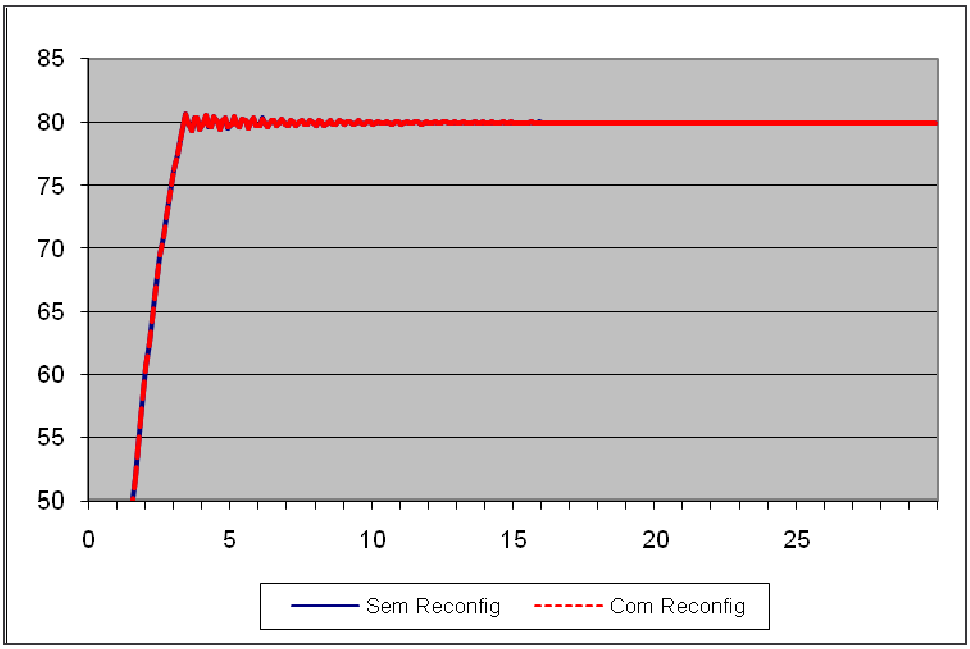


Figura 42. Evolução da velocidade no tempo com reconfiguração após estabilização

6.3.1.4 Cenário 4: Reconfiguração com perturbação simulada

Para o *setpoint* de 80 km/h, após a simulação de perturbação, os valores medidos para o tempo de estabilização e *undershoot* foram ligeiramente diferentes para a execução com e sem reconfiguração: 0,6 s e 5,85% para modalidade com reconfiguração, e 0,5 s e

3,86% para a execução sem reconfiguração. O tempo de descida foi o mesmo e teve o valor de 0,1 s.

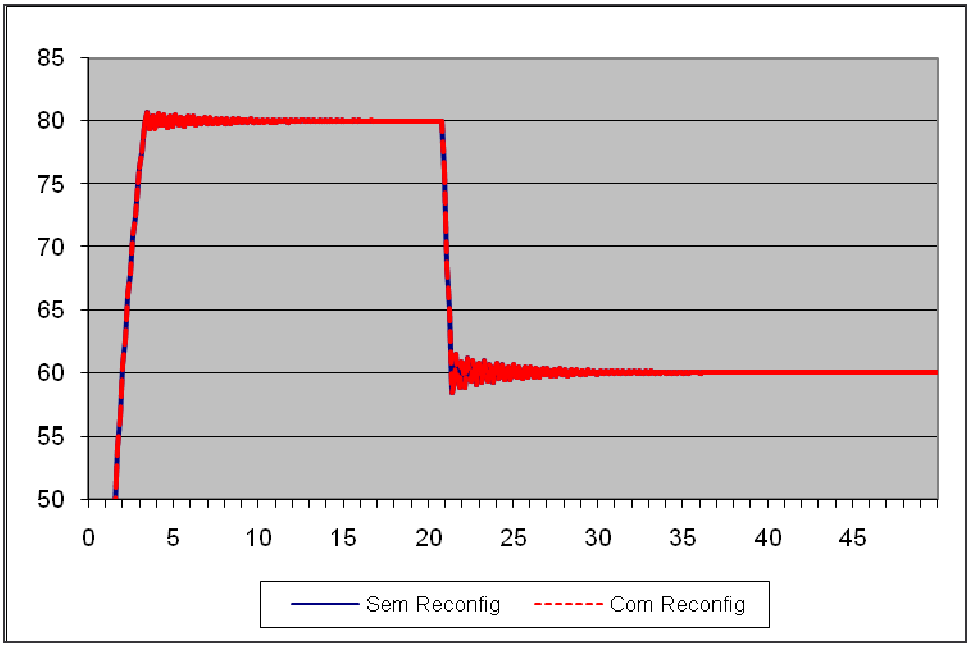


Figura 43. Evolução da velocidade no tempo em cenário com mudança de *setpoint*

Adotando-se um *setpoint* de 200 km/h, antes da perturbação, os valores dos tempos de subida e de estabilização foram de 6,1 s e 7,9 s, respectivamente. Similarmente ao medido na execução da aplicação sem o ARCOS, o valor da velocidade não ultrapassou o *setpoint*. Depois da mudança proposital do valor de *throttle*, o maior valor obtido para a velocidade foi de 202,5 km/h, e o menor valor foi de 196,1 km/h, para ambas as versões com e sem reconfiguração. Nas Figuras 44 e 45 constam os gráficos dos valores obtidos para os *setpoints* de 80 km/h e 200 km/h, respectivamente.

6.4 Discussão

Com base nos experimentos apresentados, observa-se que as ações de reconfiguração não tiveram impacto significativo no desempenho do controle do veículo. No entanto, a reconfiguração influi no comportamento do sistema, uma vez que tem como conseqüência o bloqueio de conexões entre os componentes. Por esse motivo, durante o

tempo de reconfiguração, o controlador não é atualizado com a velocidade corrente do veículo, fazendo com que o valor da atuação permaneça o mesmo nesse período.

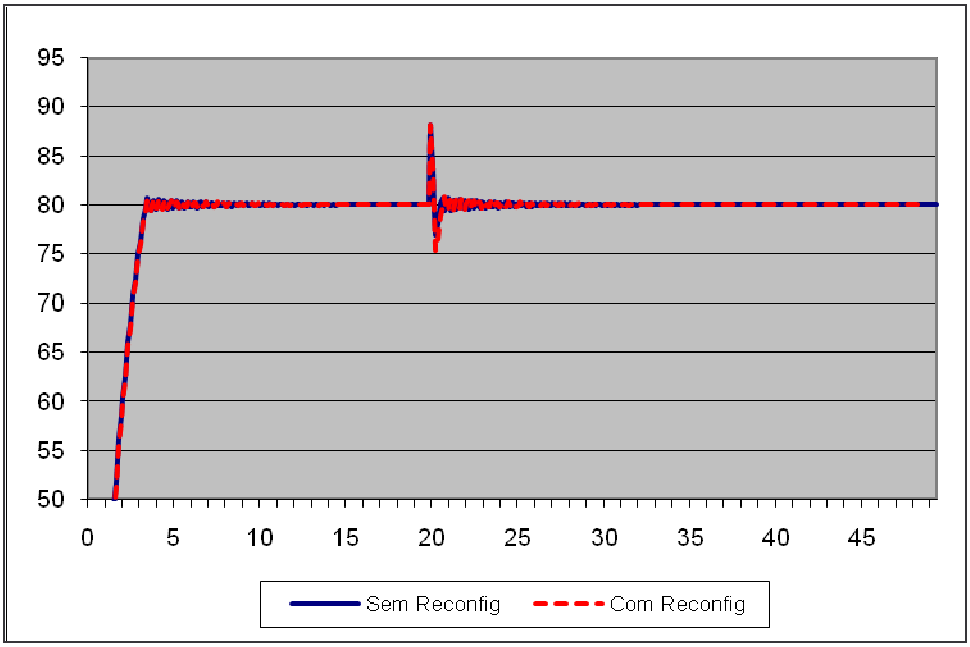


Figura 44. Evolução da velocidade no tempo em cenário com perturbação e *setpoint* em 80 km/h

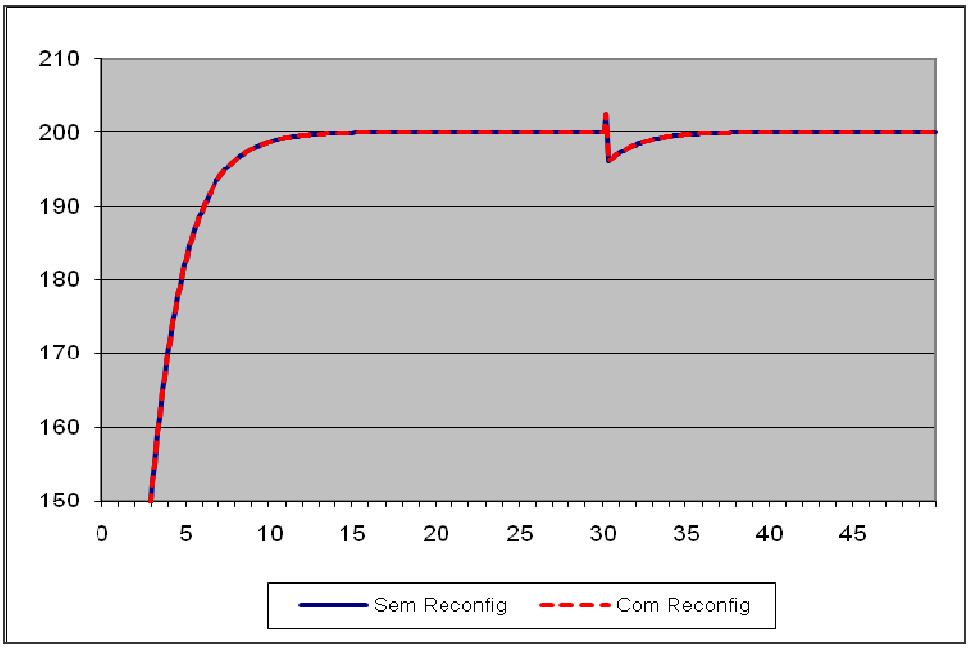


Figura 45. Evolução da velocidade no tempo em cenário com perturbação e *setpoint* em 200 km/h

No primeiro cenário, apesar de o acionamento da reconfiguração ter sido efetuado antes da estabilização do sistema, não houve comprometimento significativo no seu desempenho. Apenas os valores de *overshoot* no ARCOS foram ligeiramente distintos. No

cenário 2, uma vez que a reconfiguração se dá em um momento de estabilidade do sistema, sem influência de eventos externos, era esperado que ela tivesse pouco impacto no comportamento do mesmo.

Nos cenários 3 e 4, a reconfiguração foi acionada imediatamente após a alteração do *setpoint* e da ocorrência de perturbação, o que poderia fazer com que a resposta do sistema a esses eventos fosse prejudicada pela atraso causado pela reconfiguração. Entretanto, conforme observado nos gráficos e nas métricas do sistema, não houve influência significativa do tempo de reconfiguração no desempenho do mesmo. Tal comportamento pode ser atribuído ao fato de o intervalo de tempo em que as conexões do sistema permanecem bloqueadas ser em torno de 100 ms. Além disso, a taxa de amostragem foi de 100 ms, o que possibilita ao controlador corrigir a velocidade do veículo, em um intervalo de tempo relativamente curto. Vale observar também, que a correção efetuada pelo controlador PID utilizado é proporcional ao erro, o que agiliza a resposta do sistema à mudanças e perturbações. Espera-se que em aplicações em que seja necessário efetuar reconfigurações mais demoradas (por exemplo, com mais operações de re-conexão), a resposta do sistema a uma alteração de *setpoint* ou distúrbio externo seja mais prejudicada do que o observado nos experimentos apresentados. Entretanto, a ausência de impacto significativo no desempenho medido permite constatar a viabilidade da reconfiguração para esse tipo de aplicação, principalmente quando o planejamento e execução das mudanças são realizados de forma a considerar os requisitos funcionais e temporais da aplicação.

Capítulo 7

Trabalhos Correlatos

A seguir serão descritos alguns trabalhos que têm abordado o tema da reconfiguração dinâmica. Adicionalmente, na Tabela 7 pode ser observado um comparativo entre algumas das características que distinguem essas soluções, cada uma delas identificada pela sequência em que é apresentada nesta subseção.

7.1 Polylith – Hofmeister, Purtilo

Em (HOFMEISTER; PURTILO, 1993), uma aplicação distribuída é constituída de módulos de *software* e ligações (*bindings*) entre eles, sendo que um módulo é um processo de *software* com sua própria memória e *thread* de controle. Módulos podem se comunicar através de interfaces nomeadas, que são portas lógicas de comunicação bidirecionais, de entrada ou de saída. Mensagens são enviadas em interfaces de saída e recebidas em interfaces de entrada, sendo que a troca de mensagens é assíncrona. Ligações conectam as interfaces de módulos em uma aplicação. È utilizada a plataforma para aplicações de *software* distribuídas POLYLITH (PURTILO, 1994) que provê uma linguagem para descrição de aplicações e canal (*bus*) de *software* para gerenciamento das atividades em tempo de execução. Para

possibilitar reconfiguração foram acrescentadas algumas primitivas que incluem adição e remoção de módulos e ligações, e uma operação que solicita a um módulo que divulgue sua informação de estado em uma interface particular, e então move aquela informação para a interface de um outro módulo.

A plataforma prepara um módulo automaticamente para participar da reconfiguração, desde que o programador defina os seus pontos de reconfiguração. Tais pontos indicam quando a reconfiguração é segura e qual o estado do programa. O código fonte do programa é transformado através da adição dos comandos necessários para:

- Postergar a reconfiguração até o momento apropriado;
- Organizar e enviar o estado;
- Instalar o estado, restaurar a pilha de registro de ativação quando necessário;
- Reiniciar a execução no lugar apropriado.

7.2 Reconfiguração de aplicações baseadas em componentes em .NET – Polze, Rasche e Puhlmann

O *framework* para configuração de aplicações baseado na plataforma .Net foi inicialmente apresentado em (RASCHE; POLZE, 2003), sendo sua evolução abordada em (RASCHE; POLZE, 2005; RASCHE; PUHLMANN; POLZE, 2005). Aplicações distribuídas são descritas como grafos interconectados, sendo que componentes representam nós e arcos denotam conexões entre eles. Além disso, atributos influenciam o comportamento dos componentes.

O conjunto de componentes parametrizados e as conexões entre elas formam uma configuração da aplicação. É definida uma linguagem baseada em XML que permite descrever as possíveis configurações de uma aplicação, além de possibilitar a definição de

observadores que monitoram o valor de determinadas propriedades do ambiente, como também o estado do componente e suas propriedades internas.

O elemento central do *framework* é a infra-estrutura de reconfiguração denominada CoFRA, que implementa um algoritmo de reconfiguração baseado no trabalho apresentado por Wermelinger (WERMELINGER, 1999). Um mecanismo de adaptação avalia políticas de adaptação e aciona a reconfiguração se mudanças no ambiente são detectadas. Uma infra-estrutura de monitoramento observa o ambiente da aplicação e seus componentes, enviando informações ao mecanismo de adaptação. Se mudanças significativas são detectadas, uma requisição de reconfiguração é enviada para a infra-estrutura CoFRA que executa a reconfiguração imediatamente.

Cada componente precisa implementar uma interface específica de reconfiguração contendo métodos para conectar componentes, atribuir valores a propriedades de componentes, bloquear conexões, iniciar o processamento do componente e finalizar um componente antes de ser removido.

O suporte à instanciação de componentes é efetuado por configuradores de componentes (CoCo) que escondem a complexidade para tratar com diferentes tipos de componentes, tais como objetos CORBA, .NET e Java. Um CoCo provê métodos para criar uma instância de um componente, acessar a interface de configuração de um componente, consultar o estado de um componente e remover uma instância.

Em seu artigo de 2005 (RASCHE; POLZE, 2005), os autores analisam o comportamento temporal do algoritmo de reconfiguração implementado. Eles argumentam que aplicações de *tempo real* realizam suas atividades de forma periódica e que, para que seja possível a reconfiguração dinâmica deste tipo de aplicação, faz-se necessário que recursos de processador para comandos de reconfiguração sejam reservados. É apresentada uma

demonstração de como tais recursos podem ser calculados antes da execução da aplicação e aplicados no seu projeto e implementação.

7.3 Reconfiguração dinâmica em OSA+ - Schneider, Piciorogã, Brinkschulte

OSA+ (*Open System Architecture*) (SCHNEIDER; PICIOROAGA;, BRINKSCHULTE, 2004; BRINKSCHULTE; SCHNEIDER; PICIOROAGA, 2005) é um *middleware* extensível (*scalable*) para sistemas de *tempo real* embarcados, projetado para minimizar o consumo de memória e CPU, sendo sua arquitetura baseada em serviços. Os serviços se comunicam através de tarefas (*jobs*) , onde uma tarefa é um par formado por uma ordem e um resultado. Nesse contexto, reconfiguração significa a substituição ou movimentação de serviços.

Um serviço de reconfiguração é definido para tratar os aspectos relacionados às modificações estruturais, tais como adicionar um novo serviço, transferir estado ou remover um serviço antigo. A reconfiguração de um serviço é acionada através da chegada de uma tarefa, que, como quaisquer outras tarefas, pode possuir propriedades relacionadas, entre elas, prioridades e *deadlines* para a reconfiguração.

Caso não haja necessidade de transferência de estado entre serviços, a troca de uma versão do serviço por outra pode ser feita de forma imediata. De acordo com o trabalho, foi preciso adicionar dois métodos ao *middleware* OSA+: *TransferState* e *Switch.* O primeiro é executado pelo serviço de reconfiguração e é responsável por gerenciar a transferência de estado. O segundo foi adicionado ao *microkerne*l e é responsável por transferir as conexões do serviço antigo para o novo.

São apresentadas três abordagens para quando o estado de um serviço precisa ser transferido para outro: bloqueante, parcialmente bloqueante e não bloqueante.

- Abordagem bloqueante: Quando a chamada *TransferState* é realizada o novo serviço é bloqueado até que o antigo serviço chame *Switch* para iniciar a troca. Depois que a troca é feita, o estado é transferido.
- Abordagem parcialmente bloqueante: Quando *TransferState* é acionado a nova versão do serviço inicia a transferência de estado, sendo que a versão antiga não é bloqueada neste momento. Quando a versão antiga do serviço chama o método *Switch*, os serviços são trocados e a informação de estado remanescente é transferida.
- Abordagem não bloqueante: Como na versão anterior, a nova versão do serviço inicia a transferência de estado chamando *TransferState*. Quando a versão antiga do serviço chama *Switch*, a informação de estado remanescente a ser transferida é monitorada. Se o tempo necessário para transferir essa informação for menor que o tempo máximo aceitável de interrupção do sistema, a troca é feita e o restante da informação de estado é transferida. Caso contrário, a troca é adiada. Isso significa que a operação *Switch* retorna sem substituir os serviços e *TransferState* continua a transferir a informação de estado. Daí em diante, toda vez que a versão antiga do serviço procura por uma nova tarefa, uma chamada automática é feita ao *Switch* pelo *middleware* para verificar se a troca pode ser efetuada. No caso em que a troca não pode ser efetuada, existe a possibilidade de que, em uma nova invocação de *TransferState*, uma parte do estado já transferido para o novo serviço tenha sido modificado na versão antiga ainda em execução. Para lidar com essa situação, um serviço chamado *StateServer* monitora a transferência de estado e detecta a necessidade de transferir novamente uma informação modificada.

7.4 OpenCOM – Coulson, Blair, Grace, Joolia, Lee, Ueyama

O OpenCOM (COULSON e outros, 2004) é um modelo de componentes reflexivo com suporte à reconfiguração dinâmica. O modelo de programação consiste de primitivas para carregar componentes em unidades de gerenciamento e encapsulamento denominadas cápsulas (*capsules*) e para conectar componentes através das suas interfaces provedoras (*interfaces*) e receptoras (receptáculos) de serviço. O modelo de programação também suporta a noção de compartimentos (*caplets*), carregadores (*loaders*) e associadores (*binders*). Compartimentos são aninhados dentro de cápsulas e hospedam componentes, carregadores provêm formas de carregar componentes em vários tipos de compartimentos, enquanto associadores conectam interfaces e receptáculos localizados no mesmo compartimento ou em compartimentos distintos. Adicionalmente a essas entidades, são definidos *Frameworks* de Componentes (CF) e Meta-Modelos Reflexivos. CFs são um conjunto de componentes altamente acoplados que cooperam para efetuar alguma funcionalidade de um domínio específico e incorporam políticas para as mudanças dinâmicas de sua composição. Compartimentos, carregadores e associadores são implementados como componentes que são hospedados em CFs. Para cada um desses elementos existe um CF específico que provê operações para sua instanciação e gerenciamento.

Meta-Modelos Reflexivos são representações casualmente conectadas de aspectos do sistema, de modo a permitir sua inspeção e adaptação. No OpenCOM, o sistema é representado através de um grafo de componentes que pode ser modificado em tempo de execução para alterar a topologia do mesmo, obedecendo às políticas determinadas pelo CF.

7.5 Plastik – Gomes , Batista, Joolia, Coulson

Em (GOMES e outros, 2007) é descrito o *meta-framework* Plastik que objetiva suportar especificação e criação de sistemas baseados em *frameworks* de componentes e facilitar e gerenciar a reconfiguração em tempo de execução de tais sistemas, garantindo manutenção de integridade. A arquitetura suporta reconfiguração em tempo de execução através da integração de uma linguagem de descrição arquitetural (ADL), que é uma extensão da Acme/ Armani, com o modelo de componentes reflexivo OpenCOM.

Configuradores de Componentes localizados em cada CF interpretam especificações de operações de reconfiguração escritas em uma extensão da Acme, que são mapeadas para o OpenCOM. Para a reconfiguração programada são especificadas construções *on-do* que expressam condições avaliadas em tempo de execução que acionam ações de reconfiguração. A partir das especificações Acme, os configuradores de componentes geram os *scripts* para a realização e validação das ações de reconfiguração. Os *scripts* de validação verificam a correção das ações de reconfiguração comandadas, permitindo que sejam efetuadas ou cancelando-as, conforme o caso.

7.6 SwapCIAO – Balasubramanian, Natarajan, Parsons, Schmidt, Gokhale

O SwapCIAO (BALASUBRAMANIAN e outros, 2005) estende o CIAO através das seguintes funcionalidades adicionais: (1) mecanismo para atualizar a implementação de um componente dinamicamente e (2) mecanismo para redirecionar os clientes de um componente existente para uma nova e atualizada implementação do componente.

De acordo com os autores, os elementos principais do *framework* são:

- Compilador CIDL que suporta a opção *updatable* que aciona a geração de código para (i) definição de interfaces de fábrica (*factory*) para criação de

novas implementações de componentes, (ii) provisão de *hooks* para desenvolvedores de aplicações servidoras escolherem qual implementação de componente implantar (*deploy*), (iii) criar, instalar e ativar componentes no POA (*Portable Object Adapter*) escolhido pela aplicação e (iv) gerenciar as conexões entre portas de um componente atualizável.

- Um contêiner atualizável que provê um ambiente de execução em que implementações de componentes podem ser instanciadas, removidas, atualizadas e (re)executadas sob o controle de desenvolvedores de aplicações servidoras.
- A fábrica de componentes atualizáveis cria componentes e implementa uma fachada de invólucro (*wrapper facade*) que provê uma interface portável utilizada para implementar o padrão *Component Configurator*, utilizado pelo SwapCIAO para abrir e carregar as bibliotecas de vínculo dinâmico (DLL) em plataformas de execução heterogêneas.
- O gerenciador de repositório armazena implementações de componentes e é utilizado pela fábrica de componentes atualizáveis para procurar DLLs e localizar implementações de componentes que requerem atualização.

No SwapCIAO todas as invocações a operações de um componente são despachadas através do POA, que mantém uma tabela de despacho que armazena quantas requisições estão sendo processadas por cada componente em uma *thread*. Quando um componente está na iminência de ser removido durante uma atualização, o POA só o desativa quando o contador de referências ao componente chega a zero. Para prevenir que novas invocações cheguem ao componente enquanto estiver sendo atualizado, o contêiner bloqueia novas invocações para o componente no ORB do servidor utilizando interceptadores portáteis padrão do CORBA.

Durante o processo de atualização do componente a implementação antiga do componente é removida. Quando um cliente faz uma requisição à antiga referência depois que o componente foi removido, o POA do contêiner intercepta a requisição através de um ativador de serventes (*servant activator*). O ativador é um tipo especial de interceptador que pode criar uma implementação de componente dinamicamente, caso uma não esteja disponível ainda para atender a requisição. Uma vez que o componente tiver sido removido, o mapa de objetos ativos do POA não mais terá uma entrada correspondente para o mesmo, de forma que o ativador de serventes criará uma nova implementação de componente dinamicamente. Caso o componente a ser atualizado possua conexões com outros componentes, elas são armazenadas pelo SwapCIAO para que possam ser restauradas posteriormente.

7.7 LuaCCM – Maia, Cerqueira, Rodriguez

Maia, Cerqueira e Rodriguez (2004) conceberam um *framework* para adaptação dinâmica de sistemas, que combina os recursos da linguagem Lua (IERUSALIMSCHY; FIGUEIREDO, 1996) com o *CORBA Component Model* para possibilitar a adaptação de sistemas baseados em componentes. Os mecanismos de adaptação do *framework* são construídos sobre o LuaCCM, uma implementação do CCM que utiliza a linguagem de programação Lua, que acrescenta a possibilidade de modificar a estrutura dos componentes em tempo de execução, bem como as conexões entre eles, através do conceito de contêineres dinâmicos. Quando uma nova definição de componente é instalada, o contêiner cria um gerenciador de definição que obtém a definição do componente, a partir de um descritor ou do repositório de interfaces, e gera a implementação de classes envoltório (*wrapper*) e de contexto. Tais classes são instanciadas a cada vez que é criado um componente daquele tipo. Objetos envoltório são responsáveis por criar serventes que recebem as requisições ou eventos

e os despacham para os executores dos componentes, além de implementar as operações relacionadas a conexões entre portas. O objeto contexto armazena as referências de objetos conectados aos receptáculos do componente e é responsável por encaminhar os eventos enviados através de portas geradoras de eventos. Quando a definição do componente é modificada, as classes envoltório e contexto são adaptadas para refletir a nova definição. Adicionalmente, é possível modificar apenas uma única instância do componente, através da adaptação dos seus objetos envoltório e contexto particulares. Todas as interações entre os componentes são feitas através dos objetos envoltório e contexto, possibilitando a adição de interceptadores para modificar o comportamento dos componentes. Para facilitar a estruturação e organização da adaptação são definidos papéis e protocolos para descrever as mudanças a serem efetuadas. Os papéis descrevem as portas e interceptadores que devem ser acrescentados, incluindo as interfaces e eventos utilizados, bem como suas respectivas implementações. Através do protocolo, os novos papéis são aplicados aos componentes do sistema, de forma que a nova configuração seja efetivada.

7.8 Reconfiguration for fault-tolerant avionic systems - Ellis

Na solução descrita por Ellis (1997), a reconfiguração do sistema é feita em resposta a uma ou mais falhas de comunicação ou dos processadores. A aplicação é composto por *Functional Areas* (FA), que são a menor funcionalidade identificável que pode ser reconfigurada em um componente de *software*. Comunicação entre FA é feita através de trocas de mensagens por meio de canais de comunicação.

O sistema operacional (SO) provê uma interface virtual ao FA para suportar todas as comunicações externas e ações de configuração e reconfiguração, e mantém estruturas de dados com informações atualizadas sobre a configuração do sistema.

Todas as comunicações efetuadas com FAs em processadores distintos são feitas através de *links* inter-processadores. FAs residentes no mesmo processador se comunicam internamente através do SO. Falhas de *links* fazem com que o SO reconfigure a topologia de comunicação, direcionando a comunicação para um outro *link* disponível. Quando informados pelo SO de um novo estado de *link*, cada processador efetua sua própria reconfiguração independentemente.

A falha de um processador ou todos os *links* que o cercam separa o processador da rede, evitando utilização subseqüente de FAs nele existentes. Quando isso acontece, os FA são reposicionados em processadores disponíveis. Existem FAs especiais denominados SYSBUFF que funcionam como réplicas de FA críticos, armazenando os dados globais do FA da aplicação e agilizando o carregamento de FAs críticos.

7.9 Configuração de componentes no OSGi – Gui, De Florio, Sun e Blondia

Nos trabalhos de Gui, De Florio, Sun e Blondia (2008) é apresentado o serviço *Declarative Real-Time Component Executive* (DRCE) para controle de mudanças na configuração de aplicações que utilizam o *framework* OSGi . Essa tecnologia é formada por uma série de especificações que definem um sistema de componentes dinâmicos em Java, incluindo a descoberta e uso de serviços providos por terceiros e a instalação, atualização e desinstalação de módulos (*bundles*) sem a necessidade de reinicialização do sistema. Os autores criaram um modelo de componentes híbrido no qual uma parte suporta tarefas de *tempo real* e outra parte lida com adaptações de componentes e funções de gerenciamento. As tarefas de *tempo real* executam no RTAI, extensão do kernel Linux para *tempo real*, e as demais no *middleware* OSGi. A configuração de *tempo real* do componente, contendo informações tais como tipo de tarefa, prioridade e freqüência, fica armazenada no DRCE e é

obtida a partir de arquivos XML. Uma vez que tarefas de *tempo real* são encapsuladas como componentes OSGi, um gerente de adaptação específico para a aplicação pode acionar uma mudança estrutural no sistema, que será realizada utilizando as funcionalidades de configuração dinâmica providas pelo *middleware*.

7.10 TimeAdapt – Fritsch, Clarke

O *TimeAdapt* (FRITSCH; CLARKE, 2008) é um *framework* para reconfiguração, em tempo oportuno (*timely*), de aplicações baseadas em componentes com restrições suaves de *tempo real*. O gerente de reconfiguração de *tempo real* recebe as solicitações de mudança no sistema, especificadas por uma linguagem desenvolvida para esse fim. As solicitações de reconfiguração são objeto de um teste de admissão que determina se a reconfiguração poderá ser realizada. Reconfigurações são tratadas como tarefas de *tempo real* suaves com um tempo de liberação associado e um *deadline* suave. O teste de admissão estima a probabilidade de uma determinada reconfiguração executar em um tempo limitado com base no tempo de pior caso global, que é obtido através da soma dos tempos de cada ação de reconfiguração. Tais tempos são obtidos a partir da medição do tempo de várias execuções das ações possíveis, que são registradas em um histórico. Uma vez que são definidas previamente as possíveis configurações que o sistema pode assumir, bem como seus relacionamentos, o tempo de toda a reconfiguração é calculado obedecendo à ordem de precedência das ações. Outro fator considerado no cálculo de tempo de execução de pior caso é o tempo em que a aplicação leva para alcançar um estado quiescente para a reconfiguração, que pode ser fornecido pelo desenvolvedor do componente ou medido e estimado de forma similar à das ações de reconfiguração. O gerenciador de reconfiguração escalona e executa as ações de reconfiguração quando sua probabilidade de obedecer ao tempo limite excede um limiar

(*threshold*) especificado, sendo que são escalonadas como tarefas de *tempo real* que não sofrem preempção, com a maior prioridade possível.

O modelo de componentes utilizado no trabalho foi projetado a partir de um subconjunto do meta-modelo de arquitetura da UML 2.0 que suporta mudanças em tempo de execução entre componentes e conexões. Componentes podem ser estruturados em composições ou funcionar sozinhos (primitivos) e devem implementar uma interface para introspecção. Na composição existe um componente hierarquicamente superior que precisa implementar uma interface de reconfiguração que permite a adição, remoção e substituição de sub-componentes e a mudança de conexões entre eles.

7.11 RCA4RTES – Krichen, Hamid, Zalila, Coulette

O RCA4RTES (*Reconfigurable Computing Architectures for Real Time Embedded Systems*) (KRICHEN e outros, 2010) é um meta-modelo para especificação de arquiteturas de sistemas distribuídos de *tempo real*, bem como ações de reconfiguração que podem ser comandadas para modificar sua estrutura em tempo de execução.

O meta-modelo tem como base a linguagem AADL (*Architecture Analysis and Design Language*) e a extensão da UML MARTE, proposta pela OMG para o domínio de sistemas embarcados de *tempo real*. Através dele, sistemas podem ser compostos por várias *mode structures*, sendo que cada *mode structure* pode possuir várias configurações. Essas configurações são formadas por componentes estruturados para os quais pode ser atribuído um conjunto de propriedades não funcionais e restrições, e especificados detalhes de suas respectivas portas e comunicação. Cada configuração se relaciona com um plano de implantação (*Deployment Plan*) que especifica a implantação e configuração dos componentes estruturados no sistema.

A reconfiguração dinâmica é definida através de um conjunto de transições, que podem ser de dois tipos: (i) transições entre instâncias de *mode structures* distintas e (ii) transições na mesma *mode strucure*. Essas transições são acionadas na ocorrência de eventos específicos.

7.12 Reconfiguração dinâmica em sistemas Simplex – Feiler, Li

O trabalho descrito por (Feiler e Li (1998) propõe uma abordagem para garantia de consistência sintática e semântica em sistemas que utilizam a tecnologia Simplex.

Na abordagem Simplex, variações de componentes são utilizadas para tolerar falhas da aplicação. Uma variante é tida como líder e determina a saída do componente, sendo que a liderança pode ser modificada dinamicamente. O conceito de componente analiticamente redundante (ARC) é utilizado, de forma que as saídas das variantes podem ser diferentes, mas seu efeito na planta permanece dentro de uma região operacional. Falhas em componentes individuais são toleradas através da reconfiguração incremental do sistema quando elas são detectadas.

No trabalho, é proposta uma solução para evitar falhas através da detecção e recuperação de configurações inconsistentes através da realização de análise *offline* para determinar: (i) se uma configuração é inconsistente, devendo ser evitada como uma configuração alvo, (ii) qual a reconfiguração desejável quando uma falha ou inconsistência é observada e (iii) qual o impacto da mudança e como pode ser reduzido. Os resultados dessa análise são utilizados por um gerenciador de suporte à reconfiguração em tempo de execução. Essa análise *offline* pode identificar restrições de configuração através da verificação de inconsistências relativas à sintaxe, tipo, recursos e semântica entre variantes de componentes conectados. Através dessa análise, inconsistências em configurações alvo podem ser identificadas previamente. Desse modo, um subconjunto de configurações válidas é

determinado antecipadamente, fazendo com que apenas mudanças para configurações válidas sejam permitidas.

Uma vez que a atomicidade das transações não é garantida, é possível que o sistema permaneça em uma configuração inconsistente por um período de tempo, mas os autores argumentam que em sistemas de controle isso pode ser tolerado.

7.13 Atualização dinâmica de componentes - Xiao, Cao, You, Zhou, Mei

A provisão de funcionalidade de atualização dinâmica de componentes no *middleware* PKUAS, baseado no padrão EJB, é o foco do trabalho apresentado em (XIAO e outros, 2009). Na abordagem adotada, podem coexistir versões diferentes de componentes, sendo que as requisições dos clientes podem ser acrescidas de um número de versão, que permite direcioná-las à versão adequada do componente. Ao receber uma requisição, o *skeleton* do componente a encaminha para versão solicitada. Por sua vez, o *stub* do cliente também recebe dos *skeletons* dos componentes a informação da versão corrente, de forma que o cliente possa decidir se atualiza a versão registrada no seu *stub*. São disponibilizadas duas políticas de atualização: gradual e forçada. Na atualização forçada, todas as instâncias do componente são substituídas pela nova versão e os *stubs* dos clientes são forçados a atualizar suas versões. Caso exista necessidade de transferir o estado entre versões de componentes, a atualização é adiada até que o *middleware* sinalize que o componente se encontra em estado de quiescência e a transferência de estado seja realizada. As versões antigas dos componentes são removidas assim que finalizada a atualização das suas versões. Na atualização gradual, as versões antigas dos componentes são mantidas e os *stubs* dos clientes não são atualizados com a nova versão, de modo que somente novos clientes têm suas requisições atendidas pela nova versão do componente. Nessa modalidade, as versões antigas dos componentes só são

substituídas quando solicitado pelo cliente. Entretanto versões antigas de componentes ociosas por um período de tempo configurável podem ser removidas automaticamente.

No trabalho é apresentado um experimento para medir a sobrecarga de tempo causada pelas funcionalidades de atualização, que é considera aceitável pelos autores.

Conforme mencionado anteriormente, a Tabela 7 apresenta um comparativo dos trabalhos abordados nesta subseção, sendo que cada um deles é identificado pela sequência em que aparece no texto. As características do serviço de reconfiguração dinâmica objeto desta dissertação estão localizadas na última linha. Para fins de organização, a tabela foi dividida em duas partes, cada uma das partes contendo todos os trabalhos abordados e um subconjunto das características consideradas.

Tabela 7: Comparativo entre trabalhos correlatos – parte I

Sequência	**Plataforma de execução**	**Modelagem do sistema em componentes**	**Operações reconfiguração**
1	POLYLITH	Não	Adição e remoção de módulos e ligações; Transferência de estado.
2	CoFRA/ .Net	Sim	Adição e remoção de componentes e conexões; Ajuste de atributos.
3	OSA+	Não	Adição e remoção de serviços; Transferência de estado.
4	OpenCOM	Sim	Componentes podem ser carregados e conectados.
5	Plastik/ OpenCOM	Sim	Componentes podem ser carregados e conectados.
6	CIAO/CCM	Sim	Substituição de componentes
7	LuaCCM	Sim	Alteração da estrutura dos componentes e de conexões
8	Módulo com vários processadores e comunicação através de mensagens	Não	Redirecionamento de conexões entre FAs
9	OSGi	Sim	Monitoramento e ajuste de parâmetros de *tempo real*

Tabela 7: Comparativo entre trabalhos correlatos – parte I (continuação)

Sequência	Plataforma de execução	Modelagem do sistema em componentes	Operações reconfiguração
10	TimeAdapt	Sim	Adição e remoção de componentes e conexões
11	Meta-modelo RCA4RTES	Sim	Transições entre configurações distintas
12	Sistemas baseados no modelo Simplex	Sim	Mudança de uma configuração para outra
13	PKUAS/ EJB	Sim	Substituição de componentes
Serviço de reconfiguração dinâmica	CIAO/ CCM	Sim	Adição e remoção de componentes e conexões; Ajuste de atributos; Transferência de estado.

Tabela 7: Comparativo entre trabalhos correlatos – parte II

Sequência	Mecanismo de consistência	Preocupação com restrições temporais (*tempo real*)
1	Através de pontos de reconfiguração pré-definidos	Não há menção
2	Através do bloqueio de conexões	Existe preocupação com os aspectos temporais, mas sem recursos específicos
3	O serviço substituto pode ser bloqueado até que o substituído comande a troca	Preocupação com baixo consumo de recursos pelo *midlleware*
4	Dependência entre componentes é considerada.	Preocupação com desempenho da infraestrutura
5	Regras arquiteturais especificadas e consideradas na validação das primitivas de reconfiguração	Não há menção
6	Componente só é removido quando não há requisição em andamento e novas invocações a componentes sendo atualizados são bloqueadas	Utilização de m*iddleware* com suporte à QoS
7	Interceptadores podem receber as requisições e tratá-las	Não há menção
8	Existência de réplicas de FAs críticos	Preocupação com desempenho da aplicação

Tabela 7: Comparativo entre trabalhos correlatos – parte II (continuação)

Sequência	Mecanismo de consistência	Preocupação com restrições temporais (*tempo real*)
9	Não há menção	Utiliza SO de tempo real e evita que tarefas com restrições temporais sofram interferência de tarefas comuns
10	Inexistência de comunicação em curso e computação em andamento nas partes da aplicação afetadas pela reconfiguração	Teste de admissão das ações de reconfiguração, escalonadas como tarefas de tempo real
11	Determinação prévia de possíveis configurações	Especificação de propriedades não funcionais Garantia de cumprimento de deadline será considerada em um trabalho futuro
12	Análise previa das configurações válidas	Preocupação com o impacto da reconfiguração no desempenho da aplicação
13	Estado de quiescencia necessário para transferência de estado e substituição de componente	Preocupação com sobrecarga causada pelas funcionalidades reconfiguração
Serviço de reconfiguração dinâmica	Gerenciador de consistência para conduzir a aplicação ao estado de reconfiguração	Utilização de *middleware* com suporte à QoS; Preocupação com desempenho.

Capítulo 8

Conclusão e Trabalhos Futuros

Modificar o comportamento de aplicações em tempo de execução pode ser de grande utilidade para sistemas com alto requisito de disponibilidade, dada a importância de efetuar manutenções corretivas e evolutivas em tempo reduzido de interrupção do seu funcionamento. Além disso, o serviço de reconfiguração dinâmica pode colaborar com outros serviços, tais como gerenciamento de recursos e tolerância a falhas, de modo a manter o desempenho da aplicação em um nível adequado ou até mesmo adaptá-la de acordo com a variação das condições do ambiente computacional.

Neste trabalho, procurou-se desenvolver um serviço de reconfiguração que permitisse modificar estruturalmente uma aplicação, considerando a necessidade de manutenção de consistência no que diz respeito ao estado dos componentes. Através das várias simulações efetuadas, foi possível observar que a execução da reconfiguração não inviabiliza a manutenção dos requisitos funcionais e temporais da aplicação. Entretanto, é preciso avaliar cuidadosamente em quais circunstâncias deve ser adotada, já que sua viabilidade de aplicação depende das características e requisitos do sistema no qual será empregada. Para sistemas com requisitos restritos de temporalidade, é preciso certificar-se de que o tempo em que o componente permanece bloqueado é aceitável. Sua utilização em

conjunto com outras tecnologias, pode garantir maior previsibilidade ao sistema em que é empregada, como no caso em que é associada a mecanismos de tolerância a falhas. Já seu emprego na atualização e evolução de sistemas com criticidade moderada pode ser mais abrangente, uma vez que permite automatizar e agilizar esses procedimentos. Observa-se também, que o fato de o sistema estar apto a mudanças dinâmicas oferece a possibilidade de contornar ocorrências inesperadas, tais como erros de implementação não descobertos nos testes. Além disso, pode auxiliar outros serviços existentes, como, por exemplo, a substituição de réplicas inoperantes de um serviço de tolerância a falhas ou a realocação de componentes de nós falhos para nós operacionais (KRAMER; MAGEE, 1985).

Embora o serviço de reconfiguração apresentado neste trabalho permita modificar a aplicação em tempo de execução de forma consistente, o *Reconfiguration Manager* se limita a realizar as mudanças solicitadas, seja essa solicitação feita através de uma invocação direta ou através do registro de políticas. Como trabalho futuro, considera-se que seria de interesse dotar esse componente de uma maior inteligência, ou até mesmo criar um componente em um nível superior, dotado de maior reatividade e capaz de gerar os comandos de reconfiguração a depender das necessidades observadas da aplicação, além de avaliar a correção e viabilidade das ações de reconfiguração solicitadas. Para tanto, suas decisões poderiam ser subsidiadas pelo seu uso integrado a gerenciadores de recursos e serviços de escalonamento dinâmico.

Vale observar também que, na implementação atual, existe apenas um a cópia dos componentes *Reconfiguration Manager*, *Deployment Manager* e *Consistency Manager*, o que os caracteriza como pontos únicos de falha. Por esse motivo, a replicação desses componentes também deverá ser considerada em trabalhos futuros.

Outro aprimoramento interessante será a disponibilização de ferramentas que permitam automatizar em algum nível a adaptação do código dos componentes da aplicação que será atualizada dinamicamente, a depender da abordagem de consistência utilizada.

Pretende-se ainda, investigar a aplicabilidade no serviço aqui descrito, das funcionalidades dos novos conectores existentes nas especificações DDS4CCM (OMG, 2009) e AMI4CCM (OMG, 2010), com implementação em curso no CIAO.

Referências

ANDRADE, S. MACÊDO, R. A Component-Based Real-Time Architecture for Distributed Supervision and Control Applications. In: **Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2005)** p. 15-22. 2005.

ANDRADE, S. S. **Sistemas Distribuídos de Supervisão e Controle Baseados em Componentes de Tempo Real**. Dissertação de Mestrado. Universidade Federal da Bahia. 2006.

ANDRADE, S. S., MACÊDO, R. J., SÁ, A. S. and SANTOS, N. P. Using Real-time Components to Construct Supervision and Control Applications. In: **Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)**. Work in Progress Session. 2006.

ANDRADE, S., MACÊDO R. Engineering Components for Flexible and Interoperable Real-Time Distributed Supervision and Control Systems. In: **Anais of the 12th IEEE Conference on Emerging Technologies and Factory Automation.** Patras – Greece. 2007.

AUDSLEY, N.C. et al. Real-Time System Scheduling. In : **Predictably Dependable Computing Systems**, ed. B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, ESPRIT Basic Research Series, Springer, 1995. p.41-52.

AUSLANDER, D. M. What is Mechatronics. In: **IEEE/ASME Transactions on Mechatronics**, v. 1, n. 1, mar. 1996.

BALASUBRAMANIAN, J.; NATARAJAN, B.; PARSONS, J.; SCHMIDT, D. C.; GOKHALE, A. Middleware Support for Dynamic Component Updating. In: **Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2005)**, Agia Napa, Cyprus, October 2005.

BIDAN, C.; ISSARNY, V.; SARIDAKIS, T; ZARRAS, A. A dynamic reconfiguration service for CORBA. In: **Proceedings of IEEE International Conference on Configurable Distributed Systems,** May 1998.

BOLTON, F.; WALSHE, E. **Pure Corba**. Pearson Education. 2001.

BURNS, A. Real-Time Systems. In: **Enclyclopedia of Physical and Technology**. Academic Press, v. 14, 2002.

BRINKSCHULTE, U; SCHNEIDER, E.; PICIOROAGA, F. Dynamic Real-time Reconfiguration in Distributed Systems: Timing Issues and Solutions. In: **Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing**. 2005.

BRYAN, K.et al. Integrated CORBA Scheduling and Resource Management for Distributed Real-Time Embedded Systems. In: **Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium.** 2005.

CONAN, Denis; PUTRYCZ, Erik ; FARCET , Nicolas; DE MIGUEL, Miguel. Integration of Non-Functional Properties in Containers. In: **Proceedings of the Sixth International Workshop on Component-Oriented Programming** (WCOP), 2001.

COSTA, Fábio; KON, Fábio. Novas Tecnologias de Middleware: Rumo à Flexibilização e ao Dinamismo. In: **20º Simpósio Brasileiro de Redes de Computadores**. Búzios, 2002. p.1-61.

COTTET, F; DELACROIX, J; KAISER, C; MAMMERI, Z. **Scheduling in Real-Time Systems**. John Wiley & Sons Ltd. 2002.

COULSON, G.; BLAIR, G.S.; GRACE, P.; JOOLIA, A.; LEE, K.; UEYAMA, J.: OpenCOM v2: A Component Model for Building Systems Software. In: **IASTED Software Engineering and Applications**. Cambridge (MA), USA. 2004.

DE MIGUEL, M; RUIZ, J.; GARCIA, M. QoS-Aware Component Frameworks. In: **Tenth IEEE International Workshop on Quality of Service**, p. 161--169, May 2002.

DENG, G.; SCHMIDT, D. C.; GILL, C. D.; WANG, N. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. **Handbook of Real-Time and Embedded Systems**. CRC Press, 2007.

DINATALE, M.; STANKOVIC, J. Dynamic End-toEnd Guarantees. Distributed Real-Time Systems. In: **Real-Time Systems Symposium**, 1994.

EIDE, Eric; STACK, Tim; REGEHR, John; LEPREAU, Jay. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In: **Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium**. 2004.

ELLIS S.M. Dynamic software reconfiguration for fault-tolerant real-time avionic systems. **Microprocessors and Microsystems**, Volume 21, Number 1, July 1997, pp. 29-39(11).

FARINES, Jean-Marie; FRAGA, J. S.; OLIVEIRA, R. S. **Sistemas de Tempo Real**. Departamento de Automação e Sistemas. Universidade Federal de Santa Catarina. Florianópolis, julho de 2000.

FEILER, P.; JUN Li. Consistency in dynamic reconfiguration. In: **Proceedings of the Fourth International Conference on Configurable Distributed Systems**, 1998.

FIDGE, C. J. **Real-time scheduling theory**. SVRC Services (UniQuest Pty Ltd). Prepared for the Air Operations Division, Defence Science and Technology Organisation. Consultancy Report 0036-2, April 2002.

FLEEMAN, David; GILLEN, M.; LENHARTH, A.; DELANEY, M.; WELCH, L.; JUEDES, D.; LIU, C. Quality-Based Adaptive Resource Management Architecture (QARMA): A CORBA Resource Management Service. In: **18th International Parallel and Distributed Processing Symposium - Workshop 2**. 2004.

FRITSCH, S.; CLARKE, S. TimeAdapt: timely execution of dynamic software reconfigurations. In: **Proceedings of the 5th Middleware Doctoral Symposium**, Leuven, Belgium. ACM, New York, NY, pp. 13-18, 2008.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de Projeto – Soluções reutilizáveis de Software Orientado a Objetos**. Bookman, 2000.

GOMES , A.T.A.; BATISTA, T; JOOLIA, A., COULSON, G. Architecting Dynamic Reconfiguration in Dependable Systems. Architecting Dependable Systems IV, **Lecture Notes in Computer Science**, Vol. 4615, 2007.

HECK, Bonnie. S.; WILLS, Linda M.; VACHTSEVANOS, George J. *Software* Enabled Control: Background and Motivation. In: **Proceedings of the American Control Conference**. Arlington, VA June, pp. 25-27, 2001.

HENNING, M.; VINOSKI, Stephen; VINOSKI, Steve. **Advanced CORBA Programming with C++**. Addison-Wesley Professional. 1999.

HILLMAN, J.; WARREN, I. An Open Framework for Dynamic Reconfiguration. In: **26th International Conference on Software Engineering (ICSE'04)**, p. 594-603, 2004.

HOFMEISTER, C.; PURTILO, J. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In: **Proceedings of the 13th International Conferenceon Distributed Computing Systems**, IEEE Computer Society Press. Pittsburgh, May, p. 101-110, 1993.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. ; FILHO, W. C. Lua — An Extensible Extension language. **Software: Practice and Experience.** v, 26 p. 635–652, junho 1996.

KAO, B.; GARCIA-MOLINA, H. Deadline Assignment in a Distributed Soft Real-Time System. In: **Proceedings of the 13th International Conference on Distributed computing Systems**. 1993.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J. M.; IRWIN, J. Aspect-oriented programming. In: **ECOOP'97 - Object-Oriented Programming, 11th European Conference,** LNCS 1241, p. 220-242, 1997.

KIM, K. H.; LIU, J., MIYAZAKI, H.; SHOKRI, E. TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming. In: **Proceedings of the ISDAS '01 (Fifth International Symposium on Decentralized, Autonomous Systems)**, Dalas, TA, pp. 327-335, March 2001.

KOPETZ, Hermann. Should Responsive Systems be Event-Triggered or Time-Triggered. **IEICE Transactions on Information and Systems**, v. E76-D, n. 11, p. 1325-1332, 1993.

KOPETZ, Hermann. Distributed Fault-Tolerant Real-Time Systems: the Mars Approach. **IEEE Micro**, v. 9, n. 1, pp. 25-40, 1999.

KOPETZ, Hermann. *Software* Engineering for Real-Time: A Roadmap. In: **Proceedings of the 22nd International Conference on Future of *Software* Engineering (FoSE) at ICSE 2000**.Limerick, Ireland, 2000.

KRAMER, J; MAGEE, J. Dynamic Configuration for Distributed Systems. **IEEE Transactions on Software Engineering**, v. 11, n. 4, p. 424-436, April 1985.

KRAMER, J.; MAGEE, J. The Evolving Philosophers Problem: Dynamic Change Management. **IEEE Transactions on Software Engineering**. v. 16, issue 11. 1990.

KRICHEN, Fatma; HAMID, Brahim; ZALILA, Bechir; COULETTE, Bernard. Designing dynamic reconfiguration for distributed real time embedded systems. **10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)** p.249-254, 2010

KRISHNAMURTHY, Y.; PYARALI, I.; GILL, C.; MGETA, L.; Zhang, Y.; TORRI, S.; SCHMIDT, D. C. The Design and Implementation of Real-Time CORBA 2.0:Dynamic Scheduling in TAO. In: **Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium**. 2004.

LAPLANTE, P. A. **Real-Time Design and Analysis**. 3rd Edition. 2004.

LEUNG, J.; MERRILL, M. A note on preemptive scheduling of periodic real-time tasks, **Information Processing Letters**. 1980.

LEUNG, J; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic real-time tasks. **Performance Evaluation**. 1982.

LIU, C. L; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. **Journal of the Association for Computing Machinery**. January, 1973.

MACÊDO, R. J. A. LIMA, G. M.; BARRETO, L. P.; ANDRADE, A. M. S.; BARBOZA, F. J. R.; ALBUQUERQUE, R.; ANDRADE, S. Tratando a Previsibilidade em Sistemas de Tempo-Real Distribuídos: Especificação, Linguagens, Middleware e Mecanismos Básicos. In: **22º Simpósio Brasileiro de Redes de Computadores**, **Livro texto do minicurso**, pp. 105-163, Gramado, 2004.

MAGEE, J.; SLOMAN, M. Constructing Distributed Systems in Conic. **IEEE Transactions on Software Engineering.** v. 15, issue 6 (Jun. 1989), p. 663-675. 1989.

MAIA, R; CERQUEIRA, R.; RODRIGUEZ, N. An Infrastructure for Development of Dynamically Adaptable Distributed Components. In: **International Symposium on Distributed Objects and Applications (DOA'04) - On the Move to Meaningful Internet Systems,** Berlim : Springer-Verlag, p. 1285-1302, 2004.

MCKINLEY, Philip K. et al. Composing Adaptive *Software*. **IEEE Computer Society**, jul. 2004.

MICROSOFT. **Microsoft COM Technologies**. http://www.microsoft.com/com/ default.mspx. 2000.

MICROSOFT. **Net Framework**. http://msdn2.microsoft.com/pt-br/netframework/ default(en-us).aspx. 2000.

MOAZAMI-GOUDARZI, K.. **Consistency preserving dynamic reconfiguration of distributed systems.** Ph.D. thesis, Imperial College, London, March 1999.

GUI, N.; DE FLORIO, V.; SUN, H.; BLONDIA, C. A hybrid real-time component model for reconfigurable embedded systems. In: **Proceedings of the 2008 ACM symposium on Applied computing**. Fortaleza, Ceara, Brazil, March 2008

GUI, N., DE FLORIO, V., SUN, H., AND BLONDIA, C. A framework for adaptive real-time applications: the declarative real-time OSGi component model. In: **Proceedings of the 7th Workshop on Reflective and Adaptive Middleware**, December 2008.

OBJECT MANAGEMENT GROUP. **Data Acquisition from Industrial Systems(DAIS), Request for Proposal(RFP)**, OMG Document: dtc/99-01-02. 1999. http:/www.omg.org/techprocess/meetings/scheduling/Data_Acquisition_RFP.html.

OBJECT MANAGEMENT GROUP. **Common Object Request Broker Architecture (CORBA), v2.6**, December 2001.

OBJECT MANAGEMENT GROUP. **Common Object Request Broker Architecture (CORBA), v3.0**, July 2002.

OBJECT MANAGEMENT GROUP. **Deployment and Configuration of Component-based Distributed Applications Specification**. June 2003.

OBJECT MANAGEMENT GROUP. **Online Upgrades.** 2003.

OBJECT MANAGEMENT GROUP. **RealTime CORBA v2.0.** November 2003.

OBJECT MANAGEMENT GROUP. **CORBA Component Model Specification**. http://www.omg.org/cgi-bin/doc?formal/06-04-01. 2006.

OBJECT MANAGEMENT GROUP. **Data Distribution Service (DDS) for Lightweight CORBA Component Model (DDS4CCM)**. 2009. http://www.omg.org/spec/dds4ccm/1.0/Beta2/PDF.

OBJECT MANAGEMENT GROUP. **Asynchronous Method Invocation (AMI) for CORBA Component Model (AMI4CCM)**, Request for Proposal(RFP). 2010. http://www.omg.org/cgi-bin/doc?mars/10-12-31.pdf.

OIKAWA, Shui; RAJKUMAR, Raj. Linux/RK: A Portable Resource Kernel in Linux. In: **IEEE Real-Time Systems Symposium Work-In-Progress**. Madrid. December 1998.

OGATA, K. **Modern Control Engineering**. 4th edition, Prentice-Hall, New Jersey. 2001.

OREIZY, P. Issues in Modeling and Analyzing Dynamic *Software* Architecture. In: **Proceedings of the International Workshop on the Role of *Software* Architecture in Testing and Analysis**, 1998.

PAVAN, Allalaghatta; JHA, Rakesh; GRABA, Lee; COOPER, Saul; CARDEI, Ionut; GOPAL, Vipin; PARTHASARATHY, Sanjay; BEDROS, Saad. Real-Time Adaptive Resource Management. **IEEE Computer** 34(7): 99-101. 2001.

PAZOS, F. **Automação de Sistemas e Robótica**. Rio de Janeiro: Axcel Books, 2003.

PURTILO, J. M. The Polylith Software Toolbus. **ACM Transactions on Programming Languages and Systems**. 1994.

RAJKUMAR, R.; JUVVA, K.; MOLANO, A.; OIKAWA, S.. Resource kernels: A resource-centric approach to real-time and multimedia systems. In: **Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking**, January 1998.

RASCHE, A.; POLZE, A. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In: **Proceedings of International Symposium on Object-oriented Real-time distributed Computing**. 2003.

RASCHE, A.; PUHLMANN, M.; POLZE, A. Heterogeneous Adaptive Component-Based Applications with Adaptive.Net. In: **Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.** May 2005.

RASCHE, A.; POLZE, A. Dynamic Reconfiguration of Component-based Real-time Software. In: **Proceedings of the 10th IEEE international Workshop on Object-Oriented Real-Time Dependable Systems**. 2005.

ROMÁN, Manuel; KON, Fabio; CAMPBELL, Roy H. Reflective Middleware: From Your Desk to Your Hand. **IEEE Distributed Systems Online**, v. 2, n. 5, 2001.

SANS, R. et al. **Engineering Handbook for CORBA-based Control Systems**. Universidad Politécnica de Madrid. Madrid, 2003.

SANTOS, N. P. ; MACÊDO, R. J. de A. ; BARRETO, L. P. . Reconfiguração Dinâmica de Componentes em Sistemas Distribuídos de Controle e Supervisão, com Aplicação a Tolerância a Falhas. In: **Anais do IX Workshop de Teste e Tolerância a Falhas**, Rio de Janeiro, 2008.

SCHMIDT Douglas C.; LEVINE, David L.; MUNGEE, Sumedh.The Design of the TAO Real-Time Object Request Broker. **Computer Communications**, v. 21, n. 4, 1998.

SCHMIDT, Douglas C.; KUHNS, F. An overview of the Real-Time CORBA specification. **IEEE Computer**. v 33, issue 6, p. 56 – 63. Jun 2000.

SCHMIDT, Douglas C.; O'RYAN, C. Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures. **Journal of Systems and Software, Special Issue on Software Architecture – Engineering Quality Attributes**, Elsevier, 2002.

SCHNEIDER, E.; PICIOROAGA, F.; BRINKSCHULTE, U. Dynamic reconfiguration through OSA+, a real-time middleware. In: **Proceedings of the 1st international Doctoral Symposium on Middleware.** 2004.

SHAW, A. **Sistemas e Software de Tempo Real**. Bookman, Porto Alegre, 2003. 1ª edição.

STEWART, D, B.; KHOSLA, P, K. Real-Time Scheduling of Dynamically Reconfigurable Systems. In: **Proceedings of the IEEE International Conference on Systems Engineering**, p. 139 – 142, Aug. 1991.

SUN MICROSYSTEMS. **Java™ RemoteMethodInvocation Specification (RMI)**. Sun Microsystems, Inc. http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf. 2004.

SUN MICROSYSTEMS. **Enterprise JavaBeans™ Specification, v3.0**. Sun Microsystems, Inc. http://java.sun.com/products/ejb. 2006.

SZYPERSKI, C.; GRUNTZ, D.; MURER, S. **Component Software: Beyond Object-Oriented Programming**. Addison-Wesley. 2002.

VANEGAS, R.; ZINKY, J. A.; LOYALL, J. P.; KARR, D. A.; SCHANTZ, R. E.; BAKKEN, D. E. QuO's runtime support for quality of service in distributed objects. In: **Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)**, September 1998.

XIAO, Z., CAO, D., YOU, C., ZHOU, M., MEI, H. Towards Dynamic Component Updating: A Flexible and Lightweight Approach. In: **Proceedings of the 2009 33rd Annual IEEE international Computer Software and Applications Conference**. COMPSAC. IEEE Computer Society. v. 01, p. 468-473. July, 2009.

ZHANG, J; DIPIPPO, L; FAY-WOLFE, V; BRYAN, K; MURPHY, M. A Real-Time Distributed Scheduling Service For Middleware Systems. In: **Proceedings of the 10th International Workshop on Object-Oriented, Real-Time, Dependable Systems**, Sedona, AZ, Feb. 2005.

ZHANG, Y.; GILL, C., D.; LU, C. Configurable Middleware for Distributed Real-Time Systems with Aperiodic and Periodic Tasks, **IEEE Transactions on Parallel and Distributed Systems.** IEEE computer Society Digital Library. IEEE Computer Society. April 2009.

ZINKY, John A.; BAKKEN, David E.; SCHANTZ, Richard E. Architectural Support for Quality of Service for CORBA Objects, **Theory and Practice of Object Systems**, v. 3 nr. 1, January 1997.

WANG, Nanbor; SCHMIDT Douglas C.; GOKHALE, Aniruddha; GILL, Christopher; NATARAJAN, Balachandran; RODRIGUES, Craig; LOYALL, Joseph; SCHANTZ, Richard. Total Quality of Service Provisioning in Middleware and Applications. **The Journal of Microprocessors and Microsystems**, v. 27, n. 2, mar. 2003.

WANG, Nanbor**. Composing Systemic Aspects Into Component-Oriented DOC Middleware**. PhD thesis, St. Louis: Washington University, 2004.

WANG, S.; RHO, S.; MAI, Z.; BETTATI, R.; ZHAO, W. Real-time component-based systems. **Real Time and Embedded Technology and Applications Symposium**, 2005. RTAS 2005. 11th IEEE. p. 428 – 437, March 2005.

WERMELINGER, M. A. **Specification of software architecture reconfiguration**. Ph.D. thesis, Universidade Nova de Lisboa, September 1999.

WILLS, Linda et al. An Open Platform for Reconfigurable Control. **IEEE Controls Systems Magazine**, jun. 2001.