UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Cássia Yuri Tatibana

UM MODELO PARA PROVISÃO DE GARANTIA DINÂMICA DE TEMPO REAL EM MIDDLEWARE BASEADO EM COMPONENTES

Florianópolis

2007

Cássia Yuri Tatibana

UM MODELO PARA PROVISÃO DE GARANTIA DINÂMICA DE TEMPO REAL EM MIDDLEWARE BASEADO EM COMPONENTES

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica para a obtenção do Grau de Doutor.
Orientador: Prof. Rômulo Silva de Oliveira, Dr.
Coorientador: Prof. Carlos Montez, Dr.

Florianópolis

2007

Ficha de identificação da obra elaborada pelo autor através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Tatibana, Cássia Yuri
Um Modelo para Provisão de Garantia Dinâmica de Tempo Real em Middleware Baseado em Componentes / Cássia Yuri Tatibana ; orientador, Rômulo Silva de Oliveira; co-orientador, Carlos Barros Montez. - Florianópolis, SC, 2007.
114 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. Tempo Real. 3. Componentes de Software. 4. Middleware. 5. Garantia Dinâmica de Tempo Real. I. Silva de Oliveira, Rômulo . II. Barros Montez, Carlos. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

Cássia Yuri Tatibana

UM MODELO PARA PROVISÃO DE GARANTIA DINÂMICA DE TEMPO REAL EM MIDDLEWARE BASEADO EM COMPONENTES

Esta Tese foi julgada adequada para a obtenção do Título de Doutor, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Florianópolis, 26 de Abril 2007.

Prof. Nelson Sadowski, Dr.
Coordenador

Banca Examinadora:

Prof. Rômulo Silva de Oliveira, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Carlos Montez, Dr.
Coorientador
Universidade Federal de Santa Catarina

Prof. Lau Cheuk Lung, Dr.
Universidade Federal de Santa Catarina

Prof. Cláudio Fernando Resin Geyer, Dr.
Universidade Federal do Rio Grande do Sul

Prof. Mário Dantas, Dr.
Universidade Federal de Santa Catarina

Prof. Joni da Silva Fraga, Dr.
Universidade Federal de Santa Catarina

AGRADECIMENTOS

À Deus, aos meus pais e meus irmãos.

Aos orientadores pela dedicação e paciência.

À universidade e ao departamento pela oportunidade.

Aos amigos tão queridos por tornarem a experiência do doutorado ainda melhor.

Agradeço em especial ao Alysson, por ter me encontrado! E ao Montez, pela amizade!

RESUMO

A abordagem baseada em componentes foi desenvolvida em resposta à necessidade de lidar com a complexidade das aplicações e diminuir o ciclo de desenvolvimento do software. A separação em lógica de aplicação e parte não funcional em um componente permite que requisitos temporais sejam configurados, ao invés de inseridos ao longo do código. Como resultado, os componentes se tornam menos dependentes da plataforma subjacente e podem ser reusados em aplicações diferentes. Este trabalho apresenta um modelo para provisão de garantia dinâmica em sistemas de tempo real distribuídos baseados em componentes. O modelo desenvolvido condiciona a aceitação de um cliente à disponibilidade de recursos para satisfazer os requisitos temporais deste cliente e de clientes previamente aceitos. Este modelo permite a adoção de diferentes algoritmos para o teste de aceitação, se adequando ao modelo das tarefas escalonadas ou à capacidade da plataforma. Outra contribuição é um serviço de monitoramento de tempos de resposta de componentes, inicialmente desenvolvidos para prover dados iniciais para o modelo de garantia dinâmica. O serviço de monitoramento permite que o mecanismo de garantia dinâmica se mantenha preciso apesar da flutuação da carga computacional do servidor e permite a aplicação de algoritmos probabilistas para o modelo de garantia dinâmica.
Palavras-chave: Tempo Real, Componentes de Software, Garantia Dinâmica.

ABSTRACT

The component-based approach was developed in response to the need to cope with application complexity and reduce the software development time. The component separation of concerns allows real-time constraints to be configured instead of hard coded. As a result, components become less dependent from the underlying platform and can be reused in different applications. This work presentes a model for real-time dynamic guarantee for component-based distributed systems. According to the model, the acceptance of a client to the system is subject to the availability of resources to satisfy all clients real-time constraints. This model allows the use of different algorithms for the acceptance test, according to the application task model or the platform capacity. Another contribution is the response time monitoring service, developed to provide input data for the dynamic guarantee model. This service provides updates for the dynamic guarantee model and also allows the use of probabilistic approaches for the acceptance test.

Keywords: Real-Time, Software Components, Dynamic Guarantee.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

À medida que tecnologias de *middleware* se tornam mais presentes, novas funcionalidades são incorporadas com o objetivo de aperfeiçoá-las para fins específicos. Aviação, espaço, controle automação são apenas alguns dos exemplos de áreas de atuação desta tecnologia. Soluções concretizadas através de tecnologias de *middleware* podem ser encontradas em trabalhos de tolerância à faltas, segurança, tempo real, etc. O aperfeiçoamento do *middleware* para todas estas diversas áreas resultou em uma infraestrutura bastante completa quanto á capacidade de atender vários tipos de requisitos. Entretanto, alcançar todo o potencial desta tecnologia eliminando a sobrecarga adicional de funcionalidades desnecessarias, tornou-se uma tarefa complexa.

Os componentes de *middleware* foram desenvolvidos em resposta à necessidade de lidar com a complexidade crescente de sistemas de software. A abordagem baseada em componentes objetiva a diminuição do ciclo de desenvolvimento de software através de reusabilidade e modularização. Como unidades de implantação, componentes são desenvolvidos e instalados de forma independente, sem que uma atualização ou modificação de componente comprometa o funcionamento do resto da aplicação. A reusabilidade permite o desenvolvimento mais rápido de aplicações e facilita o processo de manutenção, atualização e extensão dos sistemas.

Os componentes apresentam uma arquitetura interna que separa a lógica da aplicação da parte não funcional do componente (parte de software relacionada a infraestrutura necessária para provisão da funcionalidade do componente). Esta separação define papéis de desenvolvimento de um componente de software A: o especialista na lógica da aplicação para o qual se destina a funcionalidade a ser desenvolvida e o integrador da aplicação que assegura que a plataforma de execução e os demais componentes cumpram com os requisitos exigidos pelo componente A. Esta separação permite que especialistas de cada área sejam capazes de construir sistemas sofisticados mesmo que não dominem todos os aspectos da plataforma envolvida no projeto. E por outro lado exige uma infraestrutura para este componente muito mais sofisticada quanto aos serviços a serem suportados; o container.

As especificações de componentes definem interfaces amplamente aceitas que permitem que componentes independentes; desenvolvidos por diferentes fornecedores (ou empresas) sejam conectados e interoperem mesmo quando estes componentes são desenvolvidos em diferentes

linguagens de programação, compiladores e plataformas (WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei, 2005). As especificações estabelecem o acordo ou contrato que torna possível a compatibilidade entre componentes.

As especificações de components mais conhecidas, o OMG *CORBA Component Model* (CCM) (Object Management Group, 2006a), *Microsoft COM+* (REDMOND, F. E., 1997) e *Sun Microsystems Enterprise Java Beans* (EJB) (Sun Microsystems, 2004), lidam de forma eficiente com requisitos orientados a negócios mas provêem suporte insuficiente para aplicações de tempo real (WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei, 2005).

No contexto desta tese, o desenvolvimento de sistemas de tempo real baseado em componentes permitiria a utilização de componentes de software com parâmetros temporais configuráveis; a execução de uma mesma funcionalidade com diferentes comportamentos temporais. O mesmo componente poderia prover diferentes níveis de serviço em tempo real (*criticality*), de acordo com os requisitos da aplicação ou as restrições de recursos do sistema.

1.1 MOTIVAÇÃO

Na literatura de tempo real existem muitos modelos e ferramentas voltadas para o desenvolvimento de sistemas baseadas em componentes. De maneira geral, estes modelos ou ferramentas foram desenvolvidas especificamente para uma classe de aplicações.

Dentro da classe de sistemas embarcados (*embedded systems*), restrições de memória e comportamento determinista são o foco central. Tanto no meio comercial como no meio acadêmico podem ser encontrados diversos trabalhos cujo foco são sistemas embarcados (*embedded systems*) (ISOVIC, D.; NORSTRM, C., 2002; VAN OMMERING, R.; LINDEN, F. van der; KRAMER, J., 2000; YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2002). Tanto os modelos de componentes quanto as ferramentas desenvolvidas nestes trabalhos objetivam acima de tudo, restrições de memória e determinismo em detrimento de aspectos como adaptabilidade ou reconfigurabilidade em tempo de execução, que são pouco ou nada exploradas. A eficácia dessa classe de componentes é medida por consumo de recursos, energia e testabilidade (MOLLER, Anders; AKERHOLM, Mikael; FROBERG, Joakim; FREDRIKSSON, Johan; SJODIN, Mikael, 2006).

Na classe de sistemas multimídia(KON, Fabio; MARQUES, Jefer-

son Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis, 2005), os tipos de dados e a natureza das aplicações admitem certa flexibilidade; a carga computacional pode ser acomodada ao limite de recursos sob pena de degradação da qualidade do serviço que geralmente pode ser tolerada. Devido a variabilidade deste tipo de tolerância por parte de aplicações multimídia, ferramentas e modelos desenvolvidos para aplicações que não objetivam qualquer determinismo são muitas vezes usadas com resultado satisfatório.

As duas classes de sistemas mencionados são, de certa forma, extremos do domínio de aplicações com restrições de tempo real. Baseado na pesquisa dentre os vários modelos de componentes existentes na literatura (apresentado no Capítulo 7), é possível concluir que muitos dos trabalhos envolvendo componentes de sistemas embarcados possuem um domínio bastante restrito quanto a portabilidade. Vários destes modelos de componentes são desenvolvidos e utilizados dentro de uma única empresa, para uma linha específica de produtos. Enquanto sistemas multimídia ou sistemas mais flexíveis quanto a restrições temporais adotam soluções que nem sempre são projetadas para sistemas de tempo real ou que não são facilmente partilhadas por outras classes de aplicações também de tempo real.

Uma grande variedade de aplicações de tempo real é distribuídas e opera sob uma demanda computacional variável e em ambientes heterogêneos. Por diversas razões, estas aplicações devem oferecer uma qualidade de serviço controlada aos seus clientes. Este é o caso, por exemplo, de videogames; servidores na Internet devem prover a qualidade requerida para um número imprevisível de jogadores pagantes. Isto requer disponibilidade de recursos (como processadores e largura de banda, por exemplo) em qualquer momento ou ao menos formas alternativas de lidar com condições de sobrecarga mantendo sob controle a degradação da qualidade do serviço.

Para sistemas como os de automação industrial ou rastreamento de alvos (MANGHWANI, P.; LOYALL, J.; SHARMA, P.; GILLEN, M.; YE, Jianming., 2005; SHNAKARAN, Nishanth; KOUTSOUKOS, Xenofon; LU, Chenyang; SCHMIDT, Douglas C.; XUE, Yuan, 2006), os requisitos tornam-se ainda mais rígidos. O projeto deste tipo de sistema envolve questões de interoperabilidade em diferentes plataformas, tratamento de atraso de comunicação e a capacidade de adaptação à mudanças das condições operacionais uma vez que a carga computacional e a disponibilidade de recursos não pode ser caracterizada com precisão *a priori*.

Ainda no domínio de sistemas embarcados e críticos, como os sistemas usados para a área do espaço ou aviação (UAVs), a flexibili-

dade quanto a carga computacional pode ser um requisito. Satélites que recebem requisições para captura de imagens ou rastreamento de eventos naturais a partir de diferentes pontos de controle em solo decidem quanto a sua capacidade em atender tais requisições baseados em requisições já previamente aceitas. Este tipo de sistema é composto por subsistemas de fornecedores diferentes, embarcados em equipamentos diferentes (sensores, atuadores e câmeras, cada um com seu próprio controlador), que operam em conjunto para completar cada uma das requisições recebidas. Configurações semelhante são utilizada em aeronaves não tripuladas de pequeno porte, usadas para rastreamento ou vigilância de territórios.

Em sistemas de tempo real dinâmicos novas tarefas podem ser ativadas em tempo de execução. Caso não exista o controle da carga computacional do servidor, novas tarefas serão ativadas até que o servidor não consiga mais satisfazer os requisitos temporais de qualquer tarefa.

Dentro de cada domínio de aplicação, componentes de software, ou modelos e ferramentas de suporte ao desenvolvimento de componentes de software de tempo real foram especializados para diferentes aspectos do que definimos como propriedades de componentes de software.

Neste trabalho detectamos a necessidade de um modelo de componentes que pudesse ser utilizado por sistemas de tempo real não restritos a qualquer domínio, cujos requisitos pudessem ser traduzidos para qualquer algoritmo clássico de tempo real e que pudesse suportar flexibilidade de requisitos de tempo real como uma propriedade não funcional, e portanto configuráveis do sistema.

Considere um sistema de controle de tráfego aéreo, cujo objetivo é guiar aeronaves no ar e no solo e garantir um fluxo de tráfego seguro e ordenado. O domínio deste sistema é dividido em torre de controle e controle de área. O domínio da torre de controle são as proximidades do aeroporto; pistas de pouso e decolagem e áreas de manobra de aeronaves em solo e até uma determinada altitude. O domínio do controle de área é delimitado pelo espaço entre os domínios da torre de controle da origem e destino das aeronaves. Aeronaves transitam entre estes domínios conforme orientações da torre de controle e do controle de área.

Até certo ponto, vôos que partem ou chegam aos domínios podem ser conhecidos com antecedência de modo que uma escala possa ser montada. Entretanto, atrasos de vôos, acidentes ou mudanças de condições climáticas que provoquem chuva, neve, ou granizo podem

exigir alterações da escala. Por isso a necessidade de flexibilidade do sistema. Quando há um aumento no número de vôos, condições climáticas severas ou quando são detectadas situações de falta de segurança, controle de torre e controle de área devem gerenciar os vôos de modo que a transição de aeronaves de um domínio para o outro não provoque a sobrecarga destes sistemas. Partidas podem ser atrasadas, aterrissagens podem ser redirecionadas, e aeronaves em cruzeiro podem reduzir ou aumentar a velocidade dentro de limites pré-estipulados.

Dentro do contexto desta aplicação, sob o ponto de vista da torre de controle, por exemplo, aeronaves são clientes. Tanto aeronaves que partem quanto aeronaves que aterrissam no domínio do controle de torre só o podem fazê-lo se o controle puder garantir a administração destas operações. A torre de controle é o servidor que pode administrar chegadas e partidas dentro dos limites de um limite de recursos. Aeronaves que precisam aterrissar são clientes que precisam do controle da torre para instruções de aproximação. A partir do primeiro contato, atualizações periódicas de instruções de aproximação são mantidas entre torre de controle e aeronave.

O tipo de serviço prestado pela torre a qualquer aeronave que requisite aproximação é o mesmo, as tarefas executadas pela torre de controle são as mesmas. Possivelmente, variações existirão conforme o porte da aeronave, o que pode exigir maior ou menor velocidade da mesma e consequente maior ou menor freqüência da atualização do controle da torre. Neste sentido, a carga computacional da torre de controle varia conforme o número de aeronaves no seu domínio e as restrições temporais impostas por estas aeronaves.

O controle de ambos os domínios, envolve a cooperação de sistemas de mapeamento, informações climáticas, rotas de aeronaves, informações de múltiplos radares e *transponders* que por sua vez provêem dados de entradas para sistemas de predição de trajetória, alerta de conflito, otimizações de chegada e gerenciamento de fluxo de tráfego. Cada um destes sistemas gerenciam recursos específicos e em combinação provêem informações na frequência necessárias para o gerenciamento do tráfego aéreo. Todos estes equipamentos caracterizam recursos diferentes a serem administrados de modo que cooperem.

Os exemplos de aplicações descritos ilustram os vários requisitos de diferentes tipos de sistemas de tempo real. Soluções baseadas em componentes de tempo real foram exploradas dentro de vários domínios, entretanto, a especificidade de cada domínio restringiu a solução desenvolvida a apenas parte dos aspectos de componentes. Esta tese trata do desenvolvimento baseado em componentes no contexto de aplicações

de tempo real distribuídas submetidas à carga computacional dinâmica. O foco é preservar as propriedades fundamentais do desenvolvimento baseado em componentes em um modelo que permita a provisão de garantia dinâmica de tempo real.

Neste trabalho propomos a integração de um mecanismo de controle de carga computacional a infraestrutura de componentes de software. Este controle se baseia nos requisitos de tempo real das chamadas a este componente em tempo de execução. O objetivo é capacitar um componente de software a avaliar sua própria situação de carga corrente, e com base nesta carga descartar ou aceitar futuras invocações por serviços. Cada um dos componentes que fazem parte do servidor avaliam, de forma independente, sua carga computacional. O resultado das avaliações individuais determina a aceitação do conjunto de chamadas futuras. Parte do critério de avaliação consiste no período durante o qual o servidor compromete seus recursos a atender estas chamadas.

Nesta tese não é assumido um comportamento específico para o sistema operacional e protocolos de rede subjacentes. Definimos modelos de componentes flexíveis o suficiente para dispor da capacidade dos sistemas operacionais e das camadas de rede existentes.

As propostas desta tese não estão obrigatoriamente atreladas a nenhum padrão de componentes existente. Entretanto, como forma de validá-las, uma implementação como prova de conceito foi desenvolvida. O modelo escolhido foi o modelo de componentes CORBA, um dos mais difundidos no domínio de componentes e mais explorado com relação a tempo real.

Embora muitos trabalhos da literatura abordem os temas desta tese, não foram identificados trabalhos que explorem, de maneira identica, o mesmo tipo de solução. O modelo proposto explora características fundamentais de componentes de software para capacitar este tipo de tecnologia a prover o controle de acesso a serviços baseado em restrições de natureza temporal. A idéia principal do modelo é que as restrições de tempo real de um serviço sejam avaliadas por componente, sendo o container a parte ideal para implementação deste controle. Pretende-se mostrar como o conhecimento que o container possui sobre os componentes e sobre o sistema podem ser usados para controlar sobrecargas transitórias ou mesmo fornecer garantia dinâmica para as requisições dos clientes.

1.2 OBJETIVOS DA TESE

O objetivo geral desta tese é apresentar um modelo baseado em componentes que inclui mecanismos que permitem um teste de aceitação em tempo de conexão (*bind*) no contexto dos sistemas de tempo real. Os mecanismos considerados são genéricos o suficiente para utilizar diferentes algoritmos de análise de escalonabilidade como teste de aceitação. Este modelo é chamado, no decorrer da tese de DGC (*Dynamic Guarantee for Components*). Visando atender o objetivo geral desta proposta, os objetivos específicos listados abaixo foram perseguidos.

- Proposição do modelo DGC, com sua descrição detalhada de modo que permita sua integração com diferentes padronizações de componentes existentes;
- Implementação de um protótipo que sirva como prova de conceito para o modelo de componentes proposto mostrando a viabilidade da realização do DGC em um padrão de componentes conhecido;
- Medições sobre o protótipo implementado, para avaliar os atrasos introduzidos pelos mecanismos do modelo DGC;
- Elaboração do serviço de monitoramento (*profiling*) dos tempos de resposta dos métodos dos componentes instalados em um dado container, a fim de suportar atividades de escalonamento baseadas nos perfis temporais dos componentes;

1.2.1 Metodologia

1.2.1.1 Classificação da Pesquisa

Segundo a classificação proposta por (SILVA, E. L.; MENEZES, E. M. , 2005) quanto à natureza da pesquisa, esta tese se enquadra como aplicada, pois gera conhecimentos para aplicação prática que podem ser dirigidos à solução de problemas específicos. Quanto à forma de abordagem do problema, trata-se de um trabalho inicialmente qualitativo (criação de modelos) e depois quantitativo (medição de protótipo e simulação). Os objetivos da pesquisa são predominantemente explicativos. Quanto aos procedimentos técnicos, a pesquisa é bibliográfica e experimental. Em um primeiro momento, foi elaborada a partir de

material já publicado, constituído principalmente de livros, artigos de periódicos e de conferências. Em um segundo momento, a fase de avaliação do modelo proposto é feita através de experimentos conduzidos via prototipagem e simulação.

1.2.1.2 Roteiro da Pesquisa

A partir da classificação previamente apresentada, a pesquisa que culminou com a escrita deste trabalho seguiu os seguintes passos:

1. Revisão bibliográfica da literatura relacionada;
2. Estudo detalhado de propostas que envolvessem assuntos em comum interesse aos desta tese;
3. Proposição de um modelo de componentes o qual inclui teste de aceitação pelo container;
4. Implementação de um protótipo como prova de conceito e medição de algumas características do protótipo ligadas a overhead;
5. Inclusão no modelo de suporte para profiling de componentes por parte do container;
6. Exemplo de uso da monitoração feita pelo container, acompanhado de avaliação por meio de simulações;
7. Escrita da tese, com base nos dados obtidos anteriormente.

1.3 ADEQUAÇÃO ÀS LINHAS DE PESQUISA DO CURSO

O trabalho descrito nesta tese está inserido no contexto da Área de Concentração em Automação e Sistemas do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina. Este trabalho está perfeitamente integrado com os demais trabalhos de pesquisa sobre tempo real, com as atividades da área de concentração e do Curso de Pós-Graduação em Engenharia Elétrica desta Universidade. Espera-se que os resultados obtidos possam contribuir para próximas teses de doutorado, dissertações de mestrado, projetos e trabalhos de graduação deste curso, assim como serem empregados no desenvolvimento de sistemas de tempo real distribuídos.

1.4 ORGANIZAÇÃO DO TEXTO

O texto da tese é resultado das diversas etapas que foram cumpridas durante o trabalho de doutorado. A primeira parte, composta pelos capítulos iniciais, corresponde a uma etapa exploratória da literatura, na qual foi necessário o levantamento dos conceitos utilizados das disciplinas de componentes e tempo real. Na segunda parte é apresentado um modelo computacional para aplicações baseadas em componentes de tempo real, em que um controle de admissão de novos clientes baseado na carga computacional de cada nodo do domínio do servidor é aplicado em tempo de conexão entre cliente e servidor. Ao final, é detalhado um serviço de monitoramento de tempos de resposta de componentes que provê os dados essenciais para a provisão da garantia dinâmica. Este texto está dividido em 8 capítulos.

O Capítulo 2 apresenta a caracterização dos sistemas de tempo real, os principais requisitos usados neste tipo de sistema e algumas das principais abordagens da literatura para o problema do escalonamento de tempo real. O Capítulo 3 introduz alguns dos conceitos sobre componentes, os quais são fundamentais para este trabalho. O texto apresenta uma caracterização da programação baseada em componentes. O Capítulo 4 contém a descrição do modelo DGC, capaz de prover garantia dinâmica para aplicações de tempo real distribuídas. O modelo permite a adoção de diferentes algoritmos, atendendo a aplicações com modelos de tarefas diversos e também a plataformas com diferentes opções de algoritmos de escalonamento. O Capítulo 5 descreve a tradução deste modelo independente de plataforma de componentes em uma arquitetura comum a várias tecnologias de componentes e descreve a implementação prova de conceito do modelo no CCM. Foram feitas medições de tempos no protótipo com o propósito de avaliar o *overhead* dos mecanismos utilizados pelo modelo. O Capítulo 6 apresenta uma proposta para um serviço de monitoramento de tempos de resposta de componentes que, num primeiro momento provê os dados de entrada para o modelo DGC, mas que pode ser utilizado para monitorar os tempos de respostas de componentes. O monitoramento provê a atualização do tempo de resposta que varia conforme a carga computacional no nodo e permite a adoção de abordagens probabilistas no teste do modelo DGC. O capitulo 7 apresenta os principais trabalhos relacionados à componentes de tempo real e uma análise sobre estas propostas com relação ao modelo DGC. Finalizando, o Capítulo 8 apresenta as considerações finais sobre o trabalho desenvolvido. São destacados os principais resultados alcançados e identificadas questões

relevantes, ainda em aberto, no contexto de componentes e tempo real.

2 SISTEMAS DE TEMPO REAL

Na realização de operações de controle de tráfego aéreo, em jogos de videogame ou na supervisão de plantas nucleares, é exigido do sistema computacional que os gerencia um comportamento imediato e acima de tudo, previsível. Para estes tipos específicos de aplicações, é importante que o sistema forneça respostas dentro de um prazo específico, sob pena de provocar danos até mesmo catastróficos ao ambiente com o qual interagem. São identificados nestes sistemas, requisitos de natureza temporal; as respostas exigidas dos mesmos não são apenas resultados esperados, são também requeridas no momento correto.

Esta é a classe de sistemas denominada de sistemas de tempo real. Em geral aplicações desta classe são também aplicações reativas, que estabelecem comunicação com o ambiente caracterizando sistemas interativos. Por isso, o conceito de sistemas de tempo real os define como sistemas que devem reagir a estímulos oriundos do seu ambiente em prazos específicos (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

A exigência do cumprimento de requisitos temporais faz com que o aspecto de correção destes sistemas envolva correção lógica (*correctness*) e correção temporal (*timeliness*) (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Uma generalização do conceito de sistemas de tempo real os define como sistemas capazes de oferecer garantias de correção temporal para o todos os seus serviços que apresentem restrições temporais. Nesta definição, se encaixam os sistemas de tempo real que fornecem serviços a um usuário humano ou a equipamentos (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Independente do fato de ser reativo ou não, o sistema de tempo real possui como conceito principal, a previsibilidade. Um sistema é dito previsível no domínio lógico e temporal quando seu comportamento pode ser antecipado antes de sua execução; quando seu comportamento independe de variações do ambiente de execução provocados por características de hardware, carga ou falha.

Para que um sistema possa ser dito previsível, é preciso considerar duas hipóteses sobre o ambiente externo: a hipótese de carga e a hipótese de falhas. A primeira, determina a carga máxima gerada pelo ambiente num intervalo mínimo de tempo. Enquanto a segunda descreve tipos e freqüência de falhas às quais o sistema está exposto durante sua execução ao passo que permanece atendendo seus requisitos

temporais e funcionais(KOPETZ, Hermann; VERISSIMO, Paulo, 1993).

A partir da determinação destas hipóteses, é assumido que um sistema previsível deve ter seu comportamento antecipado mesmo quando executa sob a pior situação de carga ocorrendo simultaneamente com a hipótese de falha.

A garantia de previsibilidade exige ainda o conhecimento detalhado sobre o comportamento temporal do sistema em relação a todos os elementos que fazem parte de sua execução (linguagem de programação, hardware, suporte). Estas propriedades caracterizam a previsibilidade determinista, que permite uma análise em tempo de projeto do comportamento temporal do sistema diante dos recursos disponíveis.

Nesses sistemas, o desempenho do caso médio não é mais importante que o desempenho do pior caso. A velocidade de processamento não garante, sozinha, bom comportamento de uma aplicação de tempo real. Um bom comportamento de uma aplicação de tempo real é verificado quando é possível garantir o atendimento de requisitos temporais de cada uma das atividades da aplicação (STANKOVIC, John A., 1988).

Quando a hipótese de carga e a hipótese de falhas não podem ser antecipadas, previsibilidades probabilistas podem ser obtidas a partir de simulações.

Quanto à garantia no atendimento de requisitos temporais, sistemas de tempo real podem fornecer garantia em tempo de projeto ou garantia dinâmica. A garantia em tempo de projeto é aquela que pode ser obtida quando existe previsibilidade determinista, enquanto a garantia dinâmica é a garantia que pode ser obtida por abordagens de melhor esforço em que a carga é dinâmica e conhecida somente em tempo de execução (MONTEZ, Carlos Barros, 2000; SHIN, A.; RAMANATHAN P., 1994).

2.1 CLASSIFICAÇÃO DE SISTEMAS DE TEMPO REAL

Tendo em vista a aplicabilidade destes sistemas, eles se apresentam numa grande variedade quanto à complexidade, criticalidade e tamanho. Sistemas de tempo real podem ser encontrados em aplicações simples e pequenas como controladores embutidos em eletrodomésticos até aplicações complexas e robustas, como aquelas implementadas para controle de tráfego aéreo (OLIVEIRA, Rômulo Silva de, 1997; STANKOVIC, John A., 1988) .

Por isso, diversas classificações são encontradas para estes sistemas, cada uma delas considerando diferentes aspectos. Sob o ponto de

vista da segurança, sistemas de tempo real podem ser críticos (*Hard Real Time Systems*) ou brandos (*Soft Real-Time Systems*). Esta classificação observa as conseqüências de uma falha temporal do sistema; quando a falha é da mesma ordem de grandeza que os benefícios do mesmo em operação normal, o sistema é caracterizado como sistema de tempo real brando. Por outro lado, se a conseqüência de uma falha temporal excede em muito os benefícios do sistema sob condições normais de funcionamento, este sistema é classificado como sistema de tempo real crítico (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Exemplos clássicos de sistemas de tempo real brando são aqueles voltados para aplicações de teleconferência, videogames, etc. Entre os sistemas de tempo real crítico se destacam aqueles de controle de tráfego aéreo, controle de plantas nucleares ou sistemas militares de defesa(FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Os sistemas de tempo real críticos são ainda divididos em Sistemas de Tempo Real Críticos Seguros em Caso de Falha e Sistemas de Tempo Real Críticos Operacionais em Caso de Falha. No primeiro caso, um ou vários estados seguros podem ser alcançados na presença de falha. No segundo, entretanto, a ocorrência de falhas parciais provoca degradação do sistema, que passa a fornecer algum tipo de serviço mínimo.

A distinção de sistemas de tempo real de acordo com o ponto de vista da implementação também é definida por alguns autores como uma classificação que define sistemas de resposta garantida (em que existe recursos suficientes para suportar a carga de pico e cenários de falhas definidos) e sistemas de melhor esforço (em que a estratégia de alocação dinâmica de recursos se baseia em estudos probabilístas sobre a carga esperada e os cenários de falhas aceitáveis).

2.2 O PROBLEMA DE TEMPO REAL E ABORDAGENS DE SOLUÇÃO

O problema de tempo real consiste em especificar, verificar e implementar sistemas ou programas, que mesmo em situações de limitações de recursos, apresentam comportamento previsível, atendendo restrições temporais impostas pelo ambiente ou pelo usuário(FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000). Considerando estes aspectos de construção, tempo real pode ser visto inicial-

mente como um problema intrínseco de programação concorrente.

Quanto ao tratamento de concorrência, são definidas duas abordagens: a abordagem síncrona e a abordagem assíncrona.

A abordagem síncrona é mais abstrata em relação aos aspectos de implementação. O principio básico desta abordagem é não considerar os tempos gastos com cálculos e comunicação; o ambiente é considerado suficientemente lento para tornar esta hipótese verdadeira. A observação dos eventos é cronológica, permitindo a eventual simultaneidade entre eles. A partir das suas premissas, a concorrência é resolvida sem o entrelaçamento de tarefas e o tempo não é tratado de maneira explícita. Esta abordagem é considerada como orientada ao comportamento da aplicação e a sua verificação (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

A abordagem assíncrona trata a ocorrência e percepção de eventos independentes numa ordem arbitrária. Ela objetiva a descrição, o mais exata possível de um sistema, baseada na observação, durante a execução, de todas as combinações de ocorrência de eventos. Por se tratar de uma abordagem orientada a implementação, leva em consideração durante a especificação e projeto, características do suporte de software e hardware da aplicação. Portanto, aspectos de portabilidade não são simplesmente assumidos; considerações a respeito de características do sistema e de implementação contribuem para o aumento da complexidade e indeterminismo (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

A abordagem assíncrona está fundamentada no tratamento explícito da concorrência e do tempo de uma aplicação em tempo de execução. Conforme esta abordagem, a questão do escalonamento de é o ponto principal do estudo da previsibilidade dos sistemas de tempo real (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

2.3 MODELOS DE TAREFAS

O conceito de tarefa é uma das abstrações básicas do problema de escalonamento. Tarefas ou processos são termos associados a unidades de concorrência em um sistema, unidades de processamento seqüencial que concorrem por recursos computacionais (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000; OLIVEIRA, Rômulo Silva de, 1997).

Um modelo de tarefas é definido pelas restrições temporais impostas ao conjunto e pelas relações de precedência e exclusão entre as

tarefas. O modelo de tarefas é parte integrante do problema de escalonamento e contribui na análise de comportamento das atividades cooperantes de uma aplicação de tempo real visando a obtenção de previsibilidade (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000; MONTEZ, Carlos Barros, 2000).

Uma tarefa pode ser ativada ou habilitada a executar a partir da ocorrência de um evento, como o términou de outra tarefa, ou pela passagem do tempo; o tempo atual é igual ao instante de tempo determinado para a ativação da tarefa (OLIVEIRA, Rômulo Silva de, 1997). De acordo com a regularidade de ativações de uma tarefa, ela pode ser classificada como periódica, aperiódica ou esporádica.

Tarefas são chamadas de periódicas quando são ativadas uma só vez a cada intervalo regular de tempo (período) por tempo indefinido. Cada ativação da tarefa descreve uma instância da mesma.

Tarefas aperiódicas ou assíncronas têm seu processamento ativado em resposta a eventos externos ou internos, caracterizando ativações aleatórias. Tarefas esporádicas caracterizam um subconjunto das tarefas aperiódicas, estas tarefas possuem um intervalo mínimo entre ativações consecutivas.

Do ponto de vista da previsibilidade, devido à possibilidade de análises de escalonabilidade em tempo de projeto, tarefas periódicas e esporádicas são usualmente utilizadas na implementação de tarefas críticas. Enquanto que tarefas aperiódicas são empregadas na implementação de tarefas de tempo real brando.

Além do período de tarefas de tempo real, outras restrições temporais fazem parte da descrição de seu comportamento no tempo.

- Tempo de Computação (*Computation Time*) (C): Tempo de uso real do processador e outros recursos do sistema para a execução completa da tarefa.

- Instante de Início (*Start Time*) (st): Instante em que a tarefa inicia o processamento.

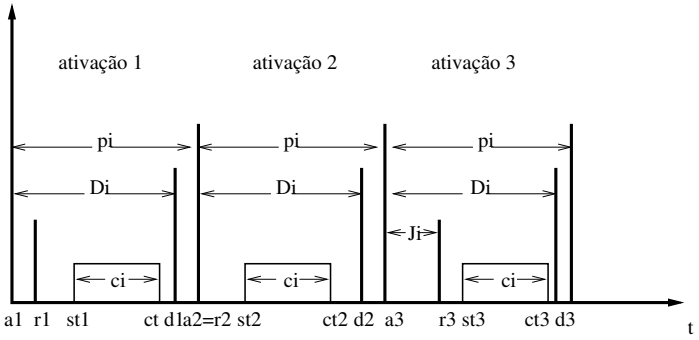
- Instante de Término (*Completion Time*) (ct): Instante de tempo em que a tarefa é concluída. Caso a tarefa execute sem interrupções a partir instante em que ganha o processador, o tempo de computação (C) corresponde ao intervalo entre o tempo de início st e o tempo de conclusão ct.

- Instante de Chegada (*Arrival Time*) (a): Instante em que o escalonador toma conhecimento da ativação de uma tarefa.

- Instante de Liberação (*Release Time*) (r): Instante em que a tarefa é incluída na fila de tarefas prontas para executar pelo escalonador do sistema.

- Jitter de Liberação (*Release Jitter*) (J): Máxima variação dos instantes de liberação das instâncias da tarefa.

O jitter de liberação ocorre sempre que o tempo de chegada a não coincide com o tempo de liberação r. Nem sempre uma tarefa é colocada na file de aptas para executar (*Ready Queue*) no mesmo instante em que é liberada. Tarefas ativadas por mensagens, por exemplo, ficam bloqueadas durante o recebimento da mensagem e tem um retardo no instante de liberação. Tarefas que esperam o polling de escalonadores ativados por tempo (tick scheduler) também sofrem atraso.



ai : Tempo de Ativação de i
ri: Tempo de Liberação de i
sti: Tempo de Início de i
Di: Deadline de i
pi: Período de i
Ci: Tempo de Computação de i
Ji: Release Jitter de i
cti: Tempo de Termino de i

Figura 1: Restrições temporais da tarefa periódica.

Conforme ilustrado pela figura 1, o comportamento de uma tarefa periódica T_i pode ser descrito pela quádrupla (J_i, C_i, P_i, D_i). Observa-se que o período (P) e o deadline (D) da tarefa são intervalos medidos a partir de cada início de período. Enquanto o deadline absoluto (d) e tempo de liberação (r) de cada ativação da tarefa são determinados a partir de períodos anteriores.

2.4 ESCALONAMENTO DE TEMPO REAL

O escalonamento é o processo de alocação de recursos do sistema e de ordenação das tarefas na fila de aptos numa escala de execução. O escalonador é o componente do sistema responsável pela gerência dos recursos e pela implementação da solução de escalonamento empregado. Ele implementa, durante a execução, a solução de escalonamento do sistema (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000; OLIVEIRA, Rômulo Silva de, 1997).

Uma solução de escalonamento mostra como o problema de alocar recursos às tarefas pode ser abordado ou mesmo resolvido. O escalonamento pode ser feito tanto em tempo de projeto quanto em tempo de execução. Soluções mistas podem ainda combinar parte do escalonamento em tempo de projeto e parte do escalonamento em tempo de execução (OLIVEIRA, Rômulo Silva de, 1997).

O algoritmo de escalonamento de sistemas de tempo real deve determinar, para um conjunto de tarefas, se existe uma escala de execução das tarefas que satisfaça seus requisitos de recursos e suas restrições temporais (MONTEZ, Carlos Barros, 2000). Tal verificação é implementada por testes de escalonabilidade que podem ou não fazer parte da abordagem de escalonamento adotada. A partir do teste, uma escala de execução é montada, definindo que tarefa utiliza qual recurso em que momento.

Um teste de escalonabilidade pode ser: suficiente mas não necessário, necessário mas não suficiente e exato. Todo conjunto de tarefas aprovado por um teste suficiente mas não necessário é escalonável mas nada pode ser dito sobre um conjunto reprovado. O conjunto de tarefas reprovado pelo teste necessário mas não suficiente não é escalonável, porém, nada pode ser dito sobre um conjunto aprovado. Testes exatos, por outro lado, aprovam conjuntos escalonáveis e reprovam conjuntos não escalonáveis (OLIVEIRA, Rômulo Silva de, 1997).

Diferentes abordagens de escalonamento são identificadas na literatura, tomando como base de classificação aspectos como previsibilidade, utilização de recursos e algoritmos de escalonamento empregados.

A carga a ser tratada pelos recursos computacionais em um determinado instante é definida pelo somatório dos tempos de computação do conjunto de tarefas que compõe a aplicação de tempo real. Quando as tarefas são bem conhecidas em tempo de projeto a carga é caracterizada como estática ou limitada. Por outro lado, quando as características de chegada das tarefas não podem ser antecipadas a carga é dita dinâmica ou ilimitada (FARINES, Jean Marie; FRAGA, Joni da Silva;

OLIVEIRA, Rômulo Silva, 2000).

Quando a carga é estática, testes de escalonabilidade podem ser aplicados e assim garantir o cumprimento de restrições temporais do conjunto de tarefas em tempo de projeto. Sob estas circunstâncias, caso seja necessário, o redimensionamento do sistema pode ser realizado de acordo com a situação de pior caso (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Conhecendo o tempo de computação e período de uma tarefa, é possível calcular a utilização de processador da tarefa. A utilização reflete a fração do tempo total do processador que a tarefa irá ocupar no pior caso. Para um conjunto de tarefas portanto, o cálculo inclui o somatório de utilização de todas as tarefas do conjunto. Algumas soluções de escalonamento se baseiam neste conceito.

Soluções de escalonamento estabelecem diferentes critérios ou regras e se baseiam em diferentes aspectos do conjunto de tarefas a ser escalonado. Quando tarefas em execução podem ser preemptadas por tarefas de maior prioridade, o algoritmo de escalonamento é dito preemptivo. Caso contrário, o algoritmo é denominado não preemptivo.

Algoritmos que realizam o cálculo da escala baseado em parâmetros de tarefas obtidos em tempo de projeto são chamados de algoritmos estáticos. Os algoritmos que baseiam seu cálculo em parâmetros mutáveis conforme a evolução do sistema, são chamados de dinâmicos.

Ainda quanto à produção da escala, algoritmos que produzem a escala em tempo de projeto são chamados de algoritmos *offline*. Os algoritmos que produzem a escala em tempo de execução são chamados de algoritmos *online*.

Pela combinação das propriedades apresentadas acima, temos algoritmos *offline* estáticos, *online* estáticos, *online* dinâmicos.

2.5 ABORDAGENS DE ESCALONAMENTO

O tratamento do problema de escalonamento tempo real pode ser dividido em teste de escalonabilidade e cálculo da escala de execução. O teste de escalonabilidade determina se as restrições temporais do conjunto de tarefas serão atendidas, considerando o algoritmo de escalonamento implementado.

Baseado no tipo de carga tratada e nas etapas de escalonamento, três grupos principais de abordagens podem ser descritos: abordagens com garantia em tempo de projeto, abordagens com garantia em tempo de execução e abordagens de melhor esforço (RAMAMRITHAN, K.; STAN-

KOVIC, J. A., 1994).

As abordagens de garantia em tempo de projeto têm como objetivo a previsibilidade determinista, e parte de um conjunto de premissas:

- A carga computacional é conhecida em tempo de projeto (carga estática);
- Os recursos no sistema são suficientes para cumprir tarefas respeitando suas restrições temporais na condição de pior caso.

Estas abordagens são dirigidas para aplicações críticas deterministas. Os dois tipos de abordagens desta categoria são: o executivo cíclico e os escalonadores dirigidos por prioridades.

No executivo cíclico, tanto o teste de escalonabilidade quanto a produção da escala são realizados em tempo de projeto. A escala produzida é executada ciclicamente pelo processador. Como a escala reflete o pior caso, embora exista a garantia de cumprimento de restrições, existe também desperdício de recursos. O pior caso é geralmente distante do caso médio e nem sempre ocorre (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Abordagens baseadas em escalonadores dirigidos por prioridade são mais flexíveis, eles realizam o teste de escalonabilidade em tempo de projeto mas produzem a escala em tempo de execução. Como o pior caso é considerado no teste de escalonabilidade, a solução implementada garante a escalonabilidade do conjunto. A ordem de execução das tarefas é determinada em tempo de execução (algoritmo *online*) por um escalonador estático (tarefas tem prioridades estáticas) ou dinâmico (tarefas tem prioridades variáveis). Exemplos de propostas que seguem esta abordagem podem ser encontrados em (AUDSLEY, N. C.; TINDELL, K.; BURNS, A.; WELLINGS A. J., 1993) e (SHA, L.; SATHAYE, S. S., 1994).

Quando a atribuição de prioridades é feita estaticamente, as tarefas recebem valores de prioridades durante o teste de escalonabilidade conforme critérios pré-determinados. A política Taxa Monotônica (*Rate Monotonic*) é um exemplo clássico dessas abordagens. Nela, a distribuição de prioridades é feita de maneira inversamente proporcional aos períodos das mesmas (MONTEZ, Carlos Barros, 2000).

Na atribuição dinâmica de prioridades, a prioridade atribuída à tarefa é alterada pelo escalonador de acordo com a situação temporal corrente da mesma. O escalonador realiza a alteração de prioridades das tarefas considerando outros atributos das mesmas e em momentos

pré-determinados pelo algoritmo. Um exemplo clássico desta abordagem é o escalonamento EDF (*Earliest Deadline First*) (LIU, Chien-Liang; LAYLAND James W., 1973) em que prioridades são atribuídas às tarefas na fila de aptos a cada liberação, conforme a proximidade de cada um de seus deadlines absolutos.

Estendendo o modelo de tarefas do taxa monotônica, o escalonamento *Deadline* Monotônico(DM) (LEUNG, J. Y. T.; WHITEHEAD, J., 1982), assume *deadlines* relativos menores ou iguais ao períodos das tarefas. As demais premissas permanecem as mesmas da abordagem Taxa Monotônica: tarefas periódicas e independentes, tempo de computação constante e tempo de chaveamento entre tarefas assumido como nulo. No *Deadline* Monotônico a atribuição de prioridades é estática e também é dita ótima para sua classe de problemas.

A premissa de tarefas independentes é bastante restritiva, grande parte das aplicações de tempo real determina alguma forma de relação de exclusão e compartilhamento de recursos entre tarefas. Tais relações determinam bloqueios em tarefas mais prioritárias que são identificados na literatura de tempo real como inversões de prioridades.

A inversão de prioridades ocorre quando tarefas compartilham um recurso guardado por um mecanismo de exclusão mútua. Uma tarefa de maior prioridade, T_1, é impedida de prosseguir a execução por estar bloqueada, a espera de um recurso sendo usado por uma tarefa de menor prioridade, T_4.

Em meio a tal situações, eventuais bloqueios podem ser provocadas sobre a tarefa de mais alta prioridade (T_1). Tarefas de prioridade intermediária (T_2 ou T_3 com valor de prioridade entre o valor de prioridade de T_1 e T_4) podem preemptar a tarefa T_4 e assim, a tarefa T_1 deve esperar que T_4 libere o recurso, mesmo estando esta tarefa a espera do processador, sendo ocupado pelas tarefas T_2 ou T_3. Na situação descrita, a tarefa de maior prioridade do conjunto considerado, T_1, permanece a espera da execução de tarefas de menor prioridade.

Quando tarefas possuem dependências entre si, inversões de prioridade são inevitáveis. Os mecanismos existentes procuram limitar o tempo durante o qual uma tarefa é bloqueada em consequência destas inversões. Alguns métodos implementam regras que permitem o conhecimento a priori do pior caso de bloqueio experimentado por uma tarefa no acesso a uma variável compartilhada (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

Duas das técnicas utilizadas para este propósito são: Protocolo de Herança de Prioridade (SHA, L.; RAJKUMAR, R., LEHOCZKY, J. P., 1990) e Protocolo de Prioridade Teto (*Priority Ceiling Protocol*) de-

senvolvidas para escalonadores de prioridade fixa. A Política de Pilha (*Stack Resource Policy*) (BAKER, T. P., 1991) é um exemplo de política usada para o controle de inversão de prioridades para escalonadores de tarefas com prioridade dinâmica. Essas técnicas são descritas em maior detalhes em (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000).

A abordagem de garantia dinâmica (em tempo de execução) trata de cargas computacionais não previsíveis; ambientes não deterministas. As situações são tratadas em tempo de execução e os recursos necessários podem ser insuficientes para os cenários de tarefas que se apresentam (sobrecarga). Nestas abordagens, tanto o teste de escalonabilidade quanto a produção da escala são feitas em tempo de execução (escalonadores *online* dinâmicos).

Este grupo de abordagens faz o uso de testes de aceitação a cada nova tarefa que chega no sistema. O teste se baseia em análises realizadas com hipóteses de pior caso sobre os parâmetros temporais da nova tarefa e do conjunto de tarefas que já existia na fila de prontos. Se o teste indica um conjunto não escalonável, a nova tarefa é descartada. Este mecanismo a assegura recursos parar tarefas previamente garantidas como escalonáveis. Estas abordagens são próprias para aplicações com restrições críticas que operam em ambiente não determinista.

Na abordagem de melhor esforço, nenhuma garantia sobre o atendimento das restrições temporais durante a execução é fornecida em tempo de projeto. A carga envolvida é dinâmica, portanto, desconhecida em tempo de projeto. Essa carga é normalmente modelada por tarefas aperiódicas em que os tempos de chegada são desconhecidos. Tais abordagens implicam na eventual ocorrência de sobrecargas transientes, caracterizadas por recursos insuficientes em determinados instantes para as tarefas liberadas para execução.

2.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou uma breve revisão bibliográfica sobre os principais aspectos de tempo real; conceitos, problemática, modelo de tarefas, e abordagens de escalonamento de tempo real.

Devido a grande aplicabilidade de sistemas desta natureza, diversos tipos de sistemas de tempo real podem ser encontrados e várias classificações podem ser atribuídas aos mesmos levando em consideração diferentes aspectos.

O escalonamento de tempo real constitui uma das principais

áreas de pesquisa no tema de sistemas de tempo real e caracteriza um problema de grande impacto na construção de sistemas desta natureza. Baseado na necessidade de cada aplicação e na caracterização das tarefas que a compõe, as diversas técnicas e algoritmos apresentados são aplicados afim de oferecer previsibilidade ás mesmas.

3 COMPONENTES

De acordo com Szyperski em (SZYPERSKI, Clemens, 1998), um componente é uma unidade de composição com interfaces especificadas através de contrato e dependências explícitas de contexto; que pode ser implantada de maneira independente; e está sujeita a composição por terceiros. Componentes devem apresentar três características essenciais: (i) isolamento: um componente deve ser implantado de maneira independente. O componente é uma unidade atômica de implantação; (ii) Capacidade de composição: um componente deve ser capaz de se conectar a outros componentes e precisa prover uma funcionalidade como uma unidade auto-contida com interfaces bem definidas; (iii) opacidade: um componente não possui estado que possa ser observado externamente, nem o ambiente e nem outros componentes devem ser capazes de acessar sua implementação ou detalhes internos.

Na literatura de sistemas embarcados várias definições para componentes podem ser encontradas. Apesar das diferenças entre estas definições, a idéia principal do desenvolvimento de software baseado em componentes é o uso de componentes como blocos de construção de grandes sistemas. Idealmente, o desenvolvimento de componentes é independente do desenvolvimento do sistema de que faz ou fará parte: componentes devem ser reusáveis em contextos diferentes (JONSSON, B.; BRINKSMA, E.; COULSON, G., 2003). A partir deste princípio, cada abordagem concretiza seu componente conforme as características mais valorizadas por seus autores.

Este capítulo apresenta a motivação, as vantagens e desvantagens e os desafios enfrentados pelo desenvolvimento de software baseado em componentes. São apresentados também os principais conceitos usados no decorrer do documento e o modelos de componentes CCM.

3.1 INTRODUÇÃO

A capacidade de especialização de software para aplicações específicas o tornam complexo e esta complexidade implica em um longo tempo de desenvolvimento, instalação, manutenção, e atualização. Um dos grandes desafios enfrentados por desenvolvedores de sistemas software é satisfazer a demanda por este produto; aumentando, ao mesmo tempo, a velocidade de produção e a qualidade do sistema final. A chave deste problema é reusabilidade. Embora não seja nova, a idéia

de reusabilidade não tem sido implementada satisfatoriamente no desenvolvimento dos sistemas computacionais atuais(SZYPERSKI, Clemens, 1998; CRNOVIC, Ivica; LARSSON, Magnus, 2002).

A orientação à objetos se baseia na composição e interação entre diversas unidades de software, mas este paradigma falhou em estabelecer mercados significativos de componentes. De acordo com (SZYPERSKI, Clemens, 1998) a definição de objetos é puramente técnica, resumidamente: encapsulamento de estado e comportamento, polimorfismo e herança. A definição não inclui noções de independência ou composição postergada (*later composition*). E mesmo que tais condições possam ser adicionadas, sua ausência levou à situação corrente; a aplicação da tecnologia de objetos na construção de sistemas monolíticos. Os sistemas não passaram a ser mais modulares simplesmente por serem construídos por uma linguagem orientada a objetos.

A tecnologia de componentes continua sendo desenvolvida de maneira a ocupar uma lacuna deixada pela orientação a objetos. Sozinho, o paradigma de orientação a objetos não é capaz de produzir software reusável, escalável e que apresente encapsulamento adequado para o estabelecimento de um mercado de componentes.

O desenvolvimento de software baseado em componentes é visto como uma promessa de aumento de produtividade justamente por promover reúso e elementos que permitam melhor gerenciamento de complexidade. O reúso permite a eliminação de várias etapas de desenvolvimento de software por aproveitar soluções (ou partes de soluções) implementadas.

Esta propriedade garante menor tempo de desenvolvimento e melhor qualidade. Em geral componentes que são reusados já passaram pela fase de testes, são aqueles que demonstram um bom desempenho como parte de outras aplicações. Em áreas como a aviação ou espaço, por exemplo, em que subsistemas de software são submetidos a processos de certificação que chegam três vezes o tempo requerido para desenvolvimento, o potencial promovido pelo desenvolvimento baseado em componentes torna-se ainda mais atrativo.

A abordagem baseada em componentes não apresenta apenas vantagens. O desenvolvimento de unidades de software reusáveis exige etapas de projeto e implementação mais complexas que o software comum. O tempo e esforço requerido para assegurar a reusabilidade do software é um dos fatores que desencorajam a construção baseada em componentes(CRNOVIC, Ivica; LARSSON, Magnus, 2002).

A análise de requisitos de componentes de software é dificultada pelo fato não ser conhecido em tempo de projeto, o contexto (ou con-

textos) em que o componente será integrado. A funcionalidade do componente e os principais requisitos podem ser determinados em tempo de projeto, mas as escolhas tomadas durante o desenvolvimento do mesmo podem, em muitos casos, determinar o quanto este componente pode ser reusado. Além disso, a análise de requisitos de componentes de software lidam também com requisitos integração de um componente em uma aplicação.

A propriedade de reusabilidade exige que o componente seja suficientemente genérico, escalável e adaptável, o que também implica em complexidade (de construção e de uso) e maior demanda por recursos. Embora o custo da manutenção de aplicações possa se tornar mais baixo pela adoção de componentes, o custo de manutenção de componentes em si pode ser bastante elevado. O componente deve ser capaz de responder a diferentes requisitos de diferentes aplicações em ambientes diversos e com diferentes graus de confiabilidade e suporte de manutenção. Componentes de software devem prover uma funcionalidade específica ao mesmo tempo que ofereça flexibilidade suficiente para ser aplicada a vários contextos.

Componentes e aplicações possuem ciclos de vida distintos, por isso, alterações introduzidas na aplicação sempre envolvem riscos de modificações que não são suportadas pelos componentes que a compõem.

Apesar das desvantagens apresentadas, o desenvolvimento baseado em componentes ainda está no início de sua fase de expansão e traz a promessa de uma abordagem bastante poderosa. Alguns autores cogitam a possibilidade de um novo papel no desenvolvimento de software; integradores que não sejam necessariamente programadores, mas que sejam capazes de montar uma aplicação a partir de componentes existentes. A partir da montagem (assembly) da aplicação, estes componentes passariam a ser atualizados remotamente, por seus respectivos desenvolvedores.

Os objetivos mais comumente perseguidos são: simplificar o desenvolvimento de sistemas através da separação da parte funcional e da parte não funcional do software; agilizar a construção de sistemas a partir de unidades de software; e a reusabilidade. No estado atual, a abordagem de desenvolvimento baseada em componentes enfrenta os seguintes desafios (CRNOVIC, Ivica; LARSSON, Magnus, 2002):

- Ainda não há consenso a respeito da especificação de componentes. Esta é uma questão importante uma vez que os conceitos básicos de desenvolvimento baseado em componentes são bastante dependentes da definição do que é um componente e como ele

deve ser especificado.

- Os modelos de componentes existentes apresentam características ambíguas, são incompletos e por isso, difíceis de usar. A relação entre arquitetura do sistema e o modelo de componente não é precisamente definida.

- O ciclo de vida do software baseado em componente tende a se tornar cada vez mais complexo porque várias fases de desenvolvimento são separadas em atividades não sincronizadas (ex: desenvolvimento de componentes pode ser independente do desenvolvimento de sistemas que usam estes componentes). O processo de engenharia de requisitos é ainda mais complexo: raramente um componente possui todas as propriedades requeridas pelo sistema, e mesmo quando isso acontece, não há garantias de que este componente funcionará de maneira ótima quando em conjunto com os demais componentes do sistema. Este tipo de problema requer outra abordagem de engenharia de requisitos, uma análise que determine se os requisitos do sistema podem ser satisfeitos através dos componentes disponíveis e quando isso não for possível a adequação destes requisitos. Devido às incertezas quanto ao processo de seleção do componente, uma estratégia é necessária para gerenciar os riscos do processo de seleção e evolução do componente.

- Mesmo que se possa especificar atributos de um componente, não há como prever as conseqüências destas escolhas em relação aos atributos do sistema resultante da composição. A relação entre os atributos de cada componente e da aplicação resultante da composição são ainda objeto de pesquisa.

- A confiabilidade de um componente é determinada conforme testes realizados pelo fornecedor. Embora a certificação seja um procedimento padrão em vários domínios, ainda não foi estabelecida para software em geral, especialmente para componentes de software. Devido à tendência de entrega de componentes em formato binário, questões relacionadas à confiabilidade de componentes tornam-se relevantes.

- A construção de sistemas complexos pode incluir vários componentes que por sua vez envolvem outros componentes. Neste cenário a configuração estrutural do sistema passa a exigir maior atenção. Aspectos a serem tratados envolvem, por exemplo, a

inclusão de um mesmo componente em duas composições do sistema.

- Como o propósito da engenharia de software é prover soluções práticas para problemas práticos, é essencial que existam ferramentas apropriadas para o sucesso do desempenho da engenharia de software baseada em componentes.

- O uso de componentes de software em domínios críticos (como sistemas de tempo real), em que requisitos de confiabilidade são rigorosos, é dificultado pela limitação do desenvolvimento baseado em componentes em garantir atributos específicos da aplicação como um todo.

3.2 CONCEITOS BÁSICOS

O desenvolvimento baseado em componentes é visto por alguns autores como uma extensão do desenvolvimento baseado em objetos. Os componentes são tratados como unidades reusáveis de implantação e composição. Entretanto, fatores como granularidade, os conceitos de composição e implantação e até mesmo o desenvolvimento de processos distinguem claramente componentes e objetos (CRNOVIC, Ivica; LARSSON, Magnus, 2002).

Para que um componente seja implantado de forma independente, é essencial que exista uma separação entre o componente e o ambiente (e de outros componentes). Um componente deve ter interfaces claramente especificadas e sua implementação deve ser encapsulada no componente não sendo acessível de forma direta pelo ambiente. Isto é o que torna um componente uma unidade implantável por terceiros(CRNOVIC, Ivica; LARSSON, Magnus, 2002).

De maneira geral, poucas variações, conforme a tecnologia ou a abordagem seguida pelo desenvolvimento são encontradas nas etapas do ciclo de vida das aplicações baseadas em componentes. De acordo com o Modelo de Componentes CORBA, por exemplo, estas etapas são: implementação, empacotamento, montagem (ou integração) e implantação.

A etapa de implementação de componentes compreende o projeto de implementação em que são definidas propriedades que especificam o que cada componente faz (funcionalidade) e como colaboram entre si e com seus clientes. A forma de interação entre componentes, clientes e serviços também é definida durante o projeto. A partir das especifica-

ções criadas no projeto, o programador cria o componente. Durante a implementação, descritores que informam o tipo de suporte necessário em tempo de execução e quais as interações a serem realizadas por este componente são elaborados.

Na etapa de empacotamento, componentes, descritores e demais arquivos necessários para configuração do mesmo são reunidos num único arquivo. Na etapa de montagem, aplicações são configuradas através da escolha (de acordo com sua funcionalidade e propriedades) das implementações de componentes necessários, restrições de instâncias de componentes são especificadas e as conexões entre instâncias de componentes são realizadas através de meta-dados.

Durante a implantação, os recursos necessários em tempo de execução (conforme indicados no descritor de implantação) são analisados. Estes recursos são preparados e os componentes são implantados para início de execução.

Para (CRNOVIC, Ivica; LARSSON, Magnus, 2002), a integração e implantação de componentes deve ser independente do ciclo de vida de desenvolvimento do componente, as atualizações de um componente não devem exigir a recompilação e montagem da aplicação. Outra propriedade importante é a característica de visibilidade, um componente deve ser visível somente através de suas interfaces, o que exige a especificação completa do componente incluindo interface funcional, características não funcionais, casos de uso, testes e demais informações.

Para que seja possível compor aplicações usando diferentes componentes, mesmo considerando os diferentes conceitos e visões que se pode ter dos mesmos, é preciso que o componente apresente (1) um conjunto de interfaces providas ou requeridas do ambiente. Estas interfaces objetivam a interação do componente com outros componentes e com a infraestrutura subjacente; e (2) um código executável que possa ser acoplado ao código de outros componentes através de interfaces.

Alguns problemas são inevitáveis no desenvolvimento baseado em componentes. Uma das dificuldades típicas é decidir como lidar com aspectos não funcionais que devem fazer parte da arquitetura do componente. Assim como os próprios componentes, estes aspectos devem ser possíveis de compor e fáceis de controlar. A separação clara de requisitos não funcionais faz com que componentes sejam mais independentes do contexto e aumenta a possibilidade de reúso dos mesmo em vários contextos.

3.2.1 Componentes e Objetos

Componentes e objetos são vistos muitas vezes como sinônimos ou elementos semelhantes. Um componente pode ser visto como uma coleção de objetos que cooperam entre si e são fortemente entrelaçados. O limite entre um componente e outros componentes ou objetos é explicitamente especificado e a interação entre eles é realizada através de interfaces. A estrutura interna do componente, por outro lado, é oculta. Os objetos dentro de um componente têm acesso à implementação uns dos outros mas o acesso à implementação de qualquer destes objetos a partir de um elemento externo ao componente é evitado.

Similaridades e diferenças entre componentes e objetos são bastante discutidas. Uma classe empacotada com as interfaces que oferece e as interfaces que requer definidas explicitamente, pode ser considerada um componente. Em geral, componentes possuem um conjunto de mecanismos de comunicação mais complexos que objetos (geralmente usam mecanismos de mensagem). Mas ao invés de conter apenas classes e objetos, um componente pode ser composto por procedimentos tradicionais, variáveis globais ou até mesmo a combinação de diferentes paradigmas de programação.

A tecnologia de objetos, quando utilizada cuidadosamente, é provavelmente um dos melhores meios de concretizar a tecnologia de componentes. Em particular, as vantagens de modelagem da tecnologia de objetos são certamente de valor quando se constrói um componente (SZIPERSKI, C., 1999).

3.2.2 Interfaces

Uma interface de um componente pode ser definida como uma especificação do ponto de acesso ao componente (SZIPERSKI, C., 1999). É através deste ponto de acesso que os clientes podem acessar os serviços providos pelo componente. Em geral, um componente oferece vários serviços, e portanto, várias interfaces.

Uma interface não oferece a implementação de qualquer operação do componente. As interfaces simplesmente nomeiam uma coleção de operações e provêem somente descrições e protocolos envolvidos nestas operações. Esta separação possibilita a (1) substituição da implementação sem modificar a interface, e desta forma, melhorar o desempenho do sistema sem que seja necessário reconstruí-lo; e a (2) adição de novas interfaces (e implementações) sem modificar implementações correntes

melhorando a adaptabilidade do sistema (CRNOVIC, Ivica; LARSSON, Magnus, 2002).

Idealmente, as interfaces seriam a única parte visível do componente. Por isso, cada cliente configura o componente através destas interfaces. Como conseqüência, é importante que em uma interface a semântica de cada operação seja bem especificada pois é usada pelo implementador das operações e pelos clientes que usam a interface.

Atualmente, nos modelos de componentes existentes as interfaces definem somente o tipo dos dados de entrada e saída e fornecem poucas informações sobre o que o componente faz. As interfaces definidas em tecnologias de componentes convencionais podem expressar propriedades funcionais, que incluem a descrição de operações providas (assinatura do componente) e o comportamento do componente. Alguns autores (BERGNER, K.; RAUSH, A.; SIHLING, M., 1999) alegam que a maioria das técnicas de descrição de interfaces como IDL (*Interface Definition Language*) se concentram apenas em descrever operações, sendo insuficientes para expressar propriedades não-funcionais como atributos de qualidade como disponibilidade, segurança, latência ou exatidão (*accuracy*) (BACHMANN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K., 2000).

As interfaces descrevem serviços providos e serviços requeridos pelo componente. Entretanto, para que estes serviços sejam efetivamente realizados, aspectos relacionados à dependência de contexto (especificação do ambiente de implantação e ambiente de execução) e interação exigem contratos que especifiquem claramente o comportamento de um componente.

3.2.3 Contratos

A maior parte das técnicas de descrição de interfaces (por exemplo a IDL) se refere somente às assinaturas das operações de um componente. Os contratos são uma tentativa de especificar o comportamento do componente. O contrato lista as restrições globais que o componente deve manter (invariantes) (MEYER, B., 1992). Para cada operação dentro do componente, um contrato lista as pré-condições a serem satisfeitas pelo cliente e as pós-condições prometidas pelo componente em retorno. As pré-condições descrevem as características que o ambiente deve prover para que as operações do contrato possam garantir as pós-condições. Este conjunto formado por pré-condições, invariantes e pós-condições constituem o comportamento do componente.

Os contratos podem ser usados também para especificar as interações entre grupos de componentes. Estes contratos são empregados de maneira diferente e são definidos em termos:

- do número de componentes participantes;
- do papel de cada componente de acordo com suas obrigações contratuais (interfaces ou variáveis para o qual deve oferecer suporte) e obrigações causais (seqüências ordenadas de ações, incluindo envio de mensagens para outros componentes);
- dos invariantes a serem mantidos pelos componentes;
- da especificação de métodos que instanciam o contrato.

O uso de contratos na especificação do comportamento de componentes que interagem entre si promove reúso e refinamento de componentes de software de granularidade mais grossa; baseados em comportamento. Os contratos permitem que o desenvolvedor de software isole e especifique explícitamente (em alto nível de abstração) os papéis de diferentes componentes em um contexto particular. Diferentes contratos tornam possível modificar e extender o papel de todo participante de maneira independente. Finalmente, novos contratos podem ser definidos fazendo a associação de participantes diferentes com papéis diferentes.

O comportamento do componente no contexto de uma aplicação pode ser bastante complexo devido aos vários papéis que este componente pode desempenhar conforme sua relação com outros componentes (ou objetos), participando em vários contratos.

3.2.4 Padrões, *Framework* e Componentes

De acordo com (ALEXANDER, C., 1979), um padrão de projeto (*design pattern*) define uma solução recorrente para um problema recorrente. Em (GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John, 1995) este conceito foi refinado pela especificação de características de padrões de projeto e seus objetivos. Os padrões capturam soluções não óbvias que não são somente princípios e estratégias abstratas. Assim como várias técnicas de solução de problemas (tal como paradigmas ou métodos de projeto de software), padrões derivam soluções a partir de princípios. Os padrões descrevem a relação entre as estruturas do sistema e seus mecanismos, não somente entre módulos

independentes. Um padrão de projeto pode ser empregado no projeto e documentação de um componente. E um componente, como entidade reusável pode ser visto como uma implementação de um padrão de projeto.

A engenharia de software baseada em componentes é a construção de software através da montagem de peças (CRNOVIC, Ivica; LARSSON, Magnus, 2002). Neste ambiente é essencial que exista um contexto dentro do qual estas peças sejam usadas. Na literatura de componentes, *frameworks* são um meio de prover tais contextos. Uma das formas de usar o termo *framework* está centrada na idéia de que o esforço de projeto durante a construção de sistemas computacionais pode ter resultados abstratos que podem ser reusados em outras situações. Um *framework* é então usado para descrever algo que não é usado somente na situação corrente especificamente, mas depois de algumas modificações, é usado em situações similares. As modificações podem ser realizadas durante a execução ou durante o tempo de projeto e é chamado de instanciação do *framework*.

Frameworks e padrões de projeto definem um grupo de participantes e as relações entre eles que pode ser reusada em situações semelhantes. A solução capturada por frameworks e padrões de projeto pode ser encapsulada por um componente de software.

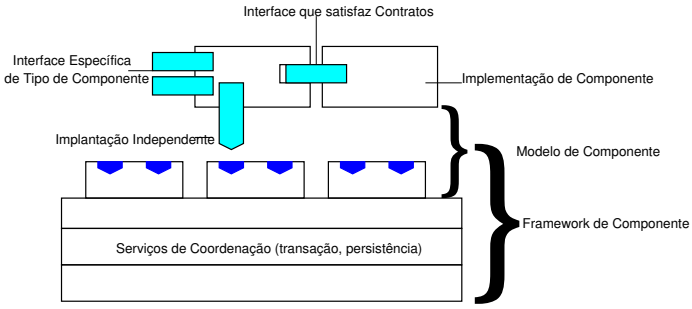


Figura 2: O conceito de Sistema baseado em componente.

A visão de Bachman et al. (BACHMANN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K., 2000) (Figura 2) relaciona alguns dos conceitos previamente mencionados. O componente implementa uma ou mais interfaces que satisfazem certas obrigações especificadas pelo contrato. Um sistema baseado

em componentes é baseado em componentes de tipos específicos, cada um destes desempenha um papel no sistema; um papel descrito por uma interface. Um modelo de componente é um conjunto de: tipos de componentes, suas interfaces e adicionalmente uma especificação dos padrões permitidos para interação entre os tipos de componentes. Um *framework* de componente provê os serviços de implantação e execução que suportam o modelo do componente.

3.3 ESPECIFICAÇÕES

Idealmente, somente através das interfaces seria possível saber a funcionalidade e dependências de contexto do componente, configurá-lo e implantá-lo. Por isso, de acordo com (CRNOVIC, Ivica; LARSSON, Magnus, 2002), a especificação de um componente é a especificação de suas interfaces.

Entretanto, para usar componentes com sucesso, são necessárias informações que vão além das informações sobre sua funcionalidade. Tais informações, incluem propriedades estruturais que definam como um componente pode ser composto com outros componentes; propriedades não-funcionais, tal como desempenho, capacidade e propriedades de ambiente assumidas; e propriedades que especifiquem a relação entre componentes similares ou relacionados. Como o desenvolvedor do componente jamais antecipará todos os aspectos do componente em que o usuário do mesmo estará interessado, é impossível alcançar especificações completas. Além disso, desenvolvedores descobrem novos tipos de dependências no decorrer de suas tentativas de usar, em conjunto, componentes desenvolvidos de maneira independente. Por isso, especificações de componentes devem ser extensíveis (CRNOVIC, Ivica; LARSSON, Magnus, 2002).

O uso de credenciais é um exemplo de especificação de componentes cujas propriedades não-funcionais podem ser estendidas. Uma credencial é uma tupla <Atributo, Valor, Credibilidade>. O Atributo é a descrição da propriedade do componente. O Valor é uma medida para esta propriedade e Credibilidade é a descrição do meio pelo qual a medida foi obtida. Esta técnica de especificação deve incluir um conjunto de atributos registrados com notações para especificar seu valor, credibilidade e meios para adição de novos atributos. Uma técnica poderia especificar alguns atributos como necessários e outros como opcionais. Este conceito é parcialmente implementado na linguagem de descrição de arquitetura UniCon (SHAW, Mary; DELINE, Robert; KLEIN,

Daniel V.; ROSS, Theodore L.; YOUNG, David M.; ZELESNIK, Gregory, 1995) que permite uma lista extensível de pares de atributo e valor para serem associados aos componentes. Outros exemplos são citados em (CRNOVIC, Ivica; LARSSON, Magnus, 2002).

3.4 MODELOS E TECNOLOGIAS DE COMPONENTES

A tecnologia de desenvolvimento baseado em componentes se baseia na dependência do *framework* de componentes. O *framework* de componentes torna possível a provisão de novos serviços de forma transparente e a remoção de informações do código fonte. Estes dois objetivos só são possíveis através da melhoria da introspecção do código fonte ou pela definição de novos formalismos. Portanto, a abordagem de componentes é usada para (1) definir componentes e suas interações com outros componentes de maneira explícita (para realizar verificações de conexão estática e automatizar o controle de conexão); (2) definir suas implementações explicitamente a partir de código nativo e interações com outros códigos nativos (para instanciar dinamicamente e controlar ciclos de vida de instâncias); e (3) definir propriedades e serviços não-funcionais de componentes de forma explícita (delegando sua realização para o *framework* de componente) (CRNOVIC, Ivica; LARSSON, Magnus, 2002).

De fato estes três objetivos foram tratados simultaneamente por diferentes desenvolvedores e resultaram em diferentes tecnologias (CRNOVIC, Ivica; LARSSON, Magnus, 2002). Um modelo de componente especifica, de maneira abstrata, os padrões e convenções impostos aos engenheiros de software que desenvolvem e usam componentes. A compatibilidade com um modelo de componentes é uma das propriedades que difere componentes de outras entidades de software.

3.5 O MODELO DE COMPONENTES CORBA

O modelo de componentes CORBA, CCM - *CORBA Component Model* é um dos modelos de componentes mais recentes e completos (CRNOVIC, Ivica; LARSSON, Magnus, 2002). Este modelo é fundamentado na experiência acumulada usando serviços CORBA, JavaBeans e EJB.

Desde 1989, o Object Management Group (OMG) tem padronizado uma especificação de *middleware* aberto para oferecer suporte às aplicações distribuídas. O CORBA permite que aplicações possam in-

vocar operações em objetos distribuídos com transparência de localização, linguagem de programação, plataforma, protocolo de comunicação, interconexões e hardware. A OMG também especifica um conjunto de Serviços de Objetos CORBA que define interfaces padronizadas para acessar serviços de distribuição, tais como serviço de nomes, negociação (*trading*) e notificação de eventos. Através do CORBA e seus serviços de objetos, desenvolvedores de sistemas podem integrar e montar aplicações e sistemas amplos e complexos usando propriedades e serviços de diferentes fornecedores (WANG, Nanbor; SCHMIDT, Douglas C.; O'RYAN, Carlos, 2001).

O objetivo principal por trás da especificação CCM é prover uma solução para a complexidade alcançada pelo CORBA e seus serviços: "A flexibilidade do CORBA oferece ao desenvolvedor um grande número de escolhas, e requer um vasto número de detalhes para ser especificado. A complexidade é simplesmente muito grande para que seja possível fazê-lo de forma rápida e eficiente.". Em (WANG, Nanbor; SCHMIDT, Douglas C.; O'RYAN, Carlos, 2001) são detalhadas uma série de limitações do modelo de objetos CORBA atacadas durante o desenvolvimento da especificação do CCM.

A primeira especificação completa do CCM foi disponibilizada somente em 2002 como parte do padrão CORBA 3.0. A especificação CCM demonstra um esforço de integração de várias aspectos da engenharia de software, resultando na descrição de aplicações sob diferentes formalismos, conforme a dimensão do tempo (o ciclo de vida, do projeto à implantação) e a dimensão abstrata (de abstrações até a implementação). Juntos, estes formalismos resultam em uma especificação muitas vezes complexa.

As aplicações baseadas em CCM objetivam a construção de aplicações a partir de componentes distribuídos heterogêneos. Os componentes desta aplicação podem ser desenvolvidos usando linguagens de programação diferentes, destinados para diferentes sistemas operacionais, e ser implantados em servidores diversos de um sistema distribuído.

A especificação CCM se divide em (MARVIE, Raphael; MERLE, Philippe, 2001):

- Modelo Abstrato: Especifica os meios pelos quais o tipo do componente é definido, bem como a definição de interfaces providas e usadas pelo mesmo. Esta definição é feita em IDL3, uma extensão da IDL padrão para definição de componentes;

- Modelo de Programação: Introduz o *Framework* de Implementa-

ção de Componentes (CIF). Este *framework* é usado para descrever a interação entre aspectos funcionais e serviços providos pela plataforma. A CIDL, linguagem de definição de implementação de componentes é a linguagem usada para especificar a relação entre a parte funcional e a parte não funcional de componentes CCM;

- Modelo de empacotamento: Define a forma com que tipos e implementações são empacotados para posterior distribuição. Um componente é distribuído sob a forma de um arquivo compactado composto por várias partes da definição do componente e um descritor XML especificando como a infraestrutura do CCM deve criar e gerenciar o componente. Este modelo permite o empacotamento de componentes individuais ou configurações inteiras de componentes;

- Modelo de implantação: define o processo de implantação de componentes em um ambiente distribuído. A partir do pacote de componentes, é determinado a instalação; criação de instâncias específicas de componentes e a interconexão entre elas a fim de formar aplicações completas;

- Modelo de execução: Baseado na idéia do container, este modelo especifica o ambiente de execução para instâncias de componentes, Graças ao container, do ponto de vista do componente, o uso de serviços de objeto CORBA é transparente;

- Meta-modelo: Define os conceitos utilizados nos modelos descritos através da UML. Em particular, o meta-modelo define as construções de linguagem presentes em IDL3 e CIDL e permite a geração automática de repositórios de tipos de componentes.

3.5.1 O Componente CCM

Os componentes CORBA descreve o uso de portas de componentes como uma forma de oferecer diferentes visões de um mesmo componentes conforme o cliente que o utiliza. No exemplo descrito em (MARVIE, Raphael; MERLE, Philippe, 2001), um componente que implementa uma máquina de bebidas apresenta diferentes conjuntos de funções conforme o cliente que o utiliza. O fornecedor de bebidas utiliza somente as portas relacionadas a provisão de estoque de bebidas

da máquina. Um comprador, por outro lado, teria acesso a portas que o permitissem a escolha e a compra do produto.

O CCM estende o modelo de objetos CORBA adicionando novos tipos, ferramentas e mecanismos. Esta extensão se reflete na IDL3, uma extensão da IDL2 (da especificação CORBA 2.X) especificado pelo modelo abstrato. A IDL3 inclui palavras chave que permitem a definição de componentes. O mapeamento das definições da IDL3 em IDL2 (também chamada de interface equivalente) torna possível que clientes, componentes ou não, possam utilizar os serviços de um componente (Figura 3).

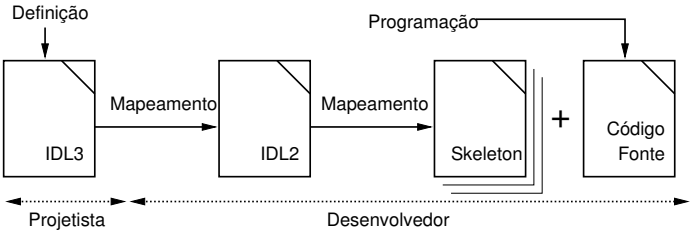


Figura 3: Mapeamento IDL3 em IDL2.

As interações de um componentes são definidas pela exposição de um conjunto de portas (Figura 4) que se dividem em:

- Facetas: Interfaces nomeadas providas por componentes para interação com clientes. Elas permitem que componentes exponham diferentes visões para seus clientes. Cada faceta engloba um conjunto de funcionalidades voltado para um cliente em particular;
- Receptáculos: Pontos de conexão que descrevem a habilidade do componente em usar uma referência provida por um agente externo. Os receptáculos especificam o que componentes podem ou precisam usar;
- Fontes de Eventos: Pontos de conexão em que eventos de um tipo específico são disponibilizados;
- Consumidores de Eventos: pontos de conexão em que eventos de um tipo específico são processados;
- Atributos: Valores expostos através de métodos específicos que são usados inicialmente para configuração do componente.

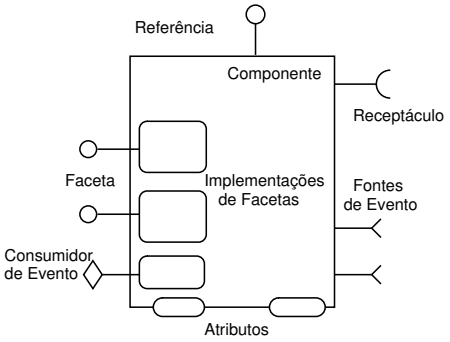


Figura 4: Portas de Componente CCM.

No CCM, as portas são variáveis nomeadas e com tipo definido, portanto facetas diferentes de um mesmo componente podem ter o mesmo tipo. Um componente agrupa definições de atributos e portas. Em geral as portas de componentes são projetadas para conexão entre componentes, mas podem também ser usadas em tempo de implantação. Os componentes CCM possuem tipo definido e podem estabelecer relação de herança com outros componentes, o que significa que o componente herda as interfaces do outro componente. Os componentes CCM também podem ser definidos através de herança, entretanto é suportada somente herança simples, de um único componente. Assim ocorre com objetos do CORBA padrão, instâncias de componentes são caracterizadas por suas interfaces e por uma referência, chamada de referência base no contexto CCM. Esta é a referência que permite aos clientes acessar portas de instâncias de componentes.

O CCM define quatro tipos de componentes: *service*, *entity*, *session* e *process*. Os componentes CCM podem ser dos seguintes tipos:

- *Service*: Os componentes deste tipo não armazenam estado ou identidade, o tempo de vida deste tipo de componente é confinado à duração de uma única invocação (uma única interação com um cliente);

- *Session*: Os componentes do tipo *session* são definidos como componentes que possuem estado transiente e identidade não persistente. O tempo de vida deste tipo de componente é igual ao tempo de duração de uma seqüência de invocações de um único cliente.

- *Process*: Os componentes do tipo *process* possuem estado persistente. A identidade deste tipo de componente é gerenciada pelo componente (a critério do desenvolvedor). Componentes deste tipo geralmente representam processos de negócios (ex: carrinho de compras em comércio eletrônico);

- *Entity*: Os componentes do tipo *entity* são similares aos componentes *process*, porém, os clientes destes componentes têm conhecimento explícito da persistência do componente. Estes componentes são usados para modelar entidades de negócios (ex: clientes de uma empresa, contas bancárias).

Além destas, outras propriedades de projeto são consideradas na especificação de cada categoria de componente. Algumas destas propriedades são: a definição de interfaces externas e internas, interfaces de *callback*, modelo de uso do CORBA, padrão de projeto do cliente, tipos de APIs externas, gerenciamento de persistência, política de tempo de vida do *servant*, o uso ou não de transações, tipos de eventos e especificação de executores. A nomeação de cada um destes elementos do conjunto de propriedades possibilita a configuração automática do ambiente de execução do componente. Estas definições são especificadas na declaração do tipo do componente e na documentação do mesmo, gerada pelo compilador da IDL e posteriormente inclusas no pacote a ser implantado.

As interfaces externas são providas por componentes para receber invocações de clientes através do container que o abriga. As interfaces internas são usadas para interação entre componente e container. Estas interfaces provêem acesso aos serviços e contexto gerenciados pelo container, tais como gerenciamento de ciclo de vida e operações de transação. As interfaces de *callback*, por sua vez, permitem ao container informar às instâncias de componentes sobre eventos importantes tais como quando uma instância de componente está disponível para algum serviço. Estas interfaces são ilustradas na Figura 5.

Para cada tipo de componente CCM existe um *home* associado. O *home* de componente CCM é responsável por atribuir chaves primárias e instanciar componentes. Além disso, o *home* gerencia o ciclo de vida do componente, contendo o método de criação, localização e destruição do componente. Assim como ocorre com componentes, *homes* são criados a partir da herança de tipos básicos, com atributos e operações "clássicas"de *homes*. Em tempo de execução, uma instância de componente só pode ser gerenciada por uma única instância de *home*. Embora possam ser gerados de forma automática, *homes* po-

dem também ser projetados com operações específicas definidas pelo projetista.

Tanto componentes como *homes* possuem atributos que são parâmetros nomeados projetados para configuração. Os mecanismos de implantação usam descritores de propriedades XML para configurar o valor inicial destes atributos em componentes e *homes*. A padronização de propriedades e interfaces para componentes, portas, atributos e *homes*, permite que clientes de componentes e desenvolvedores sejam capazes de definir quais propriedades são disponíveis e como acessá-las. O padrão CCM também define um conjunto de portas de operações genéricas que permite à ferramentas padronizadas verificar as funcionalidades do componente, configurar atributos, obter interfaces de portas e fazer a conexão em portas. Estas portas padronizadas provêem a flexibilidade necessária para composição e implantação de componentes em aplicações em execução de forma automatizada.

3.5.2 *Framework* de Tempo de Execução

O CCM enfatiza a propriedade de oferecer diversos serviços aos componentes sem exigir a modificação do código do mesmo. Esta abordagem aumenta a reusabilidade, reduz a complexidade e melhora significativamente a manutenção da aplicação final. Os componentes CORBA usam um container para implementar o acesso de clientes aos serviços do sistema usando padrões de projetos resultantes da experiência na construção de aplicações usando a tecnologia de objetos e os serviços CORBA. Os containers CCM são definidos em termos de como usam a infraestrutura CORBA subjacente e como conseqüência tornam-se capazes de gerenciar serviços como transação, segurança, eventos, persistência, serviços de ciclo de vida, entre outros. O CCM define tipos de componentes que podem lidar diretamente com serviços do CORBA, porém, enfatiza que o container gerencia estes serviços automaticamente; o código do componente simplesmente ignora os serviços associados. Para que isto seja possível, o container intercepta a comunicação entre componentes e clientes (Figura 5).

Na especificação CCM, os containers são definidos no modelo de programação. O objetivo deste modelo é descrever aspectos não funcionais para que esta parte da aplicação seja automaticamente gerada, deixando para o desenvolvedor somente a escrita do código contendo a lógica da aplicação. O mecanismo usado pelo container pode ser visto como um *framework* para integrar os serviços de objetos ao compor-

tamento funcional de componentes, tendo como base os descritores de componentes especificados durante o empacotamento.

Fazer com que implementações de componentes sejam obrigadas a configurar, através de programação, todos os mecanismos que constituem seu ambiente de execução, tais como ORB e POA (*Portable Object Adapter*), limita a reusabilidade da implementação destes componentes uma vez que componentes podem, muitas vezes, exigir configurações de ORB e POA conflitantes. Para separar os objetivos da implementação do servidor da aplicação, configuração de ORB e POA e gerenciamento de implementações de objetos em tais servidores, o CCM define dois mecanismos padrões: Servidor de componentes e container (WANG, Nanbor; GILL, Christopher, 2004).

O servidor de componentes é um servidor de implementações genérico que o *framework* CCM usa para hospedar componentes. Para prover um ambiente de execução apropriado para os componentes hospedados, um servidor de componente é responsável por configurar o ORB de forma que satisfaça os requisitos destes componentes. De forma análoga, containers são mecanismos CCM para configuração e gerenciamento de POAs. Juntos, servidores de componentes e containers provêem suporte de execução para implementações de componentes, e desta forma, separam os objetivos de programação e configuração de servidore; ORBs e POAs dos objetivos de desenvolvimento de componentes e aplicações (WANG, Nanbor; GILL, Christopher, 2004).

Uma aplicação CORBA convencional, um processo servidor contém um ORB, que cria e gerencia um POAs. Os *servants*, implementações de objetos, podem então ser ativados através deste POA. Assim como ocorre com servidores CORBA, um servidor de componente CCM contém um ORB e diversos containers, e cada um deles contém um POA. As implementações de componentes são instaladas e ativadas dentro de containers. Cada container provê limites de encapsulamento explícito entre cada conjunto de *servants* que implementa um componente e os serviços dos quais ele depende (WANG, Nanbor; GILL, Christopher, 2004).

Desta forma, os desenvolvedores deixam de ser responsáveis por criar e configurar ORBs ou POAs. Pelo mesmo motivo, uma implementação de componente não pode interagir diretamente com interfaces de objetos internos ao POA e ORB. Entretanto, para instalar e executar os componentes e homes de componentes dinamicamente em um servidor de processo genérico e padrão, servidores e componentes e homes instalados devem saber como interagir uns com os outros. Portanto, o CCM define interfaces de contâiner no modelo de programação do contâiner

para padronizar as interações entre implementações de componentes e seus contâiners (WANG, Nanbor; GILL, Christopher, 2004).

Toda a interação de uma implementação de um componente é manipulada por seu contâiner, incluindo interações com seus clientes e com o ambiente de execução que o contém. Tais interações são gerenciadas através de interfaces locais predefinidas no modelo de programação do contâiner.

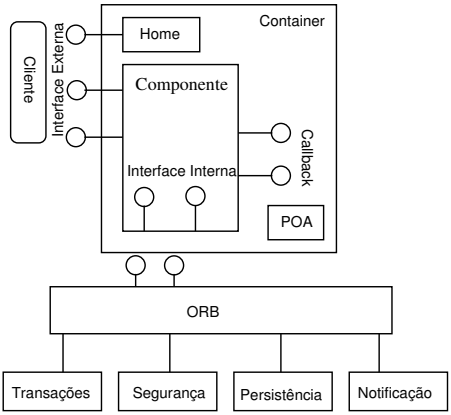


Figura 5: Container CCM.

O contâiner é um mecanismo fundamental para promover encapsulamento de componentes e o acesso transparente a serviços de objetos. O contâiner representa o ambiente de execução para uma instância de componente e provê os recursos de baixo nível como memória e processador. Além da instância de componente, o contâiner também contém uma instância do POA, especializada para o tipo de componente hospedado (e especificado no descritor do componente). A ativação, gerenciamento e uso dos serviços do POA também é delegado ao contâiner, que define ainda:

- Interfaces externas para que clientes possam utilizar o componente hospedado;
- Interfaces locais para acesso a serviços do sistema por parte do componente hospedado;
- *Callbacks* para gerenciamento de instâncias de componentes;

- Funcionalidade para a navegação entre as interfaces do componente hospedado (através do *home*);

- Conexão das facetas e receptáculos de maneira que o componente funcione como desejado;

- Canais de eventos para transmitir e receber eventos.

Tanto o acesso a qualquer uma das propriedades de um componente expostas por um contâiner, quanto o acesso do componente aos serviços de objetos CORBA são realizado através do ORB (Figura 5) (COSTA, Fabio; KON, Fabio, 2002).

O modelo de programação do contâiner inclui ainda, além dos tipos de componentes: tipos de APIs externas, tipos de API de contâiner e modelo de uso do CORBA. Tais elementos, são na realidade, nomes de relações estabelecidas entre os vários elementos que compõem o CCM. A nomeação destes vários elementos especifica uma configuração que permitirá a automação do processo de implantação do componente.

Os tipos de APIs externas (interfaces *home* e interfaces de aplicação) especificam um contrato entre o desenvolvedor do componente e os clientes do componente. São definidas em IDL e armazenadas no Repositório de interfaces para uso dos clientes.

Os tipos de API de contâiner (*Session* e *Entity*) especificam um contrato entre um componente e o contâiner. Estes dois tipos básicos definem conjuntos de API comuns e um conjunto de tipos derivados que provêem funções adicionais. A API de contâiner do tipo *session* define um *framework* para componentes que usam referências de objetos transientes. A API de contâiner do tipo *entity* define um *framework* para componentes usando referências de objetos persistentes. O modelo de uso do CORBA especifica o padrão de iteração requerido entre o contâiner, o POA e os serviços CORBA.

Para integração da parte não funcional e lógica da aplicação, o CCM provê um framework de implementação de componente (CIF), que descreve a interação entre estas partes. Para a geração de *skeletons* (estendidos em comparação com *skeletons* CORBA 2) o CIF utiliza a linguagem de definição de implementação de componente (CIDL). Os *skeletons* automatizam o gerenciamento de comportamento básico de componentes, como gerenciamento de portas, navegação entre interfaces e ciclo de vida de componentes. Além da CIDL, o CIF conta também com o *framework* definido para o POA que esconde grande parte de sua complexidade.

3.5.2.1 Estruturação do Componente

A CIDL é usada para descrever a estrutura da implementação do componente bem como seu estado de persistência. Neste caso, o componente é considerado como um conjunto de elementos comportamentais providos por executores. Dois tipos de executores são definidos: executores de componentes (que definem tipos de componentes) e executores de *homes* (que definem *homes* de componentes). A implementação de um componente é formada pela agregação de elementos com comportamento e relação específicos. A CIDL define esta agregação em uma composição (*composition*) (Figura 6). A composição é um meta-tipo que corresponde a uma definição da implementação. Nela são especificados o tipo do *home* e implicitamente, o tipo do componente. Uma definição de executor é então associada ao tipo do *home*. Esta definição especifica a relação entre o executor do *home* e outros elementos da composição. E finalmente, a composição especifica também a definição do executor.

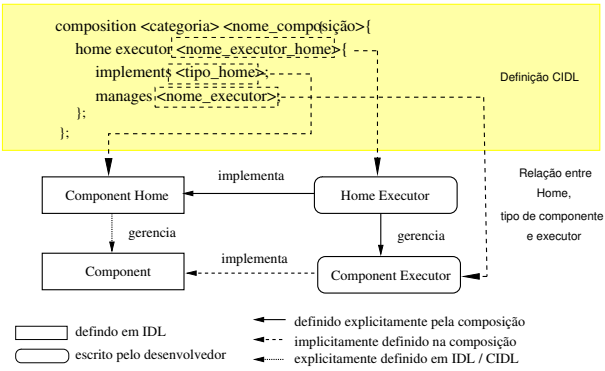


Figura 6: Definição de Composição.

As partes essencias da definição de uma composição são:

- Nome da composição;
- Categoria do ciclo de vida da implementação do componente;
- Tipo de *home* sendo implementado e implicitamente o tipo de componente (o tipo do *home* e do componente são relacionados);
- Nome do executor do *home* a ser gerado.

A implementação de aplicações CCM têm início com a definição de componentes, seus *homes* e as interfaces que este componente provê em IDL3. Um compilador deve fazer o mapeamento da IDL3 escrita pelo desenvolvedor em IDL2, a interface equivalente do componente. Para isso, o compilador usa como entrada o arquivo IDL3 definido pelo desenvolvedor, e o arquivo com a definição CIDL da composição do componente. A saída gerada pelo compilador são: o arquivo IDL2, stubs e skeletons e um arquivo descritor contendo informações a respeito do componente. O desenvolvedor deve então implementar os executores de seus componentes, que conterão o código com a lógica da aplicação. De fato, os passos de implementação de componentes apresentam variações conforme as ferramentas usadas para o processo de desenvolvimento.

3.5.3 O Modelo de Empacotamento

O modelo de empacotamento especifica como devem ser empacotados os componentes para que possam ser distribuídos e implantados. Ao final do processo de implementação de componentes, componentes (tipo de componente) e descritores são empacotados num arquivo de formato ZIP. Um descritor de pacote define o conteúdo do pacote, e informações gerais sobre o software que contém (autor, descrição, interfaces IDL, nome do componente, SO de destino, linguagem de implementação, etc). O descritor informa ainda sobre dependências de contexto da implementação.

O pacote pode conter implementações de um único componente ou um conjunto de componentes interconectados formando assim um *assembly* de componentes. Estes *assemblies* de componentes possuem um descritor que especifica quais os componentes que integram o pacote, quais as restrições de alocação e conexões entre eles.

3.5.4 O Processo de Implantação

Para que os componentes implementados possam ser usados, a ferramenta de implantação ou um arquivo para implantação (gerado pelo desenvolvedor ou por uma ferramenta) deve executar uma seqüência básica de passos para cada componente:

- Inicializar o ORB;
- Obter o serviço de nomes;

- Obter a referência do servidor do componente;
- Obter a referência do *home* do componente;
- Instanciar um contâiner no servidor;
- Instalar o *home* para o componente;
- Criar o componente a partir da chamada de método de criação do *home*;
- Configurar o componente usando métodos correspondentes;
- Conectar cada componente usando interfaces de conexão do mesmo;
- Chamar o método que sinaliza o fim da configuração do componente.

O desenvolvimento de sistemas baseados em componentes, ou a montagem do sistema a partir de componentes disponíveis, pode ser dividido em três etapas distintas: fase de configuração, fase de inicialização de instâncias e fase de uso dos mesmos.

A configuração de componentes pode ser realizada através de objetos de configuração. Os objetos de configuração encapsulam a configuração de alguns ou todos os atributos do componente e esta configuração pode ser aplicada a qualquer instância de componente de mesmo tipo que se relaciona a este objeto. Estes objetos podem acionar qualquer operação de componente disponível em tempo de configuração. Utilizando interfaces de introspecção disponíveis em todo componente CCM, o objeto de configuração descobre as funcionalidades do componente e constrói as instâncias do mesmo fazendo também as interconexões necessárias com outros componentes ou serviços do ORB. A etapa de utilização do componente tem inicio quando o cliente adquire a interface do *home* do componente.

Os descritores de implantação são usados para a escolha do tipo de contâiner a ser usado para o componente. Informações como persistência, qualidade de serviço e política de *threads* podem também fazer parte deste descritor.

Os descritores de propriedades definem parâmetros de configuração de componentes e *homes*. Estes são os parâmetros usados para configuração destes elementos através de atributos. Os valores são os valores padrão e por isso podem ser alterados posteriormente pelo usuário. Em geral este descritor é usado pelo objeto de configuração.

3.6 CONCLUSÃO

Os componentes de software têm o propósito de promover o reúso de implementações de soluções. Quando este objetivo é alcançado, a conseqüência direta é o aumento de produção de software e melhoria de qualidade do mesmo. Dentre os modelos de componentes disponíveis, o modelo de componentes CCM figura entre os mais conhecidos e estabelecidos.

O CCM é um modelo de componente do lado servidor usado para montar aplicações e implantar componentes. O CCM padroniza e automatiza o ciclo de desenvolvimento de componentes através da definição de infraestrutura de *middleware* e de um conjunto de ferramentas de suporte. Este modelo permite a definição de interfaces suportadas por componentes e automatiza a implementação e empacotamento de componentes em arquivos que podem ser automaticamente implantados no computador servidor. A arquitetura do modelo utiliza padrões de projeto já estabelecidos e permite a automatização de geração de código e utilização da infraestrutura do contâiner para gerenciar interações entre componentes e serviços do sistema. O CCM enfoca a provisão de serviços genéricos de sistema requeridos por um servidor de aplicações e implementados por um contâiner, livrando o código da aplicação de tarefas complexas e propensas a erros, e permitindo ao desenvolvedor se concentrar em detalhes da lógica da aplicação.

O CCM se concentra no mecanismo do contâiner como forma de implementação da separação de aspectos funcionais de aspectos não funcionais. Assim como o contâiner, *homes* de componentes são usados para gerenciar o ciclo de vida de componentes.

O modelo CCM é um modelo de componentes abrangente ou flexível com relação a especializações para sistemas embarcados ou tolerante a faltas, por exemplo. Entretanto, esta flexibilidade (inerente ao CORBA) oferece ao desenvolvedor uma grande quantidade de escolhas, e implica em um grande número de detalhes que devem ser especificados sob o risco de inconsistências de combinações. Na prática, o quanto o desenvolvedor terá que lidar com as complicações destas escolhas está diretamente relacionado ao suporte oferecido pela ferramenta usada no desenvolvimento destes componentes e aplicações resultantes.

4 GARANTIA DINÂMICA EM SISTEMAS DISTRIBUÍDOS BASEADOS EM COMPONENTES

Neste capítulo é apresentado o modelo DGC (*Dynamic Guarantee for Components*), para provisão de garantia dinâmica(RAMAMRITHAN, K.; STANKOVIC J. A., 1994) de tempo real em sistemas distribuídos baseados em componentes. O objetivo deste modelo é capacitar servidores de aplicações a garantir os requisitos temporais de uma carga computacional que não é conhecida a priori.

O modelo permite que condições de sobrecarga sejam detectadas antes que estas possam provocar efeitos negativos para o sistema. Qualquer ativação de tarefas no servidor só é possível quando existem recursos suficientes para atendê-la. Um teste de aceitação ao qual é submetida qualquer requisição de execução no servidor garante que clientes conectados ao sistema previamente não sejam prejudicados pelo teste de aceitação ou pela inclusão de um novo cliente. Ao mesmo tempo, um novo cliente para o qual o sistema não possua recursos suficientes é imediatamente rejeitado.

A aplicação de um teste de aceitação em tempo de conexão entre cliente e servidor utiliza um protocolo de *two phase commit* durante a primeira iteração entre cliente e servidor. Transações distribuídas (*flat distributed transactions*) são realizadas entre os vários nodos (ou servidores participantes) envolvidos na provisão da funcionalidade requerida pelo cliente. A requisição de um serviço por parte de um cliente pode ser aceita (committ) ou negada(abort).

O mecanismo é genérico o suficiente para aceitar diversos algoritmos de análise de escalonabilidade para o teste de aceitação. Portanto, o modelo pode ser adequado à plataforma do nodo alvo de implantação do componente e usar os mecanismos que esta plataforma disponibilizar.

O modelo é simples o suficiente para ser integrado a qualquer aplicação de tempo real baseada em componentes CORBA. Os requisitos de tempo real usados não são específicos do modelo e a sua integração à aplicação não exige ferramentas específicas de desenvolvimento ou modelagem.

Como o modelo promove a seleção de novos clientes com base em requisitos de tempo, ele pode ser usado como parte de um mecanismo de balanceamento de carga.

4.1 O MODELO *DYNAMIC GUARANTEE FOR COMPONENTS*

O modelo proposto se destina a sistemas distribuídos abertos em que o número de clientes é variável e não é conhecido em tempo de projeto. Neste tipo de sistema, não é possível determinar em que momento novos clientes tentarão acessar o servidor e nem como determinar a carga computacional máxima a qual o sistema poderia ser exposto. O sistema é composto por um lado cliente e um lado servidor estruturados de maneira que um número variável de clientes possa acessar o servidor em qualquer instante. Os requisitos temporais requeridos por clientes são conhecidos somente em tempo de execução. O lado servidor é composto por N ($N \geq 1$) componentes que cooperam para prover uma funcionalidade. O domínio do modelo é o conjunto de nodos que caracterizam o servidor da aplicação.

A provisão de garantia dinâmica no servidor de aplicação implica em um controle de acesso de clientes baseado no estado corrente dos recursos deste servidor. O critério usado para a aceitação ou não de um novo cliente está relacionado à capacidade dos recursos controlados e à carga computacional que a requisição de um novo cliente representa para este sistema. Esta carga computacional é caracterizada pelo: (i) número de clientes que acessam o servidor; (ii) a funcionalidade requerida pelo cliente; e (iii) pelos requisitos de tempo real que o cliente necessita para este(s) serviço(s).

Assumimos neste trabalho, um modelo de comunicação síncrona (CRISTIAN, Flaviu; AGHALI, Houtan; STRONG Ray; DOLEV Danny, 1985) com tempos de comunicação limitados e conhecidos. A aplicação cliente não é especificada explicitamente, mas as restrições temporais requeridas por ela devem ser fornecidas ao servidor no instante da primeira invocação de um serviço.

4.2 DEFINIÇÕES BÁSICAS

Um componente oferece um conjunto de métodos ou operações que implementam sua funcionalidade. A provisão de uma funcionalidade do servidor através da execução de um método ou uma seqüência de métodos de seus componentes caracteriza um **serviço** S. Os métodos que compõem a execução de um serviço podem ser implementados por vários componentes implantados em diferentes nodos, servidores participantes ou subsistemas, como partes do serviço que gerenciam dispositivos físicos diferentes.

Este tipo de construção de aplicações pode ser encontrado, por exemplo, em sistemas de aviação ou espaço. No domínio do espaço alguns satélites científicos são construídos especificamente para cada missão devido a especificidade da missão e dos equipamentos requeridos. Neste contexto, é comum que existam equipamentos e software embarcado construídos como subsistemas fechados e bastante específicos. Como consequência, um satélite é composto por um sistema heterogêneo (vários fornecedores, vários subsistemas) em que a cooperação entre estes subsistemas é determinante para o gerenciamento da missão e da obtenção do estado do satélite. O estado de um satélite é determinado pela avaliação dos vários subsistemas do satélite. Novos comandos de centros de controle (ground station), por exemplo, pode levar ao reposicionamento do satélite, ao desligamento de parte dos equipamentos ou até mesmo a adoção de um novo procedimento de resposta conforme os níveis de energia ainda existentes no satélite. Cada uma destas ações representam execuções de diferentes subsistemas. Neste sentido, cada subsistema do satélite é modelado como um nodo do sistema cuja carga se pretende controlar.

Nos sistemas de aviação não é somente a especificidade de equipamentos que leva á necessidade de controle isolado da carga computacional dos subsistemas, mas também a criticalidade da funcionalidade destes subsistemas. Neste domínio, subsistemas não críticos podem simplesmente ser removidos da escala de execução quando comprometem o cumprimento satisfatório de um serviço. O diagnóstico de uma falha ou a simples verificação do estado do sistema (satélite ou aeronave), por exemplo, pode conter maior ou menor número de detalhes conforme a janela de visibilidade (e tempo de comunicação) do satélite para um centro de controle ou o tempo de resposta requerido por um piloto.

A Figura 7 ilustra o serviço S_1, composto pela execução dos métodos $C_1.m_2$, $C_2.m_1$ e $C_3.m_1$ e o serviço S_2 composto pela execução dos métodos $C_3.m_2$ e $C_2.m_2$. Cada nodo possui o **mapeamento local** do serviço com o qual está envolvido, a relação entre o método local e o método remoto a ser invocado.

A execução seqüencial de um conjunto de métodos é a forma mais simples de modelar um serviço. Entretanto não é a situação mais frequentemente encontrada nas aplicações. Os métodos de um componente podem prover funcionalidades que dependem da cooperação de um componente remoto em meio à sua execução, este segundo componente pode também envolver um terceiro componente em sua execução caracterizando chamadas aninhadas. Com o objetivo de obter maior

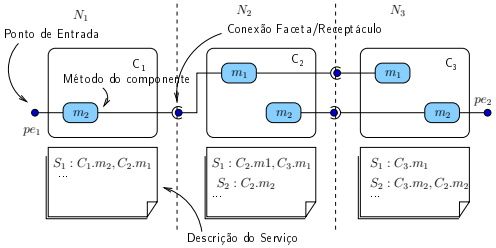


Figura 7: Fluxo de execução: mapeamento de serviços.

precisão quanto a modelagem do comportamento temporal do servidor da aplicação, adotamos a abstração de **tarefas** com prioridades fixas.

No contexto deste trabalho, cada tarefa do servidor de aplicações corresponde a execução de um bloco de código que não contenha a invocação de um método remoto. Quando um método local m_1 invoca um método remoto m_2, a descrição de m_1 é composta pela composição de dois conjuntos de informação; (i) a tarefa que tem início no instante em que m_1 foi invocado e que termina no momento em que m_2 é invocado; (ii) a tarefa que inicia no momento em que m_2 retorna e termina no fim da execução de m_1. Esta distinção de blocos de código de um método em tarefas permite a construção de escalas de execução mais precisas mesmo quando várias invocações aninhadas fazem parte da execução do serviço. Desta forma, um serviço passa a ser um conjunto de tarefas: $S = \{t_1, t_2, \ldots, t_n\}$ sendo n o número de tarefas que compõem S. O conjunto de nodos que participa na execução de um serviço S é definido como N_S. Tomando como exemplo a figura 7 temos $N_{S_1} = \{N_1, N_2, N_3\}$.

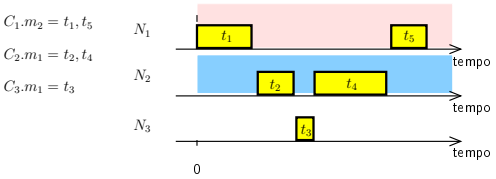


Figura 8: Decomposição de métodos em tarefas.

Um exemplo da representação de tarefas da Figura 7 é ilustrado na Figura 8. Para cada nodo do domínio, existe um conjunto de tarefas

locais executadas como parte do serviço S_1 (T_{S_1}) e um conjunto de dependências destas tarefas locais. Para o nodo N_1 da Figura 8, $T_{S_1} = \{t_1, t_5\}$ e suas dependências: $Dep_{S_1} = \{t_2\}$. Observe que o conjunto de dependências de um serviço em um nodo, inclui as tarefas resultantes de invocações a partir de tarefas locais. A tarefa t_3 também é parte do serviço S_1, mas não é chamada por nenhuma das tarefas do nodo N_1, por isso N_1 não precisa armazenar informação referente à esta tarefa. A tarefa t_3 é uma dependência do serviço S_1 em N_2, porque sua invocação determina o término da tarefa t_2. O nodo N_3 possui um conjunto vazio de dependências do serviço S_1, nenhuma tarefa é invocada pelas tarefas do N_3 como parte de S_1.

De maneira geral, apenas alguns componentes de um servidor apresentam uma interface que pode ser acessada por elementos externos ao domínio do servidor. Chamamos o meio pelo qual um componente expõe sua funcionalidade para os clientes da aplicação de **ponto de entrada** (*ep*). Cada ponto de entrada dá acesso a um ou mais serviços providos pelo servidor: $pe = \{S_1, S_2, \ldots, S_k\}$, sendo k o número de serviços que podem ser acessados pelo ponto de entrada pe. Na Figura 7, os pontos de entrada pe_1 e pe_2 são providos pelos componentes C_1 e C_3 respectivamente. O método $C_2.m_2$ da aplicação ilustrada nesta figura não pode ser acessado por um cliente, este método não apresenta interface visível para clientes. O conjunto de nodos que pode ser acessado por um ponto de entrada pe é definido como N_{pe}, sendo $N_{pe} = \{n \in N_S : n\ provê\ S \in pe\}$ e N_S o conjunto de nodos que participam na provisão de S, conforme exemplo da figura 7, $N_{pe_1} = \{N_1, N_2, N_3\}$. Nas próximas seções usaremos as definições relacionadas aos pontos de entrada.

Definimos a **profundidade de um ponto de entrada** como o número máximo de nodos que pode ser atingido através de seus serviços, isto é, o número máximo de componentes que executam tarefas em consequência de uma invocação a este ponto de entrada. No exemplo da Figura 7, a profundidade de pe_1 é 3, pois $Pr(S_1) = 3$.

4.3 TESTE DE ACEITAÇÃO

Aplicações clientes requisitam os serviços através de invocações periódicas ao servidor. A primeira invocação de um cliente está sempre sujeita à um **teste de aceitação**, que garantirá recursos para as demais requisições no caso da aceitação do cliente. Quando existem recursos suficientes para garantir a funcionalidade requerida pelo cli-

ente, conforme os requisitos de tempo-real exigidos por este cliente, o servidor retorna o resultado da invocação; o resultado da invocação do cliente é o mesmo de um servidor sem qualquer teste de aceitação (não considerando, neste caso, o tempo requerido pelo teste de aceitação). Porém, quando o servidor não consegue garantir ao cliente o serviço requerido conforme os requisitos de tempo real (falha no teste de escalonabilidade), o cliente recebe como resposta uma exceção indicando falta de recursos no servidor para garantir os requisitos do cliente.

Neste modelo, a primeira iteração entre cliente e servidor é uma invocação incluíndo um valor para cada campo da tupla: $\langle id, S, D, P, TS \rangle$: um identificador do cliente id, o serviço requerido S (identificador), o *deadline* D e período P exigidos e o intervalo durante o qual as chamadas periódicas serão invocadas (**tempo de sessão**) TS. Se existirem recursos suficientes no servidor para as requisições periódicas, estes serão alocados pelo tempo de sessão requerido.

Para cada sessão, o teste de aceitação distribuído deve ocorrer uma única vez. Os recursos (em cada um dos nodos de componentes envolvidos na provisão do serviço) são alocado para o cliente por todo o tempo de sessão requisitado. O teste ocorre na primeira invocação do conjunto de invocações periódicas, por isso, esta invocação é diferenciada das invocações subseqüentes. Verificações no servidor devem ser feitas a cada requisição recebida, para diferenciar uma chamada de um cliente já aceito (e dentro do seu tempo de sessão), de uma requisição de novo cliente (que deverá acionar o teste).

O algoritmo de teste de aceitação é recursivo, executa em todos os nodos que participam na provisão do serviço S. O primeiro passo do algoritmo é a determinação dos *deadlines* parciais de cada tarefa que compõe os métodos locais de componentes (linhas 1 a 3). A patir da nova descrição do serviço sob a forma de tarefas, o teste de escalonabilidade determina se as tarefas locais requeridas podem ser incluídas ao sistema sem que tarefas previamente aceitas sejam prejudicadas (linha 5). Se a nova carga computacional local é escalonável, o teste de aceitação é chamado nos nodos correspondentes às dependências dos métodos do serviço S (linhas 6 e 7).

Se o teste retorna verdadeiro em todos os nodos das dependências, a reserva dos recursos locais é feita pelo tempo de sessão TS requerido pelo cliente. Caso contrário, uma exceção indicando a falta de recursos em atender a requisição é retornada (linha 8, no caso de dependências, linha 13, no caso dos recursos locais).

O algoritmo executa em tempo de *bind*; durante a primeira invocação de um cliente. Em chamadas subsequentes (considerando um

Algorithm 1 Teste de aceitação distribuído.

procedure *TesteAceitação*(S, D, P)

1: **for all** $t_i \in S$ **do**
2: $d_i \leftarrow ParticionaDeadline(S, D)$
3: **end for**
4: **for all** $t_i \in S$ **do**
5: **if** escalonavel(t_i) **then**
6: **for all** $n \in N_S$ que executa tarefas de $Dep(S)$ **do**
7: **if** $\neg TesteAceitacao(S, D, P)$ em n **then**
8: **return** NO_RESOURCES;
9: **end if**
10: **end for**
11: ReservaRecurso(todas as tarefas locais)
12: **else**
13: **return** NO_RESOURCES;
14: **end if**
15: **end for**

cliente aceito), somente a identificação do cliente é necessária para controle de acesso por parte do servidor. Quando a ativação de uma tarefa relacionada à chamada de um cliente não permite uma escala possível no servidor, o cliente recebe uma exceção; o cliente falhou no teste de aceitação. O cliente pode redefinir seus parâmetros de tempo real e tentar nova invocação ao servidor, mas enquanto não passar pelo teste de escalonabilidade, terá suas invocações sujeitas ao teste.

4.4 GERENCIANDO A CARGA COMPUTACIONAL

Como ilustrado na Figura 7, grupos diferentes de componentes estão envolvidos na provisão de cada serviço. O componente C_1 está envolvido na provisão do serviço S_1 enquanto os componentes C_2 e C_3 estão envolvidos na provisão do serviço S_1 e S_2. Sob a perspectiva do Nodo 2, a carga computacional local envolve a execução de tarefas dos métodos m_1 e m_2 que devem manter as relações de precedência com outros métodos remotos que compõem os serviços S_1 e S_2.

O modelo foi projetado de forma a considerar a heterogeneidade do domínio do servidor, a diferença de capacidade dos nodos. O modelo objetiva o controle do comportamento de tempo real fim a fim, reforçando o comportamento de tempo real em cada nodo através de

proteção temporal (*temporal protection*) entre os nodos. A proteção temporal objetiva não permitir que a falha de um nodo em atender os requisitos temporais de suas tarefas comprometa outro nodo do mesmo servidor. No contexto deste trabalho, nodos cooperam para prover um mesmo serviço. A idéia de proteção temporal é usada apenas na modelagem das tarefas afim de possibilitar um teste de aceitação descentralizado.

Algoritmos de **particionamento de *deadline*** dividem um *deadline* especificado para um serviço entre tarefas de componentes. O objetivo do particionamento é determinar o *deadline* para tarefas locais em cada nodo, separando o tempo disponível para cada componente do servidor. Quando todos os *deadlines* parciais estão disponíveis, a escala local de tarefas pode ser construída em cada nodo conforme mostra a Figura 13. O *deadline* parcial de uma tarefa é usado como tempo de chegada para a próxima tarefa a ser executada no contexto do serviço. Usando como exemplo a aplicação da Figura 7, o *deadline* parcial calculado para o método $C_1.m_2$ é usado como tempo de chegada para o método $C_2.m_1$. As restrições temporais para cada tarefa dos componentes são traduzidas em cada nodo a partir do *deadline* requerido pelo cliente e a carga computacional local pode ser planejada considerando tarefas periódicas independentes (cancelamento de *jitter*).

Após o particionamento de *deadline* e o ajuste de parâmetros temporais, um teste local de escalonabilidade determina a aceitação ou rejeição de um novo cliente no sistema. Em tempo de execução, o particionamento de *deadline* e o teste de escalonabilidade são executados em cada um dos nodos que participam na provisão do serviço requisitado antes que qualquer recurso seja alocado para este cliente (Figura 11). O particionamento permite o ajuste de parâmetros temporais para cada tarefa em seu nodo; *deadlines* parciais, instantes de chegada, valores de *jitter*. O *deadline* parcial é calculado com um algoritmo específico para o modelo de tarefas usado pelo escalonador local e sua capacidade de escalonamento, o instante de chegada é recalculado em cada nodo conforme o tempo gasto pelo próprio teste de aceitação com relação ao instante de recebimento da requisição.

4.5 ALGORITMOS DE PARTICIONAMENTO DE *DEADLINE* E TESTE DE ACEITAÇÃO

O modelo DGC aceita diferentes algoritmos de escalonamento, particionamento de deadline, e teste de aceitação. Nesta seção, des-

crevemos os algoritmos *Equal Flexibility* e um teste de utilização para exemplificar o funcionamento do modelo e facilitar a descrição das seções seguintes.

O algoritmo de particionamento de deadline, *Equal Flexibility* (EQF) (KAO, Ben; GARCIA-MOLINA, Hector, 1997) objetiva a provisão de igual flexibilidade para todas as subtarefas para o qual o deadline é dividido. A **flexibilidade** de uma tarefa é definida pelo ***slack* time** e pelo tempo de execução da tarefa. O valor de *slack time* é determinado pelo *deadline* menos o tempo de execução da tarefa. O *slack time* total de um serviço é dado pelo *deadline* do serviço menos a soma dos tempos de execução das tarefas que compõem este serviço. O *slack time* do serviço é partilhado proporcionalmente entre todas as subtarefas conforme seu tempo de execução. A equação que descreve o algoritmo EQF é ilustrada abaixo.

$$d_i = a_i + e_i + [(D - a_i - \sum_{j=i}^{n} e_j) * (C_i / \sum_{j=i}^{n} C_j)]$$

onde:
d_i é o deadline parcial de uma tarefa i ($1 < i < n$ e n é o número de tarefas);
a_i é o tempo de chegada da tarefa i;
e_i é o tempo de execução da tarefa i;

Assumindo a independência das tarefas a serem escalonadas em cada nodo, podemos utilizar um teste de aceitação simples, baseado em tempos de resposta (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000). O tempo de resposta máximo (JOSEPH, M.; PANDYA, P., 1986) é o tempo transcorrido entre a chegada e o término de sua execução considerando a máxima interferência que a tarefa pode sofrer de outras tarefas de maior ou igual prioridade.

O conjunto de tarefas considerado para o teste de aceitação em cada nodo é modelado como um conjunto de tarefas independentes. Relações de precedência são substituídas por um tempo de liberação que inclui o tempo de resposta da tarefa (ou tarefas) cuja execução precede a execução da tarefa considerada no grupo em análise.

O cálculo de tempo de resposta máximo de uma tarefa é necessário a definição de uma janela de tempo R_i que corresponda ao intervalo de tempo máximo transcorrido da liberação de uma tarefa t_i até o término de sua execução. R_i corresponde ao tempo necessário, em situação de instante crítico, para execução de t_i e de todas as tarefas com prioridade maior ou igual a i ($p_i = i$). Nestas condições, o tempo de resposta máximo de R_i da tarefa t_i é dado por:

$$R_i^{k+1} = e_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil * e_j$$

onde:
$hp(i)$ é o conjunto de prioridades maior que i;
$\lceil \frac{R_i}{P_j} \rceil$ determina o número de liberações de t_j em R_i; a interferência de tarefas de prioridade maior ou igual a p_i.

A solução é conseguida quando $R_i^{k+1} = {R_i}^k$. A condição de partida para este método iterativo é ${R_i}^0 = e_i$. O método não converge quando a utilização do conjunto de tarefas é maior que 100%. Este teste determina um conjunto de n tarefas como escalonável sempre que a condição $\forall i : 1 \leqslant i \leqslant n; R_i \leqslant D_i$ for verificada.

Os algoritmos usados para particionamento de *deadline* e teste de escalonabilidade são escolhidos conforme a capacidade do nodo envolvido na aplicação. A caracterização do modelo de tarefas da aplicação também determinam que tipo de análise de escalonabilidade pode ser aplicado. O compartilhamento de recursos entre tarefas do nodo, por exemplo, exigem a utilização de mecanismos para determinar piores casos de bloqueio.

4.6 TEMPO DE *BIND*

Denominamos **tempo de *bind*** entre cliente e servidor, o momento da primeira invocação de um cliente ao servidor para requisitar uma funcionalidade. O tempo de *bind* é o tempo de maior *overhead* provocado pelo modelo.

O tempo gasto para realização do teste em tempo de *bind* para um ponto de entrada *pe* é dado pelo tempo de comunicação entre os nodos alcançados por este ponto (N_{pe}) mais o tempo do teste em cada um dos nodos.

O tempo de bind pode ser calculado pela a equação abaixo:

$$T_{bind}(ep) = 2\Delta * (Pr_{ep} - 1) + (\sum_{n \in N_{ep}} (T^n_{part} + T^n_{sched}))$$

onde:
A primeira parte da equação; $2\Delta * (Pr_{ep} - 1)$ caracteriza o tempo gasto pela comunicação entre os nodos que participam do teste de aceitação;
Δ é tempo de comunicação entre dois nodos (*oneway*);
Pr_{ep} é a profundidade do ponto de entrada *ep*, o número máximo de nodos que implementam o serviço;
A segunda parte da equação determina o tempo gasto por todos os nodos atingidos a partir de *pe* para execução de particionamento de

deadline e teste de aceitação.
T^n_{part} é tempo gasto pelo nodo n para executar o algoritmo de particionamento de *deadline*;
T^n_{sched} é o tempo gasto pela execução do algoritmo de teste de aceitação em cada um dos nodos acessados a partir de *ep*.

O tempo gasto pelo algoritmo de particionamento e teste de aceitação em cada nodo não são iguais. Algoritmos diferentes podem ser usados em cada nodo, e os métodos a serem executados como parte dos serviços requisitados podem ser compostos por um número variável de tarefas.

O valor calculado pela equação não inclui o atraso de comunicação entre cliente e servidor, é um tempo calculado para definição do tempo de chegada das tarefas no servidor e montagem da escala local de cada nodo.

O período especificado pelo cliente para o teste de aceitação é tratado no servidor como tempo mínimo entre chamadas de requisições. Em cada requisição após a aprovação do cliente no sistema, o tempo entre chegadas é usado para assegurar que clientes não invoquem componentes com frequência mais alta que especificada no teste.

4.7 APLICANDO O MODELO DGC PARA DIFERENTES GARANTIAS DE TEMPO REAL

O modelo desenvolvido pode acomodar diferentes algoritmos de teste de escalonabilidade e particionamento de *deadline*. Esta abordagem permite o aproveitamento dos vários algoritmos da literatura de modo que possa ser escolhido o algoritmo condizente com o tipo de tarefas da aplicação, ou o algoritmo que os recursos da plataforma implementem. A figura 9 ilustra possibilidades de algoritmos diferentes de particionamento de *deadline* e teste de escalonabilidade na implementação da estrutura do modelo DGC.

Aplicações de tempo real críticas: No domínio de aplicações de tempo real crítico, os algoritmos de teste se baseiam em piores tempos de execução das tarefas e situações pessimistas, como a chegada, ao mesmo tempo, de todas as tarefas que possam interferir no tempo de resposta de uma tarefa. Geralmente, o teste usado para estes sistemas são testes suficientes, portanto restritivos. Em testes restritivos o conjunto de tarefas aceito certamente é escalonável, porém entre as tarefas descartadas podem existir conjuntos escalonáveis. Em troca da garantia de um conjunto que seja escalonável, recursos são subutilizados. No

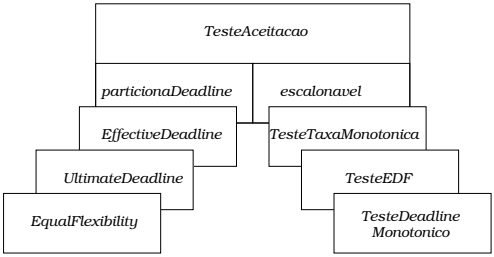


Figura 9: Adaptando o Modelo DGC.

caso de aplicações como controle de tráfego aéreo, a subutilização é justificável.

Conforme o modelo de tarefas da aplicação algoritmos diferentes podem ser utilizados: *deadline* monotônico(LEUNG, J. Y. T.; WHITEHEAD, J., 1982), taxa monotônica(LIU, Chien-Liang; LAYLAND James W., 1973) e EDF-*Earliest Deadline First* figuram entre os mais conhecidos, vários outros modelos de tarefas de tempo real crítico e testes de aceitação podem ser encontrados na literatura. O algoritmo de particionamento de *deadline* tem influência direta sobre o escalonamento das tarefas que baseiam prioridades em *deadlines*, por isso a escolha dos algoritmos deve ser feita em conjunto. Na escolha de algoritmos de escalonamento que baseiam prioridades em deadlines, *ultimate deadline* e EQF são algoritmos que não interferem com a prioridade do serviço considerando o domínio de tarefas gerenciadas pelo DGC. Quando estes algoritmos são combinados sobre uma plataforma determinista, a garantia provida é determinista. **Aplicações de tempo real brando:** Videogames são exemplos comuns de aplicações de tempo real brando. Principalmente aqueles que operam na Internet caracterizam uma aplicação exemplo bastante adequada para o modelo DGC. Considerando um número imprevisível de jogadores que podem requisitar a entrada no jogo a qualquer instante, o modelo DGC pode controlar a sobrecarga deste sistema antes que os jogadores no meio do jogo sejam prejudicados. Em aplicações desta natureza, as premissas pessimistas de tempo real crítico nem sempre são justificáveis. Como a aplicação pode absorver uma certa degradação de qualidade de serviço sem conseqüências trágicas, o melhor desempenho do sistema pode ser alcançado adotando abordagens probabilistas. As abordagens probabilistas utilizam tempos de resposta do caso médio e baseiam suas decisões em dados passados. No contexto de videogames; a aceitação de um novo cliente baseado na

estimativa da carga que este cliente provocará no sistema considerando a carga provocada por clientes que acessaram o serviço pelo mesmo ponto de entrada.

A escolha do algoritmo de particionamento de deadline segue os mesmos princípios da escolha do algoritmo para sistemas de tempo real crítico, ele interfere com a escala de tarefas no nodo de acordo com a política de prioridades adotada. Nestes sistemas, plataformas deterministas não são comumente usadas, o que o modelo DGC pode prover nestes casos é a detecção de uma sobrecarga do sistema.

A escolha de um algoritmo específico para o particionamento de *deadline* e teste de escalonabilidade depende do escalonador local do nodo, da aplicação alvo e do modelo de tarefas. Por isso, estes não são o foco deste trabalho. O modelo permite a escolha de diferentes algoritmos para estes passos.

4.8 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos o modelo *Dynamic Guarantee for Components*. Definimos suas premissas e seu funcionamento. Mostramos que o modelo é modular, admitindo diferentes algoritmos para seus mecanismos e que pode ser usado em diferentes contextos de aplicações de tempo real.

Neste capítulo descrevemos a integração de um mecanismo de controle de carga computacional à infraestrutura de componentes de software. Este controle se baseia em características inerentes ao modelo de componentes; às informações necessárias para que um componente de software seja implantado de forma independente. O mecanismo é integrado à infraestrutura exigida de qualquer modelo de componentes e permite monitoramento da carga computacional de um componente a cada requisição de serviço recebida.

A garantia de tempo real oferecida reflete o que o sistema operacional local, os mecanismos de comunicação e os algoritmos de particionamento de *deadline* e teste de escalonabilidade são capazes de suportar. Se sistemas operacionais e mecanismos de comunicação deterministicos forem usados, o sistema permite que a arquitetura proposta ofereça garantia determinísta. Porém, se o sistema operacional e suporte de comunicação apresentam comportamento probabilista, a arquitetura oferecerá somente um mecanismo de detecção de sobrecarga.

Mostramos que o domínio de informação do container sobre os componentes e sobre o sistema são ideais para o controle de sobre-

cargas transitórias ou mesmo a provisão de garantia dinâmica para as requisições dos clientes.

No próximo capitulo descreveremos como o DGC é mapeado para uma arquitetura que pode ser implementada em diferentes plataformas de componentes e ilustramos a implementação de um protótipo para prova de conceito deste modelo.

5 *MIDDLEWARE* COM GARANTIA DINÂMICA

Neste capítulo apresentamos a implementação de um *middleware* com garantia dinâmica de tempo real. O modelo DGC foi inicialmente mapeado para uma arquitetura que pode ser implementada em outras tecnologias de componentes. A implementação do modelo DGC foi desenvolvida no CIAO - *Component Integrated ACE ORB* (WANG, Nanbor; GILL, Christopher, 2004), uma implementação do modelo de componentes CORBA. Experimentos com a implementação foram executados com o objetivo de verificar o *overhead* provocado pelos mecanismos do modelo e comparar o desempenho da plataforma com e sem os mesmos.

5.1 MAPEAMENTO DA GARANTIA DINÂMICA NO CCM

De acordo com as premissas do modelo DGC apresentado no capítulo anterior, os servidores de aplicação para o qual o modelo foi projetado são compostos por componentes que se comunicam através de portas síncronas; facetas e receptáculos. A comunicação entre clientes e o servidor de aplicação pode ocorrer através de portas ou através de interfaces suportadas pelo componente. Somente os clientes *component-aware* podem acessar as funcionalidades do servidor através de facetas. Clientes *component-unaware* utilizam uma interface suportada pelo componente. Facetas de componentes dão acesso a um conjunto de métodos, já as interfaces suportadas, permitem a chamada de um único método. Por isso, clientes diferentes têm acesso a serviços diferentes de um servidor. No modelo DGC, a especificação do serviço requerido por parte do cliente prevê invocações de clientes *component-aware*, e por isso utiliza para o controle de acesso, a informação à respeito da funcionalidade requerida ao invés do controle por pontos de entrada. No caso de clientes *component-unaware*, a informação sobre funcionalidade requerida se torna redundante, mas o formato da tupla requerida do cliente para o servidor é mantida para atender os dois tipos de cliente do servidor no CCM.

As ferramentas de modelagem de componentes CCM (GOKHALE, Anirudda; NATARJAN, Balachandran; SCHMIDT, Douglas C.; NECHYPURENKO, Andrey, 2002) provêem, durante a construção dos componentes, e posteriormente durante o planejamento da composição da aplicação, as informações que serão utilizadas durante o processo de implantação destes componentes. Idealmente, durante toda a fase de desenvolvimento do

componente, descritores são preenchidos por ferramentas para serem posteriormente utilizados na preparação deste componente já em sua plataforma alvo. É esperado pelas ferramentas de implantação e configuração dos componentes CCM, descritores em linguagem declarativa contendo informações que especificam desde a composição da aplicação (como os componentes devem ser conectados) até valores para configuração de atributos de componentes.

Independente do fato do ponto de entrada de um servidor ser implementado por uma interface suportada pelo componente ou por uma porta, a distinção de componentes que possuam ponto de entrada é necessária para o modelo. Esta informação pode ser adicionada a um descritor de implantação do componente. Para clientes *component-unaware*, o tempo de *bind* corresponde à primeira invocação ao servidor. Para clientes *component-aware* o tempo de *bind* é a invocação de uma faceta que provê um serviço.

5.2 ARQUITETURA DE GARANTIA DINÂMICA NO CCM

Idealmente, os mecanismos de garantia dinâmica devem ser providos como um serviço (como serviço de persistência ou transação), cuja necessidade seria indicada de forma declarativa para um componente ou *assembly* de componentes. Desta forma, um contêiner de tempo real seria instanciado em tempo de implantação para o componente. Nesta seção descrevemos a implementação da arquitetura capaz de oferecer garantia dinâmica conforme o modelo descrito no capítulo anterior em uma plataforma existente (CIAO), e que pode ser implementada em outras plataformas de componentes CCM ou EJB.

Um gerente de aspectos de tempo real é instalado em cada nodo do servidor de aplicação (Figura 10). O gerente administra os requisitos temporais de clientes e monitora os recursos disponíveis no nodo. De modo geral, a cada primeira invocação de cliente, o gerente:

- Ajusta as restrições temporais do serviço para parâmetros de tarefas locais;
- Verifica a escalonabilidade do sistema ao aceitar este cliente;
- Ativa gerentes de outros nodos que participem na provisão deste serviço;
- Aloca os recursos reservados após a confirmação de recursos disponíveis nos outros nodos;

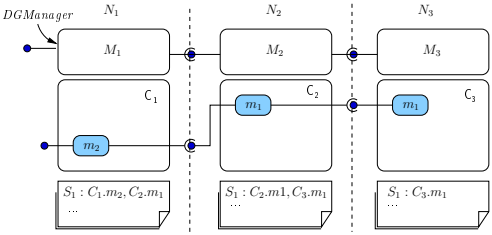


Figura 10: Relações entre *DGManagers* e componentes da aplicação.

- Garante que os clientes respeitem a frequência de invocação requisitada.

Como o número de nodos que compõem um serviço não é conhecido *a priori*, o algoritmo de teste distribuído é recursivo; executa até que todos os nodos que participam da provisão do serviço sejam alcançados. Para a aplicação ilustrada na Figura 10, as interações entre os gerentes é descrita na Figura 11.

A implementação do algoritmo de teste distribuído envolve os seguintes elementos: (i) interceptador do lado cliente, que envia restrições de tempo real junto com as requisições; (ii) interceptador do lado servidor que invoca o gerente de tempo real quando uma requisição à um ponto de entrada é recebida; e (iii) o gerente de tempo real, que controla o acesso de clientes às funcionalidades do servidor da aplicação.

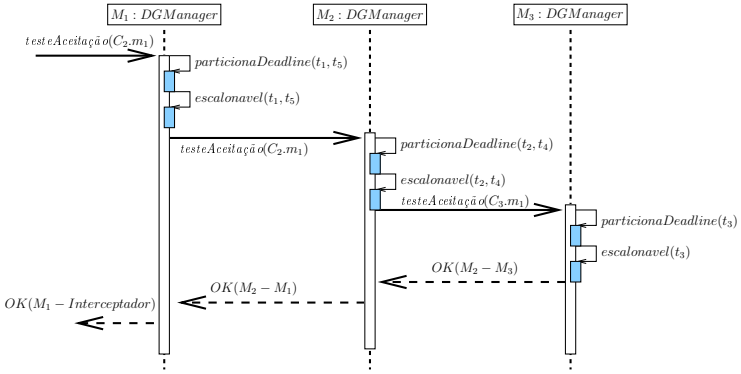


Figura 11: Exemplo de interações em tempo de execução.

5.3 IMPLEMENTAÇÃO DA GARANTIA DINÂMICA NO CCM.

A arquitetura desenvolvida para o modelo é composto por *DG-Managers*, `Node Applications` do CIAO e interceptadores portáveis do CORBA. O papel dos módulos principais que compõem o modelo são descritos nas subseções abaixo.

5.3.1 `Node Application`

O `Node Application` é parte da especificação de instalação e configuração da OMG (Object Management Group, 2003). Ele desempenha o papel de um processo servidor de componente que provê recursos computacionais para os componentes que hospeda. Baseado em metadados providos por ferramentas de instalação, os `NodeApplication` criam containers iniciais que provêem o ambiente para a criação e instanciação de componentes da aplicação (DENG,Gan; BALASUBRAMANIAN, Jaiganesh; OTE, William; SCHMIDT, Douglas C.; GOKHALE, Aniruddha, 2005).

O `Node Application` é ativado antes que a aplicação seja instalada nos nodos do domínio, ele inicia o ORB e executa enquanto a aplicação estiver em execução. A única modificação necessária ao `Node Application` é a adição do código que instala o interceptador no servidor. Isto é feito por mecanismo tradicional de instalação de interceptadores CORBA no ORB (Object Management Group, 2004a).

5.3.2 Interceptadores

Interceptadores portáveis do CORBA (Object Management Group, 2004a) foram usados tanto no lado cliente quanto no lado servidor da aplicação. Pontos de interceptação no contâiner seriam ideais para o propósito deste mecanismo, entretanto, não existe suporte para estes interceptadores no CIAO. A especificação de Qualidade de Serviço para o CCM (Object Management Group, 2006b) (que prevê tais interceptadores) ainda se encontra em desenvolvimento. Os interceptadores são ativados pelo `Node Application`, todo o código restante criado para este trabalho foi colocado nos interceptadores e em componentes CIAO padrão.

Interceptador do lado cliente O interceptador do lado cliente da aplicação adiciona as restrições de tempo real à invocação ao servidor. Estas restrições devem ser preenchidas em um descritor XML antes

que a aplicação cliente inicie a execução. Antes que a requisição seja realizada, a informação do descritor XML é preenchida na requisição para o servidor. As seguintes informações são utilizadas: identificação do cliente, período, deadline, serviço a ser invocado e tempo de sessão. Uma vez aceito, as chamadas subsequentes são enviadas somente com o identificador do cliente. Um exemplo do descritor XML usado para especificações de tempo real é ilustrado abaixo.

```
<clientdata id="123456">
        <entrypoint id = "position_facet" >
            <method id = "get_position">
                <configParameter>
                        <period>600</period>
                        <deadline>600</deadline>
                        <session>6000</session>
                </configParameter>
            </method>
        </entrypoint>
</clientdata>
```

Interceptador do lado Servidor O objetivo do interceptador do lado servidor da aplicação é ativar o teste de aceitação distribuído assim que uma invocação à um ponto de entrada é recebida no servidor. Por isso, após a implantação dos componentes do servidor da aplicação, o interceptador de requisições no lado servidor procura pela referência do DGManager através do Serviço de Nomes. Além disso, o interceptador é configurado para reconhecer invocações aos pontos de entrada do servidor de aplicação.

O interceptador é instalado no ORB do nodo servidor, por isso toda requisição que alcança este nodo é interceptada. Cabe ao interceptador filtrar as invocações de interesse, as invocações cujo objetivo é um ponto de entrada, para que o DGManager seja invocado para o teste destas requisições. As demais invocações podem simplesmente ser ignoradas (não provocam execuções no interceptador).

Caso o componente não implemente um ponto de entrada, o DGManager instalado neste nodo não será invocado pelo interceptador do nodo, mas sim por um DGManager remoto (Figura 12). Se não existe no nodo nenhum componente com um ponto de entrada, o Node Application pode ser iniciado sem interceptadores.

Quando um ponto de entrada é invocado, o interceptador extrai os dados anexados à requisição, os requisitos temporais enviados pelo interceptador do lado cliente. Com estes parâmetros o interceptador chama o método `access` do DGManager, descrito na próxima seção.

E conforme o valor retornado por este método, o método `allocate` é invocado em seguida.

5.3.3 DGManager

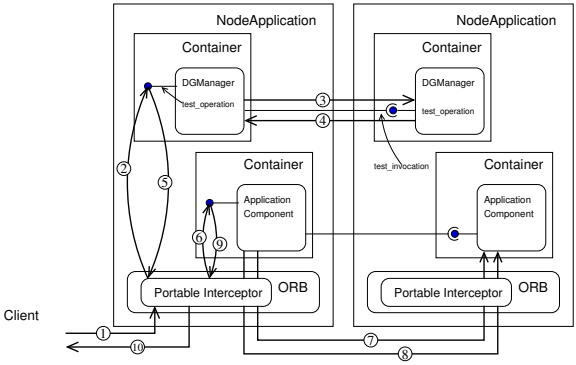


Figura 12: Fluxo de Execução: requisição de bind aceita.

O DGManager encapsula a maior parte da funcionalidade provida pelo serviço de garantia dinâmica. Ele é implementado como um componente CIAO padrão cujas instâncias são implantadas nos nodos do servidor da aplicação (Figura 12). Cada instância do DGManager monitora um ou mais componentes instalados no mesmo nodo. Sua funcionalidade é exposta através da faceta `test_operations` e através do receptáculo `test_invocation` o DGManager acessa outros DGManagers. Quando dois componentes envolvidos na provisão de um serviço são instalados em nodos diferentes, os DGManagers destes nodos são conectados, a execução do teste no primeiro DGManager desencadeia o teste no segundo DGManager.

A faceta `test_operation` provê dois métodos:

- `access`: requer como parâmetros a identificação do cliente e o método a ser chamado e retorna um valor `boolean`. Este método verifica se um cliente que tenta acessar o servidor possui permissão para fazê-lo. Se a identificação do cliente está armazenada no DGManager, significa que este passou pelo teste de aceitação e ainda se mantém dentro do seu tempo de sessão. Portanto os recursos necessários para sua requisição ainda estão alocados para

o cliente. Este método também objetiva garantir que o cliente respeite o tempo mínimo entre ativações: o instante da última requisição deste cliente é armazenado e comparado com o instante da requisição corrente. Caso o tempo entre chegadas seja menor que o período especificado pelo cliente no teste de aceitação, a requisição não é processada, e uma exceção `CORBA::NO_RESOURCES` é lançada. Para novos clientes ou clientes que excedem seu tempo de sessão, o método `access` retorna `false`.

- `allocate`: requer a identificação do cliente e suas restrições temporais. Este método é chamado quando o método `access` retorna `false`. `allocate` é o método através do qual os mecanismos de teste de aceitação são disparados. Ele executa: (i) o algoritmo de particionamento de deadline; (ii) o teste de aceitação local; (iii) o ajuste de parâmetros temporais; (v) a invocação do método `allocate` em outros DGManagers que monitoram dependências do método requerido; (vi) alocação dos recursos locais após a confirmação de outros DGManagers a respeito da disponibilidade de recursos para a chamada do cliente (Figure 11).

A comunicação entre DGManagers (passos 3 e 4 da Figura 12) também passa pelo ORB e interceptadores, assim como a comunicação entre componentes da aplicação (passos 7 e 8). Porém, estes não são ilustrados na Figura porque nesta comunicação não existe qualquer verificação ou interferência por parte dos interceptadores.

A conexão entre DGManagers assim como de componentes da aplicação é feita da maneira usual em tempo de implantação, quando é conhecido quais componentes estão envolvidos em cada serviço e em que nodos da rede os componentes da aplicação serão instalados. Após a implantação de componentes da aplicação e DGManagers, a configuração de DGManagers é realizada através dos descritores XML. Os descritores definem quais componentes cada DGManger deve monitorar e quais os métodos a serem executados como parte de cada serviço. Esta informação permite que DGManagers ajustem as restrições temporais fornecidas pelo cliente em restrições para seus métodos locais.

A classe *DGManager* gerencia uma lista de instâncias de *MonitoredComponents*. A classe *MonitoredComponents* representa o componente da aplicação cujo comportamento temporal se pretende controlar. Novas instâncias desta classe são criadas conforme a inclusão de novos componentes a serem monitorados. Atualmente, isto é feito somente uma vez, na implantação da aplicação. Entretanto, a implementação pode ser facilmente estendida para inclusão de novos componentes em

tempo de execução no caso de re-implantação de componentes.

Cada instância de *MonitoredComponent* gerencia duas listas; uma para a classe *Operation* e uma para a classe *AcceptedClient*. A classe *Operation* representa cada um dos métodos implementados pelo componente monitorado. Instâncias desta classe também são adicionadas em tempo de implantação de componentes, quando as informações a respeito das tarefas que compõem as operações são providas para o DGManager. Dois tipos de tarefas são descritas na classe *Operation*; tarefas locais que executam no nodo e tarefas remotas que incluem todas as invocações realizadas por tarefas locais. A descrição de uma tarefa local consiste de tempo de chegada e tempo de execução estimado. As tarefas remotas são dependências do componente, a descrição destas tarefas também consiste de um tempo de chegada, que na verdade é o deadline de sua tarefa predecessora (sempre uma tarefa local), e o tempo de resposta estimado. Mas também contém a referência para o componente remoto a ser invocado. A classe *AcceptedClient* representa os clientes já conectados ao servidor. As instâncias desta classe são criadas em tempo de execução e adicionadas à lista *MonitoredComponent* ou removidas desta lista de acordo com o tempo de sessão requisitado pelos clientes.

5.4 EXPERIMENTOS

O modelo descrito no capítulo anterior foi implementado usando CIAO (Component-Integrated ACE ORB), uma implementação do Lightweight CORBA Component Model (CCM) e Real-time CORBA (Object Management Group, 2002), construído sobre o TAO (The ACE ORB).

A implementação descrita nesta seção é um exemplo, foram usados algoritmos simples de particionamento de deadline e teste de aceitação baseado em utilização (FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, 2000). Os algoritmos são descritos abaixo.

5.4.1 O particionamento de deadline e o teste de aceitação.

O algoritmo de particionamento de deadline implementado pelo método `allocate` dos DGManagers nesta implementação é o *Equal Flexibility* (EQF) (KAO, Ben; GARCIA-MOLINA, Hector, 1997). O EQF, como descrito na seção 4.5, objetiva a provisão de igual flexibilidade para todas as subtarefas para o qual o deadline é dividido.

Para este algoritmo de particionamento de deadline, o DGManager precisa como dado de entrada: o tempo de execução do serviço, o tempo de execução das tarefas do método local, deadline e período requerido pelo client. O tempo de execução dos métodos remotos pode ser usado pelo DGManager para construção de uma escala de tarefas locais quando existem invocações aninhadas em um método de componente.

Na implementação, cada DGManager tem uma visão parcial do serviço; em cada nodo, o serviço é mapeado conforme as tarefas locais e suas dependências. Em cada iteração, ou em cada DGManager do serviço, é calculado somente o deadline parcial das tarefas locais considerando a proporção de seu tempo de execução com relação ao tempo de execução dos componentes que ainda não realizaram o teste. No primeiro nodo, o deadline total considerado é o deadline requerido pelo cliente (D). No segundo nodo, o deadline total considerado para o particionamento é o deadline D menos o deadline parcial das tarefas do primeiro nodo. Esta forma de mapeamento utiliza somente informação suficiente para o particionamento de deadline e montagem de uma escala local (Figura 13).

Na Figura 13, um exemplo de particionamento de *deadline* é apresentado conforme a configuração dos métodos da Figura 7. Os *deadlines* parciais calculados conforme o EQF são indicados no eixo do tempo. O serviço $S_1 = \{C_1.m_2, C_2.m_1, C_3.m_1\} = \{t_1, t_2, t_3, t_4, t_5\}$ é requisitado com deadline $D = 4800$. Assumindo tempos de execução de tarefas dados por $C_{S_1} = \{300, 200, 100, 400, 200\}$, o deadline parcial das tarefas são $d_{S_1} = \{1200, 800, 400, 1600, 800\}$.

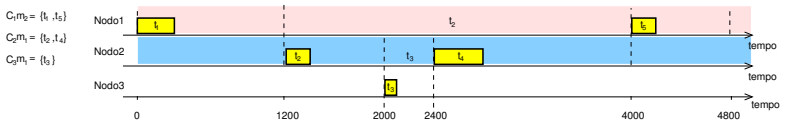


Figura 13: Particionamento de deadline: um exemplo.

É importante notar que a arquitetura não impõe um algoritmo de particionamento de deadline. Assim como o algoritmo de teste de escalonabilidade, este mecanismo depende da capacidade do escalonador. Para o teste de aceitação, um algoritmo simples, baseado em utilização foi aplicado. O algoritmo verifica se o tempo de resposta de cada tarefa, considerando a carga corrente do sistema, permite o cumprimento do *deadline* requerido pelo cliente.

Na Figura 11, os passos da execução do modelo são ilustradas conforme a configuração dos componentes mostrados na Figura 7 e a organização das tarefas descritas pela Figura 13. A seqüência: particionamento de deadline, reserva de recursos locais (assumindo a aceitação local da requisição) e a requisição do teste de aceitação em outros DGManagers é recursivo entre os nodos do servidor (Figura 11). A seqüência termina quando o último DGManager que participa da provisão do serviço é alcançado. A requisição de teste de aceitação alcança o último DGManager somente se todos os DGManagers que executaram anteriormente confirmam a disponibilidade de recursos para a requisição. Se ao menos um destes gerentes não possuírem recursos necessários para atender a requisição, uma resposta negativa é retornada para o gerente que o chamou e o próximo DGManager não é invocado.

O início do teste de aceitação distribuído é sempre provocado por um interceptador de requisições (Figure 12). Sempre que um componente da aplicação provê um ponto de entrada, as requisições destinadas a ele são monitoradas pelo interceptador que invoca o DGManager local. Os DGManagers que monitoram componentes que não oferecem pontos de entrada são invocados por outros DGManagers. O tempo de execução de DGManagers e interceptadores de requisição são contabilizados em tarefas locais aperiódicas gerenciadas por servidores de tarefas aperiódicas.

Quando o tempo de sessão de um cliente termina, o identificador do cliente é removido da lista de clientes aceitos, todos os recursos alocados pra este cliente são desalocados e chamadas posteriores deste mesmo cliente deverão ser novamente submetidas ao teste de aceitação.

5.4.2 Avaliação do *Overhead*

Os experimentos foram executados para determinar o *overhead* provocado pelos mecanismos empregados pelo DGC em aplicações com diferente número de componentes. O ambiente de execução é composto por um conjunto de cinco PCs Athlon 2.4GHz com 512 Mb de memória, executando Linux (kernel 2.6.12). As máquinas foram conectadas por uma rede Ethernet 100Mbps. A plataforma de componentes utilizada foi CIAO 0.5.3 (TAO 1.5.3).

Em todos os experimentos, executamos aplicações com um número variável de componentes (de 1 a 4 componentes). Cada componente de aplicação foi instalado em um nodo diferente com um componente DGManager também instalado no mesmo nodo. O quinto nodo do

conjunto de máquinas foi usado para disparar requisições de clientes.

Os experimentos mediram (i) o *overhead* resultante do uso dos algoritmos de particionamento de deadline e teste de aceitação previamente descritos para a primeira invocação de clientes ao servidor e (ii) o *overhead* provocado pelo uso de interceptadores em chamadas de clientes já aceitos.

O objetivo do primeiro experimento foi avaliar o *overhead* provocado pelos mecanismos usados pelo modelo, por isso, algoritmos sofisticados para o particionamento de deadline e teste de aceitação foram evitados. O teste aplicado é suficiente, ele pode deixar de aceitar tarefas que poderiam ser escalonáveis, entretanto, ele assegura que condições de sobrecarga não ocorrerão no sistema. O gráfico da Figura 15 mostra a média dos tempos de resposta e o desvio padrão correspondente em microssegundos (us) de 1000 invocações de diferentes clientes à aplicação com um a quatro componentes. Em todos os experimentos os clientes foram chamados com período de 1000ms.

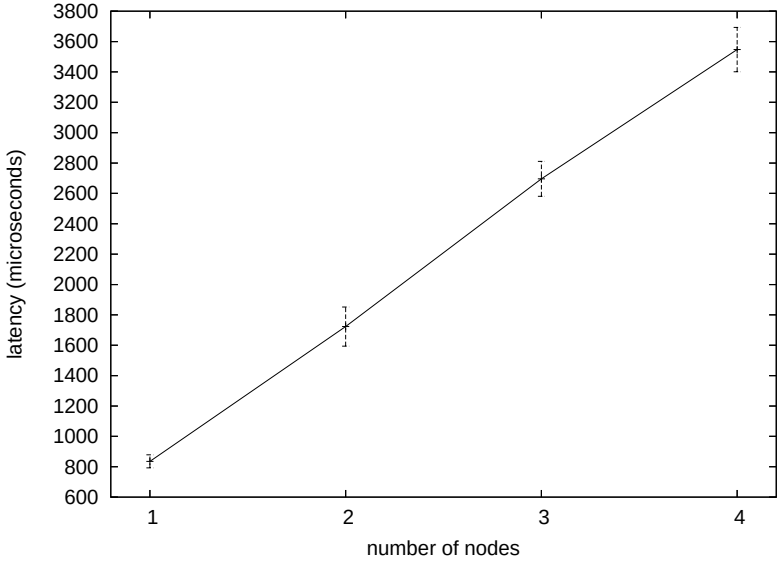
Nos gráficos é possível observar o aumento do tempo de resposta médio com o aumento do número de componentes da aplicação. A diferença na latência sofrida pelos clientes destas aplicações é de aproximadamente 1ms maior para cada componente adicionado na aplicação. Observando a diferença de tempos de resposta da aplicação: com um componente e da aplicação composta por dois componentes, podemos inferir que a diferença é provocada pela comunicação entre os dois componentes e o particionamento de deadline executados neste cenário, elementos que inexistem em uma aplicação com somente um componente. Estes experimentos demonstram que a arquitetura para garantia dinâmica implica em um custo de aproximadamente 1ms por nodo nesta configuração. É preciso considerar que estas medidas dependem do algoritmo usado para o particionamento de deadline e teste de aceitação. Valores mais altos seriam alcançados para algoritmos mais sofisticados.

O gráfico da Figura 14 ilustra o tempo de resposta médio de aplicações usando os mecanismos implementados para o modelo DGC no CIAO no cenário em que clientes já foram aceitos. Para este experimento, 1000 invocações de um único cliente foram executadas. O *overhead* da primeira invocação do cliente não é ilustrada no gráfico uma vez que 10% dos valores de tempo de resposta mais altos e mais baixos foram desconsiderados.

No gráfico da Figure 14 podemos observar que o *overhead* provocado pelo interceptador na implementação do modelo DGC em comparação com o CIAO é de aproximadamente 200us. O *overhead* é causado

pelas verificações feitas em cada requisição de cliente no interceptador do lado servidor.

Figura 14: Overhead de teste de aceitação.

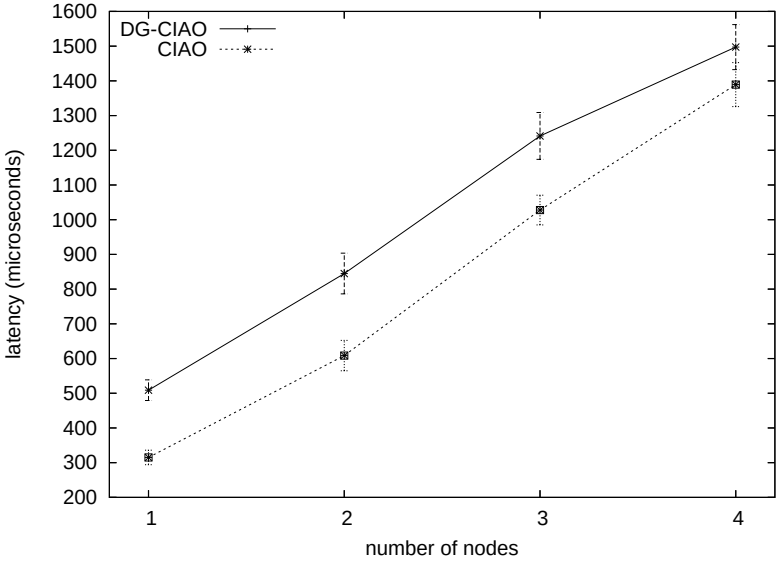


5.5 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos a arquitetura do modelo DGC enfatizando seus principais elementos e como estes se encaixam no modelo de componentes CCM. Descrevemos a implementação da arquitetura sobre uma plataforma de componentes existente, o CIAO, e ilustramos os resultados de alguns experimentos elaborados para capturar o overhead provocado pelos mecanismos utilizados em aplicações com diferentes quantidades de componentes. A arquitetura pode ser implementada em diferentes tecnologias além do CIAO, uma vez que os mecanismos utilizados se baseiam em especificações como CCM e Interceptadores portáveis do CORBA.

A dificuldade em implementar o *middleware* descrito neste capítulo reside diversos fatores; a complexidade da especificação de componentes CORBA é um deles. No sentido de prover a facilidade de desenvolvimento para especialistas de aplicações específicas, a especifi-

Figura 15: Overhead de teste de aceitação.



cação torna-se extremamente exigente com relação à infraestrutura que deve ser oferecida para os componentes. Principalmente quando se deseja acrescentar funcionalidades relacionadas à qualidade de serviço a esta infraestrutura, é perceptível a falta de diversos mecanismos como interceptadores.

Outra dificuldade no desenvolvimento do *middleware* foi lidar com uma plataforma ainda em desenvolvimento, como é o caso do CIAO. O CIAO foi escolhido por ser uma das plataformas de componentes CCM que tem infraestrutura para aplicações de tempo real e melhor condições de desenvolvimento, é a plataforma em estágio mais avançado de amadurecimento além de contar com suporte para usuários (tutoriais e listas de discussão). Entretanto, nem sempre a implementação é condizente com a especificação CCM e maneiras de contornar as diferenças sem comprometer o modelo com uma plataforma específica tiveram que ser adotadas.

O uso de tarefas de componentes ao invés de métodos de componentes como unidades do teste (granulosidade) oferecem uma melhor precisão. Porém, esta escolha só pode ser feita quando é possível quando se tem acesso a estrutura interna do componente. Este tipo de informação pode ser disponibilizado de duas maneiras: (i) o desenvol-

vedor do componente de tempo real fornece os tempos de execução de cada tarefa, e consequentemente a estruturação interna dos métodos do componente em tarefas; ou (ii) através de uma ferramenta capaz de extrair tal informação a partir da execução deste componente. Ambas as abordagens apresentam uma série de vantagens e desvantagens.

Os experimentos mostram que não existe um *overhead* exagerado com relação ao funcionamento do CIAO sem os mecanismos do modelo DGC. O valor obtido é um indicativo que deve ser avaliado pelo desenvolvedor de uma aplicação quanto ao benefício ou não do mecanismo em detrimento ao tempo de resposta acrescido pelo DGC. A arquitetura da aplicação desenvolvida e as restrições temporais exigidas desta aplicação determinarão o impacto preciso deste *overhead*.

É importante perceber que os algoritmos usados neste capítulo representam os algoritmos mais facilmente encontrados ou implementados em plataformas não especializadas. Desta maneira, estes experimentos podem ser facilmente reproduzidos. Uma característica importante do modelo proposto é que ele não limita o uso de algoritmos que podem ser integrados. Cada um dos nodos que fazem parte de um servidor podem usar algoritmos diferentes, adequados a plataforma ou funcionalidade que suporta. O que limita a utilização do modelo nestes casos é o conjunto de parâmetros de tempo real utilizados. Algoritmos que requeiram mais ou menos parâmetros do que aqueles utilizados pelo resto do sistema podem provocar a falha do modelo.

A possibilidade de aplicar o modelo a servidores compostos por plataformas heterogêneas é meramente teórica tendo em vista que nenhum experimento pode ser feito neste sentido. Mas trata-se de uma característica importante para sistemas sofisticados e de grande escala que envolvem necessariamente mais de um tipo de plataforma e consequentemente tecnologias de suporte como é o caso de aeronaves e satélites.

No próximo capítulo apresentamos um serviço para obtenção das informações necessárias para aplicação do teste de aceitação em componentes.

6 SERVIÇO DE MONITORAMENTO PARA COMPONENTES DE TEMPO REAL

Os testes de aceitação de algoritmos da literatura (LEUNG, J. Y. T.; WHITEHEAD, J., 1982; LIU, Chien-Liang; LAYLAND James W., 1973) se baseiam nos tempos de execução das tarefas. No contexto do modelo DGC, em conjunto com o período e deadline determinados pelo cliente, o tempo de execução das tarefas que compõem um serviço caracterizam a carga provocada por uma requisição. A análise desta carga determina a aceitação ou não de um novo cliente. O tempo de execução da aplicação é um dado de entrada para o modelo que pode determinar a exatidão ou eficiência do controle de admissão de clientes.

Aplicações de natureza temporal diferentes exigem tratamentos de sobrecarga diferentes em servidores de aplicação. Para que possam oferecer garantias de comportamento de tempo real mesmo em situações de pior caso de execução, sistemas de tempo real crítico adotam uma premissa pessimista. Nestes casos, os tempos de execução utilizados para os mecanismos de controle de sobrecarga destas aplicações são tempos de execução de pior caso, geralmente obtidos através de análises de pior tempo de execução (WCET). Já aplicações de tempo real brando podem muitas vezes absorver certa degradação do comportamento de tempo real (sofrer perdas de alguns deadlines) em troca de melhor aproveitamento de recursos. E por isso podem adotar estimativas de pior tempo de execução ou mesmo tempos de execução do caso médio.

De qualquer maneira, o tempo de execução é um dado de entrada para o mecanismo que controla o comportamento temporal da aplicação e deve estar disponível antes que o mecanismo entre em execução. No contexto de aplicações baseadas em componentes, esta não é uma tarefa simples, o tempo de execução de uma tarefa depende tanto do código quanto da infraestrutura que executa esta tarefa. A análise de pior tempo de execução exige, em tempo de projeto, o conhecimento a respeito da plataforma de implantação do componente que nem sempre está disponível.

Ainda não existe um padrão que defina o meio pelo qual o tempo de resposta ou execução de um componente seja disponibilizado para a infraestrutura de suporte do componente de tempo real. Esta questão é dificultada principalmente porque um dos objetivos da abordagem de desenvolvimento baseada em componentes é a portabilidade e possibilidade de implantação de componentes em aplicações desconhecidas em

tempo de projeto. Estes objetivos são de certa forma conflitantes pois o comportamento temporal do componente é influenciado tanto pela infraestrutura subjacente quanto pela integração com outros componentes.

Em sistemas de tempo real brando, os componentes geralmente executam na camada de aplicação e estão sujeitos à interferência das camadas de software subjacente. Neste contexto, os tempos de resposta agregam dados a respeito do comportamento temporal do componente na plataforma alvo e a respeito das interferências sofridas pelo componente da aplicação. Os tempos de resposta refletem também, a situação de carga computacional do sistema.

Neste capítulo apresentamos um serviço de monitoramento para componentes de tempo real. O serviço objetiva extrair do componente, dados necessários para predição do comportamento do sistema com relação à tempo real sem que isso implique na inserção de código não funcional em componentes. No decorrer deste capítulo motivamos a utilização deste serviço no contexto de aplicações de tempo real brando e descrevemos seus mecanismos e funcionamento no contexto do modelo DGC.

6.1 VISÃO GERAL DO SERVIÇO DE MONITORAMENTO

O serviço de monitoramento é baseado na idéia da infraestrutura, em conjunto com interfaces do componente (como interfaces de navegação) extraírem do componente instalado na plataforma alvo, a estimativa de tempo de execução.

O serviço de monitoramento objetiva estimar o tempo de resposta das tarefas do componente após a sua implantação na plataforma alvo. As estimativas de tempo de resposta neste caso, são obtidas pela execução do componente usando parâmetros pré-definidos pelo desenvolvedor do mesmo.

O modelo DGC baseia o teste de aceitação distribuído no isolamento temporal entre nodos do servidor. Isto implica no tratamento local da carga em cada nodo. Por isso o serviço de monitoramento dos tempos de resposta dos componentes considera somente o código executado localmente.

Para o obtenção dos tempos de execução de um método local m_1, um temporizador deve ser iniciado no instante em que m_1 é invocado e parado no instante em que m_1 retorna um resultado. Mas também é preciso parar o temporizador sempre que m_1 faz uma chamada remota,

da mesma maneira que o temporizador também deve voltar a contabilizar o tempo no instante em que esta chamada remota retorna um valor para m_1.

6.2 A FACETA DE CONFIGURAÇÃO DE ASPECTOS TEMPORAIS DO COMPONENTE

Neste capítulo, exploramos uma alternativa para obtenção de um tempo de resposta do componente baseado na execução dos mesmos na plataforma alvo utilizando parâmetros pré-selecionados pelo desenvolvedor do componente. O objetivo não é alcançar determinismo, mas obter uma estimativa de comportamento temporal do componente em sua plataforma alvo.

A provisão de informações sobre o comportamento temporal de componentes é complexa porque desenvolvedores de componentes tem acesso à estrutura interna mas não podem prever em que plataforma o componente será instalado. O profissional que instala o componente e monta a aplicação, por sua vez, tem conhecimento a respeito das conexões do componente e não sua estrutura interna. Estas informações em conjunto: estrutura interna, configuração de montagem e plataforma alvo são importantes para determinação do comportamento temporal do componente. Por isso, a extração desta informação depende de um mecanismo conjunto de ambos os passos: desenvolvimento e implantação.

Neste trabalho propomos a utilização de uma faceta de componente (**faceta de configuração** de aspectos temporais) ou interface provida pelo componente que ao ser invocada, provoque a execução de todos os métodos do componente usando parâmetros para o pior tempo de execução. E um serviço de monitoramento de tempos de resposta de componentes que executa em conjunto com a faceta de configuração. O serviço de monitoramento dos tempos de resposta em conjunto com a faceta de configuração é invocado no fim da etapa de implantação da aplicação e antes da aplicação entrar em execução. O objetivo é capacitar a infraestrutura do servidor de componente a extrair do mesmo os tempos de resposta de seus métodos exigindo o mínimo de intervenção por parte do desenvolvedor da lógica do componente. Os tempos de resposta são coletados para prover o modelo DGC com dados de entrada.

Vários conjuntos de parâmetros podem ser oferecidos pelo desenvolvedor para serem escolhidos na execução do serviço após a implan-

tação do componente: parâmetros que possam provocar o pior caso de execução são um exemplo para obtenção de uma estimativa pessimista. A idéia é permitir que através dos parâmetros pré-selecionados, o desenvolvedor possa contribuir com a obtenção de tempos de resposta na plataforma alvo.

O conjunto de parâmetros fornecidos para a execução da faceta de configuração pode ser escolhido conforme o tipo de controle de sobrecarga a ser alcançado pelo componente. Pode ser fornecido um conjunto de parâmetros que pode provocar os piores tempos de resposta ou um conjunto que possa fornecer tempos de resposta para povoar uma base de dados com possíveis valores para uso de algoritmos probabilista. O conjunto ou conjuntos podem ainda refletir diferenças da plataforma alvo e serem escolhidos pelo integrador da aplicação durante a implantação do componente.

O objetivo do serviço de extração é prover dados necessários para os mecanismos de controle de tempo real da aplicação respeitando as propriedades fundamentais da abordagem de desenvolvimento de componentes. Embora esta abordagem seja incapaz de oferecer o determinismo necessário para aplicações de tempo real crítica, realiza uma solução mais eficaz que a simples adoção de valores padrão.

Este serviço provê estimativas de tempo de resposta de métodos considerando execuções locais, entretanto os mesmos mecanismos podem ser empregados para coletar outros tipos de informação, como a seqüência de chamadas desencadeadas em uma aplicação ou tempos de comunicação entre componentes.

6.3 IMPLEMENTAÇÃO

Embora diversos trabalhos envolvendo componentes e tempo real tenham sido desenvolvidos, não existe ainda uma especificação que determine como monitorar e capturar o comportamento temporal de aplicações. A definição das interfaces para o serviço de monitoramento de tempos de resposta é ilustrado abaixo. A cada método do componente, é associado um temporizador.

```
local interface FacetTimers {
    Timer get_method_timer(
            in string method_name);
    Timer get_active_timer();
}
```

```
local interface Timer {
    readonly ulong measured_time;
    void start();
    void stop();
};

interface Profiling{
    void define_environment(...);
    FacetTimers get_facet_profile(
              in string facet_name);
    FacetTimers get_facet_profile(
              in string facet_name);
    ...
};
```

O método `define_environment` retorna informações a respeito do sistema em que executa o serviço de extração de comportamento temporal da aplicação. Interfaces de navegação do componente podem ser utilizados para recuperação dos nomes de facetas e métodos implementados pelo componente. Arquivos do sistema que tenham informações relevantes quanto a configuração do nodo podem ser lidos.

O método `get_facet_profile` retorna a referência para `FacetTimers` que provê os métodos para obtenção do temporizador ativo dentre temporizadores dos métodos da faceta (`get_active_timer`) e temporizador de um método especificado por `method_name`.

Para cada método das facetas dos componentes existe um temporizador. Através da referência de `Timer`, o temporizador pode ser parado ou reiniciado e o tempo medido por eles é armazenado na variável `measured_time`. Esta variável foi definida para armazenar um único valor, o último valor medido. Um vetor para valores recentes medidos também podem ser usados (para posterior cálculo de média) conforme a abordagem escolhida para lidar com a definição de tempo de resposta adotado. Esta hierarquia de acesso aos temporizadores corresponde a hierarquia estabelecida pelo modelo de componentes; cada faceta de componente implementa vários métodos.

A obtenção dos tempos de resposta para o modelo DGC exige mecanismos capazes de interceptar requisições que chegam e que partem dos componentes do domínio do servidor de aplicação. O container de componente seria a infraestrutura ideal para implementar o mecanismo de detecção de invocações para medições dos tempos locais. Como estrutura responsável pela parte não funcional de um componente, o container realizaria a extração de propriedades como tempo de resposta e análise de dependência entre componentes. Entretanto,

containers não interferem com requisições entre componentes uma vez estabelecido a conexão entre as portas dos mesmos. A escolha utilizada pra implementação deste serviço é o uso dos interceptadores CORBA, que já fazem parte do modelo DGC.

Os interceptadores de requisição ao servidor do CORBA, *Server Request interceptor* são instalados no ORB e interceptam todas as chamadas à métodos de objetos sobre este ORB e também o retorno destes métodos. Estes interceptadores são capazes de determinar o início e o fim da execução de um método m_1. Entretanto, estes interceptadores são completamente alheios à chamadas que m_1 realiza em componentes remotos. Portanto, se o interceptador de requisições ao servidor inicia um temporizador quando uma requisição à m_1 alcança o nodo e pára o temporizador quando m_1 retorna o resultado, a medida do tempo transcorrido da chegada da requisição até o retorno de um resultado, fornece o tempo de resposta total do método e não o tempo de execução local deste método no nodo.

Um interceptador de requisições de cliente, *Client Request Interceptor*, por outro lado, pode detectar as requisições que deixam o nodo e também detectar o momento em que a invocação retorna, mas não pode determinar qual método fez a invocação, e desta forma parar o temporizador relacionado à este método. O temporizador do método deve ser acessado por ambos os interceptadores que tem a visão parcial sobre o fluxo de execução do nodo. Entretanto, caso somente um temporizador estiver ativo, e somente um método se encontra em execução, estes problemas podem ser superados.

A Figura 16 ilustra o mecanismo em funcionamento.

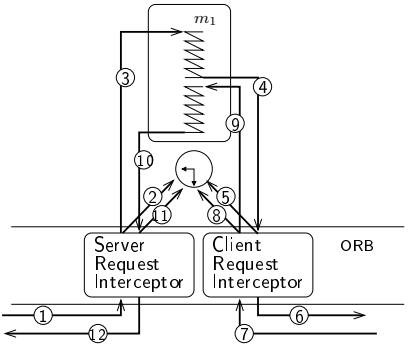


Figura 16: Coleta de tempo de resposta.

Interceptadores de requisição (*Server Request Interceptor* e *Client Request Interceptor*) foram iniciados no ORB sobre o qual o componente que implementa o método m_1 foi instalado. Os interceptadores são executados sempre que qualquer chamada alcança o nodo, e por isso, a cada requisição recebida, devem verificar se a chamada é de interesse e deve acionar um temporizador.

As verificações exigem a configuração do interceptador, que permitirá que o interceptador reconheça requisições de interesse. No caso de monitoramento de tempo de resposta, o interceptador mantém a lista dos componentes para os quais deve monitorar requisições. Chamadas que não se destinam aos componentes monitorados não provocam qualquer execução do serviço de monitoramento.

Quando um método m_1 do componente é invocado (passo 1), a chamada passa pelo ORB do nodo, e executa o interceptador de requisição de servidor. O interceptador verifica se o alvo da invocação é um objeto monitorado e (em caso afirmativo) invoca o método `get_method_timer(`m_1`)` da interface `Profiling` (passo 2). Ao receber a referência para o temporizador, inicia o mesmo e define este temporizador como ativo. A requisição prossegue execução normal (passo3) e o método m_1 inicia execução. O passo 4 da figura ilustra uma chamada de m_1 à um método remoto. Quando esta requisição alcança o ORB, o interceptador de requisições de cliente invoca o método `get_active_timer`, e para o temporizador ativo (passo 5), o temporizador de m_1. A invocação deixa o nodo (passo 6) e ao retornar (passo 7) o interceptador reativa o temporizador de m_1 (passo 8). O retorno da invocação remota é ilustrado pelo passo 9. Ao término de m_1, o método retorna o resultado (passo 10) e o interceptaodr de requisições do servidor para novamente o temporizador de m_1 e marca o mesmo como inativo.

6.3.1 Interceptadores de Container

A especificação de qualidade de serviço para componentes CORBA (Object Management Group, 2006b) é uma extensão para a especificação CCM. Esta extensão define um container denominado `Extension` que permite o uso de interceptadores portáveis de container (COPI - Container Portable Interceptor). Os interceptadores de container implementam os mesmo pontos de interceptação que interceptadores de requisição CORBA porém na camada de aplicação, na infraestrutura de suporte do componente e não no ORB. O objetivo de desenvolvimento

dos interceptadores de container é padronizar a forma com que componentes utilizam interceptadores. Estes interceptadores permitirão que o container seja capaz de recuperar informações da requisição e manipular o contexto do serviço propagado entre clientes e servidores.

Com isso, a interceptação se torna mais específica, só chegam ao interceptador, requisições destinadas ao componente monitorado.

O mecanismo proposto neste capitulo pode ser facilmente modificado para utilizar interceptadores de container ao invés dos interceptadores CORBA. Como as únicas requisições interceptadas seriam destinadas aos componentes deste container, mecanismos de seleção (filtro) de requisições recebidas não seriam necessários. Entretanto, a extração dos tempos de resposta ainda teriam que ser realizados sequencialmente; somente um temporizador pode estar ativo em um determinado momento no container. Isto porque o interceptador de requisições do servidor reconhece o início e o fim da execução de um método do componente e o interceptador de requisições de cliente reconhece apenas o início e o fim de uma invocação que deixa o container. Com os interceptadores de container, a extração de tempos de resposta pode ser feita em paralelo entre containers mas não entre métodos de um mesmo componente.

A primeira versão da especificação de qualidade de serviço para o CCM ainda se encontra em discussão. Muitas questões quanto a definição dos interceptadores e a forma com que devem ser implementados está em aberto. As implementações da especificação CCM ainda não contemplam a implementação da extensão.

6.4 MONITORANDO TEMPOS DE RESPOSTA

Os interceptadores de container permitem a monitoração de tempos de resposta somente no caso de componentes simples. O maior problema enfrentado no uso de interceptadores para este fim é a falta do conhecimento a respeito da origem de uma invocação remota: se dois métodos m_1 e m_2 de um único componente executam ao mesmo tempo, um interceptador de requisições de clientes não tem recursos para diferenciar qual dos métodos originou a invocação. Portanto não pode parar o temporizador correto. Entretanto, se um componente implementa um único método, podemos assumir um único fluxo de execução, e toda requisição que deixar o container é proveniente deste método. E isto implica na possibilidade de utilização deste mecanismo para coleta de tempo de resposta do componente durante todo o tempo em que

este componente executa.

Dentro deste contexto, o serviço de monitoramento, permite o armazenamento de um histórico de tempos de resposta de um método que reflete o estado de um nodo com relação a sua carga computacional. Este mecanismo é especialmente atraente para aplicações que aceitam mecanismos probabilistas de predição de tempo de resposta como critério para aceitação de novos clientes à um servidor de aplicação. O tempo de resposta de uma tarefa reflete, além de características inerentes ao código da aplicação, os parâmetros usados pela chamada em específico, o estado do programa com relação à suas variáveis permanentes (tamanho e conteúdo das estruturas do programa) e a carga computacional do nodo.

6.5 TESTES DE ACEITAÇÃO PROBABILISTAS: UMA APLICAÇÃO EXEMPLO

Nesta seção, descrevemos a aplicação do modelo DGC no contexto de aplicações que utilizam abordagem probabilista. O serviço de monitoramento de tempos de resposta permite a tomada de decisão a respeito da aceitação ou não de um cliente baseado no histórico de execuções do servidor.

Considere um servidor de aplicação composto somente por componentes simples, que implementem um único fluxo de execução. Cada serviço oferecido pelo servidor pode envolver vários componentes e cada serviço é definido por uma única *thread* de execução entre componentes; a provisão de um serviço envolve sempre a mesma sequência de execução dos mesmos componentes, independente dos parâmetros fornecidos pela chamada do serviço.

Neste cenário, os serviços não precisam de mapeamento em cada componente e a relação entre tarefas e método de componete é direta. Cada requisição de cliente envolve a execução da sequência prédeterminada do conjunto de componentes, conforme o serviço requisitado, e cada requisição especifica um período e um *deadline* dentro do qual o serviço deve ser provido. Conforme o modelo DGC, o *deadline* requisitado pelo cliente é particionado entre os componentes que compõem o serviço, e dentro de cada container um teste de aceitação determina a inclusão ou não das tarefas decorrentes da chamada.

Baseado nos recursos oferecidos pela monitoração de tempos de resposta, utiliza-se um algoritmo que, no instante da chegada de uma tarefa t_k, determina a probabilidade do *deadline* (parcial) D_k da tarefa

t_k ser atendido. O algoritmo determina a probabilidade do tempo de resposta de t_k ser menor que o *deadline* parcial calculado: $P(R_k <= D_k)$, onde R_k é o tempo de resposta associado com a execução de t_k. Como uma restrição adicional, o algoritmo precisa ser implementado a nível de aplicação, e não depender de suporte específico do sistema operacional subjacente.

Uma abordagem simples é utilizar o tempo de resposta de tarefas passadas para estimar a probabilidade do *deadline* de uma tarefa que chega, ser atendido. Para tanto, deve ser mantido um histórico contendo o tempo de resposta de cada tarefa executada anteriormente, para cada serviço da aplicação. O histórico pode ser usado como a PMF (Probability Mass Function) da variável aleatória R_k, e uma simples inspeção do histórico fornece a probabilidade $P(R_k < D_k)$ da tarefa t_k ter seu *deadline* atendido.

Entretanto, esta abordagem simples possui algumas desvantagens. Em função da rápida dinâmica do sistema, somente tempos de resposta recentes são relevantes para estimar o tempo de resposta de uma nova tarefa. Informações antigas, presentes no histórico, podem não ser relevantes para estimar o tempo de resposta de uma nova tarefa que chega. Além disso, o tipo de tarefa em questão pode não ter sido solicitado por um longo período, deixando o algoritmo de estimação sem dados para trabalhar.

Na realidade, o histórico completo contém a PMF do tempo de resposta das invocações de um dado serviço, observada sobre longos períodos de tempo. Para decidir sobre a probabilidade da próxima execução deste serviço atender ao seu *deadline*, é necessária a PMF desta próxima ativação. Os sistemas computacionais assumidos neste trabalho são de natureza não estacionária. A PMF da próxima ativação da função f depende do estado atual do sistema, o qual é definido por diversos fatores, tais como:

- os parâmetros fornecidos na chamada do serviço;
- valor das variáveis permanentes do programa;
- carga no computador decorrente das outras tarefas da mesma aplicação ou de outras aplicações, sobre os diversos recursos existentes;
- tamanho das filas associadas com serviços deste programa;
- alteração na demanda relativa dos diversos recursos existentes no sistema (processador, disco, rede, tabelas, etc), o que afeta diferentemente os vários serviços.

O número de estados possíveis para o sistema é imenso, quando não infinito. Entretanto, é possível aglutinar esses estados, conforme algumas propriedades que são fáceis de medir e capazes de indicar aproximadamente o tempo de resposta a ser esperado na próxima execução do serviço solicitado. Uma forma de atacar o problema é usar, para prever o tempo de resposta de um serviço S_1, não apenas as execuções passadas de S_1, mas sim o histórico de execuções passadas de todos os serviços suportados pelo programa. Entende-se que uma sobrecarga no nodo irá aumentar o tempo de resposta de todos os serviços, e não somente de S_1. Além disto, o tempo de resposta de S_1 pode ser influenciado pelo estado corrente das variáveis permanentes do programa (tamanho e conteúdo das estruturas de dados do programa). Assim, o comportamento recente de todos os serviços pode ser usado para o cálculo da previsão do tempo de resposta.

Uma análise da correlação cruzada entre os tempos de resposta dos vários serviços suportados pelo programa permitiria a identificação de quais serviços são relevantes para a estimação do tempo de resposta de quais serviços. Supondo que o programa implemente ns serviços, seriam necessárias ns_1^2 estudos de correlação, incluindo aqui as autocorrelações. Tal análise precisaria ser feita continuamente, dada a natureza não estacionária do sistema em questão. Por exemplo, a variação no tamanho de uma estrutura de dados do programa pode mudar a correlação entre dois serviços específicos. O custo de processamento desta solução é proibitivo para muitos sistemas. Análises de tendência neste caso são dificultadas pela dinâmica muita rápida do sistema.

Descrição da Abordagem

A abordagem proposta neste cenário procura aglutinar os possíveis estados do sistema em apenas dois: normal e carregado. A cada momento, o estado do sistema é identificado como normal ou carregado conforme o histórico dos tempos de resposta. Inicialmente o sistema está em estado normal. No momento que uma tarefa t_k termina, o seu tempo de resposta R_k é comparado com o tempo de resposta médio R_s associado com o serviço S. O novo estado do sistema é definido como:

- estado normal, caso $R_k <= R_s$;
- estado carregado, caso $R_k > R_s$.

O valor R_s é definido como uma média móvel, atualizada sempre que uma tarefa conclui a execução do serviço S. O novo valor de R_s, denotado por $R_s(i+1)$, é dado por: $R_s(i+1) = \alpha * R_k + (1-\alpha)R_s(i)$, onde $R_s(i)$ é o valor anterior de R_s, R_k é o último tempo de resposta observado para o serviço S e α é uma constante entre 0 e 1. O propósito

de manter R_s como uma média móvel é duplo: descartar valores antigos dos tempos de resposta e reduzir o esforço computacional, uma vez que R_s é usado pelo algoritmo de previsão toda vez que chega uma nova requisição do serviço S.

Um histórico com os tempos de resposta passados de cada método é mantido em conjunto com a informação sobre o estado do sistema no momento da chegada da requisição. Desta forma, são mantidos na realidade dois históricos em paralelo, um com os tempos de resposta observados com o sistema em estado normal, e outro com os tempos de resposta observados com o sistema em estado carregado. Sempre que uma nova requisição para o serviço S é recebida, é utilizado o histórico de tempos de resposta específico do serviço S considerando os tempos de resposta observados quando o sistema estava no mesmo estado corrente. Associando este histórico à uma PMF, é possível calcular a probabilidade condicional da tarefa atender o *deadline*; $P(R_k \leqslant D_k|estadodosistema)$. De acordo com a probabilidade estabelecida pelo servidor, o cliente é aceito ou rejeitado.

6.6 CONSIDERAÇÕES SOBRE O SERVIÇO DE MONITORAMENTO

A escolha dos mecanismos usados para a extração dos tempos de resposta, assim como aqueles utilizados para elaboração do modelo DGC procura respeitar os principios do desenvolvimento baseado em componentes. A instrumentação de componentes vai de encontro com a separação de parte funcional de componente e infraetrutura. O desenvolvedor de um componente deve ser o especialista da lógica da aplicação, não faz parte do objetivo do componente medir seu tempo de execução.

Uma solução simplificada com relação ao desenvolvimento do componente, seria a simples provisão de parâmetros de configuração via descritores. Entretanto, essa solução exige que a infraestrutura do componente na plataforma alvo (container e servidor de componente) tenha a inforamação sobre como invocar os métodos do componente implantado. A abordagem escolhida procura harmonizar exigências quanto a desenvolvimento e implantação de componentes.

A provisão de facetas para execução dos métodos de componente com parâmetros pré-selecionados não exige do desenvolvedor do componente o conhecimento não relacionado com a lógica de aplicação. Esta abordagem também é suficientemente flexível para que o modelo DGC utilize algoritmos com princípios diferentes: um conjunto de parâme-

tros de pior caso de execução pode ser usado quando o modelo DGC é configurado com algoritmos de tempo real menos flexíveis. Conjuntos de parâmetros diferentes podem ser usados quando outros algoritmos são utilizados.

A seleção de conjunto de parâmetros é condizente com o tipo de garantia a ser fornecida para a aplicação. Para aplicações de tempo real críticas, o modelo DGC é configurado com tempos de resposta correspondentes ao pior caso. Para aplicações compatíveis com abordagens probabilistas, tempos de resposta de pior caso não são adequados. Neste caso, pelo mesmo princípio de seleção de parâmetros por parte do desenvolvedor do componente, poderia-se realizar a escolha de parâmetros para a aplicação como um todo; o instalador do componente, ou o projetista da aplicação, é aquele que tem melhor conhecimento a respeito dos parâmetros mais prováveis a serem fornecidos para a aplicação em tempo de execução. Por conta disto, a configuração de componentes (extração de tempos de resposta) pode ser realizada no contexto da aplicação. Os tempos de resposta coletados são os tempos decorrentes da execução da aplicação com os parâmetros escolhidos pelo projetista. Entretanto, não é possível garantir a correspondência do comportamento da aplicação com o comportamento individual de cada compontente. Por isso a configuração, mesmo sendo realizada após a montagem da aplicação, é feita por componente. Outra vantagem da configuração por componente, é a detecção das condições de execução do componente no nodo. Tempos de resposta altos demais podem indicar a sobrecarga do nodo, esta detecção pode servir de critério para migração de componentes.

A tecnologia de componentes foi projetada para fornecer uma funcionalidade em plataformas heterogêneas e portanto, geralmente desconhecidas em tempo de projeto; característica que dificulta o uso de análise de WCET. O domínio de componentes de software no contexto deste trabalho, é a camada de aplicação, onde estão sujeitos a interferência por parte de camadas subjacentes do sistema. As prioridades das tarefas dos componentes podem determinar um comportamento temporal melhor ou pior conforme a prioridade das demais tarefas de aplicações instaladas no mesmo nodo, mas não com relação às tarefas do sistema. Os tempos de resposta refletem com melhor precisão o comportamento das tarefas da aplicação dentro do contexto em que o modelo DGC pode controlar a qualidade de serviço.

É preciso considerar que a abordagem adotada assume parâmetros simples de serem providos em tempo de projeto. Componentes que usem referências de outros componentes, por exemplo, ou parâmetros

disponíveis somente em tempo de execução não podem fazer uso desta abordagem de configuração dos tempos de resposta.

Idealmente, um serviço de monitoramento de tempos de resposta deve ser mantido durante a execução da aplicação. A atualização dos tempos de resposta deve ocorrer sempre que houver alterações no estado do sistema quanto a carga computacional. Isso garante maior precisão do mecanismo de garantia dinâmica. O uso dos interceptadores para a medição dos tempos de resposta é insuficiente para este monitoramento. Os interceptadores de requisições de cliente e servidor sozinhos não podem reconhecer o fluxo de execução entre componentes e administrar os temporizadores de cada método. As conexões entre as portas de componentes estabelecem possíveis fluxos de execução do servidor, mas não existe especificação que determine, por exemplo, que todos os métodos de uma faceta utilizem o receptáculo do componente. A determinação da *thread* de execução do servidor da aplicação depende de informação de desenvolvimento do mesmo.

6.7 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos um serviço de monitoramento de tempo de resposta para componentes de tempo real. O serviço foi inicialmente elaborado para prover os dados de entrada para o modelo DGC mas pode ser empregado para monitoramento de tempos de resposta quando componentes de estrutura simplificada são adotados.

A partir da pesquisa da literatura de componentes e tempo real, é possível perceber que um serviço com esta finalidade é essencial para qualquer das abordagens já empregadas no controle de comportamento temporal de aplicações. A forma mais comumente empregada nestas abordagens é a adoção de valores padrão como tempo de execução de tarefas ou a premissa de que tais valores serão providos por ferramentas de análise de WCET.

O mecanismo apresentado neste capítulo oferece uma solução prática para um problema bastante comum na provisão de controle de comportamento temporal fim a fim. É claro que este mecanismo se baseia em estimativas e portanto foi desenvolvido no contexto de sistemas de tempo real brando.

7 TRABALHOS RELACIONADOS

Apesar dos esforços de padronização de mecanismos voltados para qualidade de serviço em componentes, apenas um subconjunto mínimo de requisitos são comuns entre os diversos tipos de aplicações de tempo real. A integração de aspectos temporais em componentes de maneira *ad-hoc*, entretanto, resulta em sistemas difíceis de manter e componentes de software difíceis de reusar.

Uma das dificuldades no desenvolvimento de aplicações de tempo real baseadas em componentes é construir componentes como unidades independentes de implantação que permitam à aplicação final atender requisitos temporais fim a fim. O comportamento de tempo real deve ser integrado a cada componente, mas os requisitos temporais são impostos à aplicação. Como unidades independentes de implantação, componentes podem ser desenvolvidos isoladamente, mas para que possam permitir o atendimento de restrições impostas à aplicações do qual farão parte, devem ser configuráveis.

Nem sempre é possível garantir que a configuração de um componente seja suficiente para que o comportamento temporal fim a fim da aplicação seja o esperado, as conexões entre os componentes também afetam o comportamento temporal da aplicação. Considerando a variedade de aplicações de tempo real, é difícil prever os requisitos que possam ser impostos a uma aplicação durante o desenvolvimento do componente. Por isso é difícil determinar o momento correto para a configuração dos componentes, o quanto o componente pode ser configurado e de que maneira realizar a configuração. Estas questões permeiam todo o ciclo de desenvolvimento de aplicações de tempo real baseadas em componentes.

Este capitulo apresenta algumas das abordagens, extensões de modelos, arquiteturas e mecanismos encontrados na literatura que procuram resolver as questões relacionadas ao desenvolvimento de aplicações de tempo real baseadas em componentes. Cada proposta de integração de propriedades temporais ao desenvolvimento baseado em componentes enumera um conjunto de limitações que procura eliminar ou um conjunto de problemas que objetiva superar através de abordagens diferentes.

A descrição de cada trabalho está organizada em quatro grupos nas seguintes seções: frameworks de componentes, ferramentas de modelagem e síntese de sistemas de tempo real baseados em componentes, extensão de modelos de componentes para tempo real, controle

de recursos em sistemas baseados em componentes.

7.1 *FRAMEWORKS* DE COMPONENTES

Nesta seção apresentamos a descrição das características principais das abordagens de componentes RCCF e ACCORD. Estas abordagens propõem seu próprio modelo de componentes, baseado na interpretação do conceito de componentes de software sem comprometimento com nenhuma tecnologia. Além do modelo de componentes, estes trabalhos propõem seus próprios mecanismos de configuração de componentes e integração de aplicações.

O principal obstáculo que motiva estas abordagens é a integração de componentes em uma aplicações com um comportamento de tempo real fim a fim. Neste sentido, a preocupação dos autores das abordagens foi o desenvolvimento de uma arquitetura de componente que permitisse maior capacidade de reconfiguração. Tais componentes podem ser mais facilmente integrados a uma aplicação porque suportam, via configuração, os requisitos específicos da aplicação.

7.1.1 RCCF: *Framework* de Configuração de Componentes de Tempo Real.

O trabalho apresentado em (YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2000, 2002) apresenta um modelo e um *framework* de desenvolvimento de componentes que objetiva a solução de incompatibilidades encontradas durante a fase de integração de aplicações de tempo real baseadas em componentes. A solução proposta inclui: (i) um modelo de componentes que satisfaz, através da capacidade de auto-configuração de aspectos não-funcionais, requisitos temporais da aplicação em construção e (ii) um *framework* que especifica como deve ser feita a extensão de componentes para efetivar a auto-configuração dos mesmos.

No contexto do RCCF, a configuração de componentes é um processo de modificação de propriedades não-funcionais de componentes objetivando satisfazer requisitos de uma aplicação em particular (sistema final) ou permitir a integração de componentes que ofereçam suporte a propriedades temporais diferentes. A extensão do *framework* do componente é o processo de adição de novas propriedades configuráveis e adição de serviços que facilitem a configuração destas propriedades. Este processo capacita o *framework* a lidar com vários tipos de

aplicações (com diferentes restrições temporais).

Conforme (YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2000, 2002), a integração de componentes em sistemas de tempo real é ainda mais complexa devido à importância de restrições temporais impostas ao sistema. Os componentes selecionados para compor uma aplicação de tempo real devem apresentar propriedades (tais como número, prioridade e sincronização de *threads*) que permitam à aplicação final satisfazer seus requisitos temporais.

A proposta inclui um *framework* de componentes, uma técnica de configuração de componentes e uma técnica de extensão de *framework* de componente. Embora seja extensível, o *framework* de componentes enfoca aplicações de tempo real que utilizam técnicas de escalonamento de prioridade fixa como o algoritmo de Taxa Monotônica e *Deadline* Monotônico.

O *framework* de componente apresentado em (YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2002), consiste de duas seções: espaço de solução (*RTSolution*), que provê a funcionalidade principal do componente conforme publicado nas interfaces base do componente e espaço de meta-serviço (*RTManager*), que implementa serviços necessários para configuração de tempo real tal como publicado por meta-interfaces. O espaço de solução é único para um componente particular, ele implementa a funcionalidade do componente. O espaço de meta-serviço é comum a todos os componentes que estejam em conformidade com o RCCF.

O RCCF disponibliza:

- Interfaces apropriadas para especificação e recuperação de aspectos relacionados a tempo real, tais como tempo de execução no pior caso, modelo de *threads*, prioridades de *threads*, compartilhamento de recursos e protocolo de sincronização de tarefas;

- Suporte à separação do componente em funcionalidade principal e parte não-funcional;

- Suporte à extensão sistemática do *framework* padrão. O RCCF oferece suporte ao processo de configuração para que este seja adaptável para manipular a configuração de novos tipos de propriedades;

- Suporte à reflexão de serviços de configuração providos por componentes. Ele também permite que seja possível descobrir quais as propriedades configuráveis suportadas pelo componente e pelo seu *framework*.

As interfaces de componentes do RCCF são divididas entre **interfaces base** e **meta-interfaces**. As Interfaces Base oferecem acesso às funcionalidades dos componentes enquanto as meta-interfaces de componentes provêem serviços para configuração de propriedades não funcionais. A separação objetiva tornar o sistema mais fácil de ser compreendido, desenvolvolvido e configurado. A idéia é que características da parte não funcional do componente possam ser configuradas e aperfeiçoadas após o desenvolvimento da parte funcional por outros especialistas, e que este processo não interfira com a funcionalidade principal do componente.

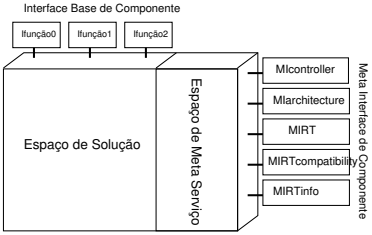


Figura 17: Arquitetura do *framework* de configuração de componentes de tempo real.

As principais funções dos meta-serviços são distribuídas entre as seguintes meta-interfaces:

- **MIcontroller:** Controla a adição e remoção de meta-interfaces;
- **MIarchitecture:** Provê serviços para recuperação de informação hierárquica de um componente composto. Também é responsável por manipular a propagação hierárquica de configuração e reflexão;
- **MIRT:** Provê serviços para facilitar a configuração, inclui um serviço que resgata informações sobre propriedades do componente que podem ser configuradas (reflexão) e um serviço que aplica instruções de configuração ao mesmo.
- **MIRTcompatibility:** A partir da especificação, realiza verificação de compatibilidade do componente e a meta-interface sendo adicionada;
- **MIRTinfo:** Publica informações sobre características de tempo

real tais como pior tempo de execução, prioridade de *threads*, sincronização de *threads*, etc.

Neste *framework* operações são distribuídas em diferentes interfaces: (i) operações de tempo de execução do componente (como serviço de invocação); (ii) operações de integração; (iii) e operações de manutenção (como análise de escalonabilidade, configuração, extensão de funcionalidade).

A camada de gerenciamento de tempo real é responsável pelas seguintes operações:

- Implementar as interfaces comuns de componentes de tempo real;
- Repassar invocações de serviços geradas para o objeto ou subcomponente apropriado;
- Gerenciar referências e outras informações de tempo de projeto sobre os objetos ou subcomponentes e conectores que residem na camada *RTSolution*;
- Implementar diversos algoritmos de verificação de compatibillidade para identificar se o componente é apropriado para ser usado em uma arquitetura específica;
- Implementar o algoritmo de configuração necessário e aplicá-lo durante a integração.

O espaço de meta-serviços especifica as propriedades temporais de cada método que compõe o componente. Esta parte do componente também contém as funções de verificação de compatibilidade, serviço de reflexão de arquitetura (informações a respeito da estrutura interna do componente) e serviço de modificação da arquitetura (extensão do componente). O controle de inserção de novas propriedades e serviços também é realizado no espaço de meta-serviço. Os serviços providos pelo componente são especificados quanto ao tempo de execução, recursos requeridos pelos métodos (que implementam os serviços do componente) e funções de consumo destes recursos. Da mesma maneira, os eventos gerados e publicados pelos componentes através de suas interfaces são detalhados com informações de freqüência e período de geração dos mesmos.

Extensão de Framework

A extensão do *framework* de componente especifica o que pode ser configurado no componente. Os autores propõem a automatização do processo de extensão dos componentes a fim de adequa-los a uma determinada aplicação de tempo real.

O processo de extensão depende de pontos de configuração (*custompoints*) presentes no código do componente. O uso de pontos de configuração é baseado no conceito de *pointcut* usado em AspectJ (KICZALES, Gregor; HILSDALE, Erik; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G., 2001). Os pontos de configuração determinam a localização, dentro do componente, em que é possível realizar a configuração - a modificação do estado interno do componente.

Uma ferramenta de extensão (descrita em (YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2002)) de *framework* permite aos desenvolvedores de componentes, especificar novos pontos de configuração e adicionar novas interfaces de serviço de configuração ao *framework*. A ferramenta gera, automaticamente, código de componente compatível com os novos pontos de configuração.

Configuração de Componente de Tempo Real

A configuração de componente é um processo de modificação de características não funcionais do componente candidato a fim de que este atenda requisitos temporais da aplicação em construção. No caso das aplicações que usam o algoritmo Taxa Monotônica (LIU, Chien-Liang; LAYLAND James W., 1973), por exemplo, a prioridade de *thread*, o protocolo de sincronização, etc. devem ser configurados para garantir o atendimento do deadline de todas as *threads*.

Para que o processo de configuração seja capaz de atender a vários tipos de aplicações e requisitos, os autores adotaram uma abordagem baseada em meta-código. O meta-código é uma instrução que detalha com exatidão como as propriedades internas do componente são manipuladas até atingir a configuração desejada.

O processo de configuração do componente consiste de 3 passos:

- O integrador de componentes escreve um PSMD (*Priority Scheduling Meta Descriptor*) para descrever a configuração desejada;
- O processador de meta-descrição dentro da ferramenta de configuração gera automaticamente o meta-codigo de baixo nível baseado no PSMD (gerado no primeiro passo);
- O processador de meta-código da ferramenta de configuração aplica o meta código ao componente candidato.

7.1.2 ACCORD: Aspectos e Componentes no Desenvolvimento de Sistemas de Tempo Real

O trabalho apresentado em (TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer, 2004) introduz o conceito de desenvolvimento de sistemas de tempo real baseado em componentes orientado a aspectos, AOSD- *Aspect-Oriented Software Development*. De acordo com os autores, determinados aspectos de tempo real não podem ser encapsulados em componentes com interfaces bem definidas. Aspectos como sincronização, otimização de memória, consumo de energia, atributos temporais permeiam muitas camadas sistema e não podem ser encapsuladas. Por isso, no contexto deste trabalho, o código relacionado com o comportamento de tempo real de uma aplicação é encapsulada vários em módulos denominados aspectos, que deve ser combinado ao código que implementa a funcionalidade da aplicação.

A utilização de AOSD e CBSD exige a provisão de métodos de análise de comportamento temporal individual de aspectos e componentes. Além de métodos para análise temporal eficiente para diferentes configurações de componentes e aspectos.

Outro obstáculo na adoção desta combinação de abordagens está na forma com que componentes são tratados nas duas abordagens de desenvolvimento. Enquanto CBSD assume que componentes são caixas pretas (informações internas de componentes não são visíveis), AOSD trabalha com componentes como caixas brancas (componente pode ser visto internamente). Portanto, é necessário o suporte para a integração de aspectos no código de componentes preservando o encapsulamento de informações (*information hiding*) deste.

A utilização das duas técnicas de engenharia de software, AOSD e CBSD, em conjunto implica na adoção de uma abstração que possibilite preservar as principais características do componente de caixa preta (CBSD), ao mesmo tempo que permite a inserção de aspectos que modifiquem o comportamento e estado interno de componentes (AOSD).

O trabalho de (TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer, 2004) investiga um método de projeto para integrar duas técnicas de engenharia de software (AOSD e CBSD) em sistemas de tempo real. O modelo proposto a partir desta investigação é o ACCORD -*Aspectual Componente-based Real-time System Development*.

O método de projeto do ACCORD decompõe sistemas de tempo real em componentes e aspectos e provê um modelo de componentes

de tempo real (RTCOM) que oferece suporte à noção de tempo e restrições temporais; restrições de gerenciamento de espaço de memória e recursos; e composição. O RTCOM objetiva oferecer reusabilidade e a propriedade de composição de software de tempo real através da combinação de aspectos e componentes.

Desenvolvimento de software orientado a aspectos

No desenvolvimento orientado a aspectos, as implementações de sistemas de software possuem, além de componentes e aspectos, um combinador (*weaver*) de aspectos; um compilador especial que combina componentes e aspectos durante um processo denominado combinação (*weaving*). Aspectos são comumente considerados uma propriedade de um sistema que afeta seu desempenho ou semântica, e permeiam (*crosscuts*) a funcionalidade do sistema (KICZALES, Gregor; LAMPING, John; MENHDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina; LOINGTIER, Jean-Marc; IRWIN, John, 1997). Aspectos de software, tais como persistência e depuração podem ser descritos separadamente e trocados de maneira independente um do outro sem atingir a estrutura modular do sistema (FLEISH, W., 1999).

Nas linguagens orientadas a aspectos, cada declaração de aspecto consiste de *advices* e *pointcuts*. Um *pointcut* implica em um ou mais pontos de junção. Um ponto de junção (*join point*) por sua vez, é o ponto no código do componente onde o aspecto deve ser incorporado (ex. um método, um tipo). Um *advice* é uma declaração usada para especificar o código a ser executado quando o ponto de junção especificado pela expressão de *pointcut* é alcançado.

O método de projeto do ACCORD

Para que seja efetivamente aplicado, o ACCORD utiliza um método de projeto constituído por:

- Um processo de decomposição com duas fases seqüenciais: (i) decomposição de sistemas de tempo real em um conjunto de componentes, e (ii) decomposição de sistemas de tempo real em um conjunto de aspectos; A fase de decomposição do sistema de tempo real em um conjunto de componentes é guiada pela necessidade de unidades substituíveis de funcionalidades que sejam fracamente acopladas, porém com forte coesão. Na segunda fase, a decomposição do sistema de tempo real em um conjunto de aspectos, são trabalhados os requisitos não funcionais e objetivos que não se

restringem ao sistema de tempo real (como gerenciamento de recursos e atributos temporais). Terminado o projeto, componentes são implementados com base no RTCOM.

- Componentes, definidos como artefatos de software que implementam um número de funções bem definidas e possuem interfaces bem definidas.

- Aspectos, definidos como propriedades de um sistema que afetam seu desempenho ou semântica, e não estão restritos a uma funcionalidade do sistema.

- Um modelo de componentes de tempo real (RTCOM) que descreve um componente de tempo real e oferece suporte a aspectos mas também reforça o encapsulamento de informações (*information hiding*). O RTCOM é projetado especificamente para: (i) permitir um processo de decomposição eficiente, (ii) oferecer suporte à noção de tempo e restrições temporais, e (iii) possibilitar análise eficiente de componentes e do sistema composto.

Aspectos em sistemas de tempo real

Os aspectos em sistemas de tempo real podem ser classificados em aspectos de aplicação, de execução e de composição (TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer, 2004).

Os **aspectos de aplicação** são aqueles que modificam o comportamento interno de componentes através de alterações no código. Esta classe engloba aspectos relacionados à configuração de componentes de acordo com a funcionalidade da aplicação que compõem. Os aspectos de aplicação influenciam ou modificam a estrutura e comportamento de componentes. De acordo com esta abordagem, propriedades e políticas de tempo real, otimização de memória, sincronização e segurança são aspectos de aplicação.

Os **aspectos de execução** se referem aos aspectos do sistema de tempo real monolítico (resultante da integração do sistema em desenvolvimento ao ambiente de execução). São aspectos considerados críticos e que fornecem informações necessárias para o sistema em tempo de execução a fim de que se possa garantir que esta integração não comprometerá o comportamento temporal ou consumo de memória.

Logo, é preciso que cada componente declare sua demanda por recursos em seu aspecto de demanda de recursos e tenha informações a respeito de seu comportamento temporal contidos em seu aspecto

de restrições temporais. Informações a respeito de compatibilidade com determinadas plataformas, sistemas operacionais de tempo real, informações sobre hardware para o qual oferece suporte são também necessárias (aspecto de portabilidade).

Os aspectos de execução garantem a previsibilidade do sistema composto, facilitam a integração ao ambiente de execução e garantem portabilidade para diferentes plataformas de hardware ou software. Esta classe de aspectos não modifica o código do componente. Ela pode ser definida como construções de fase de projeto independentes de linguagem que encapsulam objetivos relacionados ao comportamento do componente em relação ao seu ambiente de execução.

Os **aspectos de composição** descrevem: com quais componentes um componente pode ser combinado (aspectos de compatibilidade), a versão do componente (aspecto de versão) e possibilidades de estender o componente com aspectos adicionais (aspecto de flexibilidade). Esta classe de aspectos pode ser definida como uma construção de fase de projeto, independente de linguagem que encapsula objetivos relacionados com as necessidades de composição de cada componente. Aspectos de composição não alteram o código do componente.

A idéia por trás da separação de aspectos é facilitar a composição de sistemas pelo uso de uma biblioteca de aspectos. Durante a composição do sistema monolítico novos aspectos poderiam ser adicionados, flexibilizando e melhorando a integração do sistema ao ambiente de execução.

O modelo de componentes de tempo real (RTCOM)

Em (TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer, 2004), a especificação do componente RTCOM é apresentada através de notações que estabelecem a relação entre o componente e suas funções e suas dependências com relação à outros componentes. As definições não permitem, por exemplo, que exista recursividade cíclica entre operações de componentes e provêem a simplificação necessária para que análises de pior caso sejam realizadas para estes componentes.

A definição de aspectos de aplicação é influenciada por dois requisitos: preservar ao máximo o encapsulamento de informações do componente e habilitar a análise temporal dos componentes resultantes da combinação de aspectos. Para satisfazer tais requisitos, a noção de mecanismo é usada como bloco de construção de aspectos de aplicação. Esta definição possibilita o uso de linguagens de aspectos para

implementação de sistemas de tempo real e permite que compiladores de aspectos (*weavers*) existentes sejam usados para a integração de aspectos em componentes mantendo a previsibilidade do sistema. Os valores de WCET são calculados em termos de mecanismos, e a inclusão destes mecanismos sob a forma de aspectos permite o cálculo do novo valor de WCET do componente resultante.

Os autores deste trabalho utilizam definições das linguagens orientadas à aspectos, cada definição de aspecto consiste de *advices* e *pointcuts*. Um *pointcut* consiste de um ou mais *join points*. E um *join point* se refere ao ponto no código do componente em que um aspecto deve ser inserido.

Para facilitar a implementação de aspectos de aplicação em componentes, o projeto da parte funcional tem início com a implementação de operações. Após identificados os aspectos do sistema (execução, composição e aplicação) é recomendado a construção de uma tabela que relacione aspectos e componentes. A tabela reflete o efeito de um aspecto de aplicação de um componente sobre os demais componentes do sistema e objetiva auxiliar o projetista nos passos subseqüentes de projeto e implementação. Em seguida, métodos de componente são implementadas usando os mecanismos como blocos de construção. Os métodos implementados definem a política inicial do componente; funcionalidade básica e genérica do componente.

A combinação de aspectos de aplicação ao código do componente não modifica a implementação de operações, somente a implementação de métodos dentro de componentes. Portanto, métodos são a parte flexível do componente pois sua implementação pode ser modificada pela combinação de aspectos de aplicação, enquanto que operações são partes fixas da infraestrutura do componente.

Interfaces RTCOM

O RTCOM oferece suporte a três tipos diferentes de interface (Figura 18): (i) interface funcional, (ii) interface de configuração, (iii) interface de composição.

As interfaces funcionais de componentes são classificadas em duas categorias; interfaces providas e interfaces requeridas. Interfaces providas são um conjunto de operações que o componente oferece para outros componentes ou para o sistema. Interfaces requeridas são um conjunto de interfaces que o componente requer de outros componentes.

A interface de configuração é voltada para integração do sistema de tempo real com o ambiente de execução. Esta interface fornece in-

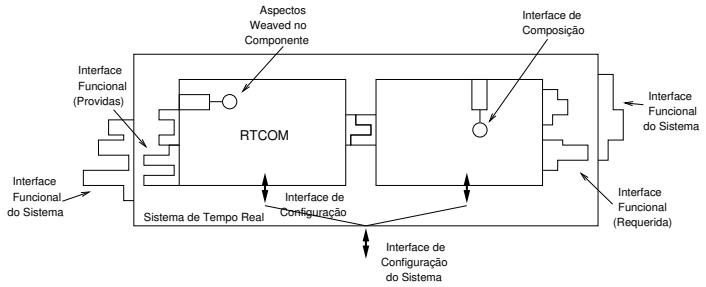


Figura 18: Interfaces oferecidas pelo RTCOM.

formações de comportamento temporal de cada componente, e reflete aspectos de execução do componente. A combinação de vários componentes resulta em um sistema que também apresenta uma interface de configuração, e permite ao projetista inspecionar o comportamento do sistema através do ambiente de execução.

As interfaces de composição correspondem a pontos de junção e são embutidas na parte funcional do componente. O compilador de aspectos identifica as interfaces de composição e as usa para inserção de aspectos. Interfaces de composição são ignoradas enquanto os componentes são compilados caso não sejam necessárias, sendo ativadas somente quando determinados aspectos da aplicação são agregados ao sistema. Assim, interfaces de composição permitem a integração de componentes e da parte de aspectos do sistema. A inserção de aspectos ao sistema pode ser feita em componentes ou no sistema resultante.

7.2 FERRAMENTAS DE MODELAGEM E SÍNTESE DE SISTEMAS DE TEMPO REAL BASEADOS EM COMPONENTES

7.2.1 CoSMIC: Síntese de Componente usando Computação Integrada à Modelo

O CoSMIC - *Component Synthesis using Model Integrated Computing* (GOKHALE, Anirudda; NATARJAN, Balachandran; SCHMIDT, Douglas C.; NECHYPURENKO, Andrey, 2002) é uma ferramenta para composição e implantação de aplicações distribuídas de tempo real baseadas em *middleware*. Esta ferramenta oferece suporte para o desenvolvimento de aplicações baseadas em componentes CCM. O CoSMIC é projetado

para (1) modelar e analisar a funcionalidade da aplicações e para (2) sintetizar meta-dados de implantação necessários para prover e certificar propriedades de qualidade de serviço fim a fim. A idéia por trás deste trabalho é a de integrar abordagens dirigidas por modelo ao desenvolvimento de aplicações distribuídas de tempo real.

As ferramentas do CoSMIC provêem a modelagem de: requisitos, políticas de gerenciamento, comportamento, e interações entre componentes. A linguagem utilizada é similar ao ESML - *Embedded Systems Modeling Language* (KARSAI, Gabor; NEEMA, Sandeep; BAKAY, Arpad; LEDECZI, Akos; SHI, Feng; GOKHALE, Aniruddha, 2002). Dois desafios em particular são o alvo do CoSMIC: (1) satisfazer múltiplos requisitos de qualidade de serviço simultaneamente e (2) lidar com complexidades decorrentes da integração de sistemas de grande escala.

De acordo com os autores, a complexidade de requisitos de qualidade de serviço, a heterogeneidade do ambiente em que componentes são implantados e a existência de sistemas e dados legados são alguns dos fatores que dificultam a integração de uma aplicação com consistência de comportamento temporal fim a fim. As ferramentas CoSMIC são projetadas para modelar e analisar a funcionalidade e os requisitos de qualidade de serviço com o objetivo de superar este obstáculo. As ferramentas do CoSMIC permitem:

- Modelar requisitos de qualidade de serviço usando UML;
- Associar o modelo com diferentes perfis de qualidade de serviço (estático e dinâmico);
- Simular e analisar comportamento dinâmico;
- Sintetizar a funcionalidade de aplicações com propriedades de qualidade de serviço em *assemblies* de componentes.

A síntese de componentes de *middleware* CCM com propriedades de qualidade de serviço para aplicações distribuídas de tempo real segue as seguintes etapas:

- Modelagem da aplicação como um todo usando as ferramentas visuais de modelagem do CoSMIC e especificação dos requisitos de qualidade de serviço da aplicação como restrições. Este passo define e particiona a funcionalidade e requisitos necessários por cada módulo da aplicação baseado no modelo geral da aplicação;
- Composição de servidores de componentes usando ferramentas do CoSMIC para combinar *assemblies* de componentes existentes e

particionamento ou definição da funcionalidade de novos componentes conforme a necessidade.

- Modelagem e síntese componentes. Caso novas implementações de componentes sejam necessárias, cada uma delas é modelada com as ferramentas do CoSMIC e posterioremente, sintetizadores de implementação geram o código a partir deste modelo;

- Validação e simulação de aplicações via ferramentas CoSMIC que verificam se uma composição da aplicação implementa suas definições de modelo corretamente;

- Implantação do sistema resultante para teste e ajuste através de ferramentas que fazem o ajuste fino dos requisitos de qualidade de serviço do CIAO para *assemblies*. Posteriormente, iterações deste processo podem ser usados para ajustes e realimentação para aperfeiçoamento do modelo geral do sistema.

O CoSMIC é fundamentado na idéia de que as dificuldades encontradas durante o processo de integração de sistemas de software são decorrentes da possibilidade de incompatibilidade de configuração de requisitos de qualidade de serviço de diferentes componentes. A garantia de satisfação de requisitos de qualidade de serviço fim a fim em aplicações de tempo real distribuídas exige explicitamente, a configuração de políticas complexas ou *plugins* de *middleware* específicos. A especificação e configuração manual destas políticas é propensa a erros e pode impossibilitar o processo de desenvolvimento.

A fim de oferecer suporte para requisitos de qualidade de serviço não previstos pela implementação do componente de *middleware*, o CoSMIC pode sintetizar módulos de *middleware* usados pelo CIAO para configurar seu comportamento e assim oferecer suporte a propriedades de qualidade de serviço não nativas necessárias por outros sistemas. O *framework* do CIAO pode então usar estes módulos específicos para configurar servidores de componentes antes de implantar os componentes.

7.2.2 Cadena: Ambiente de Desenvolvimento de Arquitetura de Componente para Sistemas Aviônicos

O Cadena (HATTCLIFF, John; DENG, William; DWYER, Matthew B.; JUNG, Georg; RANGANATH, Vekatesh, 2003) - *Component Architecture*

Development ENvironment for Avionics systems - é um ambiente integrado para desenvolvimento, análise e verificação de sistemas baseados em componentes. Este ambiente oferece suporte para construção e modelagem de componentes CCM e é empregado em aplicações baseadas no *framework Bold Stroke* da Boeing.

As aplicações alvo deste trabalho são aplicações que funciona como centro de controle de missão para um piloto de aeronave; ela gerencia *displays* da cabina do piloto, sensores táticos e de navegação e armamentos. É uma classe de sistema complexa que possui *deadlines* brandos e *deadlines* críticos, envolve grandes quantidades de processamento periódico e aperiódico e oferece suporte a milhares de modos operacionais.

Devido ao porte e complexidade deste tipo de software, o processo de manutenção requer constantes atualizações no decorrer de vários anos. Ao mesmo tempo em que exige que todas as etapas do processo de desenvolvimento sejam refeitas, a atualização objetiva preservar ao máximo o *software* legado para redução de custos e riscos.

O Cadena aplica técnicas de análise de modelos de alto nível baseado em estados e ferramentas que automatizam esta análise para o desenvolvimento de componentes (HATTCLIFF, John; DENG, William; DWYER, Matthew B.; JUNG, Georg; RANGANATH, Vekatesh, 2003). A principal motivação desse trabalho é a construção de um ambiente robusto suficiente para o desenvolvimento de sistemas reais que têm como objetivo alcançar efetivamente a aplicação de análise estática, verificação de modelo e outros métodos formais leves (ex: baseados em formulários e descritores) em sistemas baseados no CCM. O cadena provê:

- Uma coleção de formulários de especificação leves que pode ser anexado à IDL para especificar dependências intra-componentes e semântica de transição de estado de componentes;

- Ferramentas para análise de dependências que permitem o rastreamento de dependências de eventos e dados inter/intra-componentes. E algoritmos para sintetizar informações baseadas em dependência sob o aspecto de tempo real e distribuição;

- Uma infraestrutura de verificação de modelo dedicada à comunicação baseada em eventos entre componentes através de *middleware* de tempo real. Esta infraestrutura permite que modelos de projeto de sistemas (derivado da IDL CCM, descrições e anotações de montagem de componentes) sejam verificados conforme propriedades globais de sistemas;

- Código Java de *stubs* e *skeletons* gerados pelo compilador *CCM IDL to Java* do OpenCCM;

- Uma ferramenta de implantação CCM dedicada à arquitetura Boeing Bold Stroke (conexão estática de componentes com canal de eventos de tempo real) que permite a geração automática de código de implantação;

- As ferramentas do Cadena fornecem um ambiente de desenvolvimento integrado fim a fim para sistemas Java baseados no CCM.

Embora o CCM permita que componentes sejam criados e conectados (e desconectados) dinamicamente, aplicações Bold Stroke seguem a prática típica de sistemas de missão crítica/segura e empregam a alocação e configuração estática de componentes: componentes podem ser criados e conectados somente na fase de implantação do sistema. Facilidades de configuração estática incremental baseada em gráficos, textos e formulários são disponibilizadas para o desenvolvedor.

Segundo os autores, em aplicações Bold Stroke, a base da computação são os eventos disparados por tempo. O canal de eventos possui a especificação da taxa de disparo de cada evento. Acionar um evento implica no disparo da *thread* que executa o tratador relacionado, que contém chamadas de métodos e publica eventos subseqüentes.

O Cadena combina funcionalidades gráficas que facilitam para o desenvolvedor a construção de descritores que especificam a estrutura de componentes que provêem a funcionalidade da aplicação e requisitos temporais impostos sobre a mesma. Analisadores extraem destas descrições, as dependências entre componentes, as ações sobre as quais as restrições temporais se aplicam e as relações entre os fluxos de controle. Estas informações são traduzidas para analisadores de dependência e ferramentas de síntese de aspectos não funcionais.

O Cadena usa ferramentas do OpenCCM(ObjectWeb Consortium, 2007) para gerar a implementação do sistema. O OpenCCM aplica a estratégia padronizada pela especificação CORBA 3 na geração de IDL 2, a partir da qual boa parte da infraestrutura é gerada automaticamente. Para a verificação de modelo, um sistema de transição parcial é gerado com base em arquivos IDL 3 e descritores Cadena. Este sistema parcial é associado a um modelo de transição do canal de eventos do CORBA para obtenção do modelo completo de comportamento do sistema. O modelo completo de transição e algumas propriedades selecionadas do sistema são submetidas para o verificador de modelo DSpin(DEMARTINI, C.; IOSIF, R.; SISTO, R., 1999) para análise.

A construção de sistemas no Cadena deve seguir os seguintes passos:

1. Carregar uma biblioteca de componentes do domínio específico da aplicação, bem como especificações CPS associadas;

2. Definir novos componentes específicos e associar especificações de comportamento CPS;

3. Configurar conexões entre componentes (com editores da ferramenta);

4. Usar editores de dependência para examinar dependências;

5. Usar ferramentas de síntese de aspectos não funcionais para anexar distribuição e informação sobre freqüências;

6. Especificar propriedades de correção global desejadas;

7. Gerar um modelo de sistema de transição e verificar propriedades de correção através do modelo;

8. Revisar o sistema com base em realimentação obtida da ferramenta de análise.

Além dos aspectos não funcionais definidos na especificação CCM, o processo de desenvolvimento do sistema em aplicações Bold Stroke apresenta ferramentas que auxiliam ou automatizam a atribuição de prioridades ou taxas de execução para portas de consumo de eventos. As ferramentas determinam a localização (distribuição) dos componentes entre os nodos que fazem parte do sistema e identificam oportunidades de mudança de mecanismo de comunicação: entrega de eventos remota assíncrona para chamadas de métodos locais síncronas.

Verificação do modelo

Embora não haja análise de escalonamento no Cadena, as especificações de dependência podem ser usadas para auxiliar no escalonamento estático. Na implementação corrente do Bold Stroke, a análise de escalonabilidade estática é baseada na soma dos custos associados aos caminhos da árvore de chamadas deduzida de conexões entre componentes. Como as especificações de dependência do Cadena eliminam dependências impossíveis (caminhos impossíveis da árvore de chamadas), a estimativa do tempo de execução no pior caso é mais próxima da realidade.

7.3 EXTENSÕES DE MODELOS DE COMPONENTES PARA TEMPO REAL

Nesta seção, apresentamos a descrição de dois trabalhos que estendem o modelo de componentes CORBA para a provisão de comportamento de tempo real. Estes trabalhos apresentam formas de estender o modelo sem comprometer a portabilidade do mesmo com componentes CORBA padrão. Os trabalhos habilitam o uso de descritores (metadados) na configuração de mecanismos da infraestrutura de componentes para o suporte à aplicações de tempo real. Através de descritores são feitas escolha de políticas de *threads*, modelos de prioridades ou contratos para especificação de qualidade de serviço na integração entre dois componentes. O que estes trabalhos oferecem é a definição de novos descritores e uma infraestrutura que seja capaz de aplicar estes metadados na integração e implantação destes componentes.

7.3.1 CIAO: *Component Integrated ACE ORB*

O CIAO é uma implementação da especificação *Lightweight CORBA Component Model* construída sobre o TAO - *The ACE (Adaptive Communication Environment) ORB* que implementa a especificação do CORBA de tempo real(SCHMIDT, Douglas C.; LEVINE, D. L.; MUNGEE, S., 1998). O *Lightweight* CCM é uma especificação baseada no CCM, compatível com a especificação *Minimum* CORBA e desenvolvida para estabelecer um padrão de modelo de componente CORBA para aplicações embarcadas(Object Management Group, 2004b). Atualmente, o CIAO se encontra em sua versão 0.5.6, trata-se de uma plataforma ainda em desenvolvimento mas com constante atualizações.

Em (WANG, Nanbor; GILL, Christopher; SCHMIDT, Douglas C.; SUBRAMONIAN, Venkita, 2004) os autores apresentam uma extensão para o CIAO que permite a configuração de componentes quanto a aspectos de tempo real através de meta-dados. Esta implementação torna a provisão de qualidade de serviço uma parte integral do *middleware* de componentes e ao mesmo tempo desacoplada da funcionalidade do componente.

Descritores XML denominados RTCAD são usados para configuração das aplicações desenvolvidas. Um arquivo RTCAD define conjuntos de políticas que especificam políticas chaves de tempo real e configura recursos para reforçar estas políticas. Os recursos e políticas definidos no descritor RTCAD do CIAO podem ser especificados para

instâncias individuais de componentes. No descritor RTCAD, podem ser definidos diferentes conjuntos de políticas de prioridades, pool de threads, e modelos de políticas (`Server declared`, `Client propagated`).

A abordagem apresentada em (WANG, Nanbor; GILL, Christopher; SCHMIDT, Douglas C.; SUBRAMONIAN, Venkita, 2004) procura aliar a separação de parte funcional e não-funcional promovida pela arquitetura de componentes CORBA à capacidade de configuração de aspectos de tempo real, caracterizando-as como parte não funcional das aplicações desenvolvidas. Este trabalho não propõe um novo modelo de componentes ou framework de desenvolvimento, apenas estende a implementação da infraestrutura de suporte para os componente desenvolvidos em tempo de implantação e execução para que componentes de *middleware* possam utilizar a infraestrutura oferecida pelo ORB de tempo real.

7.3.2 Qedo: Objetos Distribuídos com Qualidade de Serviço

O projeto Qedo - *QoS Enabled Distributed Objects* (RITTER, Tom; BORN, Marc; UNTERSCHTZ, Thomas; WEIS, Torben, 2003) aplica a abordagem dirigida por modelo para a modelagem e implementação de qualidade de serviço em componentes de *middleware*. O trabalho estende o CCM para provisão de suporte de qualidade de serviço genérico. O objetivo deste suporte à qualidade de serviço genérico é permitir que os contratos sejam definidos conforme o domínio da aplicação em desenvolvimento, usando requisitos específicos desta aplicação.

Os contratos são usados em tempo de implantação dos componentes da aplicação. A conexão entre os componentes está sujeita a uma negociação de qualidade de serviço determinada pelos contratos. O Qedo precisa do suporte da ferramenta de implantação responsável pela conexão entre componentes para que a negociação de um contrato ocorra nesta fase. O meta-modelo do Qedo é integrado ao CCM e também à UML.

Uma ferramenta gráfica permite a modelagem da aplicação através de UML, inclusive do contrato específico para esta aplicação. A conversão desta descrição em UML para IDL também é automatizada e utiliza padrões conhecidos e comumente empregados na área de ferramentas de modelagem da área industrial. As aplicações alvo deste projeto são do domínio de telecomunicação por isso o Qedo apresenta um suporte robusto para lidar com fluxo contínuo de dados.

Para suportar os requisitos de qualidade de serviço modelados, o Qedo define uma extensão para a interface de container do modelo CCM e para o *Framework* de Implementação de Componentes (CIF). Interfaces CCM, como a de navegação de componentes, por exemplo, também são estendidas para suportar os contratos de qualidade de serviço. Estas extensões requerem que a implementação do componente interaja diretamente com a interface de qualidade de serviço do container para negociar uma categoria de contrato de qualidade de serviço. Em (RITTER, Tom; BORN, Marc; UNTERSCHTZ, Thomas; WEIS, Torben, 2003), os autores ilustram as extensões às interfaces CCM para acomodar a negociação entre componente e container e um perfil UML para qualidade de serviço.

7.4 CONTROLE DE RECURSOS EM SISTEMAS BASEADOS EM COMPONENTES

Os trabalhos descritos nesta seção objetivam o controle dos recursos de sistemas distribuídos objetivando qualidade de serviço para aplicações deste domínio. Nesta seção descrevemos cinco trabalhos: O HiDRA e *Resource Overlay* são trabalhos baseados respectivamente em controle realimentado e análise de consumo de recursos. Os mecanismos utilizados são apresentados através de formalismos que permitem garantias teóricas sobre o desempenho do sistema. O trabalho apresentado em (KON, Fabio; MARQUES, Jeferson Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis, 2005), a arquitetura integrada para o gerenciamento de componentes distribuídos, *Bulls Eye Target Manager* e QuO implementam o controle dos recursos do sistema distribuído através de uma hierarquia de gerentes de recursos no domínio. Gerentes locais atualizam o estado dos recursos para um gerente global, que toma decisões a respeito da alocação dos recursos do domínio e informam recursos locais para atuarem conforme a decisão tomada.

7.4.1 HiDRA: Arquitetura Hierárquica de Gerenciamento de Recursos Distribuídos

Em (SHNAKARAN, Nishanth; KOUTSOUKOS, Xenofon; LU, Chenyang; SCHMIDT, Douglas C.; XUE, Yuan, 2006), os autores apresentam o HiDRA - *Hierarchical Distributed Resource-management Architecture*, um *framework* baseado em técnicas de controle para o gerenciamento de re-

cursos. O HiDRA usa controle realimentado para prevenir a sobrecarga de processador e rede. Os laços de controle são estruturados de maneira hierárquica; o laço de controle do processador é um laço de controle externo ao laço de controle da rede. Os autores estendem o algoritmo de escalonamento de controle realimentado (*feedback control scheduling*) para gerenciar mais de um recurso de maneira coordenada. Monitores são associados a cada recurso controlado e periodicamente atualizam o controlador com a utilização corrente do recurso. O controlador implementa o algoritmo de controle que implementa as decisões de adaptação para cada recurso para que este atinja a utilização desejada.

A abordagem adotada neste trabalho é projetada para aplicações como sistemas de busca de alvos móveis, em que imagens de um alvo móvel devem ser processadas com a melhor qualidade possível dentro da capacidade de recursos disponíveis. A referência (*set point*) do laço de controle do processador que processa imagens recebidas é especificado durante o início do sistema. A variável de controle deste laço é a utilização do processador, o controlador do processador fornece como dado de entrada para o laço de controle da rede, a taxa de transmissão de imagens. A variável de controle do laço de rede é a utilização da largura de rede e este laço provê para o sistema, um fator de qualidade para o algoritmo de compressão de imagem.

7.4.2 Sobreposição de Recursos Baseado em Componentes

No trabalho apresentado em (WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei, 2005), os autores descrevem uma especificação de serviços de tempo real para um modelo de componentes. Uma camada de software é implementada através da definição de interfaces de tempo real para acesso aos recursos do sistema. A abordagem oferece a garantia do cumprimento de deadlines baseado na definição de classes de serviço. Uma classe de serviço i relaciona um conjunto de métodos $\Theta_{e,i}$ implementados por um componente e, uma função de chegadas $A_{e,i}$ para a invocação dos métodos deste conjunto e um deadline $D_{e,i}$ para qualquer invocação a qualquer método de $\Theta_{e,i}$. Para uma seqüência de invocações dos métodos no conjunto $\Theta_{e,i}$ que não exceda a função de chegadas $A_{e,i}$, o componente e garante um deadline limitado pelo valor $D_{e,i}$. A função de chegada estabelece um número máximo de invocações de métodos durante um intervalo de tempo I. Um tempo de pior caso de chegadas de requisições (*bursty arrival*) denotado por σ e uma taxa de chegadas ρ são utilizadas para

a definição da função de chegada. $A(I) = \sigma + \rho * I$

De acordo com esta abordagem, diferentes classes de serviço podem ser definidas de modo que ofereçam a mesma funcionalidade mas com freqüências de invocação diferentes. Na tabela abaixo, o mesmo conjunto de métodos (θ_1 e θ_2) pode ser executado em diferentes classes de serviço. O serviço de classe 2 pode realizar invocações com taxas maiores mas com restrições menos críticas (*deadline* maior).

classe i	$\Theta_{e,i}$	$A_{e,i}(I)$	$D_{e,i}$
1	θ_1, θ_2	1 + 2I	0,050
2	θ_1, θ_2	2 + 8I	0,250

Tabela 1: Classes de Serviços

Esta é a separação de requisitos de tempo real e funcionalidade oferecida pela abordagem. A relação entre os recursos e as classes de serviços a serem disponibilizadas é de responsabilidade do desenvolvedor do componente.

Na tabela, a especificação da classe de serviço considera um único componente. Quando vários componentes são envolvidos na provisão de uma funcionalidade do servidor, um teste de admissão baseado nas somas de deadlines $D_{e,i}$ e na função de chegada $A_{e,i}(I)$ de todos os componentes e envolvidos devem ser realizados. O controle de admissão é baseado em uma utilização de processador predefinida para cada componente. O mecanismo é configurado de modo que a aplicação administre uma porcentagem da utilização total do processador.

Uma implementação do modelo foi realizada em componentes EJB. A arquitetura do modelo utiliza um módulo de controle de admissão centralizado. Os clientes que desejam acessar qualquer um dos componentes do servidor devem inicialmente requisitar o acesso ao módulo de controle de acesso.

7.4.3 QuO: Objetos de Qualidade de Serviço

O QuO - *Quality Object* (SHARMA, Praveen; LOYALL, Joseph; SHANTZ, Richard; YE, Jianming, 2006) é um *framework* para provisão de qualidade de serviço dinâmico em aplicações distribuídas centradas na rede (*network-centric*) desenvolvidos pela *BBN Technologies*. Neste contexto, a qualidade de serviço dinâmica se refere à alocação e gerenciamento de recursos em tempo de execução de acordo com os requisitos da aplicação.

Essa abordagem foi desenvolvida para aplicações com requisitos temporais que possam sofrer modificações em tempo de execução. A aplicação alvo do trabalho, realiza o gerenciamento de UAVs (*Unmaned Aerial Vehicles*) com diferentes papéis: indicação de danos de batalha, vigilância e rastreamento de alvo. Os papéis estão relacionados a diferentes prioridades e requisitos de tempo real.

Conforme mudanças ocorrem na área sob monitoramento, os UAVs assumem diferentes papéis; um UAV que inicialmente objetiva vigilância poderia assumir a função de rastreamento, por exemplo. Também de acordo com a situação no domínio de monitoramento, os UAVs fornecer imagens de melhor ou pior qualidade. Todo UAV possui um receptor em terra que decodifica o sinal recebido e retransmite para uma central.

O *framework* se baseia no uso de **componentes Qoskets**, como unidades que encapsulam os mecanismos de qualidade de serviço. Todo código relativo a aspectos de tempo real é implementado dentro de um Qosket. O framework estabelece uma hierarquia de categorias de componentes Qoskets conforme o domíno de dados com o qual este Qosket interage. Os componentes Qosket *Managerial*, por exemplo, estão no topo da hierarquia, eles traduzem especificações para requisitos de qualidade de serviço e requisitos específicos da aplicação para políticas de qualidade de serviço. Na aplicação exemplo utilizada, este componente seria instalado no gerenciador dos UAVs, distribuíndo recursos para o gerenciamento dos mesmos conforme uma função de utilização. Os componentes Qosket *Mechanism* estão na base da hierarquia e acessam o controle do sistema e interfaces específicas de cada recurso utilizado pela aplicação. Os Qosket *Mechanism* são usados em cada receptor de dados de UAVs, eles controlam a qualidade da compressão das imagens para posterior transmissão para a central. De acordo com o recurso em questão, o Qosket pode controlar totalmente o recurso ou somente alterar parâmetros do mesmo.

Esta hierarquia entre Qosket exige que um Qosket correto seja acoplado a cada componente da aplicação conforme seu papel na hierarquia. Isto implica na tradução de requisitos do *Managerial* Qosket até os *Mechanism* Qoskets da aplicação.

Além da hierarquia de Qoskets, o *framework* define também padrões de composição, que determinam a relação dos Qoskets de uma aplicação, eles podem se relacionar conforme uma composição hierárquica, paralela ou seqüencial. A composição hierárquica implica na seqüência da transmissão de dados, do Qosket *Managerial* para o Qosket *Mechanism*, uma composição paralela se refere a Qoskets que rece-

bem dados de forma simultânea, e desempenham o controle de qualidade de serviço de forma independente uns dos outros. E a composição sequencial se refere a aplicações que usam a execução de componentes conforme um *pipeline*.

De acordo com os autores, os Qoskets instalados em uma aplicação podem se comunicar com tipos de dados diferentes conforme o componente ou recurso para o qual controlam a qualidade de serviço. Isto pode resultar na incompatibilidade de dados na comunicação entre Qoskets. Como não existe ainda meios para especificar a forma com que cada Qosket deve se comunicar, atualmente, o responsável pela instalação da aplicação na plataforma alvo deve conhecer a lógica da aplicação, ou trabalhar diretamente com o técnico do domínio desta aplicação.

O QuO é um projeto em andamento, várias lacunas do framework ainda não foram trabalhadas. Uma vez que os Qosket podem ser implementados para configurar até mesmo cada recurso de um nodo da aplicação, várias relações de precedência entre Qoskets devem também ser respeitadas, o que torna o processo de desenvolvimento das aplicações mais complexo. A resolução das questões de dependência entre os Qoskets e o problema da incompatibilidade de dados na comunicação entre eles são trabalhos futuros. De maneira geral, o projeto do QuO procura tornar os Qoskets mais independentes das características específicas das aplicações.

Em (WANG, Nanbor; GILL, Christopher, 2004), um trabalho combinando a capacidade do CIAO em prover qualidade de serviço via descritores XML e a capacidade do QuO é apresentado. Neste trabalho, uma extensão do CIAO oferece a infraestrutura necessária para que Qoskets sejam instalados em conjunto com os componentes da aplicação.

7.4.4 Arquitetura Integrada para o Gerenciamento de Dependências de Componentes Distribuídos

Em (KON, Fabio; MARQUES, Jeferson Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis, 2005) é apresentado uma arquitetura integrada para o gerenciamento de dependências de componentes distribuídos. A arquitetura foi implementada com objetos CORBA e objetivam o suporte para computação móvel, aplicações para telefonia celular digital, videoconferências móveis ou navegação na Web usando conexão sem fio, por exemplo.

A arquitetura define objetos configuradores gerados a partir da

especificação do componente da aplicação. A cada componente da aplicação é acoplado um objeto configurador que armazena informações a respeito das dependências do componente. Opcionalmente, os objetos configuradores podem também implementar código específico, provendo, por exemplo, políticas específicas relacionadas às dependências entre os componentes.

O serviço de gerencia de recursos, outra parte da arquitetura, é organizado como uma coleção de servidores CORBA responsáveis por: (1) manter informações sobre a utilização dos recursos em um sistema distribuído, (2) localizar o melhor nodo para executar uma determinada aplicação ou componente baseado em pré-requisitos de qualidade de serviço e (3) alocar recursos locais para aplicações ou componentes.

Este serviço usa gerentes locais em cada nodo do sistema distribuído que tem como função exportar a informação a respeito dos recursos locais para toda a rede. A rede é dividida em *clusters* e cada *cluster* é gerenciado por um gerente de recursos globais. Os gerentes de recursos locais atualizam periodicamente os gerentes de recursos globais.

Os gerentes de recursos locais também realizam o controle de admissão, negociação e reserva de recursos e o escalonamento de tarefas no nodo. O escalonador executa como um processo em nível de usuário sobre um sistema operacional convencional. Funcionalidades para tolerância a faltas também são previstas para a arquitetura.

7.4.5 *Bulls Eye Target Manager*

O *Bulls-Eye Target Manager* (ROY, Nilabja; SHANKARAN, Nishanth; SCHMIDT, Douglas C., 2006) é um serviço de provisão de recursos integrado ao CIAO. O objetivo do *Target Manager* é monitorar os recursos disponíveis em um domínio, prover estas informações para instaladores de novas aplicações no domínio, alocar recursos para os novos componentes instalados, liberar os recursos quando uma aplicação é desinstalada e facilitar a reinstalação de componentes baseado na disponibilidade dos mesmos. O *Target Manager* é estruturado em um serviço centralizado denominado *Target Manager Core*, e múltiplos *TM-Monitors*. O *Target Manager Core (TM-Core)* interage diretamente com instaladores de aplicações fornecendo informações a respeito do estado dos recursos no domínio do sistema. Para isso, recebe informação dos *TM-Monitors* espalhados em cada recurso do domínio, que atualizam o *TM-Core* periodicamente. De certa forma, o *Target Manager* é um

administrador global de recursos do domínio do sistema; todos os recursos a serem monitorados devem ser registrados no *TM-Core* com um identificador único e com a descrição de sua capacidade. Antes que uma aplicação seja instalada, o *Target Manager* é capaz de informar se existem recursos suficientes para esta instalação e quais os melhores nodos do domínio para os componentes da mesma. Em caso de detecção de sobrecarga, o *Target Manager* também pode abortar e desinstalar aplicações de menor prioridade.

7.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou alguns dos principais trabalhos da literatura envolvendo as mesmas áreas de interesse do nosso trabalho. Componentes de tempo real são o tema de trabalhos que vão desde a modelagem de dados até a construção de sistemas operacionais. Os trabalhos escolhidos para este capítulo apresentam contribuições significativas na evolução das pesquisas sobre este tema e influenciam a maneira com que aspectos de tempo real serão tratados no desenvolvimento de aplicações baseadas em componentes.

O conjunto de trabalhos analisados é heterogêneo quanto ao objetivo final, porém, o princípio em que se baseiam cada um dos trabalhos enfatiza um desafio ainda a ser ultrapassados por qualquer tecnologia de componentes atual. A tabela abaixo resume os trabalhos apresentados nesta seção com a descrição do foco principal do trabalho.

O RCCF (YAU, Stephen; TAWEPONSOMKIAT, Choksing, 2000, 2002) e ACCORD (TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer, 2004) propõem modelos de componentes cuja arquitetura seja capaz de suportar a especificação e configuração de requisitos de tempo e também absorver o impacto destes requisitos na parte funcional do componente. Estes trabalhos salientam que aspectos relacionados a tempo real irão muitas vezes interferir com a funcionalidade do componente de maneira que a separação entre parte funcional e parte não funcional de componentes possa ser extremamente complexa senão impossível. Por causa deste enfoque estes modelos propõem o uso de blocos de granularidade ainda menor para a construção dos próprios componentes; blocos relacionados à lógica de aplicação e blocos relacionados a aspectos de tempo real. Desta forma, os blocos de construção previsíveis mais as regras que controlam a combinação destes blocos em um componente são capazes de gerar componentes previsíveis.

A construção destes componentes a partir do uso de blocos exis-

Projeto	Descrição
RCCF	Definição de um modelo de componentes de tempo real
ACCORD	Definição de um modelo de componentes de tempo real
CoSMIC	Ferramenta de modelagem de aplicações CCM
Cadena	Ambiente de desenvolvimento de aplicações
Qedo	Extensão de containers CCM para QoS
Hidra	Controlador de utilização de processador e rede
Resource Overlay	Interfaces de acesso a serviços com requisitos de tempo
QuO	Gerentes de QoS em tempo de execução
Integrated Architecture	Configurador automático de dependências de componentes
Bulls Eye Target Manager	Monitores de utilização de recursos

Tabela 2: Enfoque de trabalhos relacionados

tentes é chamada de configuração de um componente. O objetivo é: (i) a obtenção de componentes e consequentemente aplicações previsíveis por construção e (ii) permitir maior flexibilidade de configuração de componentes sem comprometer a previsibilidade.

Estas abordagens potencializam a configuração de componentes e aumentam a capacidade de lidar com restrições específicas de aplicações de tempo real. Entretanto, também exigem ferramentas específicas para a construção, configuração e integração destes componentes o que implica na não portabilidade dos mesmos. A flexibilidade provida pela inserção de código (mesmo que de maneira controlada) nos componentes exige o conhecimento e alteração da parte funcional do componente.

Assim como nos modelos tradicionais de componentes (como EJB e CCM), o desenvolvimento destes modelos considera os princípios básicos de componentes; existe a separação de uma parte funcional e uma parte não funcional do componente e preocupações com relação a portabilidade destes componentes, por exemplo. As limitações que levaram os autores destes trabalhos (RCCF e ACCORD) a desenvolverem seus próprios modelos são também reconhecidas pelos autores dos trabalhos de extensão do CCM, por exemplo, para tempo real. É fato que alguns requisitos de tempo real permeiam mais de uma camada do

sistema tornando difícil o encapsulamento dos mecanismos para lidar com estas propriedades e também é reconhecido que a independência de etapas desenvolvimento e integração de componentes leva a uma limitação da configuração destes componentes para manter um determinado comportamento de tempo real fim a fim.

De acordo com a abordagem de configuração dos componentes RCCF e RTCOM (ACCORD), aspectos poderiam estender os componentes de maneira que estes pudessem permitir o controle de acesso de componentes ao sistema. A parte não funcional destes componentes opera como um container especializado e extensível quanto aos requisitos de tempo real específicos da aplicação. A parte não funcional, neste caso, é desenvolvida junto com o componente, e grande parte do que é simplesmente provido pela infraestrutura de instalação e integração de componentes CORBA, é implementado no código do componente RCCF e RTCOM. Na realidade, é injusto uma comparação direta entre estas duas abordagens e as tecnologias de componentes CORBA e EJB, por exemplo, ou suas extensões. O RCCF e ACCORD partiram de considerações a respeito de aplicações de tempo real para componentes, e não o oposto. São ambos projetos em andamento, sendo aprimorados para uma sub-área específica de aplicações de tempo real.

O Cadena e CoSMIC são ferramentas que auxiliam o desenvolvimento de aplicações de tempo real automatizando verificações e análises de consistência do projeto da aplicação. Os mesmos desafios no desenvolvimento de aplicações de tempo real baseados em componentes que motivaram as abordagens do RCCF e ACCORD são atacados pelo Cadena e CoSMIC. Estas ferramentas objetivam acompanhar o desenvolvimento de aplicações de grande porte desde a modelagem dos requisitos até geração do código para os componentes, fazendo a análise de dependências e verificando se existem incompatibilidades entre os componentes.

A abordagem proposta pelo HiDRA, o uso de controle realimentado procura adaptar a utilização de cada aplicação aos recursos disponíveis. Isto é possível em aplicações com requisitos negociáveis, que podem ser balanceados em tempo de execução, o que nem sempre é possível com qualquer tipo de aplicação. A dinâmica deste trabalho reside na administração da degradação da qualidade de imagens transmitidas conforme o aumento da carga no sistema de modo que a utilização se encaixe em uma utilização especificada. Nosso trabalho, por outro lado, reforça garantias dadas a um cliente que obtém acesso ao servidor. Uma vez que o cliente é aceito, o serviço é provido dentro dos requisitos estabelecidos, o objetivo do servidor é honrar os requisitos e não tem

poder para modificá-los.

A extensão do CIAO para permitir a configuração de políticas relacionadas a aplicações de tempo real estabelecem um padrão de configuração das funcionalidades do CORBA de tempo real para componentes. Como descrito no capítulo 6, o CIAO foi usado como plataforma de desenvolvimento do protótipo para o modelo DGC. Sozinho, o CIAO não implementa mecanismos para provisão de garantia dinâmica mas oferece suporte ao *Target Manager*.

A abordagem proposta pelo Qedo, embora adequada para algumas aplicações em que a qualidade de serviço é requisito funcional, inevitavelmente acopla a provisão de qualidade de serviço e comportamento adaptativo à implementação da aplicação e portanto compromete a reusabilidade dos componentes. A base da elaboração do nosso modelo, por outro lado, é capacitar o controle de comportamento temporal aperfeiçoando a infraestrutura de suporte ao componente sem comprometer a reusabilidade.

A abordagem adotada pelo QuO, *Target Manager*, *Resource Overlay* e pelo trabalho apresentado em (KON, Fabio; MARQUES, Jeferson Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis, 2005), a arquitetura integrada para o gerenciamento de dependências de componentes distribuídos centralizam o mecanismo que efetivamente implementa as decisões sobre uma nova implantação ou a desinstalação de aplicações de baixa prioridade em um nodo. O QuO, através de sua hierarquia de Qoskets, delega a decisão para o Qosket do topo da hierarquia e a atuação para os Quoskets da base desta hierarquia. O *Target Manager* e o trabalho de (KON, Fabio; MARQUES, Jeferson Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis, 2005) utilizam explícitamente monitores locais, que captam o estado de cada nodo do domínio e atuam nestes nodos conforme as decisões tomadas por um monitor central.

A abordagem de centralização de informações para o controle do comportamento temporal de aplicações apresenta vantagens e desvantagens. A centralização permite a tomada de decisões baseada em uma visão global do domínio, e permite a provisão de serviços como aqueles apresentados em (ROY, Nilabja; SHANKARAN, Nishanth; SCHMIDT, Douglas C., 2006). Entretanto, esta abordagem implica na utilização de monitores nos nodos do domínio para atualização do estado dos mesmos no servidor que controla o mecanismo. Isso requer a construção de monitores que sejam capazes de extrair destes recursos, o tipo de informação que o servidor central precisa para a tomada de decisão. O servidor que toma a decisão também deve ter conhecimento do impacto

da inclusão de uma nova aplicação, a carga computacional que a nova aplicação representará, nos nodos do domínio. Por isso, monitores devem ser especializados para lidar com tipos de recursos em plataformas heterogêneas.

Dentre os trabalhos apresentados na seção 7.4, o QuO é o que apresenta implementação mais avançada. Ou melhor documentação a respeito do estado da implementação. Embora encapsule os mecanismos para o controle de qualidade de serviço dentro de componentes Qoskets, o QuO não consegue manter a independência destes componentes da funcionalidade da aplicação. Cada Qosket funciona como um serviço de qualidade de serviço específico de cada componente da aplicação, e também de cada recurso utilizado por estes componentes. Quanto mais fino o controle oferecido, mais dependente o mecanismo de controle de qualidade se torna da aplicação a ser controlada. Este tipo de relação com componentes de aplicação implica em tipos de relação entre componentes Qoskets, para que estes consigam manter compatibilidade de comunicação. Um dos problemas enfrentados pelo QuO é encontrar o limite da dependência dos Qoskets e os componentes da aplicação e recursos do sistema.

No modelo proposto neste trabalho, não existe um servidor que centralize as informações. A tomada de decisões sobre a aceitação sobre novos clientes é realizada em cada nodo conforme os mecanismos que este nodo implementa para a análise de escalonabilidade. Não é necessário que todos os nodos do domínio implementem um mesmo algoritmo de escalonamento e conseqüente teste de aceitação. O teste de aceitação aplicado se refere à capacidade do nodo quanto aos algoritmos que o escalonador oferece. Existe sim, a necessidade de atualizações de nodos remotos quanto aos tempos de resposta de componentes envolvidos em um serviço, mas não são tão freqüentes quanto às trocas de mensagens necessárias por um mecanismo que emprega uma abordagem centralizada. Para que o nosso modelo ofereça serviços de informação ao responsável pela implantação de aplicações, por exemplo, seria necessário a implementação de outro mecanismo, que efetivamente centralizariam a informação do estado de cada nodo.

O modelo adotado no *Resource Overlay* (WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei, 2005) não utiliza monitores em nodos do domínio da aplicação, todas as requisições ao servidor são recebidas em um único nodo que implementa a decisão. O controle de admissão exige que todas as requisições de clientes se encaixem em uma categoria predefinida de serviço, estabelecida antes que o sistema entre em execução. Tempos de comunicação entre controle central e no-

dos do domínio do servidor ou mesmo o tempo gasto pelo mecanismo em si não são contabilizados. A abordagem proposta no (WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei, 2005) se baseia num modelo formal ainda não traduzido para uma arquitetura de software. O controle de admissão se baseia em premissas difíceis de serem alcançadas em um sistema distribuído, principalmente quando não é estabelecido o meio pelo qual os dados necessários para o controle são obtidos. O modelo DGC utiliza uma abordagem alternativa para o controle de admissão de novos clientes, o gerenciamento da heterogeneidade de plataformas é realizado através de proteção temporal entre nodos (BUTTAZZO, G.; LIPARI, G.; ABENI, L.; CACCAMO, M., 2005), e cada nodo, conforme o estado dos recursos disponíveis e conforme os algoritmos que gerenciam estes recursos, decide sobre a inclusão ou não do novo cliente no sistema.

Em (ROY, Nilabja; SHANKARAN, Nishanth; SCHMIDT, Douglas C., 2006), situações de sobrecarga detectadas são aquelas resultantes da implantação de outras aplicações no mesmo domínio (conjunto de nodos). O *Target Manager* pode informar o instalador de aplicações sobre a situação corrente dos recursos no domínio, mas não cabe ao mecanismo impedir que a aplicação seja instalada em nodos carregados. No contexto do nosso trabalho, a sobrecarga é conseqüência de um número muito alto de clientes conectados ao servidor em tempo de execução. As situações de sobrecarga são evitadas através do controle de acesso por parte de clientes ao invés da desinstalação de aplicações. A eficiência do *Target Manager* depende da qualidade de informação sobre a capacidade e utilização dos recursos no cadastro destes no *TM-Core*. Um dos desafios enfrentados no desenvolvimento deste mecanismo é a tradução deste tipo de informação de forma que o *TM-Core* informar o estado do domíno do sistema para um operador humano considerando a heterogeneidade do domínio e de tipos de informações dos recursos.

8 CONCLUSÃO

8.1 VISÃO GERAL DO TRABALHO

Esta tese apresentou um estudo sobre a implementação e uso de um modelo de programação de componentes que capacita aplicações distribuídas a usarem um suporte para garantia dinâmica de tempo real.

O trabalho pode ser dividido em duas partes básicas. A primeira compreende a descrição do modelo e arquitetura que provêem um controle de acesso à funcionalidades do servidor baseados em um teste de aceitação. O modelo emprega o particionamento de deadline para promover isolamento temporal entre nodos permitindo assim a decisão sobre a aceitação de novos clientes baseado na capacidade individual do nodo.

A segunda parte da tese apresenta um serviço para suporte à configuração do modelo de garantia dinâmica. O serviço de monitoramento captura tempos de execução do componente que podem ser usados como dados de entrada para o DGC. A implementação deste serviço é compatível com o envolvimento das etapas de desenvolvimento e implantação de componentes porque e inclui todos os elementos que contribuem para o tempo de resposta da aplicação.

Ambas as partes da tese; modelo de garantia dinâmica e serviço de configuração do modelo, foram elaborados com intuito de preservar os princípios de desenvolvimento baseados em componentes. Dentro deste limite, os mecanismos procuram extrair o máximo de informação possível a respeito do comportamento temporal da aplicação e, a partir desta informação, preservar as tarefas correntes do sistema não permitindo a sobrecarga do mesmo.

8.2 REVISÃO DOS OBJETIVOS

Esta seção revisa os objetivos da tese apresentados na seção 1.2.

- Descrição detalhada o modelo DGC proposto, de modo que permita sua integração com diferentes padronizações de componentes existentes;
  No capítulo 4 o modelo *Dynamic Guarantee for Components* é apresentado. O modelo apresentado pode ser utilizado para componentes de sistemas que adotem diferentes modelos de tarefas,

algoritmos de escalonamentos e plataformas de execução.

- Implementação de um protótipo que sirva como prova de conceito para o modelo de componentes proposto; que mostre a viabilidade da realização do DGC em um padrão de componentes conhecido. No capítulo 5, apresentamos a tradução do modelo DGC para uma arquitetura que pode ser implementada em diferentes tecnologias de componentes. Os elementos usados na implementação do DGC são comuns a vários outros modelos de componentes. A descrição do protótipo do modelo DGC implementado no CIAO, também é apresentada neste capítulo.

- Medições sobre o protótipo implementado, para avaliar os atrasos introduzidos pelos mecanismos do modelo DGC; Usando o protótipo implementado, experimentos foram executados com diversas configurações. Aplicações compostas por um número variável de componentes foram construídas e executadas. Foi medido o *overhead* provocado pelo uso dos mecanismos do modelo de garantia dinâmica e comparados com a execução da aplicação sem o controle de acesso ao servidor.

- Elaboração do serviço de monitoramento dos tempos de resposta dos métodos dos componentes instalados em um dado nodo, a fim de suportar a provisão de qualidade de serviço baseada em propriedades temporais dos componentes; O serviço de monitoramento apresentado no capítulo 6 foi inicialmente elaborado para prover o modelo DGC com os tempos de resposta das tarefas correspondentes a execução dos componentes. A estimativa de tempo de resposta reflete vários fatores que influenciam o comportamento temporal dos componentes. Quando este serviço pode ser mantido durante o tempo de execução da aplicação, atualizando tempos de resposta, o modelo DGC pode ser aplicado para controle de sobrecarga em aplicações de tempo real probabilista.

8.3 CONTRIBUIÇÕES E RESULTADOS

Esta tese apresentou as seguintes contribuições para a área de middleware para tempo real.

- A definição de um modelo para provisão de garantia dinâmica de tempo real para aplicações distribuídas baseadas em componentes.

O modelo DGC é suficientemente flexível para que possa ser aplicado à modelos de tarefas clássicos no domínio de tempo real. Não é exigido que a aplicação ou a plataforma tenham sido projetadas com requisitos específicos; não coincidentes com requisitos comumente usados por algoritmos populares em plataformas de tempo real. Esta característica permite que um servidor possa usar o DGC mesmo quando sua funcionalidade é mantida por nodos que suportam diferentes escalonadores. A escolha de algoritmos comumente implementados por sistemas existentes permite que o modelo possa ser usado de imediato em tecnologias atuais. O desenho dos sistemas que utilizam o DGC pode se basear nos algoritmos clássicos de escalonamento e utilização de recursos. A flexibilidade alcançada por esta escolha de projeto não pode ser conservada quando o mecanismo é centralizado. Embora a centralização possa simplificar o modelo, esta estratégia cria dependências que vão de encontro ao princípio fundamental do desenvolvimento baseado em componentes.

- A elaboração de um modelo que pode ser adaptado para outras plataformas de componentes que façam uso de containers ou ofereçam suporte a interceptadores de requisições.

  O estudo de componentes de software (descrito no capítulo 3) e a análise da aplicação deste conceito no âmbito de aplicações de tempo real (capítulo 7) suportam o uso do container como ponto ideal para implementação do modelo proposto. O modelo propõe mecanismos que estendem a funcionalidade do container, não exigem a alteração de uma plataforma de componentes ou a implementação de uma nova plataforma. Quase todos os modelos de componentes (conceituais ou implementações) utilizam containers (muitas vezes sob diferentes denominações) porque estes concretizam a separação de parte não funcional e parte funcional permitindo modularidade e reusabilidade. Este trabalho concretizou esta funcionalidade do container dentro do âmbito de tempo real.

- A definição de um serviço de monitoramento de tempos de resposta de métodos de componentes. Este serviço permite a extração de informação essencial a respeito da aplicação para que a predição da carga do sistema seja possível. Embora tenha sido desenvolvido para provisão de dados de entrada para o modelo DGC, o serviço de monitoramento pode ser usado para provisão

de informações para outros mecanismos de controle de comportamento temporal ou mesmo de desempenho.

O serviço de monitoramento de tempos de respostas de métodos de componentes foi desenvolvido como um serviço de container. Como ponto de interceptação da entrada de requisições a um componente, o container é também o melhor candidato para a inclusão de mecanismos para análise de tempo de execução observável de um componente. Tempos de execução são características não funcionais de componentes de software e um elemento fundamental para o teste de escalonabilidade. Principalmente porque esta característica de software é essencialmente dependente do contexto em que o software executa, a aquisição de tempos de execução observáveis devem ser funcionalidades de containers de componentes de tempo real; são projetados com o componente e executam também em tempo de implantação e execução.

- Todas as contribuições apresentadas neste trabalho ilustram mecanismos simples, porém até então ainda não explorados que utilizam a própria infraestrutura de componentes de software para agregar aspectos de tempo real comuns a aplicações de vários domínios (embarcados, distribuídos, críticos ou não críticos). Ferramentas comumente utilizadas para medição de temos de execução, por exemplo, exigem a instrumentação do código da aplicação analisada. Boa parte desta instrumentação é facilitada em aplicações baseadas em componentes, containers são comumente os pontos de entrada e saída do código do componente (início e fim de funções que implementam a funcionalidade do componente). A simplicidade torna o modelo DGC abrangente (aplicável a vários domínios) sem provocar o overhead exagerado sobre a funcionalidade de cada componente.

- Um aspecto importante salientado pelo modelo, é o particionamento ou a separação do controle de sobrecarga por componente. A implantação ou reimplantação de um componente (atualização de versão) provoca impactos somente nos componentes diretamente conectados a este componente, impacto controlado pela própria interface dos componentes envolvidos. Os demais componentes do servidor que não estejam envolvidos com o serviço provido por este componente não sofrem qualquer alteração decorrente da atualização do componente. Este tipo de contenção não seria possível em um modelo centralizado. Como mencionado anteriormente no capítulo de apresentação do modelo, o modelo

foi projetado de forma a considerar a heterogeneidade do domínio do servidor, a diferença de capacidade dos nodos. O modelo objetiva o controle do comportamento de tempo real fim a fim, reforçando o comportamento de tempo real em cada nodo através de proteção temporal (*temporal protection*) entre os nodos. A proteção temporal objetiva não permitir que a falha de um nodo em atender os requisitos temporais de suas tarefas comprometa outro nodo do mesmo servidor. No contexto deste trabalho, nodos cooperam para prover um serviço. A idéia de proteção temporal usada na modelagem das tarefas possibilita um teste de aceitação descentralizado.

8.4 PERSPECTIVAS FUTURAS

O trabalho concretizado nesta tese abre algumas alternativas para temas de trabalhos futuros a serem explorados. Talvez o trabalho mais imediato seja a adequação dos interceptadores de container ao modelo DGC. Os interceptadores de container permitem a simplificação da implementação do modelo. Somente as requisições destinadas ao componente da aplicação passam a ser interceptadas o que diminui a necessidade de implementação de verificações no interceptador. Entretanto, esta é a única modificação necessária para a arquitetura.

Prioridades também devem ser incorporadas ao modelo, a importância das tarefas deve estar relacionadas ao ponto de entrada acessado pelo cliente do componente. Desta forma, um componente pode prover funcionalidades com diferentes prioridades e o mecanismo de garantia dinâmica pode desalocar recursos de clientes com menor prioridade em favor de requisições de clientes com alta prioridade.

Algoritmos de negociação de qualidade de serviço podem ser incorporados ao modelo. Clientes que não tem a requisição atendida podem entrar em negociação com o servidor, que pode oferecer o serviço com requisitos que podem ser cumpridos ou esperar que o cliente redefina os requisitos necessários. Uma pesquisa da literatura desta área é necessária antes da definição da política adotada. Ainda neste sentido, o reparticionamento de deadlines poderia, também ser redefinido conforme a negociação entre cliente e servidor.

O modelo deve ainda ser incorporado aos mecanismos de balanceamento de carga implementado por algumas plataformas de componente. A detecção de sobrecarga no recebimento de uma requisição pode ser usada como critério para os mecanismos de balanceamento de

carga. No CIAO, estes mecanismos são estendidos para automatizar a re-instalação de um componente em outro nodo. Os componentes que sofrem sobrecarga podem ser re-instalados em nodos com maior capacidade computacional ou outras aplicações que executam no mesmo nodo podem ser desalocadas, conforme as prioridades relacionadas ou demanda destas aplicações.

O serviço de monitoramento de componentes pode ser expandido de modo a prover informação suficiente para vários algoritmos probabilistas de previsão de tempo de resposta. O balanceamento entre o *overhead* provocado por este mecanismo e sua funcionalidade é a principal preocupação neste caso uma vez que o serviço executa sempre em conjunto com a aplicação. Os próximos passos com relação a este serviço envolvem o refinamento e validação dos mecanismos empregados.

Os artigos publicados no decorrer do desenvolvimento da tese são listados abaixo:

- Um Estudo das Propostas de Modelos de Componentes para Sistemas de Tempo Real
  WTR'2004 - 6o Workshop de Tempo Real
  Gramado - RS, 14 de maio de 2004.

- Scheduling Approaches for Component-Based Real-Time Distributed Applications
  Workshop on Quality of Service for Application Servers, em conjunto com IEEE 23rd International Symposium on Reliable Distributed Systems
  Florianópolis - SC, 17 de outubro de 2004.

- Dynamic Guarantee in Component-Based Distributed Real-Time Systems
  ETFA'2005 - 10th IEEE International Conference on Emerging Technologies and Factory Automation
  Catania, Italia, 19-22 de setembro 2005.

- Real-time Dynamic Guarantee in Component-based Middleware
  ISORC'2007 - 10th IEEE International Symposium on Object and Component-Oriented Real-time Distributed Computing
  Santorini, Grécia, 7-9 de maio 2007.

O primeiro artigo apresenta os resultados das pesquisas por trabalhos relacionados e foi apresentado em um evento que envolvia projetos relacionados a diferentes tecnologias usadas dentro do âmbito de tempo real. O segundo artigo apresentou nossa proposta de utilização

dos algoritmos de escalonamento de tempo real dentro do contexto de componentes existentes. Somente no terceiro artigo apresentamos o mecanismo do DGC ainda em uma versão preliminar. Finalmente no artigo submetido ao ISORC apresentamos o DGC com a implementação do protótipo e alguns dos experimentos.

REFERÊNCIAS

ALEXANDER, C. **The Timeless Way of Building**. UK: Oxford University Press, 1979.

AUDSLEY, N. C.; TINDELL, K.; BURNS, A.; WELLINGS A. J. Deadline monotonic scheduling theory and application. In: **Control Engineering Practice**. [S.l.: s.n.], 1993. v. 1, p. 71 – 78.

BACHMANN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K. **Technical Concepts of Component-based Software Engineering**. [S.l.], 2000. Tech. Rep. CMU/SEI-2000-TR-008.

BAKER, T. P. Stack-based scheduling of realtime processes. In: **The Jornal of Real-Time Systems**. [S.l.: s.n.], 1991. v. 3, p. 67 – 90.

BERGNER, K.; RAUSH, A.; SIHLING, M. Componentware - the big picture. In: **In Proceedings of the Int. Workshop on Component-Based Engineering**. Kyoto, Japan: [s.n.], 1999.

BUTTAZZO, G.; LIPARI, G.; ABENI, L.; CACCAMO, M. **Soft Real-Time Systems: Predictability vs. Efficiency**. [S.l.]: Springer, 2005.

COSTA, Fabio; KON, Fabio. Novas tecnologias de middleware: Rumo à flexibilização e ao dinamismo. **Minicursos do 20o. Simpósio Brasileiro de Redes de Computadores**, Buzios, Rio de Janeiro, p. 1–61, Maio 2002.

CRISTIAN, Flaviu; AGHALI, Houtan; STRONG Ray; DOLEV Danny. Atomic broadcast: From simple message diffusion to byzantine agreement. In: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**. Ann Arbor, MI, USA: [s.n.], 1985. p. 200–206.

CRNOVIC, Ivica; LARSSON, Magnus. **Building Reliable Component-based Software Systems**. Norwood, MA: Artech House, INC, 2002. ISBN1-58053-327-2.

DEMARTINI, C.; IOSIF, R.; SISTO, R. dspin: A dynamic extension of spin. In: **In Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)**. [S.l.: s.n.], 1999.

DENG,Gan; BALASUBRAMANIAN, Jaiganesh; OTE, William; SCHMIDT, Douglas C.; GOKHALE, Aniruddha. Dance: A qos-enabled component deployment and conguration engine. In: **Proc. of the 3rd Working Conference on Component Deployment**. Grenoble, France: [s.n.], 2005.

FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva. **Sistemas de Tempo Real**. 1st. ed. USP: Escola de Computao 2000, 2000.

FLEISH, W. Applying use cases for the requirements validation of component-based real-time software. In: **in Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)**. Saint-Malo, France: IEEE Computer Society Press, 1999. Lecture Notes in Computer Science, vol. 1241.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns, Elements of Reusable Object-Oriented Software**. MA: Addison-Wesley, 1995. ISBN: 0201633612, pp.39-42.

GOKHALE, Anirudda; NATARJAN, Balachandran; SCHMIDT, Douglas C.; NECHYPURENKO, Andrey. Cosmic: An mda generative tool for distributed real-time and embdedded component middleware and applications. In: **In Proceedings of OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture**. Seattle, WA: [s.n.], 2002.

HATTCLIFF, John; DENG, William; DWYER, Matthew B.; JUNG, Georg; RANGANATH, Vekatesh. Cadena: An integrated development, analysis, and verification environment for component-based systems. In: **ICSE 2003**. Oregon: [s.n.], 2003.

ISOVIC, D.; NORSTRM, C. Components in real-time systems. In: **Proc. of the 8th International Conference on Real-Time Computing Systems and Applications**. Tokyo, Japan: [s.n.], 2002.

JONSSON, B.; BRINKSMA, E.; COULSON, G. Component-based design and integration platforms. In: **Embedded Systems Design: The ARTIST Roadmap for Research and Development**. [S.l.]: Springer, 2003, (LNCS).

JOSEPH, M.; PANDYA, P. Finding response times in a real-time system. **BCS Computer Journal**, v. 29, n. 5, p. 390–395, 1986.

KAO, Ben; GARCIA-MOLINA, Hector. Deadline assignment in a distributed soft real-time system. **IEEE Transactions on Parallel and Distributed Systems**, v. 8, p. 1268–1274, December 1997. No 12.

KARSAI, Gabor; NEEMA, Sandeep; BAKAY, Arpad; LEDECZI, Akos; SHI, Feng; GOKHALE, Aniruddha. A model-based front-end to ace/tao: The embedded system modeling language. In: **in Proceedings of the Second Annual TAO Workshop**. Arlington,VA: [s.n.], 2002.

KICZALES, Gregor; HILSDALE, Erik; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. An overview of aspectj. **Lecture Notes in Computer Science**, Springer-Verlag, v. 2072, p. 327–355, 2001.

KICZALES, Gregor; LAMPING, John; MENHDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina; LOINGTIER, Jean-Marc; IRWIN, John. Aspect-oriented programming. In: AKSIT, M.; MATSUOKA, S. (Ed.). **Proceedings European Conference on Object-Oriented Programming**. Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v. 1241, p. 220–242.

KON, Fabio; MARQUES, Jeferson Roberto; YAMANE, Tomonori; CAMPBELL, Roy; MICKUNAS, M. Dennis. Desing, implementation, and performance of an automatic configuration service for distributed component systems. In: **Software - Practice and Experience**. [S.l.: s.n.], 2005. v. 35, p. 667–703.

KOPETZ, Hermann; VERISSIMO, Paulo. **Real-Time and Dependability Concepts**. [S.l.]: ACM Press/Addison-Wesley Publishing Co., 1993. 441 - 446 p.

LEUNG, J. Y. T.; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In: **Performance Evaluation**. [S.l.: s.n.], 1982. v. 2, p. 237–250.

LIU, Chien-Liang; LAYLAND James W. Scheduling algorithms for multiprogramming in a hard real-time environment. **Journal of ACM**, ACM Press, New York, NY, USA, v. 20, n. 1, p. 46–61, January 1973. ISSN 0004-5411.

MANGHWANI, P.; LOYALL, J.; SHARMA, P.; GILLEN, M.; YE, Jianming. End-to-end quality of service management for

distributed real-time embedded applications. In: **Proc. of The 13th International Workshop on Parallel and Distributed Real-Time Systems**. Denver, Colorado: [s.n.], 2005.

MARVIE, Raphael; MERLE, Philippe. **CORBA Component Model: Discussion and Use with OpenCCM**. 2001. Http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2486.

MEYER, B. **Eiffel: The Language**. Upper Saddlle River: Prentice Hall, 1992.

MOLLER, Anders; AKERHOLM, Mikael; FROBERG, Joakim; FREDRIKSSON, Johan; SJODIN, Mikael. **Industrial Requirements on Component Technologies for Vehicular Control Systems**. [S.l.], February 2006. Disponível em: <http://www.es.mdh.se/publications/900->.

MONTEZ, Carlos Barros. **Um Modelo de Programação e uma Abordagem de Escalonamento Adaptativo usando RT-CORBA**. Tese (Tese (Doutorado)) — Universidade Federal de Santa Catarina, Santa Catarina, Brasil, 2000.

Object Management Group. **Real-time CORBA Specification**. August 2002. OMG Document formal/02-08-02.

Object Management Group. **Deployment and Configuration Specification**. 2003. OMG Document ptc/03-07-08.

Object Management Group. **Common Object Request Broker Architecture**. 2004. OMG Document formal/2004-03-01.

Object Management Group. **Lightweight CORBA Component Model**. May 2004. Ptc/2004-06-10.

Object Management Group. **CORBA Component Model**. April 2006. Formal/06-04-01.

Object Management Group. **Quality of Service for CORBA Components Specification**. September 2006. OMG Document ptc/06-04-15.

ObjectWeb Consortium. **OpenCCM: The Open CORBA Component Model Platform**. 2007. Http://openccm.objectweb.org/.

OLIVEIRA, Rômulo Silva de. **Escalonamento de Tarefas Imprecisas em Ambiente Distribuído**. Tese (Tese de Doutorado) — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, 1997.

RAMAMRITHAN, K.; STANKOVIC, J. A. Scheduling algorithms and operating systems support for real-time systems. In: **Proceedings of the IEEE**. [S.l.: s.n.], 1994. p. 55 – 67.

RAMAMRITHAN, K.; STANKOVIC J. A. Scheduling algorithms and operating systems support for real-time systems. In: . [S.l.: s.n.], 1994. v. 82, n. 1, p. 55–67.

REDMOND, F. E. **DCOM - Microsoft Distributed Component Object Model**. [S.l.]: IDG Books, 1997.

RITTER, Tom; BORN, Marc; UNTERSCHTZ, Thomas; WEIS, Torben. A qos metamodel and its realization in a corba component infrastructrure. In: **in Proceedings of the Hawaii International Conference on System Sciences**. Big Island Hawaii: IEEE, 2003.

ROY, Nilabja; SHANKARAN, Nishanth; SCHMIDT, Douglas C. A resource provisioning service for enterprise distributed real-time and embedded systems. In: **Proc. of the International Symposium on Distributed Objects and Applications (DOA)**. Montpellier, France: [s.n.], 2006.

SCHMIDT, Douglas C.; LEVINE, D. L.; MUNGEE, S. The design and performance of real-time object request brokers. In: **Computer Communications**. Crystal City, VA: [s.n.], 1998. v. 21.

SHA, L.; RAJKUMAR, R., LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. In: **IEEE Transactions on Computers**. [S.l.: s.n.], 1990. v. 39, p. 1175–1185.

SHA, L.; SATHAYE, S. S. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In: **Proceedings of the IEEE**. [S.l.: s.n.], 1994. v. 82, p. 68 – 82.

SHARMA, Praveen; LOYALL, Joseph; SHANTZ, Richard; YE, Jianming. Using composition of qos components to provide dynamic, end-to-end qos in distributed embedded applications - a middleware approach. In: **IEEE Internet Computing**. [S.l.: s.n.], 2006. p. 16–23.

SHAW, Mary; DELINE, Robert; KLEIN, Daniel V.; ROSS, Theodore L.; YOUNG, David M.; ZELESNIK, Gregory. Abstractions for software architecture and tools to support them. **Software Engineering**, v. 21, n. 4, p. 314–335, 1995. Disponível em: <citeseer.ist.psu.edu/shaw95abstractions.html>.

SHIN, A.; RAMANATHAN P. Real time computing: A new discipline of computer science and engineering. In: **Proceedings of the IEEE**. [S.l.: s.n.], 1994. v. 82, p. 6 – 24.

SHNAKARAN, Nishanth; KOUTSOUKOS, Xenofon; LU, Chenyang; SCHMIDT, Douglas C.; XUE, Yuan. Hierarchical control of multiple resources in distributed real-time and embedded systems. In: **Proc. of the 18th Euromicro Conference on Real-Time Systems**. Dresden, Germany: [s.n.], 2006.

SILVA, E. L.; MENEZES, E. M. . **Metodologia da Pesquisa e Elaboração de Dissertação**. Universidade Federal de Santa Catarina, UFSC: [s.n.], 2005.

STANKOVIC, John A. Misconceptions about real-time computing: A serious problem for next-generation systems. **IEEE Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 21, n. 10, p. 10 – 19, October 1988.

Sun Microsystems. **JavaBeans 3.0 Specification**. 2004. Http://java.sun.com/products/ejb/index.jsp.

SZIPERSKI, C. Components vs. objects vs. component objects. In: **In Proceedings of the Object Oriented Programming**. Munich, Germany: [s.n.], 1999.

SZYPERSKI, Clemens. **Real-time object-oriented modeling**. England: Addison-Wesley and ACM Press, 1998. ISBN 0-201-17888-5.

TESANOVIC, Aleksandra; NYSTROM, Dag; HANSSON, Jorgen; NORSTROM, Christer. Aspects and components in real-time system development: Towards reconfigurable and reusable software. **Journal of Embedded Computing**, Cambridge International Science Publishing, February 2004.

VAN OMMERING, R.; LINDEN, F. van der; KRAMER, J. The koala component model for consumer eletronics software. **IEEE Computer**, IEEE, March 2000.

WANG, Nanbor; GILL, Christopher. Improving real-time system configuration via a qos-aware corba component model. In: **In Proc. of Hawaii International Conference on System Sciences (HICSS)**. Honolulu, Hawaii: [s.n.], 2004.

WANG, Nanbor; GILL, Christopher; SCHMIDT, Douglas C.; SUBRAMONIAN, Venkita. Configuring real-time aspects in component middleware. In: **In Proc. of the Conference on Distributed Objects and Applications (DOA 2004)**. Cyprus, Greece: [s.n.], 2004.

WANG, Nanbor; SCHMIDT, Douglas C.; O'RYAN, Carlos. An overview of the corba component model. In: **Component-Based Software Engineering: Putting the Pieces Together**. Essex - England: Addison-Wesley, 2001.

WANG, Shengquan; RHO, Sangig; MAI, Zhibin; BETTATI, Riccardo; ZHAO, Wei. Real-time component-based systems. In: **Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium**. San Francisco, CA, USA: [s.n.], 2005. p. 428–437.

YAU, Stephen; TAWEPONSOMKIAT, Choksing. Component customization for object-oriented distributed real-time software development. In: **Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing(ISORC 2000)**. [S.l.: s.n.], 2000.

YAU, Stephen; TAWEPONSOMKIAT, Choksing. An approach to object-oriented component customization for real-time software development. In: **Proc. of the Fifth IEEE International Symposium on Object-Oriented Real-time Distributed Computing**. [S.l.: s.n.], 2002.