Fábio Favarim

COMPONENTES EM UM ESQUEMA DE TOLERÂNCIA A FALTAS ADAPTATIVA

FLORIANÓPOLIS
2003

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

COMPONENTES EM UM ESQUEMA DE TOLERÂNCIA A FALTAS ADAPTATIVA

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Fábio Favarim

Florianópolis, Março de 2003.

O USO DE COMPONENTES EM ESQUEMAS DE TOLERÂNCIA A FALTAS ADAPTATIVA

Fábio Favarim

'Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, Área de Concentração em *Sistemas de Informação*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

_____
Prof. Joni da Silva Fraga, Dr.
Orientador

_____
Prof. Edson Roberto De Pieri, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

_____
Prof. Joni da Silva Fraga, Dr.
Presidente

_____
Prof. Frank Augusto Siqueira, Dr.
Co-orientador

_____
Prof. Lau Cheuk Lung, Dr.

_____
Prof. Luis Fernando Friedrich, Dr.

_____
Prof. Carlos Barros Montez, Dr.

A meus pais, Laudino e Maria,
por todo amor dedicados
a mim e a meus irmãos.

AGRADECIMENTOS

"E se clamares por entendimento, e por inteligência alçares a tua voz, se buscares como a prata e como a tesouros escondidos a procurares, então compreenderás o temor do Senhor, e descobrirás o conhecimento de Deus. Porque o Senhor é quem dá a sabedoria, e da sua boca vem o conhecimento e o entendimento." Provérbios 2:3-6

A Deus, por ter permitido a conclusão com sucesso desse importante empreendimento.

Meu agradecimento especial aos meus orientadores Joni da Silva Fraga e Frank Augusto Siqueira, por terem me dado a oportunidade de me dedicar a essa área de pesquisa. Além disso, agradeço pela generosidade de dividir comigo suas experiências e conhecimentos, e principalmente pelo encorajamento que me deram durante a realização desse trabalho..

Aos meus pais, pelo apoio constante e incentivo ao estudo e por nunca terem medido esforços para que eu tivesse uma excelente educação.

Ao meu amigo Douglas, que durante todo esse tempo, mesmo de longe sempre me incentivou na conquista desse título.

Aos meus amigos do GOU (Grupo de Oração Universitário) que durante o curso, foram como uma segunda família.

Muito obrigado aos membros da minha banca, pelo tempo despendido na leitura do meu trabalho e por participarem da minha defesa.

A todos os professores, colegas e funcionários do Programa de Pós-Graduação em Engenharia Elétrica e do Departamento de Controle e Automação (DAS), que de alguma forma contribuiram para a realização deste trabalho.

Enfim, a todos que de alguma forma contribuiram para a conclusão deste trabalho, quero expressar meus sinceros agradecimentos.

À CAPES pelo apoio financeiro.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica.

COMPONENTES EM UM ESQUEMA DE TOLERÂNCIA A FALTAS ADAPTATIVA

Fábio Favarim

Março/2003

Orientador: Joni da Silva Fraga, Dr.
Co-Orientador: Frank Augusto Siqueira, Dr.
Área de Concentração: Sistemas de Informação.
Palavras-chave: Tolerância a Faltas, CORBA, CCM, Componentes de Software, Adaptação em Sistemas Distribuídos.
Número de Páginas: xii. + 95

Componentes de software representam um importante passo no sentido de sistematizar a produção de software, além de trazer redução nos custos e no tempo de desenvolvimento do software. O desenvolvimento baseado em componentes de software consiste da composição das aplicações através de um conjunto de partes de software, denominada de componentes. O desenvolvimento baseado em componentes pode ser realizado empregando tecnologias já existentes. Porém, essas tecnologias não dão suporte a tolerância a faltas. Esta dissertação apresenta um modelo de tolerância a faltas baseado em componentes para a construção de aplicações distribuídas. O modelo TFA-CCM permite que requisitos de QoS guiem a seleção da configuração de serviços replicados em tempo de execução, utilizando um conjunto de componentes que tratam dos aspectos não-funcionais da aplicação.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

COMPONENTS IN A SCHEMA OF ADAPTIVE FAULT TOLERANCE

Fábio Favarim

March/2003

Advisor: Joni da Silva Fraga, Ph.D.
Co-Advisor: Frank Augusto Siqueira, Ph.D.
Area of Concentration: Information Systems
Keywords: Fault Tolerance, CORBA, CCM, Software Components, Adaptation in Distributed Systems.
Number of Pages: xii + 95

Software components represent an important step towards automating software production, reducing costs and development time. Component-based development consists in composing applications using software parts, called components. Component-based development can employ existing technologies. However these technologies do not have support for fault tolerance. This master thesis presents a model for building distributed applications with fault-tolerance requirements by using software components. The AFT-CCM model allows that QoS requirements be used to select the configuration of replicated services during execution time, employing a set of components that deal with the non-functional aspects of the application. The characteristics of this model and the results obtained with its implementation are described along this master thesis.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivações e Objetivos

A abordagem de programação baseada em componentes de software sempre foi apontada como base de qualquer ferramenta automatizada ou técnica de sistematização da produção de software, fundamentada na composição de programas a partir de componentes pré-existentes (SZYPERSKI, 1998).

Várias das características encontradas em outros estilos de programação podem ser encontradas nesta abordagem, além dos aspectos de redução nos custos e no tempo de desenvolvimento do software (SZYPERSKI, 1998). Entre estas, uma das mais importantes talvez seja a flexibilidade propiciada pela abordagem de componentes para a programação e manutenção de programas distribuídos. A utilização de elementos auto-contidos e a interação através de interfaces bem definidas fazem de componentes unidades de configuração que permitem a implementação e a manutenção de programas distribuídos de maneira eficiente.

Sistemas de software atuais estão cada vez mais distribuídos e operam em ambientes dinâmicos como a rede mundial – a Internet. As aplicações distribuídas com requisitos de tolerância a faltas são difíceis de serem construídas e mantidas, se consideramos a complexidade e as características destes ambientes. A tolerância a faltas costuma ser construída fazendo uso de suportes de *middleware* e fornecendo mecanismos que permitam a adaptação de técnicas de redundâncias às mudanças nestes sistemas de larga escala.

Nos últimos anos o conceito de componentes vem sendo integrado a várias tecnologias de middleware existentes. Exemplos desses esforços são o CCM (CORBA *Component Model*) (OMG, 2002a), que é parte integrante das especificações CORBA 3.0 (OMG, 2002b); EJB (Enterprise JavaBeans) desenvolvido pela Sun (SUN MICROSYSTEMS, 2001). Estas tecnologias fornecem um suporte limitado para tolerância a faltas, geralmente na forma de mecanismos de persistência de dados.

Este trabalho tem o objetivo de estudar a tolerância a faltas adaptativa como forma de se adequar à mudanças que ocorrem no ambiente de execução. As mudanças podem ter relação direta com a freqüência em que as falhas ocorrem, assim a adaptação de mecanismos de tolerância a faltas de acordo com essas variações, permitirá o nível de confiabilidade e disponibilidade do sistema seja mantido.

A configuração dinâmica é usada neste trabalho como instrumento para prover a tolerância a faltas adaptativa. Um nível de confiabilidade ou de disponibilidade requisitado pela aplicação pode ser mantido com diferentes configurações de replicações, alocando somente os recursos necessários para a obtenção dos requisitos desejados.

O presente trabalho se propôs desde o seu início a examinar a possibilidade de utilização de técnicas adaptativas de tolerância a faltas tomando como base a tecnologia de componentes. Diante disto desenvolvemos o TFA-CCM – Tolerância a Faltas Adaptativa baseada em Componentes CORBA, que implementa um modelo de suporte a tolerância a faltas adaptativa totalmente transparente à aplicação.

O TFA-CCM é formado por componentes de software que são responsáveis por implementar técnicas de tolerância a faltas, definindo e controlando o comportamento de um serviço replicado. Esse modelo fornece um conjunto de componentes que permitem fazer o monitoramento, a detecção e a reconfiguração de aplicações distribuídas. O modelo apresenta soluções práticas para integrar requisitos de qualidade de serviço (QoS) que devem nortear a seleção da configuração de serviços replicados. Deste modo, diferentes níveis de qualidade de serviço (QoS) podem ser especificados, visando atender diferentes requisitos de tolerância a faltas. O TFA-CCM tem mecanismos no sentido de reconhecer a necessidade de reconfigurar e de efetivar mudanças no sistema visando atender os requisitos de QoS sem que aspectos como desempenho e estabilidade sejam duramente comprometidos.

1.2 Estrutura da Dissertação

Esta dissertação está dividida em seis capítulos. Este capítulo inicial descreveu o contexto onde o trabalho está inserido, a sua motivação e os objetivos.

O capítulo 2 contém os conceitos relacionados a tolerância a faltas. Além disso, são apresentadas as abordagens para que a tolerância a faltas seja alcançada.

O capitulo 3 apresenta os conceitos sobre componentes de software. O capítulo também apresenta dois modelos de componentes distribuídos, o EJB (*Enterprise JavaBeans*) e o CCM (CORBA *Component Model*). Ao fim do capítulo, são feitas algumas comparações entre os dois modelos apresentados.

O capítulo 4 apresenta uma discussão detalhada sobre a proposta do modelo de tolerância a faltas adaptativa proposto.

O capítulo 5 discute alguns aspectos relacionados à implementação do modelo de tolerância a faltas adaptativa proposto no capítulo 4. Medidas de desempenho, no sentido de avaliar o modelo também são apresentadas neste capítulo.

Finalmente, o capítulo 6 apresenta as principais conclusões deste trabalho.

Capítulo 2

Tolerância a Faltas

A grande profusão de serviços disponíveis em redes de computadores nos últimos anos, principalmente na Internet, tem aumentado as expectativas dos usuários em relação a qualidade de serviço. Dentro destas expectativas estão a necessidade de confiabilidade, disponibilidade e o desempenho destes serviços, que caracterizam qualidades de serviço normalmente fundamentadas sobre a idéia de "replicação". Ou seja, a utilização de técnicas de replicação é uma solução usual para aumentar a disponibilidade e o desempenho de serviços oferecidos em sistemas distribuídos.

O oferecimento contínuo de serviços mesmo na presença de falhas parciais caracteriza o que se entende como disciplina de tolerância a faltas. Técnicas de replicação e outros mecanismos usados na manutenção da disponibilidade dos serviços fazem parte da fundamentação dessa disciplina.

Neste capítulo são apresentados alguns conceitos de tolerância a faltas usados neste documento. A classificação das faltas segundo seus efeitos também é abordada. Na seqüência as técnicas de replicação utilizadas em sistemas distribuídos são mostradas, bem como os modelos clássicos que fazem uso de replicações para a tolerância a faltas em sistemas distribuídos. Por fim, são introduzidos os conceitos que envolvem tolerância a faltas adaptativa.

A terminologia em português na área de tolerância a faltas é ainda bastante controversa. Neste texto os termos e definições estão baseados em um trabalho conjunto entre o INESC de Portugal e o LCMI/UFSC (VERÍSSIMO; LEMOS, 1989).

2.1 Faltas, Erros e Falhas

Alguns autores definem o conceito de "Segurança de Funcionamento" como a qualidade de serviço que faz com que usuários tenham confiança nos serviços de um sistema (LAPRIE, 1992). A tolerância a faltas, faz parte do suporte não funcional que acrescenta esta qualidade.

Para se entender as imperfeições (falta, erro e falha) às quais está sujeito o funcionamento de um sistema, é importante a compreensão do que é o serviço de um sistema. O serviço de um sistema é o comportamento do mesmo observado por seus usuários. Ou seja, a percepção de serviço é a percepção de funcionamento do sistema observada através de sua interface. Qualquer subsistema, programa ou pessoa caracteriza o "usuário do sistema", através de trocas que faz com o mesmo fazendo uso da interface do sistema.

O serviço dito falho é aquele que não está de acordo com suas especificações. As possíveis causas de falhas são as **faltas** (*fault*), fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionadas. Em qualquer uma destas situações, a existência de faltas em um componente[1] do sistema leva a manifestações internas chamadas de **erros** (*error*), que caracterizam um comportamento errôneo do sistema. Os erros quando não tratados podem resultar em uma **falha** (*failure*) do sistema (manifestação externa), ou seja, a observação através da sua interface o desvio do serviço oferecido do serviço especificado. A Figura 2.1 ilustra o relacionamento entre falta, erro e falha.

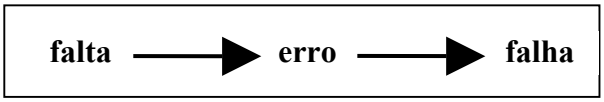

Figura 2.1 – Falta, Erro, Falha.

A ocorrência de falhas parciais (falhas de componente) em um sistema distribuído pode ser causada por erros provocados a partir de *hardwares* faltosos, por erros de *software* devido a elementos faltosos provenientes de um projeto mal sucedido (faltas de projeto) ou por causas externas ao sistema (ex: queda de energia). Se um serviço distribuído não estiver preparado para lidar com a ocorrência desses problemas, poderá trazer prejuízos ao usuário do serviço.

[1] Um componente pode ser um módulo, um objeto, um processo, etc.

Um componente de um sistema distribuído é dito correto (não faltoso) se, para uma dada entrada, produz uma saída que está de acordo com as especificações. Ou seja, uma saída é considerada correta se está dentro das expectativas do usuário e se entregue dentro de um limite de tempo especificado.

Para ilustrar, considere um computador que fornece serviços de acesso ao banco de dados de uma biblioteca, que tem uma memória com defeito em alguns de seus *bits*, esta falta pode provocar a interpretação errada da informação armazenada. Pode-se notar que uma falta pode resultar em nenhum erro. Aqueles *bits* da memória podem nunca ser usados e portanto, a falta pode não se manifestar sob a forma de um erro.Além disso, um erro nem sempre conduz a uma falha de serviço.

Várias faltas podem resultar em um mesmo erro. Isto faz com que a atividade de identificação de uma falta que ocasionou um erro seja uma tarefa difícil. No exemplo citado acima, se nos *bits* do dispositivo faltoso fossem gravados informações a respeito da autorização de acesso a um determinado serviço, a negação de acesso a usuários autorizados poderia ocorrer no sistema.

2.2 Classificação das Faltas

É muito importante a classificação de faltas levando em conta as "semânticas de falhas" associadas que definem as maneiras pelas quais um serviço pode falhar. É a partir da identificação dos modos de falhas de um sistema que se dimensiona os mecanismos para tolerar e tratar suas faltas. As semânticas de falhas são definidas segundo domínios de tempo e de valor, o que leva à seguinte classificação (CRISTIAN, 1991):

- **Falhas por parada (*crash*)**: são aquelas falhas que causam o travamento ou a perda do estado interno do componente defeituoso. Com este tipo de falha, um serviço nunca se submete a uma nova transição de estado. Um exemplo desse tipo de falha ocorre quando a máquina em que um processo está executando fica travada, bloqueando a execução de seus códigos.

- **Falhas por omissão:** são aquelas que fazem com que um serviço não responda a algumas requisições de serviço. O comportamento de atender algumas requisições e não responder a outras é intermitente ou transitória.

- **Falhas por temporização**: são aquelas que fazem com que um componente responda a uma requisição de serviço fora do intervalo de tempo especificado. Estas falhas podem ser por antecipação, quando a ocorrência de um serviço é percebida antes do tempo especificado, ou por atraso, quando o serviço ocorre após o tempo especificado. Um exemplo deste tipo de falha é quando uma rede de comunicação está congestionada e a mensagem demora a chegar ao seu destino (falha por atraso). Vale ressaltar que uma falha de temporização só se caracteriza se a especificação do sistema impuser restrições temporais ao serviço. Falhas de temporização também são chamadas de falhas de desempenho na literatura.

- **Falhas por valor**: são aquelas que ocorrem quando uma resposta é devolvida com o valor fora do especificado, porém dentro do intervalo de tempo especificado. Um exemplo desse tipo de falha é quando uma mensagem é corrompida (alterada) na rede de comunicação.

- **Falhas Arbitrárias**: esta categoria de falhas engloba todas as classes de falhas citadas acima. Nas falhas arbitrárias um componente tem as suas falhas envolvidas com erros nos domínios de valores e de tempo, simultaneamente ou não. Exemplos desse tipo de falha são componentes que mandam valores diferentes pertencentes ao domínio de valores especificado na mesma comunicação com diferentes destinatários (alguns autores citam como falhas de dupla face). O envio de valores diferentes para diferentes receptores, correspondendo a mesma mensagem, implica que os valores sejam todos especificados como dentro do domínio de valores para que não sejam detectados por mecanismos baseados em sintaxes. Este tipo de falha também é chamada de **maliciosa** ou **bizantina**.

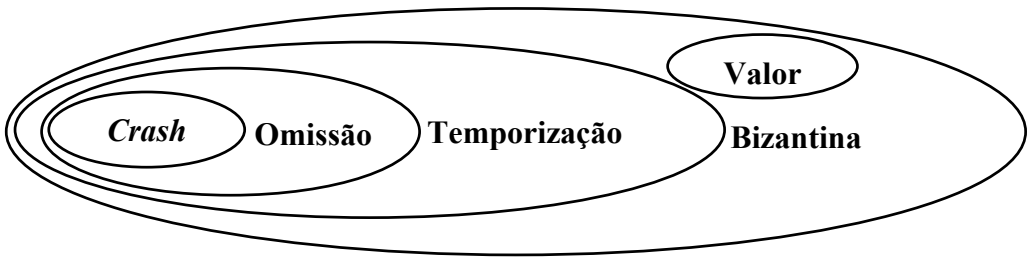

Figura 2.2 – Classificação da semântica de falhas em Sistemas Distribuídos.

A Figura 2.2 apresenta a classificação das falhas apresentadas nesta seção de acordo com sua severidade. A falha por *crash* é a mais restritiva das classes de falha enquanto a falha arbitrária representa a menos restritiva.

2.3 Tolerância a Faltas

Tolerância a faltas é uma propriedade ou qualidade do sistema que tenta garantir os serviços de forma ininterrupta, de acordo com a sua especificação, mesmo na presença de faltas, através do uso de redundância (JALOTE, 1994). As redundâncias podem ser de software, hardware ou no tempo. A tolerância a faltas envolve normalmente várias fases. Geralmente, são identificadas as fases de detecção de erros, confinamento de erros, processamento de erros e tratamento de faltas (ANDERSON, LEE, 1981, LAPRIE, 1992):

- **A detecção de erros:** é a fase mais importante da tolerância a faltas; é a partir da detecção que os mecanismos que compõem a tolerância a faltas são ativados. As técnicas de detecção envolvem testes (normalmente testes não completos). A literatura é bastante vasta na descrição de testes nos vários níveis de um sistema. A detecção de um erro não é imediata ou mesmo completa em um sistema. Erros podem não ser detectados ou mesmo apresentar uma latência em suas detecções.
- **Confinamento de erros:** significa delimitar a propagação de erros, a fim de evitar que se propague para o restante do sistema, pois quando um erro é detectado, outros componentes já podem ter sido influenciados pelo mesmo. Isto é decorrente do atraso existente entre a manifestação da falha e a detecção do erro (latência do erro). Durante este período, muita informação incorreta pode ter sido disseminada entre os elementos do sistema, levando à ocorrência de outros erros. Portanto, antes de tentar recuperar, é necessário avaliar a extensão do dano causado ao sistema. Técnicas de estruturação modular de sistemas com definição de interfaces apropriadas são usualmente empregadas no sentido de conter a disseminação de erros.
- **Processamento de erros:** depois de identificado o erro e o estrago causado por este, é necessário eliminar os mesmo a fim de levar o sistema novamente para um estado livre de erros. As técnicas envolvidas no processamento de erros se dividem em **compensação de erros** e **recuperação de erros** (ANDERSON, LEE, 1981, LAPRIE, 1992):

- **Compensação de erros:** envolve a redundância de informação ou de processamento para mascarar os efeitos de elementos faltosos eventuais. As técnicas de compensação de erro são baseadas na replicação ativa de componentes.

- **Recuperação de erros:** A recuperação de erro consiste em substituir um estado incorreto por um estado livre de erros. A recuperação pode ser de duas formas: recuperação por retrocesso (*backward error recovery*) e recuperação por avanço (*forward error recovery*). Ambos os tipos estão apresentados na Figura.2.3.

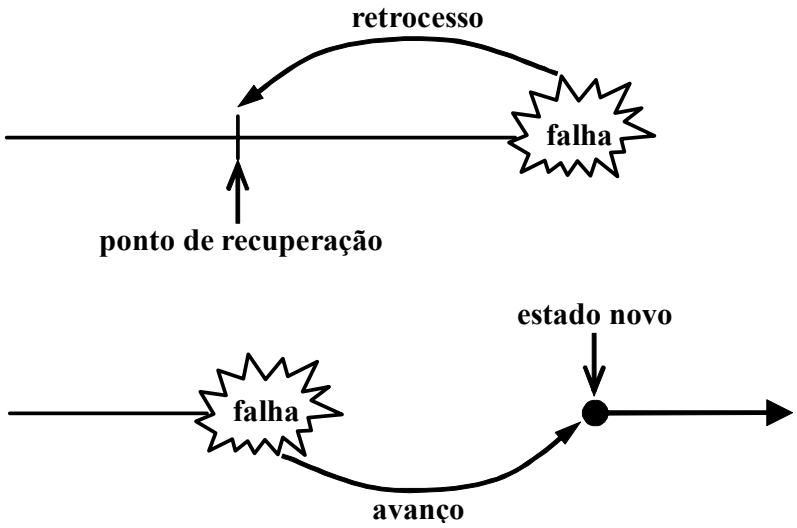

Figura.2.3 – Recuperação em retrocesso e em avanço.

Na recuperação por retrocesso, o estado do sistema é retornado a um estado anterior. Desta forma, a recuperação por retrocesso do erro exige que informações de estado do sistema sejam regularmente salvas em pontos específicos de execução do serviço, denominados de pontos de recuperação (*checkpoint*), em um repositório de memória estável. Quando da detecção de um erro, o estado do sistema é restaurado com os valores do último ponto de recuperação estabelecido.

Na recuperação por avanço, nenhum estado anterior está disponível. A recuperação do erro consiste da tentativa de transformar o estado falho em um estado futuro, a partir do qual o sistema retorna a sua operação normal.

- **Tratamento de Falhas:** Embora a técnica de recuperação de erros tenha retornado o sistema para um estado livre de erros, ainda são necessários procedimentos que visam impedir a reativação de uma falta. O importante é garantir que a mesma falha não se repetirá imediatamente, retornando o sistema para um estado errôneo. Assim, após o processamento de um erro, é necessário identificar os elementos faltosos, de modo a recuperá-los ou mesmo retirá-los do sistema (tratamento de faltas).

2.4 Técnicas de Replicação

Técnicas de replicação contribuem na melhora da confiabilidade, no aumento da disponibilidade e possivelmente na melhora do desempenho do sistema. Técnicas de replicação são uma alternativa para que serviços continuem a ser oferecidos em sistemas distribuídos, mesmo na presença de falhas. O uso de replicação torna possível implementar serviços tolerantes a faltas, onde o usuário possa ter abstraído as dificuldades de interação e da manutenção do serviço replicado. Ou seja, é criada uma interface única de serviço replicado onde o usuário interage como se estivesse ativando o serviço de um componente único.

O uso de técnicas de replicação requer a utilização de protocolos de coordenação no sentido de manter a consistência de estado e a transparência do conjunto. Estes protocolos também são responsáveis pelo controle da concorrência e pela recuperação em situação de falha parcial ou total das réplicas. Os protocolos são distintos para as diferentes abordagens de replicação.

São três as técnicas de replicação normalmente identificadas na literatura (POWELL, 1991): Replicação Passiva, Ativa e Semi-Ativa. As abordagens de replicação passiva e semi-ativa isolam elementos faltosos de um serviço replicado do sistema, enquanto que a abordagem de replicação ativa é capaz de mascarar falhas parciais de réplicas do conjunto. Para a escolha de cada uma das abordagens é necessário levar em conta o tipo de aplicação, a classe de faltas que se deseja tolerar e as características do sistema distribuído (LUNG, 2001).

2.4.1 Replicação Ativa

Nesta abordagem todas as réplicas não faltosas recebem as requisições, processam de

forma paralela e produzem as mesmas saídas, conforme ilustrado na Figura 2.4. A técnica de replicação ativa é também chamada de Máquina de Estados (SCHNEIDER,1990). Para garantir a consistência de estados das réplicas implica na necessidade de garantir o determinismo[2] das réplicas. Nesta abordagem, em caso de falha, o resultado correto pode ser obtido no mesmo instante, uma vez que as réplicas que não apresentam falhas já realizaram o processamento, não havendo a necessidade de recuperar o estado das réplicas.

Diferentes estratégias podem ser usadas para enviar o resultado para o cliente. Pode ser enviado o primeiro resultado que chegar; ou todos os resultados podem ser concatenados em seqüência e enviados ao cliente; ou os resultados passam por um votador que seleciona o resultado que a maioria votou.

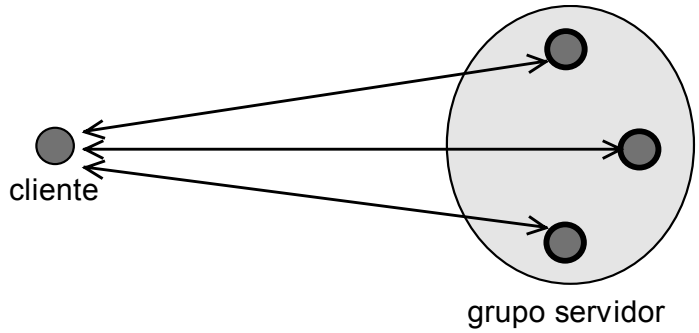

Figura 2.4 – Replicação Ativa.

Esta abordagem é capaz de tolerar todos os tipos de falhas (parada, omissão, temporização, valor ou arbitrária). É a abordagem mais apropriada para aplicações que necessitam de serviços ininterruptos com sobrecarga mínima em situações de falha, como as aplicações de tempo real, pois as falhas parciais são mascaradas.

Um sistema usando essa abordagem deve especificar o grau de replicação baseado na semântica de falhas que irá suportar. Seja t o número máximo de réplicas faltosas, têm-se as seguintes especificações:

- **falhas arbitrárias e de valor:** a replicação deve possuir pelo menos $2t + 1$ réplicas, para que o resultado do processamento replicado possa ser considerado válido (CRISTIAN, 1985). Para que ambos os tipos de falhas sejam mascarados, o

[2] Uma replicação é determinista ou formada por réplicas deterministas se réplicas idênticas e não faltosas, partindo do mesmo estado inicial e processando as mesmas entradas, produzirem as mesmas saídas (SCHNEIDER, 1990). O determinismo de réplicas é uma condição para a consistência de estado entre réplicas.

cliente deve receber o resultado correspondente ao da maioria correta: $t + 1$ réplicas corretas. A decisão do resultado depende de uma votação majoritária sobre os valores produzidos a partir de cada réplica do conjunto.

- **falhas por parada e omissão:** nestas semânticas são necessárias pelo menos $t + 1$ réplicas formando a replicação, uma vez que basta apenas uma única réplica enviar o seu resultado ao cliente (JALOTE, 1994). Para essas classes de falhas o mecanismo de replicação ativa torna-se mais simples por não exigir nenhuma técnica de votação, sendo apenas necessário algum tipo de protocolo que faça com que somente um resultado seja utilizado. Esse tipo de processamento replicado pode ser implementado de duas formas (POWELL, 1991):
  - um protocolo é executado entre as réplicas corretas com objetivo de decidir qual delas enviará o resultado ao destino; ou
  - todas as réplicas enviam seus resultados ao destino, que seleciona um resultado ignorando os demais.
- **falhas de temporização:**
  - **por atraso:** nesta semântica são usadas as mesmas premissas das falhas por parada e por omissão em relação ao número de réplicas: $t + 1$ réplicas formam o conjunto.
  - **por antecipação:** nesta classe de falhas são necessárias $2t + 1$ réplicas, a exemplo do que já acontecia com as falhas arbitrárias e por valor.

2.4.2 Replicação Passiva

Na replicação passiva apenas uma réplica (a primária) recebe, processa e responde as requisições. As outras réplicas são passivas (*backup*) e têm seus estados atualizados periodicamente a partir da réplica primária, através de mecanismos de *checkpointing* (Figura 2.5). Caso a réplica primária falhe, uma réplica *backup* assumirá a função de primária (POWELL, 1991). Uma destas réplicas *backups* deve, a partir do último *checkpoint*, executar as requisições dos clientes posteriores ao *checkpoint* até chegar ao estado da antiga réplica primária.

Em relação à abordagem de replicação ativa, as replicações passivas apresentam um

custo de processamento menor, uma vez que somente uma réplica realiza o processamento de uma requisição de serviço. Além disso, esta abordagem tem a vantagem de não necessitar do determinismo de réplicas, uma vez que o primário impõe seu estado sobre as réplicas *backups* periodicamente.

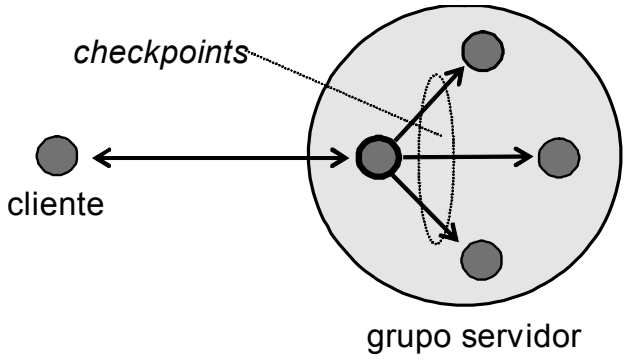

Figura 2.5 – Replicação Passiva.

Esta abordagem é limitada a um espectro de falhas mais restritivo: tolera falhas parciais por parada e por omissão. Também não é a replicação mais apropriada para aplicações que necessitam de serviços ininterruptos, pois existe uma sobrecarga relevante em tempo de execução para a recuperação do estado da primária, no caso de falha da mesma.

2.4.3 Replicação Semi-Ativa

A abordagem de replicação semi-ativa apresenta características de ambos os modelos anteriores: as réplicas são todas ativas, porém sendo uma privilegiada (POWELL, 1991). Uma representação gráfica dessa abordagem é apresentada na Figura 2.6. A replicação semi-ativa tenta eliminar as limitações das duas outras técnicas anteriores, como o uso de protocolos de difusão atômica para garantir o determinismo das réplicas na replicação ativa e o atraso gerado devido a recuperação na replicação passiva.

Nesta abordagem todas as réplicas executam os pedidos de serviço e somente a réplica privilegiada é responsável por impor a ordem de execução dos pedidos como também pela resposta aos clientes. Quando a réplica privilegiada falha, uma das réplicas restantes assume o seu papel. Como nesta abordagem todas as réplicas são ativas, não ocorre a recuperação de estado baseado em retrocesso. Este modelo permite a execução de funções não deterministas, uma vez que as decisões são impostas pela réplica privilegiada.

Esta abordagem suporta as semânticas de falhas por parada, omissão, temporização e valor. No entanto para essa última classe é necessário um mecanismo de votação na réplica líder. O comportamento bizantino do líder não é tolerado neste tipo de replicação.

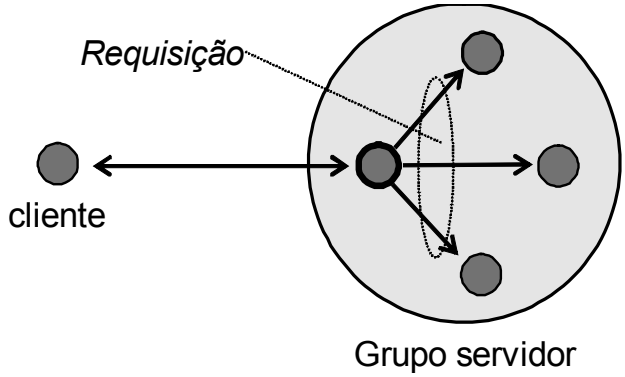

Figura 2.6 – Replicação Semi-Ativa.

2.5 Modelos de Replicação

Vários modelos têm sido introduzidos com o propósito de fornecer tolerância a faltas e que de certa forma se apresentam como extensões das abordagens citadas acima. A seguir apresentamos alguns modelos mais difundidos e presentes na literatura, examinados segundo a ótica das três abordagens apresentadas (ativa, passiva e semi-ativa).

2.5.1 Modelos de Replicação Ativa: Replicações Competitiva e Cíclica

Nestes modelos, todas as réplicas processam os pedidos de serviço, isto é, todas são ativas, porém somente uma responde ao cliente. Em ambos modelos, cada réplica tem um controlador associado (Figura 2.7) responsável pela recepção, difusão e comparação de mensagens, ficando a réplica correspondente dedicada ao processamento das requisições. Todas as mensagens trocadas entre controladores são transmitidas através de um protocolo de difusão atômica.

Estes modelos de replicação toleram falhas por temporização, o que inclui as semânticas de falha por parada, omissão e temporização por atraso, e também toleram falhas arbitrárias, que compreendem todo o espectro de falhas (POWELL, 1991).

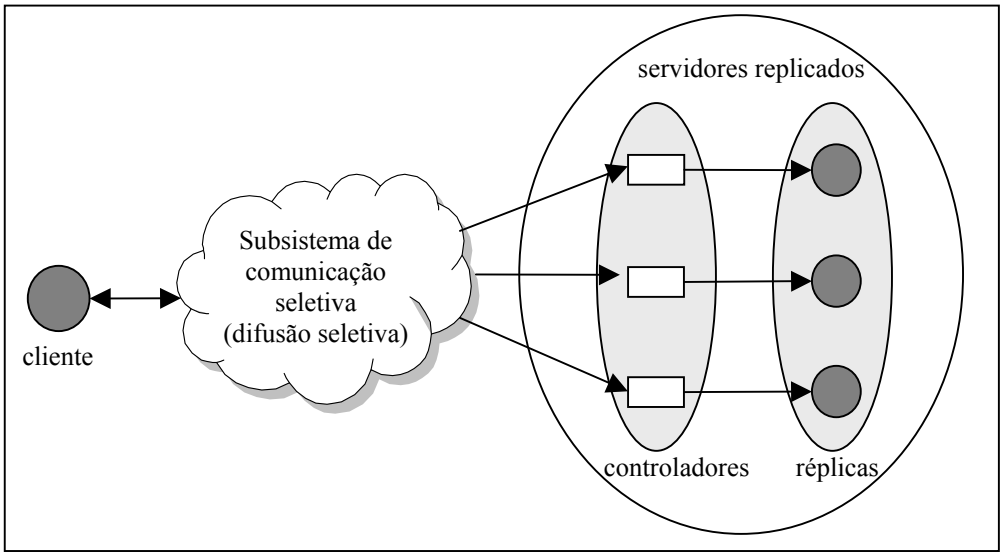

Figura 2.7 – Controladores e Réplicas (Replicação Ativa)

2.5.1.1 Replicação Ativa Competitiva (POWELL, 1991)

Este modelo é caracterizado pela competição entre as réplicas, onde somente a mais rápida responde a requisição.

Na hipótese de faltas por temporização, a requisição do cliente é difundida aos controladores do grupo, os quais a repassam às réplicas associadas. No recebimento da resposta de sua réplica, o controlador verifica se já recebeu uma mensagem de outro controlador do grupo. Se não possuir, o controlador concatena um identificador à mensagem de resposta e a envia (mensagem e identificador) para os controladores do grupo. Se o controlador recebe a sua mensagem, como a primeira difundida entre o grupo de controladores, saberá que sua réplica é a mais rápida e, portanto, terá que enviar a resposta ao cliente. No caso em que ao receber a sua mensagem que foi difundida no grupo de controladores, já possua uma mensagem de outro controlador, saberá que não é o primeiro e descartará sua mensagem.

No caso de faltas arbitrárias, é acrescido um mecanismo de comparação com a finalidade de validar o resultado a ser enviado ao cliente. Para que t falhas sejam toleradas, é necessário ter no mínimo $2t+1$ réplicas. Ao receber o resultado de sua réplica, o controlador adiciona sua assinatura ao resultado e difunde a mensagem a todos os controladores que fazem parte do grupo. O controlador no processo de comparação que verificar que a $t+1$ *ésima* mensagem correta recebida é a sua, é o responsável pelo envio da resposta ao cliente.

Em todos os casos acima, após o envio da resposta para o cliente, o controlador da réplica mais rápida difunde uma mensagem de *fim_de_processamento,* que indica o fim da execução da requisição do cliente no grupo. Com esta última mensagem, torna-se possível a detecção de falhas do controlador da réplica mais rápida e a sua substituição.

2.5.1.2 Modelo de Replicação Ativa Cíclica (POWELL, 1991)

Neste modelo de replicação ativa, os controladores são configurados em forma de um anel lógico, com um *token* associado. A ordem de posse do *token* é definida segundo a seqüência do anel. O controlador detentor do *token*, ao receber uma requisição, é responsável pelo envio da resposta ao cliente e também por repassar o *token* para o próximo controlador do anel lógico, o qual será responsável por enviar a resposta ao cliente na próxima requisição.

Para tratar com hipóteses de faltas de temporização, o mecanismo usado é semelhante ao do modelo competitivo. A diferença é que o controlador que detém o *token* é o responsável por enviar a primeira mensagem para o cliente e pela difusão para os outros controladores da mensagem de *fim_de_processamento*, além de ser responsável pelo repasse do *token* para o próximo controlador especificado no anel lógico.

Na semântica de falhas arbitrárias, a replicação ativa cíclica também faz o uso de assinaturas. No recebimento de uma resposta pelo controlador que detém o *token*, é criada uma mensagem especial (mensagem e assinatura) que é difundida a todos os outros controladores que fazem parte do grupo e o *token* é passado para o próximo controlador do anel lógico. A mensagem *token* vai sendo passada no anel lógico, agregando mensagens e assinaturas, até que o controlador que possui o *token* verifique que $t + 1$ mensagens sejam corretas (mensagens iguais). Este controlador é o responsável pelo envio da resposta ao cliente e pelo envio de uma mensagem para os outros controladores informando o final do ciclo de processamento da requisição. Se ao término do ciclo do anel lógico, o total de $t + 1$ respostas iguais não tiver sido alcançado, então mais de t falhas ocorreram no sistema.

2.5.2 Modelo de Replicação Líder/Seguidores

O modelo de replicação líder/seguidores (*leader/followers*) utiliza a técnica de replicação semi-ativa. Esse modelo é implementado no sistema Delta-4 (POWELL, 1991).

Neste modelo todas as réplicas são ativas, porém somente a réplica líder responde ao

cliente. A réplica líder é responsável por estabelecer a ordem de processamento e por retransmitir as requisições às outras réplicas (seguidoras). É necessário um protocolo de difusão confiável que garante pelo menos a ordenação FIFO no envio das mensagens entre a réplica líder e suas seguidoras. Desta forma, a réplica líder impõe a ordem de execução dos pedidos, ou seja, é responsável pelo determinismo das réplicas. Para que faltas de valor sejam toleradas, um mecanismo de votação deve ser utilizado na réplica líder.

Um *crash* do líder pode ser detectado através do sistema de comunicação, quando o cliente tenta retransmitir uma mensagem para o líder ou pelos seguidores que recebem mensagens periódicas de "*I am alive*" ("estou vivo") do líder. Neste modelo o comportamento bizantino do líder não pode ser detectado.

2.5.3 Modelo de Replicação Coordenador/Coordenados

O modelo de replicação coordenador/coordenados (*coordinator/cohorts*) utiliza a técnica de replicação passiva. Este modelo é implementado usando o sistema ISIS (BIRMAN, 1985).

Nessa técnica, a cada requisição do cliente é atribuído um coordenador, enquanto as outras réplicas operam como coordenados. Cada coordenador pode somente atender a uma requisição de cada vez, no entanto várias requisições podem ser atendidas concorrentemente. Assim é possível a existência de vários coordenadores atendendo a requisições distintas no mesmo instante.

A escolha de um novo coordenador a cada requisição, necessita que todas as réplicas tenham a mesma visão dos participantes (*membership*), permitindo saber quem são os coordenadores e coordenados. O protocolo GBCAST do ISIS fornece essa visão. As requisições dos clientes são enviadas aos membros do grupo através do protocolo ABCAST do ISIS, um protocolo de difusão confiável com ordem total, que assegura que todas as réplicas não faltosas recebem as requisições do cliente na mesma ordem relativa.

Ao término da execução de uma requisição, o coordenador envia um *checkpoint* aos coordenados e depois envia o resultado do processamento da requisição ao cliente. O envio do *checkpoint* e da resposta ao cliente é feita através do protocolo de difusão confiável com ordenação causal CBCAST do ISIS. O uso desse protocolo garante que todos os coordenados atualizem seus estados antes que uma requisição futura seja executada.

Quando ocorre uma falha em um coordenador, um novo é eleito dentre os coordenados. Como todos os membros têm a mesma visão dos outros membros do grupo e usam o mesmo algoritmo de eleição, a escolha do novo coordenador é feita sem que nenhuma mensagem seja trocada.

2.6 Tolerância a Faltas Adaptativa

Normalmente, tolerância a faltas em sistemas distribuídos tem sido provida através de combinações de redundâncias de software e hardware, definidas estaticamente. Se considerarmos as características dinâmicas dos sistemas distribuídos atuais que aumentam em escala dia a dia, a noção fixa e pré-estabelecida na coordenação destes modelos de redundâncias torna os mesmos totalmente ineficientes. Além disso, incorre em alto custo, pois sempre será necessário alocar recursos de forma estática procurando atender o pior caso.

Sistemas distribuídos confiáveis podem se beneficiar de políticas adaptativas que utilizam os recursos do sistema de maneira mais eficiente e flexível. Portanto, a tolerância a faltas adaptativa tem o intuito de fazer o uso destas políticas adaptativas na gerência de redundâncias, de maneira a se manter eficaz, mesmo diante de mudanças freqüentes do seu ambiente computacional.

Com o uso de políticas estáticas, o nível de confiabilidade fornecido pelo sistema será sempre limitado aos recursos redundantes que ele usa. Com um nível de confiabilidade configurável pode-se fazer um melhor uso destes recursos disponíveis, de maneira a alocar somente os recursos necessários para obter a confiabilidade desejada, reduzindo assim os custos e conseguindo melhorar o desempenho dos sistemas.

Um software adaptativo é usualmente visto como aquele que pode ter a sua configuração modificada no sentido de responder às mudanças no seu ambiente de execução. As alterações de configuração baseadas em uma tolerância a faltas adaptativa podem ser conduzidas, por exemplo, por mudanças nos padrões de comunicação de um sistema distribuído, na freqüência de ocorrências de falhas parciais ou mesmo por novas necessidades da aplicação (HILTUNEN; SCHLICHTING, 1996). Um software adaptativo pode envolver a troca de algoritmos, em tempo de execução, para atender as mudanças do ambiente (CHEN et al., 2001).

2.6.1 Definição de Tolerância a Faltas Adaptativa

Tolerância a faltas é considerada um aspecto muito importante para os sistemas atuais, porém o custo para fornecê-la é muito alto, pois depende das semânticas de falhas (Seção 2.2), da redundância desejada, do número ou da freqüência de faltas que podem ocorrer.

Segundo Kim e Lawrance (1990), tolerância a faltas adaptativa é conseguida com mecanismos que satisfaçam requisitos de tolerância a faltas variáveis, dinamicamente, através da utilização eficiente (e adaptativa) de uma quantidade limitada e variável de recursos de processamento redundantes.

Tomando como base o citado acima, podemos definir tolerância a faltas adaptativa como a propriedade que permite um sistema manter e melhorar a confiabilidade do sistema através da adaptação a mudanças no ambiente computacional ou nas políticas de tolerância a faltas.

2.6.2 Modelo para Tolerância a Faltas Adaptativa

Um modelo para sistemas adaptativos é apresentado em (HILTUNEN; SCHLICHTING, 1996). Este modelo pode ser aplicado em uma grande quantidade de problemas, tal como algoritmos distribuídos, *membership* de protocolos de difusão confiável (CRISTIAN, 1991), esquemas de tolerância a faltas, etc.

Este modelo divide o processo de adaptação em três fases: detecção de alteração do ambiente ou de política (*change detection*), acordo (*agreement*) e ação (*action*) (HILTUNEN; SCHLICHTING, 1996).

Detecção de Alteração

Esta é a fase onde é feita a detecção de possíveis alterações no ambiente e a decisão de quando uma alteração no sistema deverá ativar uma adaptação.

A execução de cada uma das fases da adaptação é feita de acordo com um conjunto de políticas. Esta fase é executada de acordo com a política de detecção, que especifica a condição sob a qual uma adaptação no sistema será realizada, ou seja, quando é detectada uma alteração no sistema.

Acordo

Nesta fase é obtido o acordo entre todas as entidades do sistema distribuído onde a adaptação se faz necessária, visando decidir se os mesmos concordam com a adaptação. Normalmente esse processo envolve algum tipo de algoritmo distribuído de acordo, onde todos os envolvidos concordam com as mudanças propostas. Se houver uma entidade centralizadora responsável por tomar as decisões, não há a necessidade de algoritmos de acordo.

Esta fase é regida por duas políticas: política de resposta e política de votação. A *política de resposta* especifica a resposta de um sítio quando o acordo inicia. Por exemplo, se um sítio concorda ou discorda que uma mudança seja efetuada. A *Política de Votação* especifica sob quais condições o resultado do acordo é positivo (quando a alteração ocorreu) ou negativa (quando a alteração não ocorreu). Se o acordo for quantitativo, a política de votação também descreve como combinar as respostas, como por exemplo, a quantidade mínima de votos necessária para que uma adaptação ocorra.

Ação

É a fase onde a mudança do comportamento do sistema é efetivada, ou seja, quando o sistema é adaptado para corresponder as novas exigências. Em alguns casos a adaptação se dá de maneira bem simples, com a mudança de alguns parâmetros de execução, tal como o intervalo de monitoramento para verificar se um determinado sítio está ativo. Porém em outros casos, faz-se necessário efetuar alterações mais significativas, como a mudança da técnica de replicação utilizada.

Esta fase é executada de acordo com duas políticas: política de ação e política de temporização. A *política de ação* especifica a ação que deverá ser realizada como o resultado de uma alteração. A *política de temporização* especifica quando a ação será efetuada.

A Figura 2.8 esboça o modelo geral para sistemas adaptativos. Nela pode-se observar uma linha tracejada que sai da fase de ação e volta para a fase de acordo, pois em algumas vezes o processo de adaptação necessita que as fases de acordo e ação sejam executadas mais de uma vez, ou seja, são necessários vários *rounds*.

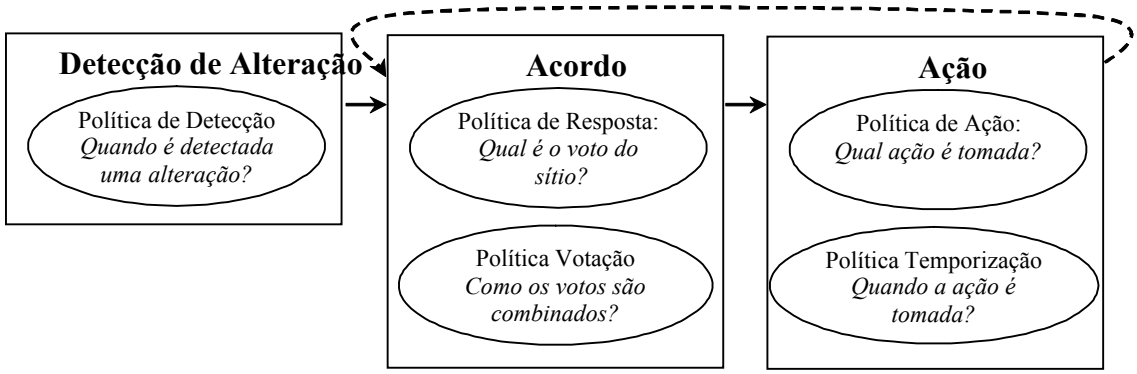

Figura 2.8 – Modelo de Sistemas Adaptativos (HILTUNEN; SCHLICHTING, 1996).

O sistema adaptativo normalmente continua executando suas funções de forma simultânea às fases de detecção de alteração e de acordo. Um sistema adaptativo pode, portanto, adaptar-se diversas vezes para manipular diferentes tipos de alterações no seu ciclo de vida.

No sistema também poderá ocorrer a adaptação inversa, o que significa que o sistema, depois de determinado tempo executando uma mudança pode verificar que não se faz mais necessário operar esta alteração de configuração, podendo então o sistema voltar ao seu estado anterior de configuração.

2.7 Referências em Tolerância a Faltas Adaptativa

Esta seção apresenta alguns trabalhos da literatura que visam prover mecanismos de tolerância a faltas adaptativa.

2.7.1 Adaptação dirigida pela semântica de falhas

O custo em fornecer tolerância a faltas incorre em custos extras em termos de desempenho e de consumo de recursos. As falhas em um sistema podem ser relativamente raras, porém uma sobrecarga é gerada pelos mecanismos de tolerância a faltas mesmo que nenhuma falha no sistema ocorra. Em (CHANG et al., 1998) é apresentada uma abordagem para reduzir o custo no fornecimento de tolerância a faltas, através de adaptações às semânticas de falhas no sistema. Nesta abordagem é usado um algoritmo para situações sem falhas e quando as mesmas ocorrerem, durante a execução da aplicação, o algoritmo de aplicação é trocado por um outro algoritmo envolvendo redundâncias, fornecendo uma tolerância a faltas própria para a semântica de falhas que ocorreram.

Nesta abordagem, primeiro é identificado uma hipótese de falhas que é valida na maior parte do tempo de execução do sistema, que é chamado de modelo básico de falhas (*baseline failure model*). Ou seja, sistemas que raramente experimentam falhas podem ser desprovidos desse modelo básico. Em muitas situações, pode não ser determinado um modelo básico de falhas. A abordagem inclui ainda um modelo de falhas estendido (denotado por *fm*), que inclui o modelo *baseline* e falhas menos restritivas. Além do modelo *fm*, tem-se um algoritmo de aplicação que inclui redundâncias para o modelo de falhas ativo que deve estar contido em *fm* e ao mesmo tempo ser uma extensão do modelo de falhas *baseline*. A abordagem define ainda um algoritmo de detecção que identifica falhas que estão em *fm* e não pertencentes ao modelo de falhas ativo (*Det(fm)*).

As relação de modelos de falhas usados nesta abordagem são:

$$M_{baseline} \subseteq M_{ativo} \subseteq M_{fm}$$

Essa abordagem prevê a execução simultânea do algoritmo que tolera faltas com semântica no modelo ativo de falhas (que pode ser o *baseline*) e do algoritmo para detecção de falhas no modelo *fm*. Quando é detectado uma falha no modelo *fm* e fora do modelo ativo, então ocorre a troca de algoritmo de tolerância a faltas para atender a semântica contida no modelo *fm*. Isto determina um novo modelo ativo que estende o modelo anterior com o novo tipo de semântica de falha detectado no sistema e pertencente a *fm*.

Na Figura 2.9 são mostrados dois algoritmos denotados por *Alg(baseline)* e *Alg(fm)*, usado com os modelos *baseline* e *fm*, respectivamente. Um algoritmo *Det*(*fm*) é usado para detectar se uma falha em *fm* ocorre durante a execução do sistema quando *Alg(baseline)* está ativo. Os algoritmos *Alg*(*baseline*) e *Det*(*fm*) são executados simultaneamente até que uma falha em *fm* seja detectada. Depois da falha detectada, o algoritmo *Alg*(*fm*) passa a ser ativo com seu respectivo modelo de falhas (*fm*). Esta abordagem adaptativa é denotada por *Adap*(*baseline*→ *fm*). A Figura 2.9 mostra o funcionamento da abordagem, assim como o custo da execução da mesma.

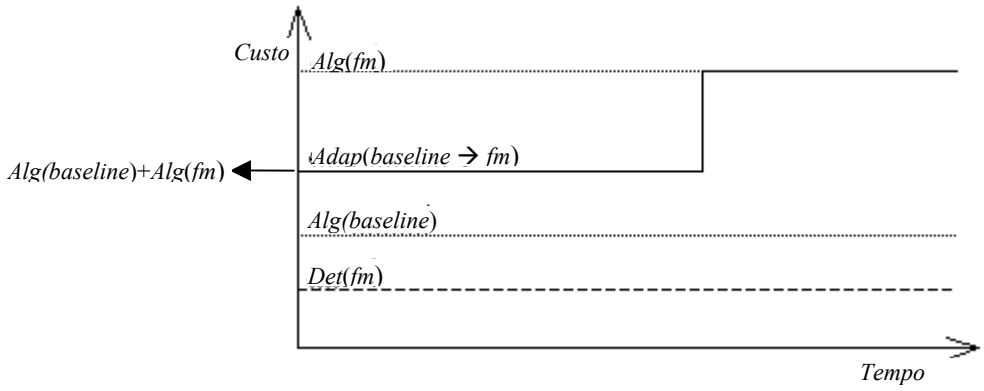

Figura 2.9 – Idéia básica da adaptação para os modelos de falhas (CHANG et al., 1998).

Observando a Figura 2.9 é possível verificar que o custo do algoritmo *Adap(baseline→fm)* é a soma do custo dos algoritmos *Alg*(*baseline*) e *Det*(*fm*), o que faz essa abordagem ser melhor somente se o custo desta soma for menor que *Alg*(*fm*) ao longo do ciclo de vida do sistema.

Em (CHANG et al., 1998) a abordagem proposta é aplicada ao problema do *broadcast* atômico. São usados como solução três protocolos de *broadcast* atômico apresentados em (CRISTIAN et al., 1985): um deles tolera faltas por omissão, o outro tolera faltas por temporização e o último tolera faltas bizantinas.

A estrutura básica para o protocolo de *broadcast* adaptativo é simples, consistindo inicialmente de um protocolo que não é tolerante a faltas *Alg*(*Nenhum*) e três algoritmos de detecção: *Det*(*Omissão*), Det(*Temporização*), Det(*Bizantina*). O projetista do sistema deverá escolher para uma aplicação, a hipótese de falhas inicial que deverá ser usada e o algoritmo de detecção desejado. Por exemplo, se inicialmente o modelo de falhas a ser tolerado é para não tolerar nenhuma falha, então, o algoritmo escolhido é o *Alg*(*Nenhum*) e se o algoritmo de detecção for detectar falhas por omissão, *Det*(*Omissão*). Quando uma falha por omissão for detectada, o algoritmo é trocado para o algoritmo de *broadcast* que tolera falhas de omissão, isto é, Alg(*Omissão*). Neste caso, o modelo ativo de falhas passa a ser o que inclui falhas por omissão

Se projetarmos o modelo de sistemas adaptativos apresentado na seção anterior nesta abordagem de tolerância a faltas adaptativa dirigida pela semântica de falhas, teremos que a fase de detecção do modelo coincide com o algoritmo *Det()*; a fase de acordo não é

necessária por que o algoritmo *Det()* é implementado de forma centralizada; e a fase de ação é contemplada com a definição do novo algoritmo tolerante a faltas adequado a nova situação de semântica de falhas.

2.7.2 Chameleon

O *Chameleon* (BAGCHI et al., 1998) é uma infraestrutura adaptativa que fornece diferentes níveis de requisitos de confiabilidade para aplicações distribuídas. O *Chameleon* faz o uso de ARMORs (*Adaptive, Reconfigurable, Mobile Objects for Realiability*), objetos móveis adaptativos e reconfiguráveis para prover a confiabilidade exigida.

O *Chameleon* pode, seletivamente, usar combinações de ARMORs a fim de prover diferentes níveis de confiabilidade. Além de fornecer um ambiente de tolerância a faltas altamente especializado, ARMORs também são independentes de localização – eles podem executar suas ações em qualquer nó de uma rede heterogênea. Os ARMORs também formam o mecanismo através do qual nova funcionalidade pode ser introduzida no sistema de maneira incremental.

Suporte de gerenciamento

Qualquer computador em uma rede pode estar configurado para fazer parte do ambiente *Chameleon*. O *Chameleon* possui o FTM (Gerenciador de Tolerância a Faltas), parte fundamental da adaptação fornecida pelo *Chameleon*. Uma vez instalado – e pode ser em qualquer sítio de uma rede, invoca um *daemon* (que tem a função de manipular as comunicações com *hosts* remotos) e um *heartbeat* ARMOR (usado na detecção de *hosts*) posicionado junto a um serviço de aplicação em um *host* remoto. Finalmente, o FTM cria um Backup que monitora o FTM e que ao detectar um erro, torna-se o novo FTM ativo. Uma vez que o FTM Backup é configurado, tem-se um ambiente *Chameleon* estável pronto para aceitar e servir solicitações do usuário.

Outros nós na rede podem pedir para se unir ao ambiente *Chameleon* através do FTM. Sob a requisição de um novo nó para se unir à infra-estrutura, o FTM envia o código necessário para compilar e executar um *daemon* no nó que deseja unir-se ao *Chameleon*.

Suporte de instalação

O usuário submete uma aplicação ao FTM com requisitos de confiabilidade especificadas em uma linguagem declarativa própria que o FTM deve interpretar. O FTM

seleciona uma estratégia de execução de tolerância a faltas apropriada, através da qual os requisitos declarados de tolerância a faltas podem ser satisfeitos. Para realizar esta seleção, o FTM tem um registro de várias estratégias de execução de tolerância a faltas e a descrição das semânticas de falhas associadas.

Depois que o FTM seleciona uma estratégia de execução particular, ele lança os gerenciadores locais (*surrogate managers*) para realizar a execução da aplicação tolerante a faltas. Estes gerenciadores irão centralizar vários dos controles locais de uma aplicação. Existem gerenciadores locais para cada aplicação no sistema. O FTM instala um gerenciador local em qualquer sítio da rede, onde a sua instalação consiste do envio do código do mesmo e das bibliotecas de componentes exigidas pelo *Chameleon* para o nó no qual o gerenciador local será instalado. No nó de recepção, o *daemon* recebe o código, compila-o e executa o mesmo. Esta recompilação no nó destino faz com que se tenha flexibilidade de suporte multi-plataforma.

Todos os ARMORs do *Chameleon* são registrados junto ao FTM e estão disponíveis para uso por qualquer gerenciador local. Um gerenciador local instala os ARMORs comuns (*common* ARMORS) necessários para completar os requisitos da aplicação. Por exemplo, gerenciadores locais responsáveis por executar uma aplicação instalam três cópias de um ARMOR de Execução (*Execution* ARMOR), um para cada réplica da aplicação e um ARMOR Votador (*Voter* ARMOR). Depois que gerenciadores locais instalam com sucesso os ARMORs necessários para executar a aplicação do usuário, notificam o FTM informando a localização de cada ARMOR. Desta forma, o FTM tem formado uma visão de configuração global do sistema para uso durante a recuperação de eventuais falhas parciais.

Com o uso dessa arquitetura configurável de ARMORs é possível fornecer diferentes níveis de disponibilidade, de acordo com a necessidade de cada sistema. Fazendo um comparativo dessa abordagem com o modelo de sistemas adaptativos (seção 2.6.2), verifica-se que a fase de detecção é contemplada pelo FTM através da consulta aos *heartbeat* ARMORs nos sítios onde estão instaladas as réplicas. A fase de acordo não fica explicíta por que o FTM é centralizado. A fase de ação é executada também pelo FTM, que é responsável por reconfigurar o sistema em conseqüência das mudanças que ocorreram no sistema. O que se pode concluir é que a adaptação neste caso é contemplada por mecanismos de configuração.

2.7.3 AQuA

A arquitetura AQuA (*Adaptive Quality of Service Availability*) (CUKIER et al., 1998) visa fornecer tolerância a faltas adaptativa para aplicações distribuídas. A arquitetura AQuA permite que os programadores de aplicações declarem os níveis de confiabilidade desejados. O AQuA tenta satisfazer o nível de tolerância a faltas requisitado através da configuração do sistema em função das mudanças de recursos do sistema e devido às falhas ocorridas.

Segundo Cukier et al. (1998) a arquitetura AQuA tem uma série de diferenciais em relação a outras propostas de fornecimento de tolerância a faltas:

- O nível de abstração em relação à tolerância a faltas vista pelos usuários é elevado se comparado com outros sistemas, permitindo ao usuário um controle de alto nível sobre os tipos de faltas que podem ser toleradas e o nível desejado de disponibilidade de um objeto distribuído;
- O sistema é adaptado dinamicamente em resposta às falhas parciais que ocorrem no sistema, visando manter o nível de disponibilidade desejado;
- AQuA reconhece que aplicações podem ter diferentes níveis de disponibilidade no decorrer de seu ciclo de vida, fornecendo reconfiguração do sistema em resposta a mudanças nos requisitos da aplicação ou do ambiente do sistema.

A arquitetura AQuA usa o QuO (*Quality Objects*) (ZINKY et al., 1997, LOYALL et al., 1998) para especificar os requisitos de qualidade de serviços (QoS) em nível de aplicação. Também faz uso do gerenciador de confiabilidade do PROTEUS (SABNIS et al. 1999) para configurar o sistema em resposta a falhas parciais e aos requisitos da aplicação. Por sua vez, o Ensemble (HAYDEN, 1998) é usado no AQuA para fornecer serviços de comunicação de grupo. Ainda o AQuA fornece aos objetos das aplicações uma interface CORBA, através do *gateway* AQuA. Um *gateway* AQuA faz a conversão de mensagens entre os processos em nível de comunicação Ensemble para mensagens IIOP, entendidas pelos ORBs, no CORBA. A seguir, cada uma dessas tecnologias integrantes da infraestrutura AQuA serão brevemente apresentadas.

Ensemble

O sistema de comunicação de grupo Ensemble é usado pelo AQuA para assegurar

comunicação confiável entre grupos de processos, garantindo a entrega atômica das difusões (*multicast*) detectando e excluindo membros de grupos que falhem por *crash*.

O Ensemble assume falhas de processos por *crash* e usa mecanismos de detecção de falhas de processos através de mensagens "*I am alive*". O AQuA usa os resultados deste mecanismo de detecção como entrada para o Proteus, o qual fornecerá mecanismos de configuração para o processo de recuperação.

Quality Objects

QuO (*Quality Objects*) permite às aplicações especificar os requisitos de qualidade de serviço(QoS) no nível de aplicação através de um "contrato", o qual especifica as ações a serem tomadas baseadas no estado do sistema distribuído e nos requisitos desejados pela aplicação. O QuO permite definir medidas de QoS expressas em alto nível, através do uso de uma linguagem de descrição de contrato (CDL – *Contract Description Language*). Um contrato especifica os elementos do sistema que precisam ser monitorados de modo a avaliar o nível de QoS corrente, e as ações a serem tomadas quando ocorrer mudanças no sistema. O AQuA faz o uso de QuO para transmitir para o Proteus os níveis de disponibilidade desejados pelas aplicações nas suas configurações. Um contrato é especificado em tempo de configuração, isto é, quando a aplicação está sendo configurada para iniciar a execução.

Proteus

A tolerância a faltas em AQuA é fornecida pelo Proteus, o qual gerencia dinamicamente a replicação dos objetos distribuídos de maneira a tornar a aplicação confiável nos níveis requisitados pelo usuário.

Através da tradução dos requisitos de disponibilidade expressos em alto nível, transmitidos através do QuO para o Proteus, é tomada a decisão de como fornecer tolerância a faltas. Isso envolve a escolha do tipo de replicação, ativa ou passiva, o tipo da votação a ser usada (algoritmo e localização), o grau de replicação, o tipo de faltas a serem toleradas (crash, valor ou temporização), a localização das réplicas, entre outros fatores. Além disso, Proteus também é responsável por implementar o esquema de tolerância a faltas escolhido.

Proteus consiste basicamente de um conjunto de monitores, votadores, fábricas de objetos e um gerenciador de replicação. O gerenciador de replicação é composto por um conselheiro (*advisor*) e um coordenador de protocolo. Os monitores implementam temporizadores para detectar faltas por atraso ou omissão. Os votadores decidem qual das respostas das réplicas apresentar como resposta de uma replicação. As fábricas criam e destroem objetos nos *hosts.* O gerenciador de replicação recebe informações dos QuO a respeito da disponibilidade dos objetos remotos e usando seu conselheiro toma as decisões do tipo de tolerância a faltas que deve ser fornecida. O coordenador de protocolo concretiza as decisões tomadas pelo conselheiro.

Gateway

A comunicação entre todos os componentes da arquitetura AQuA é realizado através do uso de *gateways*, que traduzem as invocações dos objetos CORBA em mensagens que são transmitidas via Ensemble. A comunicação entre cada componente e um *gateway* é feito através de mensagens IIOP. Os *gateways* também são responsáveis por implementar os votadores e os monitores, além de concretizarem vários esquemas de replicação.

Na Figura 2.10, é apresentada a arquitetura do AQuA.

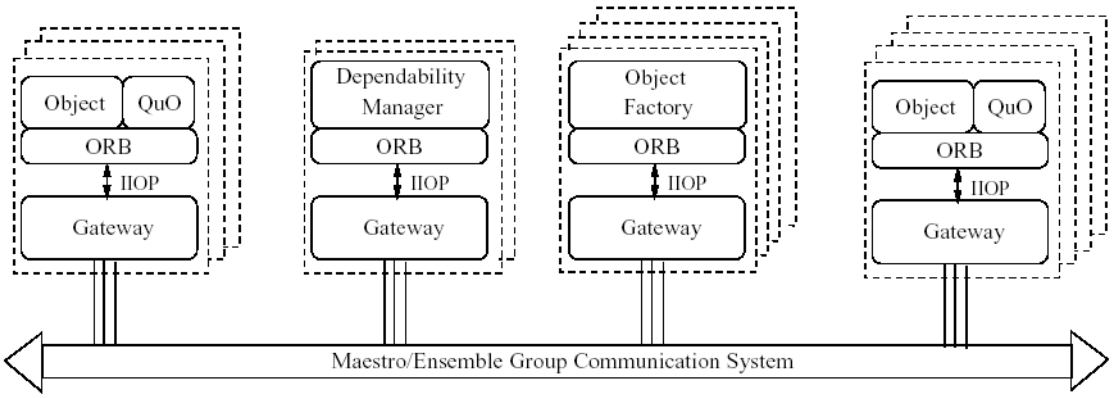

Figura 2.10 – Arquitetura AQuA (CUKIER et al., 1998).

Comparando o AQuA com o modelo de sistemas adaptativos, nota-se que a fase de detecção, acordo e ação são efetuadas pelo Proteus, através de monitores, votadores, fábricas de objetos e um gerenciador de replicação. O AQuA é a experiência mais expressiva presente na literatura sobre tolerância a faltas adaptativa.

2.8 Conclusão

Neste capítulo foram apresentados os principais conceitos sobre tolerância a faltas e também a classificação das faltas segundo as semânticas de falhas associadas.

Em seguida foi apresentado um estudo sobre técnicas de replicação (ativa, passiva e semi-ativa). A abordagem de replicação ativa tem a necessidade do determinismo entre as réplicas. Esta abordagem apresenta um baixo *overhead* em caso de falha do sistema, visto que todas as réplicas são ativas e, portanto, o mascaramento da falha de uma réplica é praticamente instantâneo. Ao contrário, a abordagem de replicação passiva apresenta um *overhead* maior, devido ao tempo gasto para recuperar a réplica primária, visto que é necessário recuperar o estado da réplica. No entanto, a abordagem de replicação passiva não necessita do determinismo entre as réplicas, pois somente a réplica primária é ativa e impõe o estado às réplicas *backups*.

Alguns dos principais modelos de replicação presentes na literatura também foram apresentados. Os modelos apresentados toleram basicamente faltas de *crash*, omissão e temporização.

Aspectos que envolvem tolerância a faltas adaptativa foram introduzidos neste capítulo, assim como um modelo geral para sistemas adaptativos. O uso de mecanismos de tolerância a faltas adaptativa permite manter e melhorar a tolerância a faltas de um sistema através da adaptação a mudanças no ambiente ou nas políticas de tolerância a faltas.

Por fim, foram apresentados alguns trabalhos da literatura que empregam o conceito de tolerância a faltas adaptativa.

Capítulo 3

Componentes de Software

O desenvolvimento de software que antes era baseado em abordagens fundamentadas em blocos monolíticos, apresentava grandes dificuldades quando o software tinha uma certa complexidade. A implementação, a confiabilidade e a manutenção envolviam altos custos e tempos significativos. Novas abordagens de programação surgidas no mercado a partir da metade dos anos oitenta ajudaram a reduzir os tempos e custos de desenvolvimento. Entre estas abordagens estão a programação orientada a objetos (MEYER, 1988) e a programação baseada em componentes (SZYPERSKI, 1998); ambas enfatizam a reusabilidade de software. A orientação a objetos teve uma larga utilização desde então devido a sua flexibilidade no trato com a reusabilidade e a fácil expressão do mundo real em suas metodologias *bottom-up* (BOOCH, 2000).

A tecnologia de componentes tem ressurgido recentemente através de suportes *middleware* para programação distribuída (OMG, 2002a, SUN MICROSYSTEMS, 2001). A programação por componentes se apresenta como plenamente adequada a ambientes distribuídos onde as necessidades de evolução e de configuração dinâmica estão presentes. Por definir bem os limites do reuso de componentes esta última abordagem atende melhor as necessidades citadas de ambientes distribuídos que a programação orientada a objetos (SZTAJNBERG, 2002).

A programação baseada em componentes tem se disseminado rapidamente, o que pode sugerir que esta tecnologia tenha surgido recentemente. Entretanto, a idéia de construir aplicações a partir do reuso de pedaços de software pré-existentes não é nova, ela vem desde a década de 60. Na Conferência de Engenharia de Software da OTAN em 1968,

durante a “crise do software”, a idéia do reuso de software foi introduzida por Doug MacIlroy (MCILROY, 1968). McIlroy propôs uma indústria de software capaz de produzir componentes de software reutilizáveis. Segundo ele, os desenvolvedores poderiam criar seus programas a partir da composição de componentes de software pré-definidos e implementados, ao invés começar o programa como um todo a partir do nada.

A idéia da utilização de partes pré-existentes na construção de sistemas maiores vêm sendo utilizada em outras áreas há muito tempo. Na indústria automobilística, por exemplo, os automóveis são construídos com base em componentes pré-fabricados que são interconectados. Um conjunto bem escolhido de componentes pode ter muitas possibilidades de configuração e os produtos finais podem ser fabricados de forma mais rápida e mais confiável.

Como os desenvolvedores de software estão constantemente em busca de novas formas de construir aplicações em menor tempo e com menor custo sem afetar sua qualidade e eficiência, a tecnologia de componentes pode ser vista como solução para atender tais exigências.

Neste capítulo, serão apresentados alguns conceitos relacionados a componentes. Em seguida, serão apresentados dois modelos de componentes, dando maior ênfase no modelo de componentes do CORBA (CCM), pois entendimento dos seus principais recursos é de fundamental importância nesse trabalho.

3.1 Componente de Software

3.1.1 Caracterização de Componentes

O desenvolvimento de aplicações baseadas em componentes consiste em fazer a composição das aplicações a partir de pedaços de códigos existentes, os quais são denominados componentes, permitindo o reuso de software (KRUEGER, 1992, SZYPERSKI, 1998).

São várias as definições para componentes de software apresentada na literatura (BACHMAN et al., 2000, BROWN; WALLNAU, 1998). Segundo Szyperski (1998), “Componente é uma unidade de composição com interfaces contratualmente especificadas e apenas com dependências de contexto explícitas. Um componente pode ser instalado

isoladamente e estar sujeito à composição por terceiros".

A partir da definição, conclui-se que os componentes devem ser construídos tão independentes de contexto quanto possível, de forma a permitir sua reutilização em diferentes contextos. Quanto mais um componente é independente de contexto, mais facilmente pode ser reutilizado.

Um componente ideal deve oferecer o conjunto certo de interfaces e não ter dependências de contexto. Entretanto, tal qualidade é difícil, senão impossível de ser encontrada em um componente. Tecnicamente, um componente deveria vir com todos os softwares dos quais depende agrupados consigo, mas claramente isto vai contra o princípio da programação baseada em componentes.

A definição também indica que um componente encapsula sua implementação, interagindo com o ambiente externo (outros componentes ou aplicações) através de interfaces bem definidas, que especificam claramente o que o componente requer e fornece. As interfaces servem para a comunicação entre componentes ocultando dos usuários os detalhes de implementação. A especificação de um componente é, normalmente publicada em separado de seu código fonte por meio da declaração de suas interfaces. Desta forma, componentes podem ser vistos como produtos que são feitos disponíveis com o objetivo de serem compostos por terceiros formando programas.

As tecnologias de componentes definem um novo tipo de programador, o programador de sistemas que funciona como um *montador de sistema* (*system assembler*), utilizando componentes bem definidos para criar suas aplicações. O *montador de sistema* não precisa conhecer os detalhes de implementação de componentes, mas sim ser um conhecedor dos serviços providos por um componente.

Baseado no citado acima, assumimos neste texto que um componente de software é uma unidade binária desenvolvida para uma finalidade específica e que colabora facilmente com outros componentes no objetivo de formar aplicações. Os componentes interagem entre si apenas através de interfaces bem definidas.

A programação baseada em componentes promove a separação entre a lógica das implementações (parte funcional) e o gerenciamento das aplicações (parte não funcional). Desta forma, os componentes implementam somente aspectos funcionais; serviços de

gerenciamento e políticas ficam a cargo de uma infraestrutura de componentes.

3.1.2 Benefícios da programação baseada em componentes

A tecnologia de componentes de software apresenta diversos benefícios para o processo de desenvolvimento de software, entre os quais aponta-se (SZYPERSKI, 1998):

- Maior flexibilidade na programação e manutenção de sistemas: novas funcionalidades podem ser adicionadas a sistemas existentes de acordo com as necessidades de evolução ou atualização. As funcionalidades existentes podem ser substituídas sem causar nenhum impacto a outras partes do sistema, tanto do ponto de vista estrutural como funcional. A manutenção do sistema é simplificada, pois basta substituir um componente por outro que possua as mesmas características de interface.

- Redução no tempo de desenvolvimento: a disponibilidade de uma ampla variedade de componentes, que atendam as necessidades do cliente, diminui os tempos de projeto de programas complexos. Desta forma, é possível acompanhar de forma veloz mudanças que ocorrem no mercado, garantindo dessa forma a competitividade.

- Redução dos custos: o custo de desenvolvimento é menor, devido ao fato de um componente ser desenvolvido para atender uma diversidade de aplicações e não somente uma única aplicação.

- Aumento na confiabilidade dos sistemas: como um componente é utilizado em vários sistemas, os erros podem ser detectados mais rapidamente e por sua vez, as devidas correções também podem ser feitas de forma mais rápida. Assim, os componentes tendem a se estabilizar mais rapidamente, tornando os sistemas mais confiáveis.

3.1.3 Interfaces

"Uma interface é uma coleção de operações utilizadas para especificar um serviço de uma classe ou de um componente" (BOOCH et al., 2000). As interfaces dos componentes são os pontos de acesso aos serviços oferecidos e usados pelo componente. Normalmente, um componente possui vários pontos de acesso, cada ponto provendo um

serviço diferente. As interfaces separam as especificações dos componentes de suas implementações. Assim, faz com que os detalhes de implementação não sejam conhecidos pelos usuários.

É de grande importância a boa definição das interfaces de um componente (parte explícita e visível), separado da respectiva implementação (parte interna), como potencial de sucesso no uso dos componentes como integrantes de sistemas maiores. Através de interfaces, os usuários de um componente podem conectá-los a outros componentes sem se preocupar com sua implementação. As interfaces definem apenas assinaturas de métodos, não suas implementações. Portanto, uma interface pode ser vista como uma coleção de métodos abstratos, que especificam os serviços providos por um componente ou por uma classe. Um mesma interface pode ser realizada (implementada) por diferentes componentes ou classes; e um componente ou classe pode implementar diferentes interfaces (BOOCH et al., 2000).

A conexão entre componentes somente é possível através de interfaces compatíveis (SZYPERSKI, 1998). A Figura 3.1 ilustra como os componentes se combinam a partir de suas interfaces.

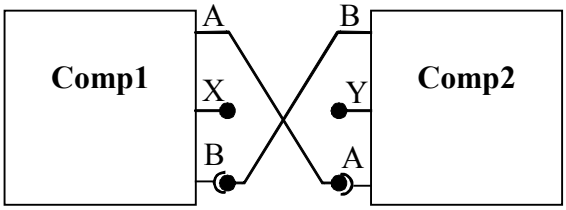

Figura 3.1 – Conexão de Componentes.

No exemplo mostrado na Figura 3.1, o componente Comp1 fornece as interfaces A e X e requer a interface B. O componente Comp2 fornece as interfaces B e Y e requer a interface A. As conexões entre os componentes (Comp1 e Comp2) são estabelecidas através das interfaces compatíveis A e B.

Conforme o exemplo em UML mostrado na Figura 3.2, é possível mostrar o relacionamento entre um componente e suas interfaces. A primeira representação mostra a interface na sua forma canônica, ocultando os métodos abstratos da interface. A segunda representação é a forma expandida que permite a visualização dos métodos abstratos da interface, os quais são implementados pelos componentes ou classes que concretizam a

interface. No exemplo da Figura 3.2, tem-se o componente Cliente.java dependendo da interface IConta, que por sua vez é realizada pelo componente CCorrente.java.

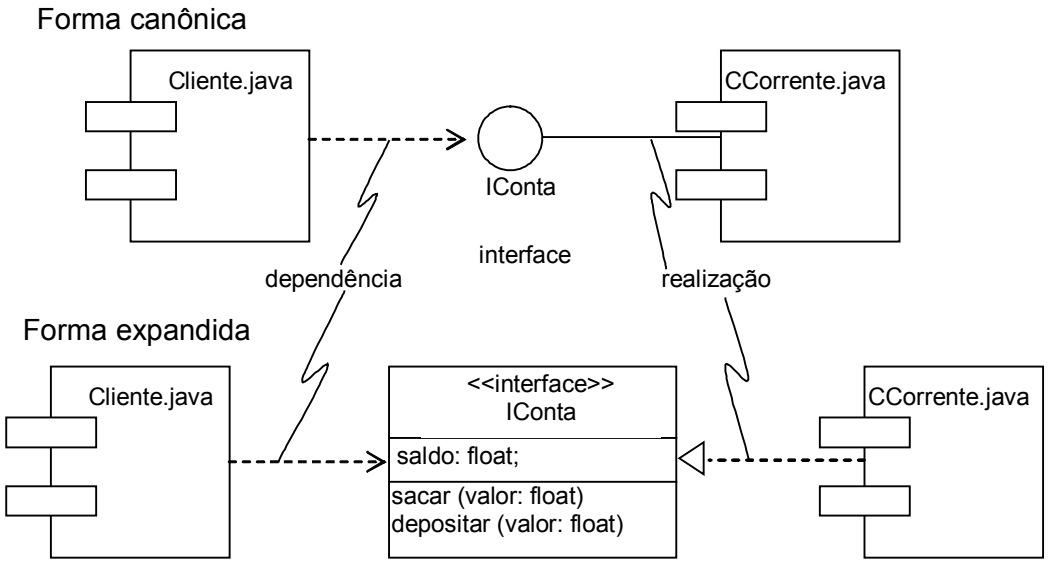

Figura 3.2 – Componentes e Interfaces.

3.1.4 Empacotamento

À medida com que a quantidade de componentes, classes, entre outros elementos num sistema cresce, tem-se a necessidade de organizá-los em grupos maiores, chamados de pacotes (*packages*) (BOOCH et al., 2000).

Nos pacotes são agrupados conjuntos de elementos com funções afins, para um dado domínio de aplicação (sistemas contábeis, controle de estoque, etc)[3]. Portanto, o pacote forma um espaço de nome (*namespace*), significando que os elementos do mesmo tipo necessitam ser declarados de maneira única no contexto do pacote que os contêm. Os pacotes poderão conter até mesmo outros pacotes, permitindo decompor os modelos de maneira hierarquizada.

Todo pacote deve ter um nome que seja distinto dos demais pacotes. Tendo nomes distintos dos pacotes, pode-se ter componentes distintos com nomes iguais em diferentes pacotes. Por exemplo, pode-se ter o componente Cliente no pacote X e o componente Cliente no pacote Y.

Geralmente são utilizados vários componentes para a composição de uma aplicação,

[3] A noção de *packages* é a mesma de *frameworks* em linguagens orientadas a objetos.

organizados em pacotes para possam ser disponibilizados comercialmente, o que é conhecido como *Component Off-The-Shelf* (COTS), ou "Componente de Prateleira".

3.1.5 Reutilização

A reutilização de software enfatiza o reaproveitamento de softwares existentes para auxiliar na construção de novos sistemas (DÍAZ, 1993, KRUEGER, 1992). A programação baseada em componentes é fundamentada na reutilização de software: todo ou quase todo esforço de programação na implementação de um sistema é feito pela composição de componentes existentes. O processo de construção de software através da reutilização de componentes permite acréscimos significativos de produtividade e qualidade, e uma redução no custo de desenvolvimento como resultado da redução do tempo de codificação.

Com a reutilização de componentes, a confiabilidade dos sistemas pode ser melhorada porque os componentes já foram submetidos a testes durante o seu desenvolvimento original e podem já ter comprovado a sua eficácia em outros sistemas, reduzindo então os custos de manutenção de software.

3.1.6 Modelo de Componentes

Os componentes precisam ter um comportamento e características esperadas para participar de uma estrutura de componentes, de maneira a tornar possível sua interação com o ambiente e outros componentes. Desta forma, eles precisam funcionar de acordo com um conjunto de regras. Estas regras são definidas por um modelo de componentes. O modelo de componentes especifica como o componente publica suas interfaces, métodos e eventos, provendo as diretrizes para a criação e implementação dos mesmos (ORFALI; HARKEY, 1998, BACHMAN et al., 2000).

Os principais modelos atuais de componentes são: EJB (*Enterprise* Java *Bean*s) (SUN MICROSYSTEMS, 2001), desenvolvido pela Sun Java e CCM (CORBA *Component Model*) (OMG, 2002a), parte integrante das especificações do CORBA 3.0 (OMG, 2002b), padronizado pela OMG, *Object Management Group*. Estes modelos estão apresentados nas próximas seções.

3.2 EJB (Enterprise Java Beans)

O modelo de componentes *Enterprise JavaBeans* (EJB) (SUN MICROSYSTEMS,

2001) foi introduzido pela *Sun Microsystems* em 1998. O EJB é um modelo de componente para uso no servidor, destinado ao desenvolvimento de aplicações distribuídas, usando a linguagem de programação Java. As aplicações construídas a partir de componentes EJB podem ser implantadas em qualquer plataforma de servidor que ofereça suporte a especificações EJB.

De acordo com o EJB, um componente é chamado de *enterprise bean*, mas no texto chamaremos apenas de *bean*. Cada um dos tipos de *beans* definidos nas especificações EJB é executado dentro de um *container*. O *container* é um suporte de execução para *beans* que provê serviços de suporte (ciclo de vida, gerência de estado, segurança, transação e persistência) para estes. Vários *beans* podem ser implantados no mesmo *container.*

Um *bean* não interage diretamente com outro *bean* ou serviço. Essa interação é feita através do *container*. Com a interposição do *container* entre os componentes de aplicação e os serviços de suporte, é possível ter transparência na utilização destes últimos. Desta forma, o desenvolvedor de um *bean* pode somente focalizar o trabalho no desenvolvimento da lógica da aplicação. Os aspectos não relacionados com a funcionalidade da aplicação (políticas, qualidade de serviços, etc) estarão concentrados no *container*. Um *container* é localizado dentro de um servidor de aplicação que, por sua vez, fornece um ambiente de execução para um ou mais *containers.* O servidor de aplicação gerencia os recursos de baixo nível e os aloca para *containers* quando necessário.

A interação entre um cliente[4] e o *bean* é feita através de duas interfaces, a interface EJB *Home* e a interface EJB *Object*. A interface **EJB *Home*** fornece acesso aos serviços de ciclo de vida do *bean*. Os clientes podem usar essa interface para criar, remover ou buscar uma instância existente do *bean*. O *container* registra automaticamente esta interface para cada tipo de *bean* instalado no mesmo, através da API JNDI (Java *Naming and Directory Interface*) (THOMAS, 1998). Isso permite ao cliente localizar a interface EJB *Home* de um tipo de *bean* para criar, localizar ou destruir uma instância do mesmo. A interface **EJB *Object*** fornece acesso remoto aos métodos do *bean*, usando Java RMI sobre IIOP. Esta interface representa a visão do cliente de um *bean*.

Quando o cliente invoca uma operação dessas interfaces (*EJB Home* ou *EJB Object*),

[4] O cliente de um componente pode ser um outro componente implantado no mesmo ou em outro *container*, um programa Java qualquer ou até mesmo um programa escrito em outra linguagem.

o *container* intercepta cada chamada e desta forma, implementa os controles não funcionais necessários na execução do *bean* alvo.

A Figura 3.3 mostra um *bean* dentro de um *container*, onde o cliente interage com o *bean* através das interfaces *Home* e *Object*.

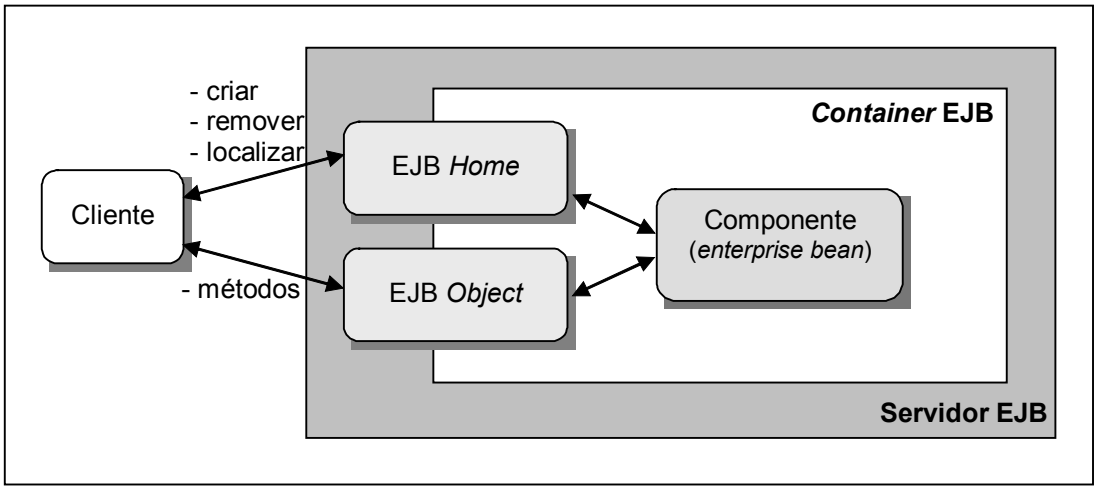

Figura 3.3 – *Container* EJB.

O modelo EJB define três tipos de *bean*:

- *entity bean* – componente que representa uma entidade de negócio que possui informação de estado armazenada em um meio persistente. Isto implica, que o *bean* em si é persistente.
- *session bean* – componente que normalmente contém métodos relacionados a lógica do negócio, tais como efetuar cálculos, transferir fundos entre contas correntes, etc. Este tipo de componente não tem estado persistente, isto é, possíveis informações de estado são mantidas somente enquanto o componente estiver sendo usado pelo cliente. Os *session beans* subdividem-se ainda em dois subtipos:
  - *stateful* (com estado): neste tipo de *bean* o estado é mantido durante uma sessão cliente-*bean*. Quando o cliente remove o *bean*, a sessão termina e o estado desaparece. Este tipo é normalmente usado quando é conveniente manter estado entre pedidos do cliente.
  - *stateless* (sem estado): não mantêm qualquer estado entre requisições

subseqüentes de um cliente em particular. Quando um cliente invoca um método, as variáveis da instância do bean podem representar um estado, mas apenas durante a invocação, pois esse estado deixa de existir quando o método termina.

- *message driven bean* – permite a integração do componente ao JMS (Java *Message Service*), que dá suporte a mensagens assíncronas, de forma a permitir aos EJBs receber mensagens através do JMS e executar métodos que façam o tratamento das mensagens recebidas.

As regras associadas aos serviços de suporte necessários ao *bean* estão em um arquivo descritor de implantação (*deployment descriptor*), especificadas na linguagem XML (eXtensible Markup Language) (W3C, 1998). Essas regras são definidas em tempo de projeto ou mesmo em tempo de execução usando uma ferramenta. Em tempo de execução, o *container* executa os serviços de suporte acordo com as regras definidas nestes arquivos de "*deployment*" (THOMAS, 1998).

3.3 Modelo de Componentes CORBA (CCM)

Desde 1989, o *Object Management Group* (OMG) tem se preocupado em especificar um padrão aberto de middleware para a programação distribuída orientada a objeto chamado de CORBA (*Common Object Request Broker Architecture*) (OMG, 2002b). O CORBA, permite a invocação de operações em objetos distribuídos sem a preocupação quanto a localização do objeto, linguagem de programação, plataforma, protocolo de comunicação ou hardware. A OMG também especifica um conjunto de objetos de serviço CORBA, com interfaces padronizadas para acessar estes serviços distribuídos comuns, tal como: serviço de nomes, segurança, notificação de eventos, entre outros. Através do CORBA e seus objetos de serviço, desenvolvedores de sistemas podem integrar e montar aplicações amplas e complexas usando propriedades e serviços de diferentes fornecedores.

Infelizmente, o modelo de objetos do CORBA, apresenta algumas limitações no seu uso (WANG, 2000):

- As interconexões entre objetos ficam disseminadas entre suas implementações (e não através de interfaces), o que dificulta operações de configuração dinâmica. Na programação orientada a objetos é sempre difícil ter uma visão completa de uma

configuração de aplicação.

- O uso dos serviços comuns do CORBA, tal como os serviços de transação, de persistência, de ciclo de vida, de segurança, de notificação de eventos, necessita ser codificado dentro dos objetos, junto com a implementação da funcionalidade do objeto. Isso faz com que o desenvolvimento de aplicações torne-se mais complexo, pois o desenvolvedor deve se preocupar com aspectos técnicos do uso dos serviços de suporte do que somente se preocupar com a parte funcional da aplicação.

- Não existe um padrão para a implantação de objetos.

Para suprir tais limitações, a OMG desenvolveu o Modelo de Componentes CORBA, ou CORBA *Component Model* (CCM) (OMG, 2002a). A especificação CCM foi concluída no final de 2002, constituindo uma das maiores inclusões ao padrão CORBA 3.0. O CCM estende o modelo de objetos CORBA, organizando os objetos em componentes, facilitando o trabalho de modelagem, desenvolvimento, empacotamento, implantação e execução dos componentes. O CCM define também um ambiente padrão para tratar com os serviços CORBA geralmente utilizados (parte não-funcional). O *container* do CCM fornece suporte para os serviços de transação, de persistência, de ciclo de vida, de segurança e de notificação de eventos, tornando possível então a separação dos aspectos funcionais dos componentes de suas necessidades de políticas e serviços de suporte (aspectos não funcionais).

O CCM está estruturado em diversos modelos, os quais têm por objetivo especificar a modelagem, o desenvolvimento, o empacotamento, a implantação e a execução dos componentes. Estes modelos são abordados nas próximas subseções.

3.3.1 Modelo Abstrato (Modelagem)

O modelo abstrato permite aos desenvolvedores definir as interfaces do componente usando a linguagem IDL3[5]. Essas definições são usadas somente para fins de projeto, pois a definição em IDL3 é mapeada para a IDL2[6]. Para cada construção em IDL3 existe uma regra de mapeamento para a IDL2. Esse mapeamento é realizado por um pré-compilador

[5] Extensão da IDL tradicional do CORBA para a definição de componentes.
[6] É a IDL tradicional do CORBA, mas que no CCM recebe o nome de IDL2

para esse fim. A IDL2 gerada a partir desse mapeamento recebe o nome de “IDL equivalente”.

Componentes são definidos usando a palavra-chave `component`. Esta palavra-chave representa um novo meta-tipo em CORBA, a qual é uma especialização de uma interface. Cada definição de componente definida em IDL3 tem uma “interface equivalente” associada em IDL2.

```
interface <component_name> : Components::CCMObject {...};
```

Como acontece com as interfaces com objetos CORBA, um componente também pode ser estendido através do mecanismo de herança, porém somente herança simples é permitida. Além da herança um componente também pode dar suporte a uma ou mais interfaces.

A Figura 3.4 mostra o conceito de herança aplicado aos componentes, como também das interfaces relacionadas. Nessa figura pode ser observado um componente A, que dá suporte a uma interface I, um componente B que herda o componente A, todos esses, definidos pelo desenvolvedor. Pode-se observar também, que a interface B, herda a interface A (estas duas interfaces são geradas através do mapeamento da IDL3 em IDL2) e que a interface A por sua vez, herda a interface *CCMObject* (pré-definida), que é herdada por todos os componentes implicitamente. E por fim pode ser visto, que a interface *CCMObject* herda três outras interfaces: a interface *Navigation*, a interface *Events* e a interface *Receptacles* (pré-definidas).

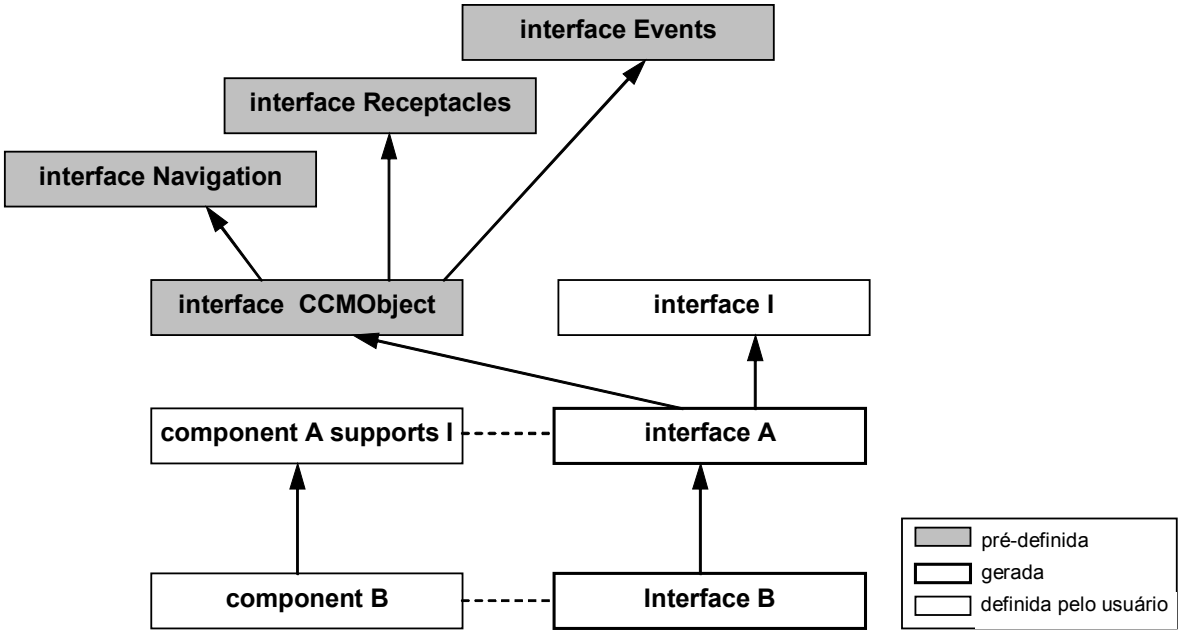

Figura 3.4 – Herança de componentes e interfaces relacionadas.

Um componente é composto por atributos e portas. Atributos são propriedades do componente que têm como propósito a configuração do mesmo[7]. As portas são pontos de conexão entre os componentes, através das quais os componentes interagem. A cada definição de porta em IDL3 é feito um mapeamento específico em IDL2. São definidos quatro tipos de portas (OMG, 2002a), mostradas na Figura 3.5:

- Facetas (Facets): são as interfaces através das quais um componente oferece seus serviços a clientes.
- Receptáculos (Receptacles): são interfaces usadas por componentes, no acesso a serviços de outros componentes.
- Produtores de Eventos (Event Sources): são interfaces que emitem eventos de um tipo específico para um ou mais consumidores de eventos.
- Consumidores de Eventos (*Event Sinks*): são as interfaces pelas quais um componente é notificado da ocorrência de eventos de um determinado tipo.

Como objetos CORBA, as instâncias de componentes também são caracterizadas por

[7] Os atributos são usados principalmente para configurar o componente. Atributos podem ser usados por ferramentas de configuração para predefinir valores de configuração para um componente, visando estabelecer o comportamento do componente. O CCM permite operações para acesso e modificação dos atributos, as quais podem provocar exceções.

suas interfaces e por uma referência – a referência do componente. Esta referência permite que os clientes tenham acesso às portas da instância do componente.

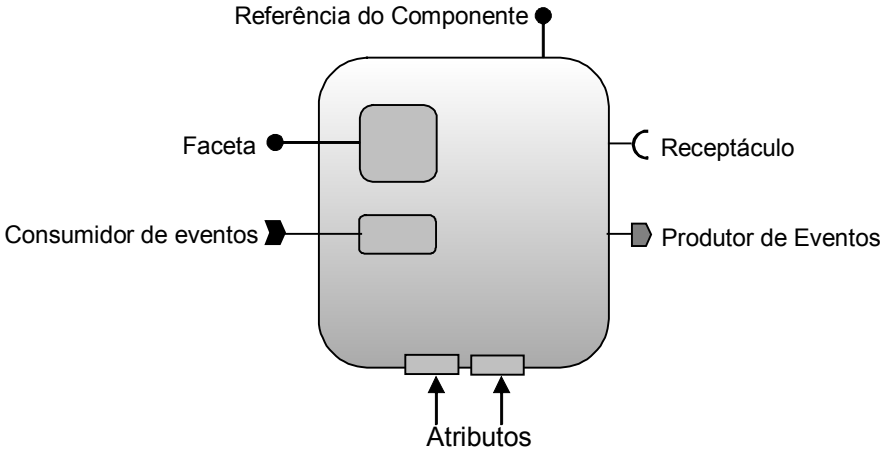

Figura 3.5 – Mecanismo de Portas.

As portas são na verdade interfaces expostas ou usadas pelo componente. As facetas e os consumidores de eventos são as interfaces expostas pelos componentes, enquanto que os receptáculos e as fontes de eventos são as interfaces usadas pelos componentes. Os diferentes tipos de portas são usados em tempo de implantação e de execução para a conexão entre os componentes.

A conexão entre as portas deve ser feita através de portas compatíveis. Desta forma, facetas devem ser conectadas com receptáculos, pois estas estão fundamentadas no mecanismo de comunicação síncrona, enquanto que os produtores de eventos devem ser conectados com consumidores de eventos, porque fazem uso do mecanismo de comunicação assíncrona (serviço de eventos).

3.3.1.1 Facetas

No contexto de objetos CORBA, um objeto tem uma única interface que agrupa todas as operações disponíveis. Portanto, quando um objeto herda de diversas interfaces, ele terá todas as operações das interfaces agrupadas. Os componentes superam essa limitação através do uso das facetas, onde cada uma expressa um aspecto funcional do componente.

Cada componente poderá prover zero ou mais facetas. Cada faceta tem sua própria referência, denominada referência de faceta. Como um componente pode prover várias

facetas, este poderá então ter diversas referências de objetos, onde cada referência corresponde a uma faceta. Assim, um componente tem uma referência para ele, e nenhuma ou várias referências de facetas.

Como objetos, componentes também contam com o conceito de encapsulamento, onde a implementação de cada faceta é encapsulada pelo componente. Desta forma, as facetas não podem ser dissociadas do componente a que pertencem, o que implica que as facetas existirão enquanto o componente existir.

Através da interface equivalente, os clientes poderão navegar entre as facetas do componente e se conectar as portas do componente. Cada porta do componente é mapeada em um conjunto de operações na interface equivalente, as quais fornecem as funcionalidades de acordo com o tipo de porta. A Figura 3.6 mostra o relacionamento entre o componente, suas referências e as facetas (OMG, 2002a).

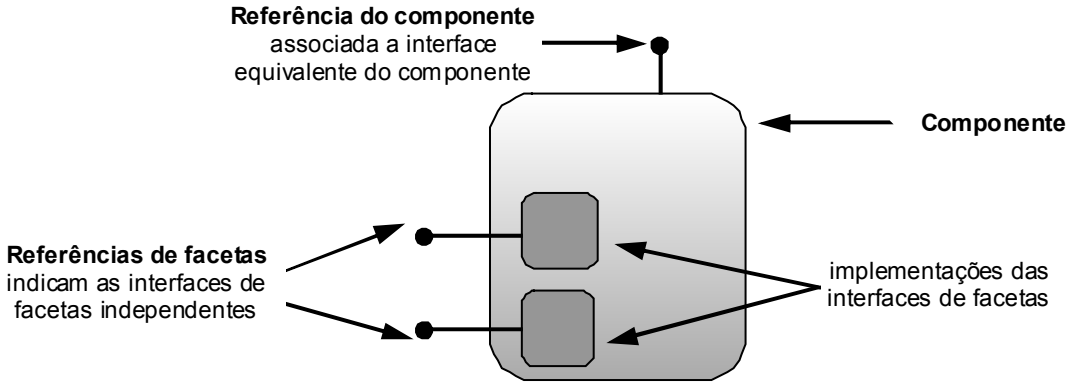

Figura 3.6 – Interfaces e Facetas do Componente.

3.3.1.2 Receptáculos

Um receptáculo é uma porta que permite a um componente, através de uma referência (referência de componente, referência de faceta ou referência de objeto), estabelecer uma conexão com uma faceta do mesmo tipo de um outro componente e, portanto, permite invocar operações desta interface. O relacionamento entre um componente e uma referência é chamado de conexão, conseqüentemente, quando é estabelecida uma conexão entre as duas partes é dito que estão *conectadas.* Portanto, um receptáculo representa um ponto de conexão conceitual, pois esta abstração é concretizada em um componente através de um conjunto de operações de estabelecimento e gerenciamento de conexões. Um componente pode ter zero ou mais receptáculos.

Os receptáculos podem ser de dois tipos distintos: *simplex* ou *multiple.* O receptáculo do tipo *simplex* permite que somente uma única conexão ao receptáculo pode existir em um dado período de tempo, enquanto que o tipo *multiple* permite que existam diversas conexões simultâneas ao mesmo receptáculo.

3.3.1.3 Eventos

O modelo de eventos utilizado pelo CCM é compatível com o modelo de notificação de eventos da especificação dos serviços comuns do CORBA (OMG, 2001). Para receber eventos, um consumidor precisa se subscrever junto aos produtores de eventos. O CCM usa somente o modelo *push*, onde o produtor de eventos informa a todos os consumidores que um evento foi produzido.

Os produtores de eventos são responsáveis por produzir eventos para os consumidores de eventos, através de portas "*event source*". Existem dois tipos de produtores de eventos: o tipo *emitter* que permite somente um consumidor para seus eventos e o tipo *publisher* que habilita vários consumidores aos seus eventos. No tipo *emitter*, os eventos são colocados em um canal de eventos compartilhado, o qual pode ser usado por um ou mais produtores de eventos. No tipo *publisher*, é usado um canal de eventos para cada *publisher*, onde consumidores devem se inscrever para receber os eventos publicados pelo produtor. Um componente pode ter zero ou mais *emitters* e/ou *publishers*.

Uma porta "*event sink*" permite a um componente receber eventos produzidos por produtores de eventos. Não há distinção entre consumidores de eventos, como no caso dos produtores. A porta "*event sink*" é usada para para receber eventos tanto de *emitters* quanto de *publishers*.

3.3.1.4 Declaração de portas e atributos

Como já dito, um componente é composto por portas e atributos, portanto, a declaração de um componente, consiste da declaração das portas e atributos que compõem o componente. A declaração básica de um componente em IDL3 na Figura 3.7 mostra as palavras chaves que definem facetas, receptáculos, produtores e consumidores de eventos e atributos.

```
component <nome_do_componente> {
      //faceta
      provides <tipo_interface> <nome_faceta >;

      //receptáculo do tipo simplex
      uses <tipo_interface> <nome_receptaculo>

      //receptáculo do tipo multiple
      uses multiple <tipo_interface> <nome_receptaculo>;

      //produtor de eventos (emitter)
      emits <tipo_evento> <nome_produtor>;

      //produtor de eventos (publisher)
      publishes <tipo_evento> <nome_produtor>;

      //consumidor de eventos (event sink)
      consumes <tipo_evento> <nome_consumidor>;

      //atributo
      attribute <tipo_attributo> <nome_attributo>;
};
```

Figura 3.7 – Declaração de Componente em IDL3.

3.3.1.5 Homes

As *homes* são interfaces de gerenciamento de instâncias de componente de um tipo. A *home* equivale ao operador *new* em modelos orientados a objetos, na criação de instâncias; mas também fornece outras operações de ciclo de vida de componentes, como: operações para remoção de instâncias de componentes e operações para localização de instâncias de componentes, etc (WANG, 2000).

Um componente é definido independentemente de sua *home*, mas uma *home* deve especificar o tipo de componente que gerencia. Muitas *homes* podem gerenciar o mesmo tipo de componente, porém não as mesmas instâncias, ou seja, uma instância de componente é gerenciada por somente uma única *home*.

Opcionalmente uma *primary key* pode ser associada a uma instância de componente. As *primary keys* são identificadores únicos de instâncias dos componentes; servem para a localização das mesmas no servidor.

Dois tipos de operações opcionais são definidos nas *homes*: *factorys* e *finders*, que retornam a referência para uma nova instância de componente ou para uma instância existente, respectivamente. *Finders* utilizam a *primary key* para localizar a instância do componente correspondente. As implementações das *factories* e *finders* são geradas

automaticamente pelo CCM, porém o desenvolvedor do componente pode criar suas próprias *factories* e *finders*, de acordo com suas necessidades.

O mapeamento das definições da *home* é dividido em três partes. As interfaces explícitas que agrupam as operações declaradas pelo desenvolvedor, as interfaces implícitas que agrupam as operações genéricas de uma *home* de componente (geradas pelo CCM) e por fim, a interface final que herda estas duas interfaces.

3.3.2 Modelo de Programação (Desenvolvimento)

O principal objetivo do modelo de programação é descrever as partes não funcionais (serviços do sistema, tal como persistência, ciclo de vida, etc.) que o componente utiliza, permitindo que esta parte da implementação seja gerada automaticamente, de maneira que o desenvolvedor de componentes precisará somente suprir a implementação da parte funcional (lógica do negócio).

De modo a integrar essas duas partes o CCM define o *Component Implementation Framework* (CIF), o qual descreve como a parte funcional e a parte não funcional devem interagir. O CIF utiliza a CIDL (*Component Implementation Definition Language*), uma linguagem declarativa usada para descrever a estrutura da implementação do componente assim como descrever a persistência do estado do componente.

O CIF usa as descrições em CIDL, juntamente com a descrição em IDL do componente para gerar, através do compilador CIDL, os esqueletos de implementação do componente. Estes esqueletos de implementação implementam as especificações IDL2 de um componente, fornecendo operações para descobrir as portas do componente, navegação entre as portas e operações para efetuar as conexões (operações de conexão e de desconexão de uma faceta a um receptáculo, operações de inscrição ou de cancelamento junto a um produtor de eventos, entre outros).

O compilador CIDL também gera os descritores de componente, que têm como função especificar as características do componente, como a descrição de suas portas, o tipo do componente, o tipo de ciclo de vida utilizado, entre outras. Os descritores de componentes estão descritos na seção 3.3.4.

O processo de geração de um componente é esboçado na Figura 3.8, onde a partir do

arquivo em IDL3 contendo a descrição do componente, é gerado pelo compilador IDL3/IDL2, o arquivo com o mapeamento da IDL3 em uma especificação IDL2. A partir da IDL2, são gerados pelo compilador IDL, a *stub* de cliente e o *skeleton* de servidor. Através do arquivo IDL3 e do arquivo CIDL o compilador CIDL gera a estrutura inicial de implementação do componente e o descritor do componente. Então este "esqueleto" de implementação, o código da implementação do componente e o *skeleton* do servidor são compilados por um compilador de uma linguagem de programação que irá gerar o código executável do componente. O código executável do componente e seu descritor são empacotados em um *package*, que poderá conter uma ou mais implementações de um componente, onde cada uma corresponde à implementação do mesmo componente, porém para diferentes plataformas. Para gerar o código do cliente, o compilador da linguagem de programação usa o *stub* do cliente e o código do cliente.

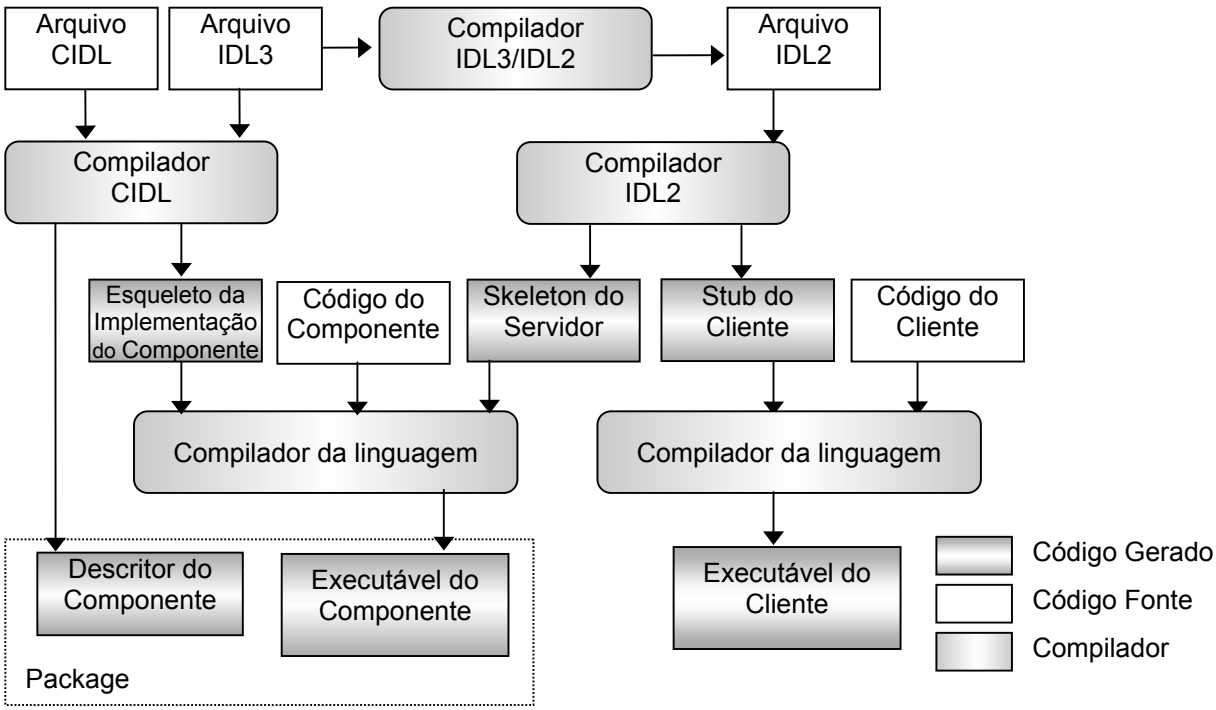

Figura 3.8 – Processo de geração do Componente

Um componente é considerado um conjunto de elementos comportamentais providos por seus executores. Executores são as implementações geradas a partir da CIDL. Dois tipos de executores são definidos: executores de componentes (*executor*) e executores de *homes* (*home executor*). A implementação de um componente, é composta pela agregação

de elementos com comportamento e relacionamento específicos. A CIDL é usada para definir essa agregação, chamada de composição (composition).

A definição básica de uma composição em CIDL é mostrada na Figura 3.9.

```
composition <categoria> <nome_da_composição> {
      home executor <nome_do_home_executor> {
            implements <home_type>;
            manages <nome_do_executor>;
      };
};
```

Figura 3.9 – Composição em CIDL.

Um composição é basicamente definida por:

- Nome da composição: que identifica a composição.
- Categoria do componente: que especifica a que categoria o componente pertence, podendo ser: *service*, *session*, *process* ou *entity*. As categorias de componentes estão descritas na seção 3.3.3.
- *Home* do componente: A composição específica o tipo de *home* do componente, a qual é definida em IDL3.
- Executor de *home*: a composição especifica uma definição para o executor de *home*. O executor de *home* é quem implementa a *home* do componente. O nome do executor de *home* será usado como nome do elemento de programação que implementa a *home* do componente (por exemplo, o nome da classe). O executor de *home* é gerado automaticamente pelo CIF.
- Executor de Componente: a composição especifica uma definição para o executor do componente que é quem implementa o componente. O nome do executor de componente será usado como o nome para o elemento de programação que implementa o componente. O executor de componente é gerado automaticamente pelo CIF. O executor de componente pode ser dividido em segmentos, ou seja, sua implementação pode ser dividida em diversas classes, onde cada classe pode implementar uma faceta distinta do componente, sendo que cada uma pode ter requisitos de persistência distintos.

Na Figura 3.10 é mostrado de maneira esquemática a composição apresentada anteriormente. Nesta figura é possível observar que o componente não está explicitamente especificado, ficando a sua especificação implícita na especificação do tipo da *home*.

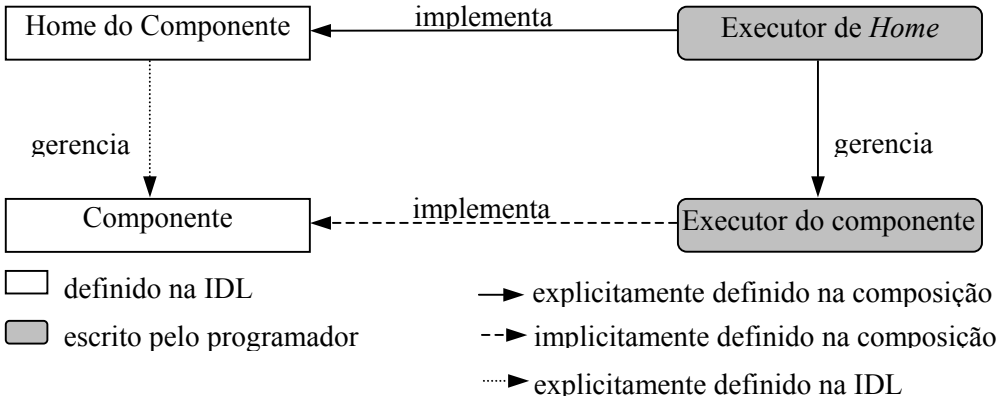

Figura 3.10 – Estrutura de uma composição e seus relacionamentos

3.3.3 Modelo de Execução

Os componentes são executados através de um *container*. O *container* é o mecanismo responsável por prover um ambiente de execução para um componente, fornecendo um acesso transparente aos serviços do CORBA (aspectos não funcionais), tais como: persistência, transações, segurança, notificação de eventos e ciclo de vida. O *container* pode ser visto como um *framework* responsável por integrar os serviços do CORBA. O principal objetivo do *container* é a separação de parte funcional e parte não funcional da construção de um sistema. Desta forma, o programador de aplicação é responsável por apenas implementar operações relacionadas à lógica da aplicação, enquanto o *container* gerencia toda a parte não funcional.

Componentes e *homes* são implantados em *containers* através de ferramentas de implantação (seção 3.3.5). Na especificação de componentes CORBA não está definido explicitamente o número de componentes que pode existir dentro de um *container*, ficando a cargo de quem implementa a especificação definir.

A comunicação entre componentes e *container* é realizada através de chamadas locais; o *container* ativa as operações do componente através de interfaces de *callback*, que são implementadas pelo programador do componente, e o componente ativa operações do *container* através de interfaces internas, implementadas pelo *container*. A interação dos clientes e o componente é feita através das interfaces externas, definidas através da IDL do

componente e sua *home*.

Como no modelo de objetos CORBA, os componentes também necessitam do POA (*Portable Object Adapter*) para encaminhar as solicitações dos clientes aos respectivos *servants*. Porém ao contrário como acontecia com o modelo de objetos, onde o programador do objeto era responsável por instanciar o POA e configurá-lo de acordo com as políticas desejadas, no modelo de componentes é o *container* que executa essa tarefa. Portanto, um *container* além de conter o componente, ele também possui uma instância do POA especializada para esse componente. No descritor do componente estão descritos os aspectos não funcionais utilizados pelo componente, assim como as políticas para a criação do POA.

Na Figura 3.11 é mostrado o modelo de execução, com os elementos citados acima. Na figura pode ser observado que *container* obtém acesso aos serviços do CORBA através do ORB (*Object Request Broker*).

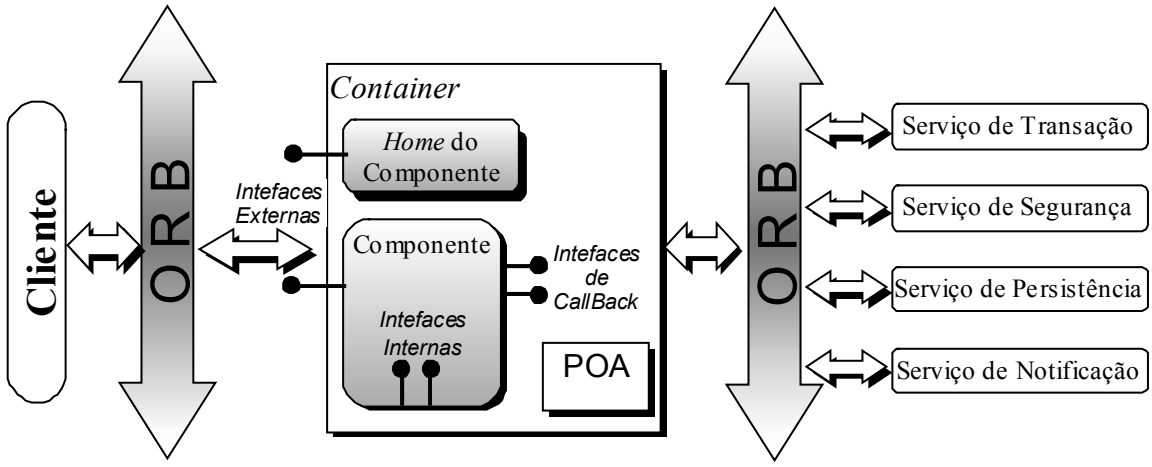

Figura 3.11 – Arquitetura do modelo de execução.

O modelo de execução então especifica tipos de APIs externas, tipos de API de *container*, modelo de uso do CORBA e tipos de componentes.

Os tipos de APIs externas (interfaces *home* e interfaces do componente) especificam um contrato entre o programador do componente e os clientes do componente. Elas são definidas em IDL3 e armazenadas em um repositório de interfaces para uso dos clientes.

Os tipos de APIs de *container* especificam um contrato entre um componente e o seu *container*. Dois tipos de API de *container* são definidos. A API do tipo *session* que define

um *framework* para componentes que usam referências de objetos transientes e a API do tipo *entity* que define um *framework* para componentes que usam referências de objetos persistentes.

O modelo de execução do CORBA define o padrão de interação entre o *container*, o POA e os serviços do CORBA. Três modelos de interação são possíveis no CCM, definidos com base no tipo de referência de objetos (transiente ou persistente), e no tipo de mapeamento de um *servant* para o ObjectID:

- *stateless* – usa referências de objetos transientes em conjunto com um servant POA que pode suportar qualquer ObjectID, isto é, um *servant* que pode ter mais de um identificador de objeto;
- *conversational* – usa referências de objetos transientes em conjunto com um *servant* POA que é dedicado a um ObjectID específico;
- *durable* – usa referências de objetos persistentes em conjunto com um *servant* POA que é dedicado a um ObjectID específico.

Os tipos de componentes definidos na especificação CCM autorizam combinações de tipos de API de *container* e modelos de interação do CORBA (Tabela 3.1).

Modelo de interação	**API do *Container***	**Categoria do Componente**
stateless	*session*	*serviço*
conversational	*session*	*sessão*
durable	*entity*	*processo*
durable	*entity*	*entidade*

Tabela 3.1 – Definição das categorias de componentes.

Como pode ser visto na Tabela 3.1, o CCM define quatro tipos de componentes:

- **Serviço:** são componentes que não têm estado e nem identidade. O seu ciclo de vida está limitado à duração de uma requisição de serviço. Esses componentes são geralmente usados para representar funções que não necessitam armazenar informações durante invocações de um mesmo cliente. Um exemplo deste tipo, é um componente que executa algum tipo de cálculo para outros componentes, quando uma requisição de serviço é feita nesse componente, ele executa o cálculo

e retorna o resultado.

- **Sessão:** são componentes que têm estado transiente e sua identidade não é persistente. O estado do componente existe enquanto existir o componente, portanto, o estado do componente será perdido quando o servidor onde o componente está instanciado parar de funcionar. Cada componente do tipo sessão é geralmente associado com um cliente, portanto o ciclo de vida deste tipo de componente é a seqüência de invocações de operações solicitadas pelo cliente, como por exemplo, uma cesta de compras em uma aplicação de comércio eletrônico, onde enquanto estiver sento feita a compra, os itens adicionados ao carrinho serão guardados até que o cliente encerre a compra ou desista dela.

- **Processo:**: são componentes com estado e identidade persistentes. O estado do componente não é visível para o cliente, enquanto a identidade pode ser visível somente se o desenvolvedor definir operações para este fim. O estado desse tipo de componente pode ser gerenciado pelo *container* ou pela implementação do componente. Componentes deste tipo geralmente representam processos de negócios, tais como um empréstimo ou criação de uma conta ao invés de representarem entidades, tais como clientes ou contas.

- **Entidade:** são componentes com estado e identidade persistentes. O seu estado e a identidade são visíveis para o cliente. A identidade desse tipo de componente é visível para seus clientes através de uma chave primária (*primary key*) declarada na *home* do componente. O estado desse tipo de componente pode ser gerenciado pelo *container* ou pela implementação do componente. Este tipo de componente é geralmente usado para modelar entidades de negócio com clientes de um empresa, fornecedores, contas-correntes em um banco. A maior diferença entre componentes do tipo processo e entidade é que componentes do tipo processo não expõem sua identidade persistente para o cliente (a não ser através de operações definidas pelo desenvolvedor).

São definidas quatro políticas de controle de ativação e desativação dos componentes (OMG, 2002a):

- *Método*: ativação/desativação a cada chamada de método, limitando o uso de memória ao tempo de duração da operação, mas acrescentando o custo de ativação

e desativação do componente.

- Transação: ativação/desativação a cada transação, ou seja, a memória permanece alocada durante a transação ou sessão de computação.
- Componente: o *container* ativa o componente quando é feita a primeira chamada a alguma de suas operações, e o desativa quando requisitado pela aplicação, liberando a memória utilizada pelo componente.
- *Container*: o *container* ativa o componente quando é feita a primeira chamada a alguma de suas operações e continua ativo até o *container* determinar a necessidade de desativá-lo. Portanto, a memória permanecerá alocada até que o *container* decida liberá-la por questões de necessidade de memória, de tempo de uso, etc.

3.3.4 Modelo de Empacotamento/Distribuição

O modelo de empacotamento define como os componentes devem ser empacotados para posterior distribuição. Este modelo permite o empacotamento tanto de componentes individuais quanto de configurações inteiras de componentes.

No contexto de componentes CORBA, um pacote (*package*) de software é um arquivo no formato "ZIP", contendo uma ou mais implementações de um tipo de componente e um conjunto de descritores. A possibilidade de ter mais de uma implementação para o mesmo tipo de componente, deve-se ao fato de fornecer implementações do componente para diferentes linguagens e plataformas.

Os descritores são escritos usando o OSD *(Open Software Description*), o qual é uma definição de tipo de documento (DTD) XML (*eXtesible Markup Language* (W3C, 1998)). Os descritores contidos em um pacote são: o descritor do pacote, o descritor do componente e o descritor de propriedades. Para facilitar a identificação dos tipos de descritores, o CCM padroniza a extensão para o nome do arquivo de cada tipo de descritor.

Um pacote de software pode ser implantado separadamente ou pode ser agrupado com outros pacotes de componentes de modo a formar um pacote *assembly*. Um pacote *assembly* é um conjunto de componentes inter-relacionados. Este tipo de pacote contém, além de pacotes de componentes; um descritor *assembly* que especifica os componentes

que o integram e as conexões entre componentes. O arquivo *assembly* tem a extensão ".aar" (*assembly* archive) e o descritor *assembly* tem a extensão ".cad" (c*omponent assembly descritor*).

Os descritores de pacotes de software fornecem informações gerais sobre o software, como autor, companhia, descrição do componente, entre outros. Além disso, o descritor do pacote provê a descrição de todas as implementações do componente, e características associadas como sistema operacional, linguagem de implementação, etc. Um arquivo descritor de pacote tem a extensão padrão ".csd" (*component software descriptor*).

O descritor do componente descreve a estrutura do componente, tal como as interfaces herdadas, as interfaces suportadas, as portas do componente, o tipo do componente, tipo de ciclo de vida utilizado, etc. Utilizando as informações contidas neste descritor, uma ferramenta de implantação pode conectar diversos componentes para construir uma aplicação completa. As informações contidas neste descritor são usadas para escolher o tipo de *container* que será usado pelo componente, como também definir o tipo de ciclo de vida utilizado. Um arquivo descritor de componente tem a extensão padrão ".ccd" (CORBA *component descriptor*). O compilador CIDL gera uma parte do conteúdo deste descritor, o restante é escrito pelo programador do componente, como o tipo de ciclo de vida utilizado pelo componente.

O descritor de propriedades é usado para configurar as instâncias de um componente ou de uma *home*. Um arquivo descritor de propriedades tem a extensão padrão ".cpf" (*component property descriptor*). O descritor de propriedades pode ser referenciado pelo descritor do pacote para definir os valores *default* para o componente, como também pode ser referenciado pelo descritor de *assembly* para definir os valores iniciais para os componentes e *homes*.

3.3.5 Modelo de Implantação

O modelo de implantação fornece meios para a instanciação de uma aplicação distribuída composta por componentes. Durante o processo de implantação é realizada a implantação e ligação de uma topologia lógica de componentes ao ambiente computacional físico. A topologia lógica de componente é especificada pelo pacote *assembly* ou pelo pacote de software individual.

De maneira geral, os passos básicos para o processo de implantação são:

- definir em quais *hosts* os componentes serão instalados. Esta informação é geralmente resultante da interação entre a ferramenta e o usuário;
- instalar as implementações dos componentes nos *hosts*, de acordo com a plataforma destino;
- instalar os servidores e *container*s nos *hosts*, caso ainda não existam;
- criar as instâncias das *homes* e dos componentes nos *hosts*;
- conectar os componentes como especificado no *assembly* descritor, facetas com receptáculos, produtores com receptores de eventos.

Todo o processo de implantação de componente ou pacote *assembly* recebe suporte de uma ferramenta de implantação em conjunto com objetos que auxiliam neste processo. O processo de implantação de componentes é dependente da ferramenta de implantação.

3.4 Trabalhos Relacionados

Está seção apresenta alguns trabalhos relacionados a componentes CORBA. O primeiro deles trata da replicação de componentes CORBA, o segundo trata de uma abordagem de reconfiguração dinâmica para suporte de tolerância a faltas e o último trata da adição de requisitos de QoS a componentes CORBA.

3.4.1 Replicação de Componentes CORBA

Em (MARANGOZOVA; HAGIMONT 2002) é apresentada uma abordagem para replicação de componentes CORBA. Nessa abordagem são usados objetos de interceptação, que são responsáveis por capturar as requisições feitas para o componente de modo a disparar as ações necessárias para o gerenciamento da replicação. Os objetos de interceptação permitem integrar o gerenciamento da consistência das réplicas com o processamento da aplicação, sem modificar a necessidade de modificar o código do componente. A manutenção da consistência entre as réplicas é realizada através de interconexões entre as cópias para propagar as ações de consistência. Essas interconexões recebem o nome de “ligações de consistência” (*consistency link*).

As ligações de consistência implementam protocolos de consistência que estabelecem as relações de consistência entre as réplicas e fornecem o tratamento necessário para manter estas relações válidas. A maioria dos protocolos de consistência precisa acessar os dados internos do componente com o objetivo de ter acesso a informações de estado do componente. Deste modo, primitivas de transferência de estado devem ser implementadas pelo componente.

Os objetos de interceptação implementam a mesma interface do componente que será replicado, o que implica que toda a vez que se desejar replicar uma nova aplicação é necessário implementar um novo objeto de interceptação com a mesma interface do componente de aplicação.

Esta abordagem pode ser aplicada a qualquer aplicação composta por componentes CORBA. A replicação é adicionada à aplicação em tempo de implantação (configuração), através do programa de implantação dos componentes. Como esses programas são responsáveis por controlar a criação das instâncias dos componentes e a suas interconexões, eles também são responsáveis pela criação das réplicas e da conexão destas com outros componentes. Em um exemplo simples de replicação (Figura 3.12), o programa de implantação deve conectar um componente cliente a uma cópia local de um componente servidor e fazer as conexões (ligações de consistência) entre a réplica local e a réplica remota para mantê-las consistentes. No exemplo de replicação citado, a técnica de replicação é a passiva (seção 2.4.2)

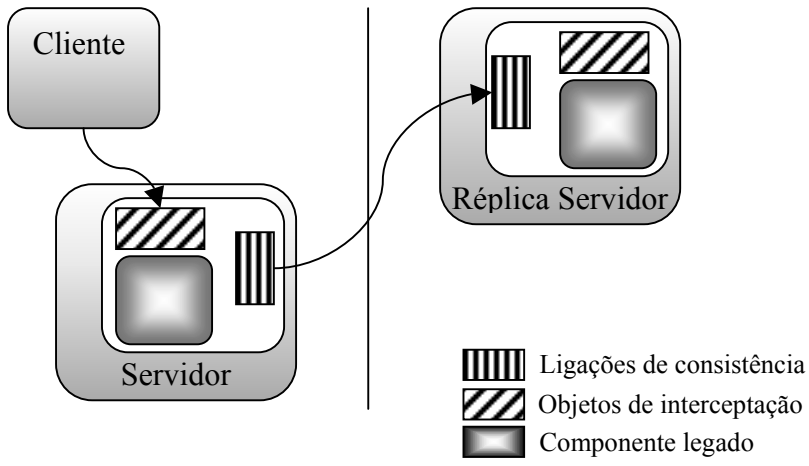

Figura 3.12 –Esquema Simples de Replicação (MARANGOZOVA; HAGIMONT 2002).

3.4.2 Reconfiguração Dinâmica com Suporte de Tolerância a Faltas

Em (BATISTA; CARVALHO, 2002) é apresentado um mecanismo usado na composição de aplicações, que oferece suporte para seleção dinâmica de componentes e inclui aspectos de tolerância a faltas, fornecendo uma execução livre de faltas enquanto o mecanismo estiver controlando a aplicação. Esse mecanismo é chamado de "*conector genérico*", que faz a seleção dinâmica de componentes através da assinatura de um método que o componente deve prover. Neste trabalho o objetivo vai além de fazer a invocação de um método em um determinado componente e retornar os resultados. O que se deseja é executar tais tarefas e garantir que qualquer invocação feita pelo conector genérico seja livre de faltas.

Conector genérico

O conector genérico permite que se escreva uma aplicação estabelecendo apenas os serviços que se deseja usar, sem determinar o(s) componente(s) que oferece(m) tais serviços. O conector genérico seleciona dinamicamente os componentes que oferecem os serviços requeridos que irão compor a aplicação. No processo de seleção destes componentes, o conector genérico usa, como critério de busca, a assinatura dos métodos correspondentes aos serviços desejados a serem localizados.

Quando um método é invocado sobre o conector genérico este invoca uma função de busca para procurar por componentes que oferecem o serviço especificado na chamada. A busca é realizada em um repositório padrão (Serviço de Nomes ou *Trading*) ou sobre uma tabela de configuração, que é uma tabela gerenciada pelo conector genérico, contendo a identificação dos componentes selecionados para executar serviços invocados pela aplicação via o conector genérico. Esta tabela funciona como uma espécie de *cache*. O conector genérico pode ser instruído pelo programador da aplicação para realizar a busca no repositório ou para consultar primeiro a tabela de configuração. O comportamento padrão é primeiro consultar a tabela de configuração (*cache*) e só procurar no repositório se nenhum componente for encontrado na tabela de configuração.

Ao encontrar um componente que satisfaça tal critério de busca, o conector monta a requisição CORBA, registra o método e seu provedor na tabela de configuração, caso ainda não esteja na tabela e, então, invoca o serviço sobre o componente selecionado. Por fim, após a execução do serviço o conector retorna os resultados.

Este mecanismo introduz uma grande flexibilidade na modelagem da aplicação, uma vez que o desenvolvedor não precisa se preocupar com quais componentes irão executar os serviços. Outro aspecto refere-se ao potencial de reconfiguração dinâmica que é introduzido pelo conector genérico, pois a cada chamada de um mesmo serviço, diferentes componentes podem ser selecionados para executá-lo.

Tolerância a faltas no Conector Genérico

O conector genérico inclui um mecanismo de tolerância a faltas, cujo objetivo é garantir que o mecanismo tente satisfazer todas as invocações sob sua responsabilidade resolvendo, sempre que possível, as falhas decorrentes da não disponibilidade de um componente selecionado. Neste caso, ao invés de relatar o problema para o usuário, o conector seleciona um outro componente que ofereça o serviço solicitado. A falha de uma invocação só é propagada para o usuário caso o conector genérico tenha esgotado as possibilidades de encontrar algum componente para executar o serviço.

De modo a fornecer tolerância a faltas, o conector genérico possui um tabela, chamada de tabela de grupo de componentes. Esta tabela mantém a lista das identificações dos componentes que fornecem o método invocado sob o conector genérico. Na primeira busca feita no repositório padrão, a função de busca irá retornar uma lista de todos os componentes cuja interface tem o método invocado sob o conector genérico e esta lista será armazenada na tabela de grupo de componentes. Isto evita uma nova busca no repositório, quando ocorrer alguma falha em algum componente, o que tornaria o processo de buscar por um novo componente demorado. Essa idéia de uma tabela de grupo de componentes segue a idéia de grupos de objetos de um determinado tipo, útil para fornecer suporte a tolerância a faltas. Quando da ocorrência de uma chamada para um componente faltoso, não houver outro componente na tabela de grupo que possa atender esta chamada, é realizada uma busca no repositório padrão; caso não seja encontrado um componente que atenda ao critério de busca, então o cliente que efetuou a chamada é avisado.

Nessa abordagem não existe garantia que sempre existirá um componente disponível, o que se garante é somente que quando houver pelo menos um componente registrado no repositório, este será selecionado para executar determinado serviço.

3.4.3 QoS em Componentes CORBA

Embora técnicas de desenvolvimento de software baseado em componentes estejam se tornando maduras para aplicações de negócios, elas ainda não são adequadas para aplicações de missão critica, tais como aplicações de tempo real e aplicações embutidas. Em (BALASUBRAMANIAN et al., 2003) é apresentado o CIAO (*Component-Integrated ACE ORB*), que é uma extensão do CCM. O CIAO foi projetado com o intuito de trazer a tecnologia de componentes para o desenvolvimento de aplicações de tempo real e aplicações embutidas, permitindo estabelecer políticas de QoS e de tempo-real.

O CIAO é baseado no TAO, que é um ORB de tempo real, que implementa mecanismos para tentar alcançar requisitos de QoS de aplicações distribuídas. O CIAO aprimora o TAO para simplificar o desenvolvimento de aplicações de tempo real e embutidas, permitindo aos desenvolvedores especificar de maneira declarativa as políticas de QoS no momento da composição da aplicação.

O CIAO estende o descritor de propriedades de componentes. Este arquivo especifica as propriedades de QoS desejadas, tais como tamanhos dos *buffers* de entrada a serem alocados, porção da largura de banda da rede a ser reservada, e a prioridade dos pacotes de rede enviados pelo componente. O CIAO também estende o *container* de modo a fornecer uma interface para o gerenciamento das políticas de QoS e interagir com os mecanismos necessários para atender as políticas de QoS.

Os desenvolvedores de componentes podem usar o CIAO para desacoplar os aspectos de QoS da implementação de componentes através da declaração dos requisitos de QoS no descritor de propriedades do componente.

3.5 Conclusão

Este capítulo mostrou os principais conceitos relacionados ao desenvolvimento de software baseado em componentes.

O uso de componentes de software no desenvolvimento de aplicações tem se mostrado vantajoso. O processo de construção de software através da reutilização de componentes reduz o tempo de desenvolvimento devido à redução da quantidade de código que precisa ser escrita, levando a acréscimos significativos de produtividade. A qualidade e

a confiabilidade da aplicação construída através da composição de componentes também pode ser melhorada, pois componentes podem já ter sido testados durante seu processo de desenvolvimento e também ter comprovado sua eficácia em outras aplicações, o que leva à redução dos gastos com manutenção. A manutenção do sistema é facilitada, pois será localizada somente no componente onde se faz necessário, através, por exemplo, da troca do componente.

Além da conceituação sobre o desenvolvimento de software baseado em componentes, foi apresentada uma descrição sucinta do modelo de componentes EJB e uma descrição mais detalhada do modelo de componentes do CORBA (CCM), visto que este último foi o adotado para o desenvolvimento deste trabalho.

De modo geral, componentes EJB são comparáveis aos componentes CCM. De fato, a especificação CCM pode ser vista como um super-conjunto do EJB, sendo que a interoperabilidade entre EJB e CCM faz parte da especificação deste último. O CCM define o modelo de componentes sobre o modelo de objetos CORBA, trazendo todas as vantagens deste, como independência de linguagem e sistema operacional, enquanto que o modelo do EJB é baseado em objetos do Java, o que faz com que o EJB seja dependente de linguagem. No CCM a descrição das interfaces dos componentes é feita em IDL, ao contrário do EJB na qual as interfaces são declaradas em Java.

Estes modelos de componentes promovem a separação dos aspectos funcionais dos não funcionais, fazendo com o que desenvolvedor do componente somente se preocupe com a lógica do negócio que o componente se propõe a resolver. Os aspectos não funcionais são deixados a cargo do *container* (ambiente de execução para os componentes), que é responsável por gerenciar os serviços usados pelo componente. Porém, muitos serviços ainda não são providos pelo *container*, como requisitos de qualidade de serviço, de tolerância a faltas e de tempo real.

Apesar das vantagens trazidas pela utilização de componentes de software para o desenvolvimento de aplicações distribuídas, uma aplicação que possua requisitos temporais, de tolerância a faltas, balanceamento de carga, entre outros, não encontra suporte para atendimento destes requisitos nos modelos de componentes atuais. Com isto, propomos um modelo que dá suporte a requisitos de tolerância a faltas para o modelo de componentes CORBA.

Capítulo 4

Modelo TFA-CCM

Neste capítulo é apresentada a proposta de um modelo de tolerância a faltas adaptativa, o TFA-CCM – *Tolerância a Faltas Adaptativa baseada em Componentes CORBA*, que consiste de um conjunto de componentes CORBA, fornecendo suporte de tolerância a faltas adaptativa para aplicações distribuídas. Este suporte de tolerância a faltas é totalmente transparente para a aplicação.

O modelo apresenta soluções capazes de integrar requisitos de QoS que devem guiar a seleção da configuração dos serviços replicados. Deste modo, diferentes níveis de qualidade de serviço (QoS) podem ser especificados, visando atender diferentes requisitos de tolerância a faltas. O TFA-CCM tem mecanismos capazes de reconhecer a necessidade de reconfiguração e da efetivação de mudanças no sistema, que visam atender os requisitos de QoS sem que aspectos como desempenho e estabilidade sejam duramente comprometidos.

Inicialmente será apresentada uma descrição geral do modelo TFA-CCM, seguido por uma descrição detalhada de cada um dos seus componentes.

4.1 Descrição Geral do TFA-CCM

Como abordado na seção 3.8, componentes CCM são executados dentro de um *container*, que é responsável por fornecer um ambiente de execução aos mesmos, provendo serviços de forma transparente. Porém, no conjunto de serviços fornecidos pelo *container* não está incluído o suporte para tolerância a faltas. A partir dessa necessidade, foi proposto

o TFA-CCM. O TFA-CCM é formado por um conjunto de componentes, que juntos visam implantar um esquema de tolerância a faltas adaptativa para as aplicações também compostas por componentes CCM.

O suporte de tolerância a faltas fornecida pelo TFA-CCM é adaptativo, pois permite ao programador de uma aplicação especificar requisitos de qualidade de serviço (requisitos de QoS), definindo níveis desejados de disponibilidade e de confiabilidade do serviço. Através desses requisitos, o suporte de gerenciamento do TFA-CCM seleciona as configurações necessárias de maneira a atender os requisitos especificados. O suporte de gerenciamento atua nas partes que fornecem a tolerância a faltas (partes não funcionais), de forma a reconfigurá-las em função da freqüência em que as falhas ocorrem nos componentes de aplicação de modo a levar o serviço a se distanciar das expectativas de QoS do usuário.

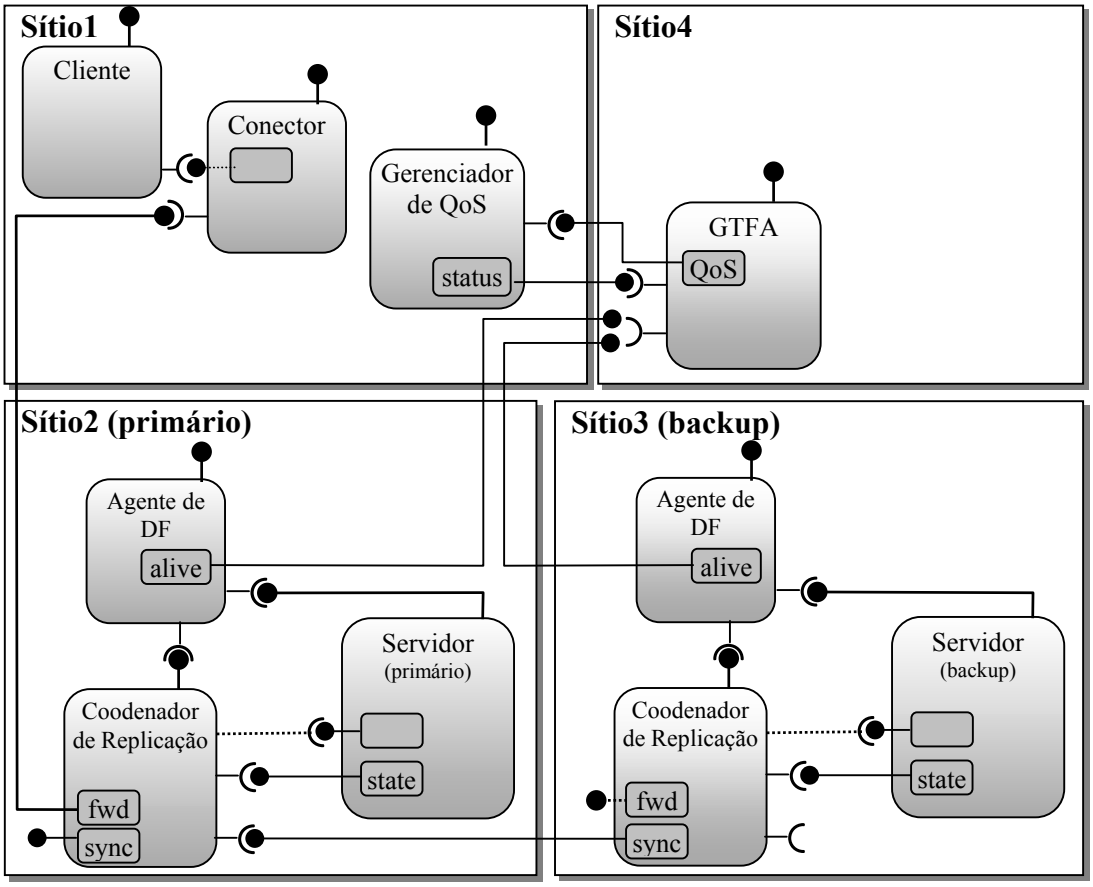

Figura 4.1 – Visão geral do TFA-CCM.

Os componentes que compõem o TFA-CCM podem estar dispostos em vários *hosts* (sítios). A Figura 4.1 ilustra todos os componentes que formam o suporte necessário do TFA-CCM, em uma possível configuração, na qual os componentes estão dispostos em 4 sítios de um sistema distribuído. Essencialmente, estes componentes não funcionais fazem a configuração e o monitoramento da aplicação. A configuração do serviço apresentada na figura usa a técnica de *replicação passiva* com duas réplicas do servidor da aplicação. Nesta configuração, o componente cliente situado no sítio 1 está conectado aos serviços fornecidos pelo componente servidor situado no sítio 2 (réplica primária). O cliente obtém acesso aos serviços do servidor através de um componente chamado *conector*, que tem a função de tornar transparente para o cliente as possíveis mudanças de configuração do servidor. Por exemplo, a troca de réplicas, onde a réplica *backup* substituiria a primária na configuração, não afeta em nada o comportamento do cliente. Na falha da réplica primária, o conector é redirecionado para a réplica *backup* do servidor.

Para cada serviço definido como tolerante a faltas existe um *gerenciador de tolerância a faltas adaptativa* (GTFA – Figura 4.1), que tem a função principal de determinar uma configuração, baseado nas condições do sistema, na freqüência das falhas parciais e nos requisitos de QoS exigidos para a aplicação.

As ações necessárias do GTFA para que uma nova configuração seja alcançada são ativadas a partir de dados de monitoramento da configuração atual comparados com os requisitos do usuário. O monitoramento para verificação de possíveis falhas nos componentes é realizado pelos *agentes de detecção de falhas* (agentes de DF na Figura 4.1). A cada réplica do serviço é associado um agente de DF, localizado no mesmo sítio, que é o responsável pelo envio dos dados de monitoramento ao GTFA.

O *gerenciador de QoS* (seção 4.3) mostrado na Figura 4.1, é usado para especificar os requisitos de QoS no suporte do TFA-CCM. Os requisitos de QoS especificados são repassados ao GTFA, que os interpreta e define uma configuração adequada buscando atender tais requisitos.

Os aspectos relacionados com a aplicação, no caso o servidor tolerante a faltas, são representados pelos componentes replicados que executam os aspectos funcionais da aplicação, e pelo coordenador da replicação responsável pelos aspectos não funcionais, que implementa os algoritmos referentes à coordenação da técnica de replicação selecionada.

O TFA-CCM possui entre suas premissas faltas de *crash* (faltas de parada) para processadores ou componentes. A comunicação no TFA-CCM é assumida como confiável tomando como base as propriedades do protocolo IIOP (*Internet Inter-ORB Protocol*) do CORBA. O IIOP faz uso da pilha TCP/IP como serviço subjacente e está fundamentado na semântica *at most once*, permitindo que eventuais retransmissões de mensagens de pedidos ou respostas provocadas por perdas sejam filtradas no sentido de manter uma só invocação do método requisitado. O suporte de exceções do CORBA é usado para notificar que o alvo de uma invocação não está respondendo e, portanto, está falho (detecção de sítio ou componente em parada). Os *crashes* de componentes ou de sítios são tratados pelo GTFA, que tenta adequar a configuração da aplicação aos requisitos do usuário, restituindo réplicas ou trocando a técnica de replicação.O *crash* do GTFA é tratado pelo gerenciador de QoS.

4.2 Conector

O conector é um mecanismo fornecido pelo TFA-CCM, o qual provê suporte para interconexão dinâmica de componentes CORBA, oferecendo uma maneira transparente de reconfigurar a aplicação. Esse conector é genérico, não ficando restrito a interconectar componentes pré-determinados, mas a atender qualquer tipo de componente.

Quando um cliente deseja fazer o uso de um servidor tolerante a faltas, o cliente deve usar um conector. O conector manipula as requisições dos clientes, permitindo que uma mensagem com a requisição alcance a uma das réplicas do servidor. Desta forma, mantém-se a abstração de uma chamada sobre um servidor único, não replicado. Assim sendo, a transparência das mudanças de configuração no servidor é total para os componentes clientes. Por exemplo, a mudança da técnica de replicação ou do componente primário (caso o técnica de replicação o possua) não afetará em nada o componente cliente.

As setas numeradas na Figura 4.2 indicam o trajeto normal de um pedido de serviço de um cliente: (1) o pedido emitido pelo cliente é passado para o conector, (2) o conector transfere o pedido para o coordenador de replicação, que (3) então aciona a réplica local do componente replicado. No recebimento da resposta (4), o componente coordenador executa o protocolo de coordenação relacionada com a técnica de replicação em uso e então devolve a resposta ao cliente (5 e 6).

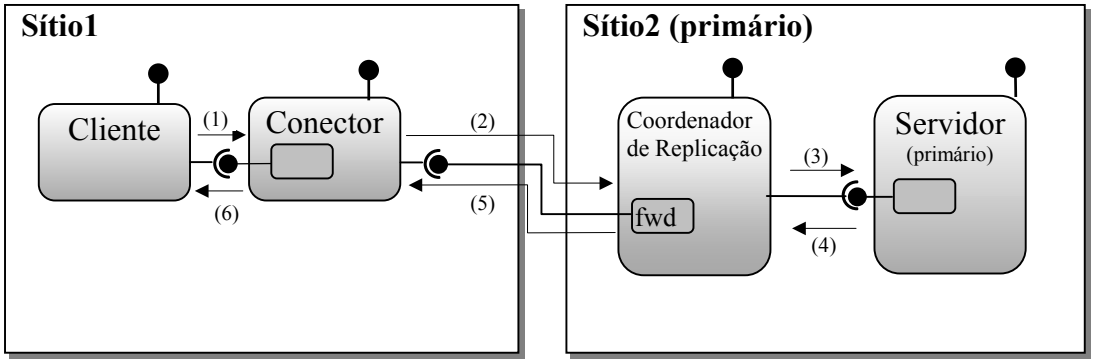

Figura 4.2 – Trajeto de um pedido de serviço.

4.3 Gerenciador de QoS

O gerenciador de QoS é responsável por definir os requisitos de QoS desejados, relacionados especificamente à tolerância a faltas. Os requisitos de QoS são especificados em termos de níveis de QoS, através de uma interface gráfica, onde cada nível especifica uma configuração desejada do sistema. Depois de definidos os níveis de QoS, estes são passados ao GTFA para que o mesmo realize as ações necessárias de modo a atender o nível de QoS especificado.

Em cada nível de QoS, são especificados os seguintes requisitos:

- O número de réplicas;
- A técnica de replicação utilizada;
- O intervalo de monitoramento das réplicas;
- O tempo máximo de espera (timeout) das respostas ao monitoramento dos componentes;
- O intervalo de checkpoint, que determina o número de requisições e o intervalo de tempo entre cada atualização de estado (replicação passiva).

Diferentes níveis de QoS podem ser especificados através do gerenciador de QoS. De modo a manter o nível de confiabilidade desejado, as transições entre os níveis de QoS devem ser definidas. A transição de um nível de QoS para outro se dá quando o nível de QoS corrente não atende mais aos requisitos de confiabilidade desejados. O gerenciador de QoS permite que sejam especificadas as ações a serem tomadas nas seguintes situações:

- Falha de uma réplica de um componente;
- Falha de um sítio utilizado pela aplicação;
- Falha do GTFA.

A detecção de um determinado número de falhas ou a ausência de falhas em um certo intervalo de tempo pode ocasionar mudanças de nível de QoS. Neste caso, condições de transição se tornarão válidas e ocasionarão a mudança da configuração do sistema.

Várias condições de transição podem ser especificadas ligando níveis de QoS diferentes com o objetivo de manter os requisitos de confiabilidade. São atribuídos graus de prioridade às condições de transição. Com isto, caso mais de uma condição de transição seja satisfeita em um determinado instante, será escolhida aquela que tiver o maior grau de prioridade.

Quando ocorrer uma transição entre dois níveis de QoS, o GTFA irá adequar a configuração da aplicação de acordo com os novos requisitos de QoS em vigor. Através do gerenciador de QoS, o usuário pode ainda modificar manualmente os requisitos de QoS ou ativar transições entre níveis, levando o GTFA a reconfigurar a aplicação.

Ao definir os níveis de QoS desejados e transições entre estes níveis, o programador cria uma máquina de estados, conforme ilustrado na Figura 4.3.

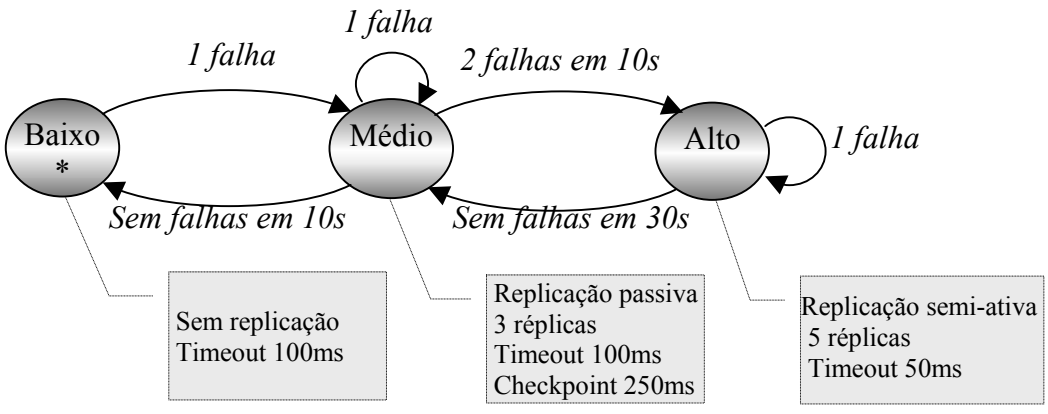

Figura 4.3 – Exemplo de Especificação de QoS.

Na Figura 4.3, tem-se 3 níveis de QoS especificados. O sinal de * (asterisco) define em qual nível o sistema deve iniciar. Portanto, conforme a figura, o sistema inicia executando no nível de QoS baixo, onde nenhuma técnica de replicação é utilizada e o

intervalo de monitoramento para detectar falhas do componente servidor é de 100ms. Quando uma falha ocorrer no componente uma transição será ativada, levando à reconfiguração do sistema. Agora o sistema passa a executar no nível de QoS médio, com replicação passiva, com três réplicas, etc. No decorrer da execução do sistema, podem ocorrer diversas transições entre os níveis de QoS, de modo a garantir o nível de confiabilidade desejado. Porém, se nenhum dos níveis de QoS puder ser alcançado o sistema volta a operar na configuração anterior do sistema.

Os requisitos de QoS podem ser mudados a qualquer momento, em tempo de execução, bastando apenas especificá-los novamente através do gerenciador de QoS que, por sua vez, aciona o GTFA para que este se encarregue de realizar as mudanças necessárias de configuração.

4.4 Gerenciador de Tolerância a Faltas Adaptativa (GTFA)

O GTFA é um dos principais componentes que compõem o TFA-CCM, possuindo duas funções principais. A primeira função é escolher o nível de QoS inicial que deve ser atendido.

A segunda função do GTFA é efetuar as ações necessárias de modo a atender aos requisitos especificados pelo nível de QoS selecionado. As ações que podem ser realizadas são:

- Implantar novas réplicas (componentes, agentes de DF e gerenciadores de replicação) ou restaurar réplicas já existentes que tenham apresentado falha;
- Determinar os sítios onde serão implantadas novas réplicas;
- Trocar a técnica de replicação, que envolve basicamente a substituição do coordenador de replicação;
- Definir qual o componente primário, caso a técnica de replicação o possua;
- Modificar os intervalos de monitoramento e de checkpoint das réplicas.

O papel desempenhado pelos agentes de DF (seção 4.5) é fundamental para o bom funcionamento desta função do GTFA. As ações para a manutenção do nível de QoS, são ativadas a partir de dados de monitoramento da configuração atual comparados com os

requisitos do usuário. Esses dados de monitoramento são fornecidos pelos agentes de DF.

O mecanismo de detecção está baseado em chamadas do GTFA ao método is_alive dos agentes de DF (seguindo o modelo *pull*[8] de monitoramento). O retorno normal ou na forma de exceções CORBA destas chamadas é determinante na detecção de falhas e na composição de informações sobre a configuração atual. Quando estas informações estão em desacordo com o nível de tolerância a faltas selecionado, o GTFA atua dinamicamente, reconfigurando o sistema, de modo a manter o nível de confiabilidade desejado. Caso o GFTA não consiga atender o nível de QoS desejado, o sistema volta a atuar na configuração do sistema.

E por fim, o GTFA também é responsável por fornecer informações de status do sistema para o gerenciador de QoS. Estas informações envolvem a técnica de replicação que está sendo utilizada, a localização dos componentes replicados, estatísticas das falhas ocorridas, etc. O envio periódico destas informações permite a detecção de falha do próprio GTFA a partir do gerenciador de QoS, que disponibiliza estas informações ao usuário. A retomada do GTFA em outra máquina ou na mesma é simples devido à flexibilidade da programação com componentes. O estado do GTFA pode ser facilmente recuperado, por ser salvo em *checkpoints* em um meio de armazenamento persistente.

4.5 Agente de Detecção de Falhas (Agente de DF)

Os agentes de DF são responsáveis por detectar falhas nos componentes replicados e também nos coordenadores de replicação. No TFA-CCM são previstos dois níveis de detecção de falhas: nível de sítio e nível de componente. Esses dois níveis de detecção estão baseados no suporte de chamada remota de métodos do CORBA.

Na detecção de falhas de sítio, o agente de DF recebe periodicamente mensagens do tipo “você está vivo?” do GTFA através de chamadas de métodos do CORBA. A falha de um sítio é assumida quando retornar ao GTFA uma exceção referente à invocação sobre um agente de DF, conforme mostrado na Figura 4.4. Em situação normal, informações de estado da configuração local são retornadas ao GTFA. A falha do agente de DF será assumida como falha nos outros componentes ligados a ele (componente replicado e

[8] Neste modelo o objeto detector de falhas envia periodicamente mensagens para o objeto monitorado para verificar se está ativo (OMG, 2001; SERGENT, 1999).

coordenador de replicação).

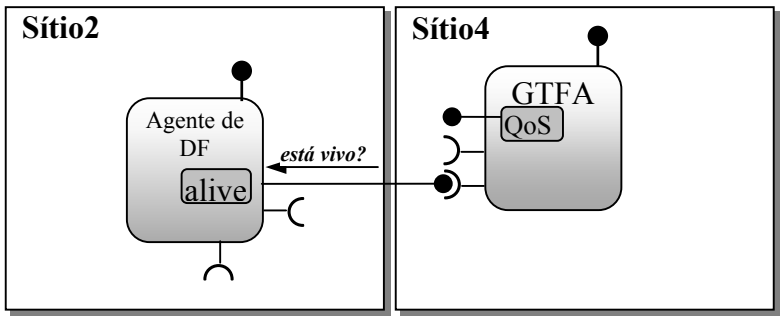

Figura 4.4 – Detecção de Falhas a Nível de Sítio.

O segundo nível é concretizado na configuração pelas ligações de cada agente de DF a um componente replicado e a um coordenador de replicação, conforme mostrado na Figura 4.5. As eventuais falhas dos componentes replicados e coordenadores são detectados quando ocorrer uma exceção referente a chamadas periódicas de mensagens também do tipo "você está vivo?". As falhas de componentes são reportadas ao GTFA pelo agente de DF nas chamadas periódicas de monitoramento.

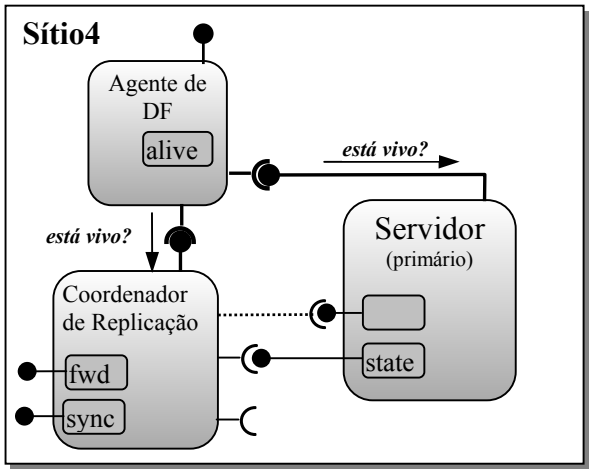

Figura 4.5 – Detecção de Falhas a Nível de Componente.

4.6 Coordenadores de Replicação

Os coordenadores de replicação têm a função de implementar os protocolos de coordenação das técnicas de replicação. Os componentes de coordenação de uma técnica de replicação trocam informações de modo a manter a consistência entre os componentes replicados do servidor. Portanto, operações de salvamento de estado, definição de réplica privilegiada (réplica primária), construção de *logs* de operações, e outros aspectos não

funcionais referentes à técnica usada são executadas pelos algoritmos implementados a partir destes componentes de coordenação. No TFA-CCM, a simples troca do coordenador resulta na troca da técnica de replicação na configuração.

No TFA-CCM várias técnicas de replicação (seção 2.4) podem ser utilizadas. Além de coordenadores para estas técnicas, é proposto também um coordenador que não implementa nenhuma técnica de replicação. Este é usado quando se deseja apenas requisitos de disponibilidade, não havendo, portanto, da transferência de estado, *logs*, etc. Este tipo de coordenador lida com só uma instância do componente, que ao falhar permite como tratamento a reinstalação da instancia *host*.

4.7 Conclusão

Este capítulo apresentou o TFA-CCM, o qual oferece mecanismos flexíveis para a construção de software baseado em componentes CORBA com requisitos de tolerância a faltas. De modo a prover a tolerância a faltas adaptativa, o TFA-CCM foi construído a partir de um conjunto de componentes de software, que combinados, fornecem requisitos de tolerância a faltas para aplicações baseadas em componentes.

O modelo TFA-CCM adapta a configuração do sistema levando em conta os requisitos de confiabilidade do usuário em relação a um serviço de aplicação. Qualquer aplicação baseada em componentes CORBA pode fazer uso do TFA-CCM para oferecer requisitos de tolerância a faltas.

Além disso, preocupou-se em maximizar a transparência no uso TFA-CCM, possibilitando que o modelo seja utilizado sem qualquer alteração nos componentes que necessitarem requisitos de tolerância a faltas, e nem mesmo no suporte de execução do modelo de componentes.

De modo a prover a tolerância a faltas adaptativa, o TFA-CCM foi construído a partir de um conjunto de componentes de software, que combinados, fornecem requisitos de tolerância a faltas para aplicações baseadas em componentes.

Capítulo 5

Aspectos de Implementação e Resultados

De maneira a verificar a viabilidade do modelo apresentado, este capítulo tem por objetivo descrever os aspectos relacionados à implementação do TFA-CCM, assim como efetuar uma análise dos resultados obtidos. Mostraremos de forma geral como foi implementado cada um dos componentes que compõem o TFA-CCM. Apresentaremos algumas medidas de desempenho que tem o objetivo de verificar a sobrecarga (*overhead*) imposta pela utilização do TFA-CCM. Por fim, são discutidos os nossos propósitos neste modelo com a literatura estabelecida da área.

5.1 OpenCCM

Toda a implementação do modelo TFA-CCM teve como plataforma de desenvolvimento o OpenCCM versão 0.2 (MARVIE et al., 2002, MARVIE; MERLE 2001), que é uma implementação parcial da especificação do CCM. O OpenCCM é totalmente desenvolvido em Java. O OpenCCM foi escolhido pois no início deste trabalho era a primeira implementação não comercial de código aberto disponível. A versão do Java utilizada foi a 1.4.

No momento da implementação, a versão disponível do OpenCCM não abrangia todos os modelos especificados pelo CCM, contemplando apenas o modelo abstrato e o modelo de implantação definido pela especificação do CCM. O OpenCCM usa o padrão CORBA (CORBA2) como base e é implementado como uma camada adicional que permite a definição das características de componentes CORBA (CORBA3).

As noções básicas do modelo abstrato do CCM são as que envolvem componentes e portas. No OpenCCM, componentes, assim como portas, são representadas por objetos do CORBA padrão. Portanto, suas referências estão de acordo com a IOR (*Interoperable Object Reference*).

Um componente contém uma referência para a sua implementação, assim como referências para suas portas. Essa estrutura é a base para as facilidades de introspecção[9] usadas durante a fase de implantação dos componentes. Um programa de implantação usa a introspecção de modo a obter as referências das portas do componente e para estabelecer as interconexões entre os componentes. As referências de portas também são usadas pelos clientes em tempo de execução para invocações dos métodos providos pelas portas.

A implementação do OpenCCM conta com um compilador para a IDL3, que provê o mapeamento das definições em IDL3 para as definições da IDL2. As interfaces IDL2 geradas durante a fase de mapeamento permitem o uso de um compilador IDL padrão e assim asseguram a compatibilidade com o CORBA2.

No OpenCCM, a implantação (instalação) de uma aplicação é feita através do programa de implantação, o qual possui instruções para instalação de arquivos, criação de instâncias de componentes, configuração de componentes e interconexão das portas dos componentes. A interconexão de portas é feita através de interfaces de gerenciamento de portas geradas pelo OpenCCM que são adicionadas ao componente.

O OpenCCM define diversas classes que fornecem um ambiente de implantação simples, manipulado através de uma API simples. Esta versão não implementa o modelo de execução (*container*) e nem o modelo de programação. Desta forma, o uso de serviços que deveriam ser fornecidos pelo *container* não é transparente para componentes de aplicação.

Esta versão do OpenCCM é compatível com três diferentes ORBs, o ORBacus 4.0.5 (IONA TECHNOLOGIES, 2001), OpenORB 1.0 (OPENORB, 2001) e VisiBroker 4.1.1 e

[9] A introspecção (*introspection*), também referenciada como reflexão estrutural (*structural reflection*), refere-se ao processo de obter informação estrutural do programa e usá-la no próprio programa (TATSUBORI, 1999).

4.5 (BORLAND, 2001). Dentre esses ORBs, foi escolhido o ORBacus para o desenvolvimento do trabalho, por ser um ORB disponível gratuitamente e por ter uma boa documentação. No entanto, qualquer um dos ORBs poderia ser utilizado, visto que todos seguem as especificações da OMG.

O ORBacus 4.0.5 é um ORB (*Object Request Broker*) que está de acordo com a especificação CORBA 2.3.1 (OMG, 1999c) com mapeamentos completos de IDL (*Interface Definition Language*) para as linguagens C++ (OMG, 1999a) e Java (OMG, 1999b), tendo o IIOP (*Internet Inter-Orb Protocol)* como protocolo de comunicação nativo, que também padronizado pela OMG. O ORBacus fornece suporte aos serviços básicos do CORBA, como os serviços de nomes, de eventos e de propriedades. Além do suporte completo a programação dinâmica, como a interface de invocação dinâmica (DII), interface de esqueleto dinâmico (DSI), repositório de interfaces e de implementações, *Portable Object Adapter* (POA).

5.2 Implementação do Modelo TFA-CCM

Conforme descrito na seção anterior (seção 4), o modelo TFA-CCM é composto por um conjunto de componentes de software, que visam fornecer suporte de tolerância a faltas adaptativa a serviços de aplicação em ambientes distribuídos. Os componentes do TFA-CCM permitem fazer o monitoramento, a detecção e a reconfiguração dos mecanismos de tolerância a faltas de aplicações distribuídas. A seguir são descritos os aspectos envolvidos na implementação de cada um dos componentes do TFA-CCM.

5.2.1 Conector

O conector, como já descrito, tem o objetivo de fornecer transparência de localização dos componentes replicados. Para fazer a conexão entre componentes é necessário que as portas de interconexão sejam do mesmo tipo. De modo a tornar o conector genérico (independente do tipo de porta de comunicação), foi utilizada a interface de esqueleto dinâmico (DSI) do CORBA, que permite receber invocações a métodos de interfaces que não são implementadas pelo objeto (no caso, o conector). Esta invocação dinâmica está indicada na Figura 5.1 pela seta (1). Quando uma requisição chega ao conector, esta é encaminha para o coordenador de replicação primário através da conexão com a faceta `fwd` deste.

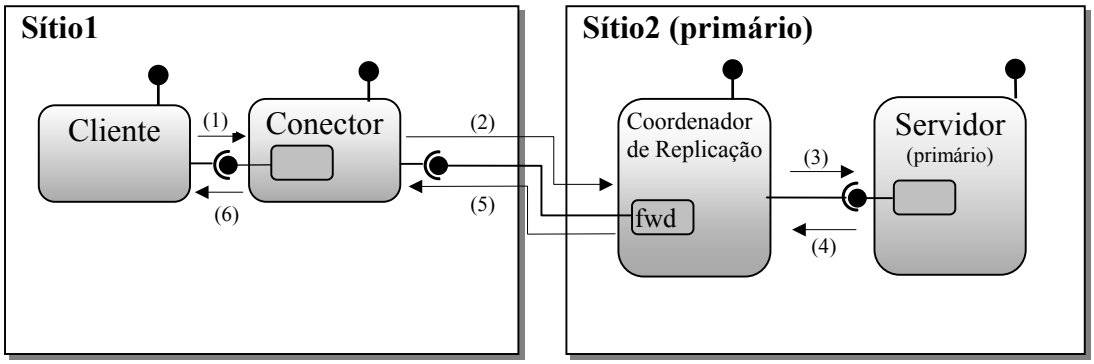

Figura 5.1 – Invocação dinâmica do conector.

5.2.2 Gerenciador de Tolerância a Faltas Adaptiva (GTFA)

O GTFA tem como principal função executar as ações necessárias para reconfigurar o sistema de modo que os requisitos de QoS especificados pelo gerenciador de QoS sejam alcançados. Esses requisitos são repassados pelo gerenciador de QoS através da faceta `QoS`, fornecida pelo GTFA. A Figura 5.2, mostra a definição em IDL3 do componente GTFA, assim como a definição da faceta `QoS`. Como pode ser observado na Figura 5.2 a faceta QoS possui dois métodos:

- `set_QoS (QoS_Requirements)`: este método é usado pelo gerenciador de QoS para enviar os requisitos de QoS ao GTFA. O parâmetro `QoS_Requirements` é uma lista que contém as definições dos níveis de QoS.
- `LogInformation (on)`: que é usado pelo gerenciador de QoS para informar ao GTFA, se deseja ou não receber as informações sobre as ações que estão sendo executadas pelo GTFA.

Após recebidos os requisitos de QoS, o GTFA pode executar as ações necessárias para configurar o sistema de modo a suportar o nível de QoS onde os requisitos são atendidos. As possíveis ações que o GTFA pode executar foram descritas na seção 4.4.

O GTFA também é responsável por fazer o monitoramento dos agentes de DF, para detectar possíveis falhas (*crash*) nestes, caracterizando a falha do sítio onde o agente se encontra, ou detectar as possíveis falhas (*crash*) nos componentes monitorados pelo agente de DF, ou seja, a réplica-componente e o coordenador de replicação.

O monitoramento é feito pelo GTFA em intervalos de tempo definidos pelo usuário

através no gerenciador de QoS. O GTFA invoca o método `is_alive()`, provido pela faceta `alive` do agente de DF, no sentido de detectar uma possível falha (*crash*) do agente de DF. O retorno ao método `is_alive()` é uma estrutura contendo a situação dos componentes monitorados. Essa estrutura é composta por duas variáveis: a variável `faults_in_component` e a variável `faults_in_coordinator`, que indicam a quantidade de vezes que possíveis falhas ocorreram na réplica-componente e no coordenador de replicação, respectivamente. A representação em IDL desta estrutura é mostrada na seção 5.2.3 na Figura 5.3

```
module tfa-ccm {
   struct alive_status {
       short faults_in_coordinator;
       short faults_in_component;
   };
   interface ialive {
         alive_status is_alive();
   };
   enum conectiveMode{ none, or, and };
   struct QoS_Transition {
       string        transitionName;
       short         priority;
       short         componentFault;
       short         componentTime;
       short         hostFault;
       short         hostTime;
       conectiveMode connective;
       string        nextLevel;
   };
   typedef sequence <QoS_Transition> QoS_Transitions;
   enum replMode{ none, passive, semi-active };
   struct QoS_Level {
       string        levelName;
       boolean       initialLevel;
       replMode      replType;
       short         numberReplicas;
       short         intMonitorAgent;
       short         timeoutAgent;
       short         intMonitorComp;
       short         timeoutComp;
       QoS_Transitions transitions;
   };
   typedef sequence <QoS_Level> requirements;
   interface iQoS {
         void set_QoS (in requirements QoS_Requirements);
         void LogInformation (in boolean on);
   };
   component GTFA {
       provides      iQoS        QoS;
       uses          istatus     status;
       uses multiple ialive      alive;
   };};
```

Figura 5.2 – Descrição em IDL3 do GTFA.

Na chegada de uma resposta ao método `is_alive()` ou mediante o esgotamento do tempo de espera (*timeout*) definido pelo nível de QoS corrente, o GTFA verificará se os requisitos de QoS estão sendo mantidos. Caso não esteja, as ações necessárias para a manutenção daqueles requisitos serão tomadas. Uma falha ou a sua ausência em um determinado intervalo de tempo pode levar a uma transição entre níveis de QoS na máquina de estados que define as variações de configuração do sistema. Por exemplo, a resposta ao método `is_alive()` pode informar que o componente replicado falhou duas vezes durante o último intervalo de monitoramento do GTFA. De acordo com o nível de QoS corrente, o GTFA pode não efetuar nenhuma ação, pode reimplantar toda a estrutura instalada no *host* em outro lugar, ou pode efetuar outra ação descrita no nível de QoS corrente.

Para a implantação de componentes o TFA-CCM usa a API de implantação especificada pelo CCM e implementada parcialmente pelo OpenCCM, fornecendo métodos para a instalação de arquivos com a implementação nos sítios destino, para a instanciação dos componentes, para a interconexão de componentes e para configuração dos atributos em tempo de implantação. Os métodos para a configuração dos atributos em tempo de implantação não são implementados na versão 0.2 do OpenCCM, tornando necessário a implementação no nosso trabalho de tal funcionalidade. Para isso, foi utilizado o mecanismo de introspecção da linguagem Java.

De forma a manter uma visão global de todos os componentes implantados no sistema o GTFA mantém uma estrutura de dados contendo essas informações. Essa estrutura contém a referência de todos os componentes, da localização de cada componente, a técnica de replicação que está sendo utilizado, entre outras informações. Estes dados permitem que as reconfigurações necessárias no sistema sejam efetuadas, como por exemplo interligar as portas dos componentes ou remover componentes de algum sítio. O GTFA também usa esses dados para fornecer informações sobre o funcionamento do sistema ao gerenciador de QoS, como a técnica de replicação que está sendo utilizada, a localização dos componentes replicados, e falhas que acontecem nos componentes.

A falha do GTFA e a conseqüente perda destas informações comprometem todo o funcionamento do sistema. Contudo, para evitar tal situação, as informações de estado do

GTFA são gravadas em um meio de armazenamento persistente. Desta forma, caso o GTFA apresente falha e tenha que ser reimplantado em outro sítio, é possível recuperar o seu estado. O OpenCCM na versão 0.2 não implementa os *containers*, que são responsáveis pelo gerenciamento do estado persistente do componente. Com isto, no nosso trabalho tivemos que implementar mecanismos que permitissem o armazenamento persistente.

Se o OpenCCM implementasse o *container*, o componente GTFA seria implementado como um componente do tipo *entity*, cujo estado é persistente e seu estado poderia ser gerenciado de forma transparente pelo *container*. Caso o *container* fosse implementado, os outros componentes que compõem o TFA-CCM seriam do tipo *session*. A escolha do tipo *session* para o restante dos componentes que compõem o TFA-CCM deve ao fato da perda do estado do componente não comprometer a consistência do sistema.

5.2.3 Agentes de Detecção de Falhas (Agentes de DF)

O agente de DF é responsável por fazer o monitoramento de uma réplica do componente e de um coordenador de replicação, no sitio onde está implantado. Esse monitoramento é feito através da chamada do método `_non_existent()` da classe `Object` que é implicitamente herdada por todos os componentes. Não havendo resposta em um determinado período de tempo (*timeout*) ou se exceções ocorrerem na chamada ao método `_non_existent()`, o agente de DF armazena essa informação em um buffer para depois ser repassada ao GTFA.

O agente de DF provê a faceta `alive`, a qual fornece o método `is_alive()` para o monitoramento de falhas. A Figura 5.3 mostra a definição em IDL3 do componente agente de DF. Como já explicado na seção 5.2.2, a faceta `alive` é usada pelo GTFA para a detecção de falhas no agente de DF, assim como nos componentes ligados a ele (componente replicado e o coordenador de replicação).

```
module tfa-ccm {
   struct alive_status {
         short faults_in_coordinator;
         short faults_in_component;
   };
   interface ialive {
         alive_status is_alive();
   };
   component FDAgent {
         provides ialive   alive;
         uses  Components::CCMObject   server;
         uses  Components::CCMObject   coordinator;
   };
};
```

Figura 5.3 – Descrição em IDL3 do Agente de DF.

5.2.4 Coordenadores de Replicação

Os coordenadores de replicação têm a função de implementar a técnica de replicação utilizada. Foram implementados os coordenadores que implementam as técnicas de replicação passiva, semi-ativa e um coordenador que concentra aspectos não funcionais de um componente não replicado. A técnica de replicação ativa não foi implementada, pois requer mecanismos de comunicação de grupo, que não são fornecidos pelo ORB utilizado.

Todo componente coordenador de replicação tem duas facetas, a faceta `sync` e a faceta `fwd`. A faceta `sync` é usada pelos coordenadores de replicação passiva e semi-ativa para sincronização do estado de uma nova réplica com a réplica primária. Além desse uso, o coordenador que implementa a replicação passiva usa essa faceta para sincronizar o estado das réplicas em caso de mudança do estado do componente primário.

A faceta `fwd` é implementada por todos os tipos de coordenadores de replicação, para que as requisições dos clientes sejam repassadas do conector para o coordenador de replicação. Além desse uso, essa faceta também é usada pelo coordenador que implementa a técnica de replicação semi-ativa para repassar as requisições recebidas dos clientes para os coordenadores acoplados às réplicas seguidores (*followers*). A faceta `fwd` só tem um método que é responsável por encaminhar a invocação do método requisitado pelo cliente ao servidor replicado. A Figura 5.4 mostra a definição em IDL3 do coordenador de replicação.

```
   module tfa-ccm {
      struct NamedValueFT {
         string          name;        // argument name
         any             value;       // argument
         unsigned long   flags;      // argument mode flags
      };
      typedef sequence <NamedValueFT> NVListFT;

      interface ifwd {
          any forwardRequest(in string op, in NVListFT params,
                             in CORBA::TypeCode resultTypeCode);
      };
      interface isync {
             void syncState(in State state);
      };

      component Coordinator {
         attribute      string      replType;
         provides       ifwd        facet_fwd;
         provides       isync       facet_sync;
         uses           MgtState    state;
         uses           isync       sync;
      };
};
```

Figura 5.4 – Descrição em IDL3 do Coordenador de Replicação.

Como o coordenador não sabe antecipadamente qual a interface IDL (porta de comunicação) ao qual está associado, isto é a réplica-componente, é usado o mecanismo de invocação dinâmica do CORBA (DII), o qual permite em tempo de execução descobrir como fazer as chamadas aos métodos da interface da réplica ativa.

Os componentes construídos para serem replicados devem implementar a interface `MgtState` (Figura 5.5), que fornece uma forma padrão para a atualização do estado do componente. Essa interface fornece as operações para recuperação (`get_state`) e atualização (`set_state`), que são idênticas às operações de acesso a estado na especificação FT-CORBA (OMG, 2000). O estado de um componente é codificado em uma seqüência de bytes. De modo a garantir a interoperabilidade, é recomendado ao desenvolvedor do componente adotar o padrão de codificação de dados CORBA – CDR-Common Data Representation (OMG, 2002b) – para codificar a estrutura de dados em uma seqüência de bytes.

```
module tfa-ccm {
   typedef sequence<octet> State;
   interface MgtState {
      State    get_state() raises (NoStateAvailable);
      void     set_state(in State s) raises (InvalidState);
   };
};
```

Figura 5.5 – IDL para Atualização de Estado.

5.2.5 Gerenciador de QoS

O Gerenciador de QoS fornece uma interface gráfica para a especificação dos requisitos de QoS (Figura 5.6). A especificação dos requisitos de QoS consiste da definição dos níveis de QoS, onde são definidas as características de cada nível e também as transições entre os níveis.

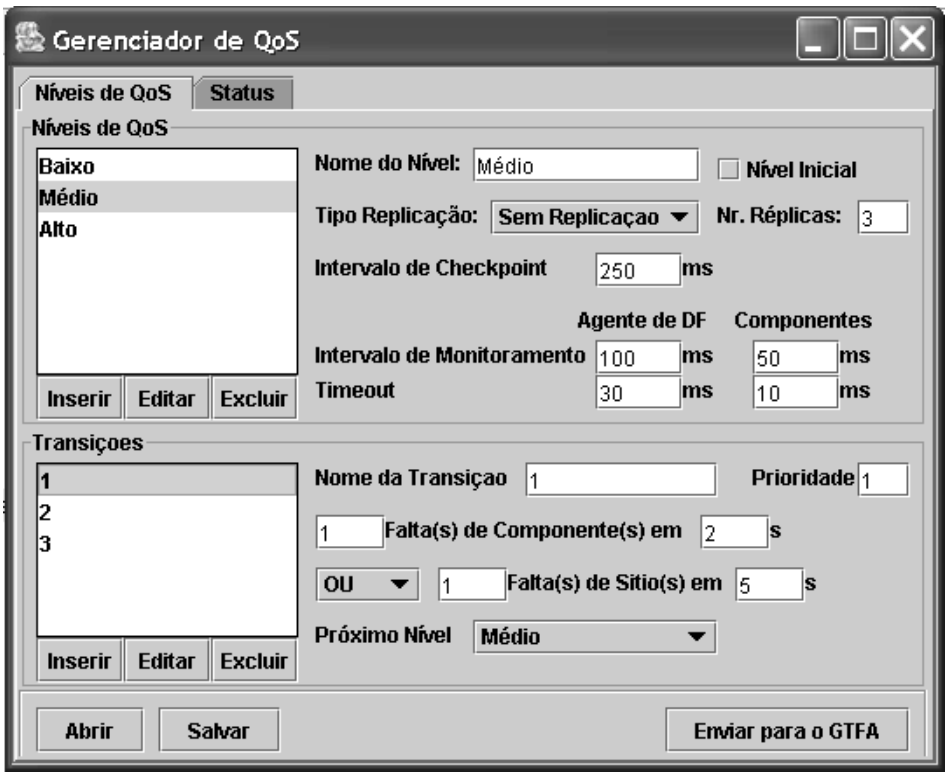

Figura 5.6 – Tela do Gerenciador de QoS para Edição dos Níveis de QoS.

Como pode ser observado na Figura 5.6, vários parâmetros devem ser especificados para cada nível de QoS, conforme descrito abaixo:

- *Nome do Nível*: específica um nome qualquer que define o nível do QoS.

- *Nível Inicial*: define se esse nível será o nível inicial. Isto permite que quando os requisitos de QoS forem passados para o GTFA, este saiba em qual nível o sistema é iniciado;
- *Técnica de replicação*: define a técnica de replicação utilizada, que pode ser: sem replicação, passiva ou semi-ativa;
- *Número de Réplicas*: define o número de réplicas do componente para manter o grau de tolerância a faltas desejado;
- *Intervalo de Checkpoint*: determina o intervalo entre cada atualização de estado, usado somente se for utilizada a técnica de replicação passiva.
- *Intervalo de Monitoramento e Timeout do agente de DF*: define o intervalo de monitoramento e o tempo de reposta (*timeout*) do agente de DF monitorado para determinar se está faltoso;
- *Intervalo de Monitoramento e Timeout dos componentes replicados*: define o intervalo de monitoramento e o tempo de reposta (*timeout*) dos componentes monitorados pelo agente de DF para determinar se estão faltosos.

Para cada nível de QoS devem ser definidas as transições que fazem a máquina de estados evoluir para outros níveis. A Figura 5.6 ilustra os parâmetros que definem as transições da máquina que podem ser explicitadas como:

- *Nome da Transição*: específica um nome qualquer que define a transição;
- *Prioridade*: define qual a prioridade da transição. Usada quando vários níveis puderem ser ativados simultaneamente, definindo qual deles deve ser usado, de acordo com a precedência; Quando menor for o valor atribuído maior é a prioridade.
- *Quantidade de faltas em componentes e quantidade de tempo decorrido:* indica a quantidade de faltas em componentes e/ou tempo especificado para que uma transição seja ativada;
- *Quantidade de faltas no sítio e quantidade de tempo decorrido:* indica a quantidade de faltas que ocorrem em um sítio e/ou o tempo especificado para que

uma transição seja ativada;

- *Conectivo entre faltas em componentes e em sítios*: é um conectivo "e/ou" que permite fazer a combinação dos dois itens anteriores.

O gerenciador de QoS também recebe do GTFA informações de status a respeito do sistema através da faceta `status` (Figura 5.7). Estas informações são mostradas ao usuário através da interface gráfica do gerenciador de QoS, como mostrado na Figura 5.8. Quando o gerenciador de QoS percebe que não está mais recebendo informações do GTFA, ele poderá então reinstalar o GTFA em outro sítio. Enquanto o GTFA está sendo recuperado o sistema continua funcionando normalmente.

```
module tfa-ccm {
   typedef sequence <string> seqString;

   enum replMode{ none, passive, semi-active };
   struct inf {
      string     levelName;
      replMode   replType;
      short      numberReplicas;
      short      faultsInHostLevel;
      short      faultsInHostTotal;
      short      faultsInComponentLevel;
      short      faultsInComponentTotal;
      seqString  replicaLocation;

   };
   interface istatus {
      void setStatus (in infS infStatus);
      void updateLog (in infLog seqString);
   };

   component QoSManager{
      provides istatus     status;
      uses     iQoS        QoS;
   };
};
```

Figura 5.7 – IDL do Gerenciador de QoS.

Figura 5.8 – Tela do Gerenciador de QoS para informar o Status do Sistema.

5.3 Resultados Obtidos

De modo a verificar o desempenho da implementação do TFA-CCM, foram executados alguns testes[10]. Os testes foram executados em uma rede local Ethernet de 100Mbps composta por computadores Pentium IV 1.6Ghz com 256Mb de memória RAM, sistema operacional Linux Mandrake versão 9.1 e Java JDK 1.4.

O primeiro teste mensurou o tempo de resposta das chamadas ao componente replicado em diferentes configurações, isto é, com diferentes técnicas de replicação. As seguintes configurações foram montadas: sem a adição do TFA-CCM, com coordenador que não implementa nenhuma técnica de replicação, com coordenador de replicação passiva, e com coordenador de replicação semi-ativa, usando duas réplicas nos dois últimos casos. Nos testes foram utilizados um componente com apenas uma faceta com um método que não executa nenhuma instrução. Deste modo, foi minimizada a influência do tempo de processamento do método nos resultados dos testes, avaliando apenas a sobrecarga gerada pela adição do TFA-CCM.

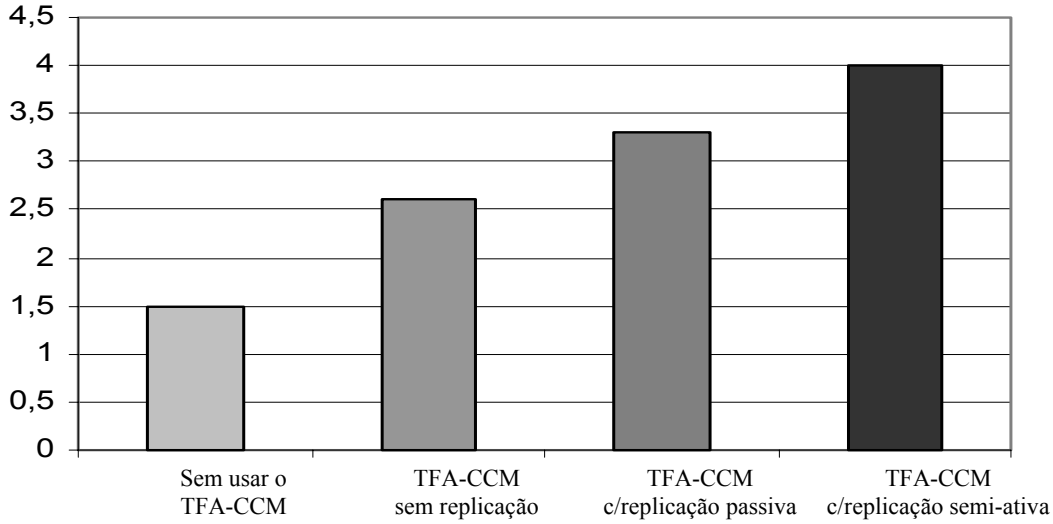

Figura 5.9 – Desempenho do TFA-CCM.

No gráfico da Figura 5.9, é possível observar que a utilização do modelo, mesmo sem implementar nenhuma técnica de replicação, aumenta o tempo de resposta da chamada em 1.1ms em relação à chamada sem o TFA-CCM. Esta diferença chega a 1.8ms quando utilizamos a técnica de replicação passiva, e atinge 2.5ms quando é usada replicação semi-ativa. O aumento no tempo de resposta era esperado, e pode ser considerado aceitável, tendo em vista que o TFA-CCM inclui uma série de componentes responsáveis por fornecer requisitos de tolerância a faltas.

Um consideração importante é que a replicação semi-ativa apresentou um desempenho pior que a replicação passiva em nossas implementações. Isto se explica pelas nossas implementações destas replicações: enquanto na replicação passiva o coordenador primário envia *checkpoints* ao secundário após responder ao cliente, na replicação semi-ativa se dá justamente ao contrário; o primário (*leader*) envia suas cópias de requisições do cliente ao seguidor (*follower*) antes dos seus processamentos e respectivas respostas ao cliente.

O segundo experimento realizado verificou o tempo médio gasto para criar uma réplica do componente em novo um sítio, juntamente com o seu coordenador de replicação e o seu agente de TFA, e o tempo médio para trocar a técnica de replicação. Neste experimento foi observado que o tempo médio gasto para a implantação de uma nova réplica foi de 320ms, enquanto o tempo médio gasto para fazer a troca de coordenadores,

[10] Os resultados foram obtidos a partir da média de 1000 execuções.

alternando entre replicação passiva e semi-ativa, foi da ordem de 250ms. Neste último teste a replicação semi-ativa, durante a redefinição da réplica primária (réplica líder) apresentou um desempenho ligeiramente superior que a replicação passiva. Isto representa que em situação de falha do primário a recomposição da replicação semi-ativa se dá de maneira mais rápida que a replicação passiva.

5.4 TFA-CCM x outras abordagens

Nenhum trabalho semelhante ao nosso foi encontrado na literatura, que é o fornecimento de aspectos de tolerância a faltas na forma de QoS adaptativo a aplicações baseadas em componentes. No entanto, alguns trabalhos que estão de certa forma relacionados foram apresentados. Trabalhos que visam fornecer tolerância a faltas adaptativa foram apresentados na seção 2.7, também trabalhos que visam prover tolerância a faltas ou requisitos de QoS para aplicações baseadas em componentes foram apresentadas na seção 3.4.

Nesta seção apresenta-se um breve comparativo entre cada um dos trabalhos e o TFA-CCM.

Em (CHANG et al., 1998) é apresentado uma abordagem para reduzir o custo no fornecimento de tolerância a faltas, através de adaptações conduzidas pelas semânticas de falhas no sistema. Nesta abordagem, quando falhas ocorrerem durante a execução da aplicação, o algoritmo pode ser trocado por um algoritmo tolerante a faltas que considera a nova situação de semântica de falhas. Esses algoritmos e a troca entre eles são definidos em tempo de compilação, ao contrário do TFA-CCM que permite definir as políticas de troca, permite também definir as técnicas de replicação, e tudo isto pode ser feito em tempo de execução.

A arquitetura AQuA (seção 2.7.3)visa fornecer tolerância a faltas adaptativa para aplicações distribuídas. AQuA permite que os programadores de aplicações especifiquem os níveis de confiabilidade desejados, que são alcançados através da configuração do sistema em função da disponibilidade de recursos e de acordo com as falhas ocorridas. Apesar do TFA-CCM e AQuA possuir mecanismos de especificação de QoS com funcionalidade equivalente, no AQuA os requisitos de QoS são definidos em tempo de

compilação, enquanto que o TFA-CCM permite que os requisitos de QoS sejam modificados em tempo de execução, tirando proveito da flexibilidade propiciada pela utilização de componentes de software.

O *Chamaleon* (seção 2.7.2) é uma infra-estrutura adaptativa, que permite que diferentes níveis de requisitos de disponibilidade sejam fornecidos através de uma arquitetura de ARMORs. O *Chamaleon* pode usar seletivamente combinações de ARMORs a fim de prover diferentes níveis de disponibilidade, que podem ser introduzidos de maneira incremental no sistema. Apesar de empregar mecanismos de composição semelhantes aos existentes em modelos de componentes, o *Chamaleon* não segue um modelo de componentes padrão como o CCM, usado neste trabalho, ou o EJB.

Em (BATISTA; CARVALHO, 2002), um conector é antes de tudo um suporte para configuração, que busca entre várias opções de componentes servidores, aquele que possui a assinatura mais adequada para atender a invocação de um cliente. Se na chamada do componente – que é executada pelo conector – ocorre alguma exceção, a procura de uma nova alternativa de componente servidor é imediatamente iniciada. Na proposta TFA-CCM, conectores assumem um papel mais simples, que é o de um elemento redirecionador de invocações, que apenas reenvia as chamadas para os componentes replicados na configuração. Aspectos não funcionais são tratados pelos coordenadores de replicação. A nossa opção por um conector simples foi determinada pelo uso do CCM, onde por definição o *container* já concentra muitos dos aspectos não funcionais de uma aplicação.

Em (MARANGOZOVA; HAGIMONT, 2002) é apresentada uma abordagem para replicação de componentes CORBA. Nessa abordagem são usados objetos de interceptação, que são responsáveis por capturar as requisições feitas para o componente de modo a disparar as ações necessárias para o gerenciamento da replicação. Esses objetos de interceptação possuem a mesma interface do componente que será replicado, o que implica que toda a vez que se desejar replicar uma nova aplicação é necessário implementar um novo objeto de interceptação com a mesma interface do componente de aplicação. Já no modelo TFA-CCM, é empregado um conector genérico, que independe da interface do componente replicado e não precisa ser modificado para ser utilizado em diferentes aplicações. Adicionalmente, o TFA-CCM é um mecanismo que além de prover replicação para as aplicações, também tem todo um mecanismo para o gerenciamento das

réplicas, tornando possível por exemplo, a criação de novas réplicas automaticamente.

Em (BALASUBRAMANIAN et al., 2003) é apresentado o CIAO, uma extensão da especificação do CCM para suporte de QoS. Essa proposta difere-se um pouco do TFA-CCM pois está preocupado em fornecer requisitos de QoS relacionados a tempo real e aplicações embutidas, enquanto o TFA-CCM trata da tolerância a faltas. Este trabalho foi relacionado pois trata-se de um esforço de adicionar novos serviços (aspectos não funcionais) aos componentes CORBA.

5.5 Conclusão

A implementação do TFA-CCM demonstrou a aplicabilidade do modelo. Testes realizados com a implementação inicial do modelo mostraram que os seus custos de desempenho são aceitáveis, tendo em vista o ganho em confiabilidade propiciado pelo TFA-CCM. Com o avanço das implementações da especificação do CCM, acredita-se que melhor desempenho pode ser obtido. A versão 0.5 do OpenCCM que foi lançada no início de 2003, implementa mais características da especificação do CCM, como implementação de parte do modelo de programação e execução (*container*). Como o TFA-CCM é composto por um conjunto de componentes que seguem a especificação do CCM, em tese pode ser usado em outras implementações do modelo CCM.

As ferramentas e tecnologias utilizadas no desenvolvimento do TFA-CCM (OpenCCM, arquitetura CORBA e a linguagem Java) baseiam-se em padrões abertos, o que pode ser considerado como um importante fator para alcançar a interoperabilidade e a independência de plataforma da aplicação desenvolvida.

Um comparativo entre o TFA-CCM e outras abordagens relacionadas foi apresentado, tornando possível verificar a contribuição trazida pelo TFA-CCM .

Capítulo 6

Conclusões e Perspectivas Futuras

Esta dissertação é antes de tudo um exercício que explora as possibilidades da tecnologia de componentes em middleware. Tomando como base esta tecnologia foi proposto um suporte de tolerância a faltas adaptativa. A tolerância a faltas é adaptativa no sentido de permitir a aplicação que utiliza o suporte de tolerância a faltas se modificar de acordo com as mudanças que ocorrem no ambiente, isto é, se adequar, por exemplo a freqüência em que as falhas ocorrem, de maneira a manter o nível de confiabilidade e disponibilidade do sistema mesmo na ocorrência de falhas no sistema.

Diante disto, apresentamos o TFA-CCM – Tolerância a Faltas Adaptativa baseado em Componentes CORBA – que implementa um modelo de suporte a tolerância a faltas adaptativa totalmente transparente à aplicação. O TFA-CCM é formado por um conjunto de componentes de software que permitem fazer o monitoramento, a detecção e a reconfiguração do suporte de tolerância a faltas. O modelo apresenta soluções práticas para integrar requisitos de QoS que devem guiar a seleção da configuração de serviços replicados.

Enfim, ao propor o TFA-CCM, acredita-se ter contribuído efetivamente para a adição rápida e eficiente do suporte de tolerância a faltas para aplicações baseadas em componentes. Devido aos componentes do suporte propostos tratarem essencialmente com aspectos não funcionais, estes podem ser reutilizados por qualquer aplicação que desejar ter requisitos de tolerância a faltas. Os testes realizados com o TFA-CCM atenderam aos objetivos esperados desta dissertação.

Uma possível continuidade desta dissertação seria o aprimoramento da implementação do TFA-CCM, com a sua evolução para novas implementações da especificação do modelo CCM. Outra possibilidade seria fornecer o suporte de tolerância a faltas adaptativa através do *container*, uma vez que este é responsável por fornecer acesso aos serviços não funcionais utilizados por um componente. Neste caso, poderia se estender as especificações de tolerância a faltas do CORBA – que são caracterizados como serviços comuns – para que também fossem aplicadas ao modelo de componentes CCM, visto que a especificação do FT-CORBA (OMG, 2000) é voltada para objetos e não para componentes de software.

Dentro da perspectiva de fornecer suporte a serviços (não funcionais) não previstos pelo CCM, propõe-se a adição de outros requisitos de QoS no TFA-CCM, além de tolerância a faltas, como requisitos de tempo real e balanceamento de carga. Além disso, seria interessante permitir a especificação dos níveis de QoS através de uma linguagem declarativa, usando por exemplo, o XML. Por fim, propõe-se o desenvolvimento de uma aplicação para a aplicação do modelo TFA-CCM.

Glossário

Agente de DF	– Agente de Detecção de Faltas
API	– Application Program Interface
CCM	– CORBA Component Model
CIDL	– Component Implementation Definition Language
CIF	– Component Implementation Framework
CORBA	– Common Object Request Broker Architecture
DII	– Dynamic Invocation Interface
DSI	– Dynamic Skeleton Interface
EJB	– Enterprise Java Beans
GTFA	– Gerenciador de Tolerância a Faltas Adaptativa
IDL	– Interface Definition Language
IIOP	– Internet Inter-ORB Protocol
OMG	– Object Management Group
OMG	– Object Management Group
OpenCCM	– Open CORBA Component Model
ORB	– Object Request Broker
POA	– Portable Object Adapter
QoS	– Qualidade de Serviço
TFA-CCM	– Tolerância a Faltas Adaptativa baseada em Componentes CORBA

Referências Bibliográficas

ANDERSON, T.; LEE P. A. Lee. *Fault Tolerance – Principles and Practices*. Prentice-Hall, 1981.

BACHMAN, F. et al. Volume II: Technical Concepts of Component Based Software Engineering. Technical Report CMU/SEI-2000-TR-08, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 2000.

BAGCHI, S. et al. The Chameleon Infrastructure for Adaptive, Software Implemented Fault Tolerance. In: *17th IEEE Symposium on Reliable Distributed Systems*. p. 261–267, West Lafayette, Indiana, 1998. IEEE Computer Society.

BALASUBRAMANIAN, K., et al. Towards Composable Distributed Real-time and Embedded Software. In: *Proceedings of the 8th Workshop on Object-oriented Real-time Dependable Systems*. Guadalajara, Mexico, January, 2003, IEEE.

BATISTA, T. V.; CARVALHO, M. G. Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support. *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers. v.65, n.4, Abril, 2002.

BIRMAN, et. al. Implementing Fault-Tolerant Distributed Object. *IEEE Transactions on Software Engineering*. v.6 n.11, p.502-508. Junho, 1985.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML-Guia do Usuário*. Ed. Campos. Rio de Janeiro, 2000.

BORLAND, Software Corporation. *Reference Quide, Visibroker 4.5 for Java.* 2001. http://info.borland.com/techpubs/books/vbj/vbj45/pdf_index.html.

BROWN, A. W.; WALLNAU, K. C., The Current State of CBSE. *IEEE Software*. v.15, n.5, p. 37-46. Setembro, 1998.

CHANG, I.; HILTUNEN, M.A.; SCHLICHTING, R. D. Affordable Fault Tolerance Through Adaptation. *IPPS/SPDP Workshops 1998*. p. 585-603. Abril, 1998.

CHEN W.K.; HILTUNEN, M.A.; SCHLICHTING, R.D. Constructing Adaptive Software in Distributed Systems. In: *Proceedings of the 21st International Conference on Distributed Computing Systems*. p. 635-643. Phoenix. AZ. Abril, 2001. IEEE

Computer Society.

CRISTIAN, F. Understanding Fault-Tolerant Distributed systems. *Communications of the ACM*, v.32, n.2, p.56-78, fevereiro, 1991.

CRISTIAN, F.; AGHILI, H.; STRONG, R.; DOLEV, D. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement, In: *15th International Symposium on Fault-Tolerant Computing*, p.200-206, Ann Arbor, Michigan, USA. Junho, 1985. IEEE Computer Society.

CUKIER, M. et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In: *17th IEEE Symposium on Reliable Distributed Systems*. p. 245-253, West Lafayette, Indiana, 1998. IEEE Computer Society.

DÍAZ, R.P., Status Report: Software Reusability. *IEEE Software*. v.10, n.3, p. 61-66. Maio, 1993.

HAYDEN, M. G. *The Ensemble System*. PhD Thesis, Department of Computer Science, Cornell University, 1998.

HILTUNEN, M.A.; SCHLICHTING, R.D. Adaptive Distributed and Fault-Tolerant Systems. *International Journal of Computer Systems Science and Engineering*. v.11, n.5, p. 125-133, setembro, 1996.

IONA TECHNOLOGIES. *ORBacus for C++ and Java,* version 4.0.5. 2001. http://www2.iona.com/support/docs/orbacus/4.0.5/OB-4.0.5.pdf

JALOTE, P. Fault Tolerance in Distributed System. Prentice-Hall, 1994.

KIM, K.H.(Kane); LAWRENCE, T. Adaptive Fault Tolerance: Issues and Approaches. In: *Proceedings of Second IEEE Workshop on Future Trends of Distributed Computing Systems*. p. 38-46, Cairo, Egypt, 1990. IEEE Computer Society.

KRUEGER, C. W. Software Reuse. *ACM Computing Surveys*. v.24, nr.2, p. 131– 183, Junho, 1992.

LAPRIE, J. C. (Ed). *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer Verlag, 1992.

LOYALL, J. P. et al. Specifying and Measuring Quality of Service in Distributed Object Systems. In: *1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. p. 43–52. Kyoto, Japão, 1998.

LUNG, L. C. Experiências com Tolerância a Faltas no CORBA e Extensões ao FT-CORBA para Sistemas Distribuídos de Larga Escala. Tese (Doutorado em

Engenharia Elétrica) – Centro Tecnológico, UFSC, Florianópolis, 2001.

MARANGOZOVA, V.; HAGIMONT, D. An Infrastructure for CORBA Component Replication. In: *Proceedings of the* 1st *IFIP/ACM Working Conference on Component Deployment*. 2002. p. 222-232, Berlin, Alemanha. 2002.

MARVIE, R., MERLE, P., e VADET, M. *The OpenCCM Plataform*. 2002. http://corbaweb.lifl.fr/OpenCCM/

MARVIE, R.; MERLE, P. CORBA Component Model: Discussion and Use with OpenCCM. Submitted to Informatica-An International Journal of Computing and Informatics, Special Issue on Component Based Software Development. 2001.

MCILROY, M. D. Mass Produced Software Components. In: *Proceedings of the NATO Software Engineering Conference*. p.138-150. Garmisch, Germany. Outubro, 1968.

MEYER, B. *Object-Oriented Software Construction*. Englewood Cliffs: Prentice Hall, 1988.

OMG. Object Management Group. *IDL/C++ Language Mapping*. OMG Document formal/99-07-45. 1999.

OMG. Object Management Group. *IDL/Java Language Mapping*. OMG Document formal/99-07-53. 1999.

OMG. Object Management Group. *The Common Object Request Broker Architecture v2.3.1*. OMG Document formal/99-10-07. 1999.

OMG. Object Management Group. *CORBA Components*. OMG Document formal/02-06-65. 2002.

OMG. Object Management Group. *The Common Object Request Broker Architecture v3.0*. OMG Document formal/02-06-33. 2002.

OMG. Object Management Group. *Event Service Specification v 1.1*. OMG Document formal/2001-03-01. 2001.

OMG. Object Management Group. *Fault-Tolerant CORBA Specification v.1.0*. OMG Document ptc/2000-04-04, 2000.

OPENORB, The Community. 2001. http://openorb.sourceforge.net/

ORFALI, R.; HARKEY, D. *Client/Server Programming with Java and CORBA*. Second Edition. John Wiley & Sons Inc. EUA, 1998.

POWEL, D. *Delta-4 Architecture Guide.* Esprit II P2252, Delta-4 Phase 3, agosto, 1991.

SABNIS, C. et al. PROTEUS: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA. In: *7th IFIP International Working Conference on Dependable Computing for Critical Applications*. p. 137–156. San Jose, CA, USA, 1999.

SCHNEIDER, F.B., Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial. *ACM Computing Survey*. v.22, n.4, p.299-319, Dezembro, 1990.

SERGENT, N., Défago, X. e Schiper, A. *Failure Detectors: implementation issues and impact on consensus performance*. Techical Report SSC/1999/019. École Polytechnique Fédérale de Lausanne, 1999.

SUN MICROSYSTEMS. Enterprise JavaBeans Specification v2.0. 2001.

SZTAJNBERG, A. Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos. Tese (Doutorado em Engenharia Elétrica) – COPPE/UFRJ, Rio de Janeiro, 2002.

SZYPERSKI, C. Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley Publishing Co., 1998.

TATSUBORI, Michiaki. *An Extension Mechanism for the Java Language*. Dissertação (Mestrado em Engenharia) - Universidade de Tsukuba. Ibaraki, Japão, Fev. 1999.

THOMAS, A. Enterprise JavaBeam™ Technology – Server Component Model for the Java™ Plataform. Patricia Seybold Group, EUA. Dezembro, 1998.

VERÍSSIMO, P.; LEMOS, R. de. *Confiança no Funcionamento: Proposta para uma Terminologia em Português*. Publicação conjunta INESC e LCMI/UFSC. 1989.

W3C (1998). *eXtensible Markup Language (XML) v1.0*. World Wide Web Consortium.

WANG, N.; SCHMIDT, D.C.; O'RYAN, C. An Overview of the CORBA Component Model. In: *Component-Based Software Engineering: Putting the Pieces Together* (G.Heineman e B.Council, eds.). Reading, Massachusetts: Addison-Wesley, 2000.

ZINKY, J. A. et al. Architectural Support for Quality of Service for CORBA Objects. *Theory e Practice of Object Systems*, v.3, n.1, p. 53-73, Abril, 1997.