UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Experiências com Tolerância a Faltas no CORBA e Extensões ao FT-CORBA para Sistemas Distribuídos de Larga Escala

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

Lau Cheuk Lung

Florianópolis, Maio de 2001.

"Experiências com Tolerância a Faltas no CORBA e Extensões ao FT-CORBA para Sistemas Distribuídos de Larga Escala"

Lau Cheuk Lung

'Esta Tese foi julgada adequada para obtenção do Título de Doutor 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. Jean-Marie Farines, Dr.
Co-orientador

Prof. Raimundo J. de Araújo Macêdo, Dr.

Prof. Paulo J. Esteves Veríssimo, Dr.

Prof. Francisco Vilar Brasileiro, Dr.

Prof. Udo Fritzke Júnior, Dr.

Aos meus pais Lau Shing Lit
e Lau Yeung Pik Yee.

Aos meus irmãos Lau Cheuk Lun
e Jim Lau.

AGRADECIMENTOS

Agradeço aos professores (e meus amigos) Joni da Silva Fraga e Jean-Marie Farines por esses anos de empenho e incentivo. Aos bolsistas do projeto GroupPac: Ricardo Padilha e Luciana Souza, pelas conversas e contribuições que tanto enriqueceram este trabalho.

Agradeço também a todos os professores, colegas e funcionários (Wilson e Marcos) do Programa de Pós-graduação em Engenharia Elétrica que de alguma forma contribuíram para a realização deste trabalho. Em especial, agradeço a meus amigos: Karina Barbosa, Carlos Brandão, Evandro Cantú, Antônio Cunha, Frederico Freitas, Cristian Koliver, Augusto Loureiro, Jerusa Marchi, Carlos Montez, Rafael Obelheiro, Ricardo Padilha, Cristiane Paim, Luciano Rottava, Frank Siqueira, Luciana Souza, Isabel Tonin, Cesar Torrico, Michele Wangham, Carla Westphall pela convivência e apoio durante essa jornada.

A Deus.

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

Experiências com Tolerância a Faltas no CORBA e Extensões ao FT-CORBA para Sistemas Distribuídos de Larga Escala

Lau Cheuk Lung
Maio/2001

Orientador: Joni da Silva Fraga, Dr.
Co-Orientador: Jean-Marie Farines, Dr.
Área de Concentração: Sistemas de Informação.
Palavras-chave: Tolerância a faltas, CORBA, Redes de Larga Escala, Sistemas abertos.
Número de Páginas: 230.

RESUMO: Esta tese apresenta nossas experiências com tolerância a faltas no CORBA e uma proposta de extensão das especificações Fault-Tolerant CORBA [OMG00] para sistemas distribuídos de larga escala. A motivação deste trabalho foi a inadequação ou a falta de definições nas especificações FT-CORBA que permitissem atender a requisitos de tolerância a falta para sistemas de larga escala, tal como a Internet. Neste trabalho, é apresentado um conjunto de soluções, englobando detecção de falhas, *membership* e comunicação de grupo, que visam principalmente atender aspectos de escalabilidade, necessários quando são tratados sistemas desta natureza. A principal contribuição deste trabalho é a proposta de um modelo de hierarquia de domínios de tolerância a faltas que facilita o gerenciamento e comunicação de grupo interdomínio.

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

Experiences with Fault Tolerance in CORBA and Extensions on FT-CORBA for Large Scale Distributed Systems

Lau Cheuk Lung

May/2001

Advisor: Joni da Silva Fraga, Ph.D.
Co-advisor: Jean-Marie Farines, Ph.D.
Area of Concentration: Information Systems
Keywords: Fault Tolerance, CORBA, Large Scale Network, Open Systems
Number of Pages: 230.

ABSTRACT: This thesis presents our experiences with fault tolerance in CORBA and a proposal to extend the Fault-Tolerant CORBA specifications [OMG00] for large-scale distributed systems. Our motivations for this proposal are that FT-CORBA does not assist, properly, to the requirements of fault tolerance in large-scale networks, such as Internet. In this document, we present a set of solutions (including failure detection, membership, group communication) that aim, mainly, to treat scalability aspects in large-scale network. The main contribution of this thesis is the proposal of a model of hierarchy to fault tolerance domains, which facilitates the management and communication for inter-domain groups.

SUMÁRIO

CAPÍTULO 1
Introdução

1.1 Motivações

As necessidades de aplicações confiáveis impulsionaram pesquisas em tolerância a faltas em ambientes de sistemas distribuídos. A obrigatoriedade de serviços oferecidos continuamente, mesmo na presença de falhas parciais no sistema, de certa maneira é favorecida pela disponibilidade natural de processadores e hardware nestes sistemas. Deste modo, usualmente, são utilizadas técnicas de replicação em serviços críticos.

Mas as características de sistemas distribuídos fracamente acoplados colocam implicações na manutenção da consistência entre as réplicas do serviço. Os principais problemas estão em se manter a consistência interativa [Lamport82] e a detecção de falhas dos diferentes componentes membros da replicação nestes ambientes. Normalmente, estas dificuldades são cobertas pelas funcionalidades de suportes de comunicação de grupo. Diante disso, um esforço considerável tem sido feito no sentido da introdução de conceitos de grupo na programação distribuída, no desenvolvimento de produtos comerciais [Birman91, ISIS95, IONA95].

Por outro lado, nos últimos anos, a área de sistemas distribuídos tem avançado também na direção da padronização aberta. Esta tendência vem no sentido de combater soluções proprietárias devido aos altos custos de desenvolvimento de sistema que o uso das mesmas traz – por exemplo, custos provenientes da dificuldade de manutenção de software, interoperabilidade e portabilidade. Atualmente, muitas aplicações distribuídas estão seguindo o paradigma de orientação a objetos e considerando o CORBA (*Common Object Request Broker Architecture*) [OMG96] como a melhor alternativa de se adequar a sistemas abertos. No entanto, as especificações iniciais do CORBA não contemplavam requisitos de tolerância a faltas, fundamentais em sistemas distribuídos. Isto motivou vários grupos de pesquisa no sentido de propor soluções para esse problema. As

preocupações iniciais eram investigar meios para disponibilizar serviços de comunicação de grupo na arquitetura CORBA. Estas experiências podem ser classificadas em três abordagens: *Integração* [Maffeis95, ISIS95], *Serviço* [Felber98] e *Interceptação* [Fraga97, Moser98, Lau00a]. Paralelo a isso, outros trabalhos, seguindo essas mesmas abordagens, propuseram meios para suporte a tolerância a faltas em *middlewares* CORBA [Killijian98, Chung98, Lau99a], alguns desses trabalhos incorporam também comunicação de grupo no suporte.

O amadurecimento dessas pesquisas e a crescente demanda por aplicações confiáveis em sistemas distribuídos foram determinantes para que a OMG (*Object Management Group*) resolvesse então, lançar um edital convidando empresas e instituições de pesquisa a submeterem propostas no sentido de introduzir tolerância a faltas no CORBA [OMG98a]. Como resultado disso, após alguns anos de trabalho, foi publicada então, as especificações do *Fault-Tolerant* CORBA (FT-CORBA) [OMG00a]. Essas especificações, que ainda devem passar por varias revisões (e extensões), definem um conjunto de interfaces de serviços e facilidades úteis para a implementação de técnicas de replicação em ambientes distribuídos heterogêneos. As especificações FT-CORBA atendem apenas aos requisitos básicos para tolerância a faltas, definindo algumas interfaces bastante genéricas, de fácil entendimento, e úteis em praticamente todas as aplicações que requerem tolerância a faltas em sistemas distribuídos.

No entanto, nestas especificações não existe a definição de interface para um serviço de comunicação de grupo, fundamental na implementação de replicações ativas em sistemas distribuídos. Segundo o documento [OMG00a], as necessidades relacionadas com a comunicação de grupo podem, no momento, ser supridas por uma ferramenta proprietária de comunicação de grupo. Ademais, é parte do plano de diretrizes da OMG especificar, num futuro próximo, um serviço de comunicação de grupo no padrão CORBA.

Por outro lado, redes altamente distribuídas (tal como a Internet), têm caracterizado o que é normalmente identificado como aplicações e sistemas distribuídos de larga escala. Sistemas de larga escala são caracterizados pela grande distribuição espacial, com um caráter aberto, integrando quantidades significativas de recursos computacionais, assinalados pela sua heterogeneidade. Projetar sistemas escaláveis tem sido, nos últimos

anos, alvo de pesquisa na comunidade de sistemas distribuídos, principalmente em padronizações de sistemas abertos, tal como o CORBA.

Requisitos para tolerância a faltas em redes de larga escala envolvem complexidades adicionais tanto na comunicação como na gestão dos componentes do sistema distribuído. Em uma análise detalhada dessas especificações, verificamos que muitas das necessidades para desenvolvimento de aplicações tolerantes a faltas em redes de larga escala ainda não foram abordados no FT-CORBA. Portanto, é um problema aberto e uma das nossas motivações neste trabalho.

1.2 Objetivos da tese

Nesta tese, nossas pesquisas são concentradas em trabalhos, desenvolvidos nos últimos seis anos, que tratam da tolerância a faltas em sistemas distribuídos adotando arquiteturas abertas. A nossa aproximação com a arquitetura CORBA foi natural, mas também impulsionada pela rápida aceitação da mesma.

Este texto mostra a nossa evolução nestes anos. A princípio de uma maneira original, sem trabalhos para confrontar os nossos esforços, e depois com o surgimento de vários trabalhos com os mesmos propósitos o que serviu para aprofundar questões que nortearam nossa conduta durante este período. O princípio básico era utilizar tecnologias abertas, tal como CORBA, no desenvolvimento de sistemas distribuídos obtendo, portanto, todas as vantagens que aportam tais soluções na redução dos custos de desenvolvimento e de manutenção nesses sistemas. E para não perder as características de interoperabilidade, propriedade fundamental de qualquer sistema aberto, era necessário saber como introduzir suporte de grupo e mecanismos de tolerância a faltas na arquitetura CORBA sem que isso representasse em qualquer modificação nas especificações do padrão.

Nossas pesquisas em tolerância a faltas no CORBA, se dividem em duas fases: antes e depois da padronização do FT-CORBA. Na primeira fase, os trabalhos mostram a nossa procura por novos paradigmas e conceitos na programação orientada a objetos que nos permitissem incluir os suportes necessários a partir de plataformas CORBA sem que isto representasse em modificações nas especificações. Foi com este intuito que usamos a Reflexão Computacional e o conceito de interceptação [Lau95, Lau96b]. A nossa evolução

foi também diante da bibliografia relacionada. A partir destas perspectivas, as pesquisas conduziram na definição de duas propostas:

- MetaFT [Lau96a, Fraga97, Lau99c]: é um modelo de programação que explora as facilidades do paradigma de reflexão computacional na implementação de técnicas de replicação em ambientes heterogêneos. O modelo adota uma estrutura reflexiva segundo a abordagem de meta-objetos [Maes87];

- GroupPac 1.0 [Oliveira99a, Oliveira99b, Lau00a, Lau00d]: é uma plataforma que fornece uma coleção de objetos de serviços para suporte a processamento de grupo e/ou tolerância a faltas. Estes objetos podem ser combinados, constituindo *frameworks*, para implementar técnicas de replicação. Um exemplo de uso destes serviços é o CosNamingFT [Lau99a, Lau99b] – um serviço de nomes CORBA tolerante a faltas.

Com a disponibilidade das especificações FT-CORBA, a segunda fase do texto da tese consistiu, inicialmente, no estudo dessas especificações [Lau00c, Fraga01] e na implementação e adequação do GroupPac a esses novos padrões [Lau01b]. Os trabalhos tomaram novos rumos. Uma vez que normalmente a necessidade de padrões abertos se faz sentir mais presente em redes de larga escala, a questão que nos colocamos foi a de prover tolerância a faltas a partir do CORBA mas considerando aspectos de escalabilidade. Nesta segunda fase do texto, é mostrado nossos esforços primeiro em verificar as limitações das especificações FT-CORBA e depois em propor um modelo que adapte as especificações FT-CORBA para aplicações distribuídas de larga escala [Lau00a, Lau00b, Lau01a].

1.3 Organização do texto

A organização deste texto reflete as etapas desenvolvidas durante as pesquisas realizadas no doutoramento. Esta tese está dividida em sete capítulos. Os capítulos de dois a quatro descrevem a primeira fase (antes da padronização do FT-CORBA) das nossas experiências. Os capítulos seguintes, com exceção da conclusão, descrevem nossos trabalhos a partir da padronização das especificações FT-CORBA.

O capítulo 2 introduz uma série de conceitos sobre requisitos para introduzir suporte de grupo e tolerância a faltas no CORBA, e apresenta as abordagens, acompanhado de exemplos, para alcançar isso. Discussões e considerações sobre estas abordagens são apresentadas ao final deste capítulo.

No capitulo 3, são apresentados nossas duas propostas (MetaFT e GroupPac 1.0) para introduzir tolerância a faltas no CORBA. Aspectos de implementação e medidas de desempenho, de ambos, são também apresentados. Ao fim deste capítulo, são feitas discussões e comparações com as outras propostas encontradas na literatura.

Um serviço de nomes CORBA tolerante a faltas é apresentado no capítulo 4, juntamente com aspectos de implementação e medidas de desempenho, no sentido de exemplificar a aplicabilidade do GroupPac 1.0 para fornecer suporte a técnicas de replicação.

O capítulo 5 apresenta, de forma resumida as especificações FT-CORBA e os nossos esforços, refletido no GroupPac 2.0, no sentido de implementar estas especificações.

A proposta de adaptação das especificações FT-CORBA para sistemas de larga escala é apresentado no capítulo 6. Também, ao final deste, um conjunto de considerações sobre esta proposta são apresentadas. Complementando o capitulo, é apresentada nossa experiência na implementação do modelo proposto.

No capítulo 7 são apresentadas as principais conclusões e perspectivas de trabalhos futuros.

Além disso, dois anexos são introduzidos para facilitar leitores não familiarizados com a literatura e conceitos utilizados neste texto. No primeiro, Anexo A, é apresentada a taxonomia usada neste texto e que está baseada em [Laprie92]. Neste mesmo anexo foi colocado um texto resumindo as principais técnicas de replicação usadas em sistemas distribuídos. No anexo B, é descrito um estudo sobre comunicação de grupo em redes de larga escala. São apresentados o problema da escalabilidade em comunicação de grupo e também algumas propostas da literatura de algoritmos para solucionar este problema. Este estudo serviu de base para nossas propostas.

CAPÍTULO 2
Suporte de Grupo e Tolerância a Faltas no CORBA

2.1 Introdução

A arquitetura CORBA é uma plataforma aberta definida através de um conjunto de especificações e de conceitos para que objetos distribuídos, em um ambiente de rede heterogêneo, possam ser acessíveis através de uma interface bem definida. No entanto, as especificações do CORBA inicialmente não apresentavam uma visão muito clara, em termos de conceitos e de necessidades, de suporte para a introdução da noção de grupo de objetos e de mecanismos para tolerância a faltas na programação distribuída aberta. Devido a isto, o padrão CORBA tem sido objeto de extensão em vários protótipos e mesmo produtos para garantir o suporte para grupo. Em particular, podemos citar as propostas: Orbix+Isis [IONA95], Electra [Maffeis95a], OFS [Liang96], OGS [Felber98], Ethernal [Moser98] e GroupPac (apresentado no capitulo 4) que incluem a noção de grupo de objetos como uma das funcionalidades disponíveis a partir de um ORB.

Inicialmente, este capítulo apresenta uma descrição sucinta do modelo de objetos distribuídos CORBA padronizado pela OMG. Em seguida, levantamos um conjunto de requisitos gerais desejáveis para o processamento em grupo em arquiteturas abertas. Com base nestes requisitos, apresentamos um conjunto de propostas, encontradas na literatura, que tratam da inclusão de suporte de grupo e/ou tolerância a faltas em *middlewares* CORBA. Estas propostas são classificadas segundo as abordagens de integração, de serviço e de interceptação. Por fim, uma comparação entre estas propostas é feita à luz dos requisitos e conceitos identificados anteriormente. É importante ressaltar que este levantamento de requisitos, as abordagens e o estudo comparativo apresentado em [Lau95, Lau96c], foram realizados muito antes da padronização do FT-CORBA [OMG00a] pela OMG.

O intuito deste capítulo é mostrar a adequação destas proposições de ORBs a modelos usuais de técnicas de replicações encontrados na literatura de tolerância a faltas. A discussão sobre requisitos gerais para suporte de grupo e os conceitos CORBA, nos próximos itens deste capítulo, está fortemente fundamentada no trabalho descrito em [ISIS93, Liang90].

2.2 Arquitetura CORBA

O OMG (*Object Management Group*) é uma organização, criada em 1989, formada por um grupo de mais de 800 empresas (tais como IBM, SUN, DEC, Microsoft, Novell, etc) com o objetivo de especificar um padrão aberto para a programação distribuída orientada a objeto chamada arquitetura CORBA. Esta arquitetura é um conjunto de mecanismos padronizados e de conceitos que se baseiam no modelo cliente-servidor em que objetos distribuídos encapsulam um estado interno e se fazem acessíveis através de uma interface bem definida. Segundo essa arquitetura, métodos de objetos remotos podem ser ativados de forma transparente em ambientes distribuídos heterogêneos através de um ORB (*Object Request Broker*). O ORB, num sentido mais genérico, é um canal de comunicação para objetos distribuídos. No CORBA, é possível também obter interoperabilidade entre objetos desenvolvidos em diferentes linguagens de programação (C, C++, Ada, Cobol, Java, etc.), onde as diferenças de cada uma são mascaradas pelo CORBA através do mapeamento adequado da IDL (*Interface Definition Language*) [OMG96] para a linguagem de programação desejada. Cada objeto CORBA tem sua interface especificada em IDL, uma linguagem declarativa, sem nenhuma estrutura algorítmica, com sintaxes e tipos predefinidos baseados na linguagem C++. Portanto, o uso da linguagem de definição de interface (IDL) permite tratar das heterogeneidades do ambiente computacional, tais como os diferentes tipos de maquinas, sistemas operacionais e linguagem de programação.

2.2.1 O modelo de objeto

Esta seção apresenta as terminologias e os aspectos conceituais do modelo de objetos definido pela OMG [OMG96]. Um sistema de objetos é uma coleção de objetos onde os requerentes do serviço (os clientes) são isolados dos provedores dos serviços (os

servidores) através de interfaces bem definidas. Um *cliente* – não necessariamente um objeto – envia requisições de serviço (invocações de métodos) a um objeto servidor. Toda *requisição* pode ser interpretada como um evento que ocorre em um valor de tempo qualquer, provocando o processamento no servidor. A uma requisição estão associadas informações sobre o endereço do objeto alvo (referência de objeto), da operação requisitada e dos parâmetros associados à operação. Uma *referência de objeto* é definida como sendo um valor associado a um objeto particular que identifica o mesmo quando usada em uma requisição. No CORBA, um objeto pode ser identificado por diferentes referências de objeto.

No modelo CORBA, os objetos só podem ser criados através de instancia ou destruídos por meio de requisições de um cliente. Isto é devido ao fato de não existirem mecanismos de construtores e destrutores – como em algumas linguagens de programação orientada a objetos – nas especificações IDL/CORBA. A criação de um objeto por um cliente é revelada na forma de uma referência de objeto que denota um novo objeto.

Todos os objetos deste modelo são descritos através de suas *interfaces* que descrevem o conjunto de operações que podem ser requisitados pelos seus respectivos clientes. O uso da interface permite ocultar do cliente aspectos relacionados à implementação do serviço. O conceito de interface está diretamente relacionado ao conceito de classe no modelo de objetos clássico. Tanto que no modelo de objetos da OMG é permitido aplicar, por exemplo, os conceitos de encapsulação, herança e polimorfismo.

2.2.2 A arquitetura OMA

O objetivo da OMG é permitir o crescimento da tecnologia a objetos e influenciar sua direção no sentido de cumprir os requisitos e conceitos estabelecidos na OMA (*Object Management Architecture*) – uma arquitetura que fornece a infra-estrutura conceitual para todas as especificações da OMG. As regras definidas nesta arquitetura de referência visam auxiliar a construção de componentes de software interoperáveis, reutilizáveis e portáveis, baseada em interfaces padrões e abertas, e orientadas a objetos. Esta arquitetura é composta de quatro componentes principais (figura 2.1): o ORB, os objetos de serviço, as facilidades comuns e os objetos da aplicação. Nos próximos itens desta seção, será descrito cada um destes componentes da arquitetura OMA.

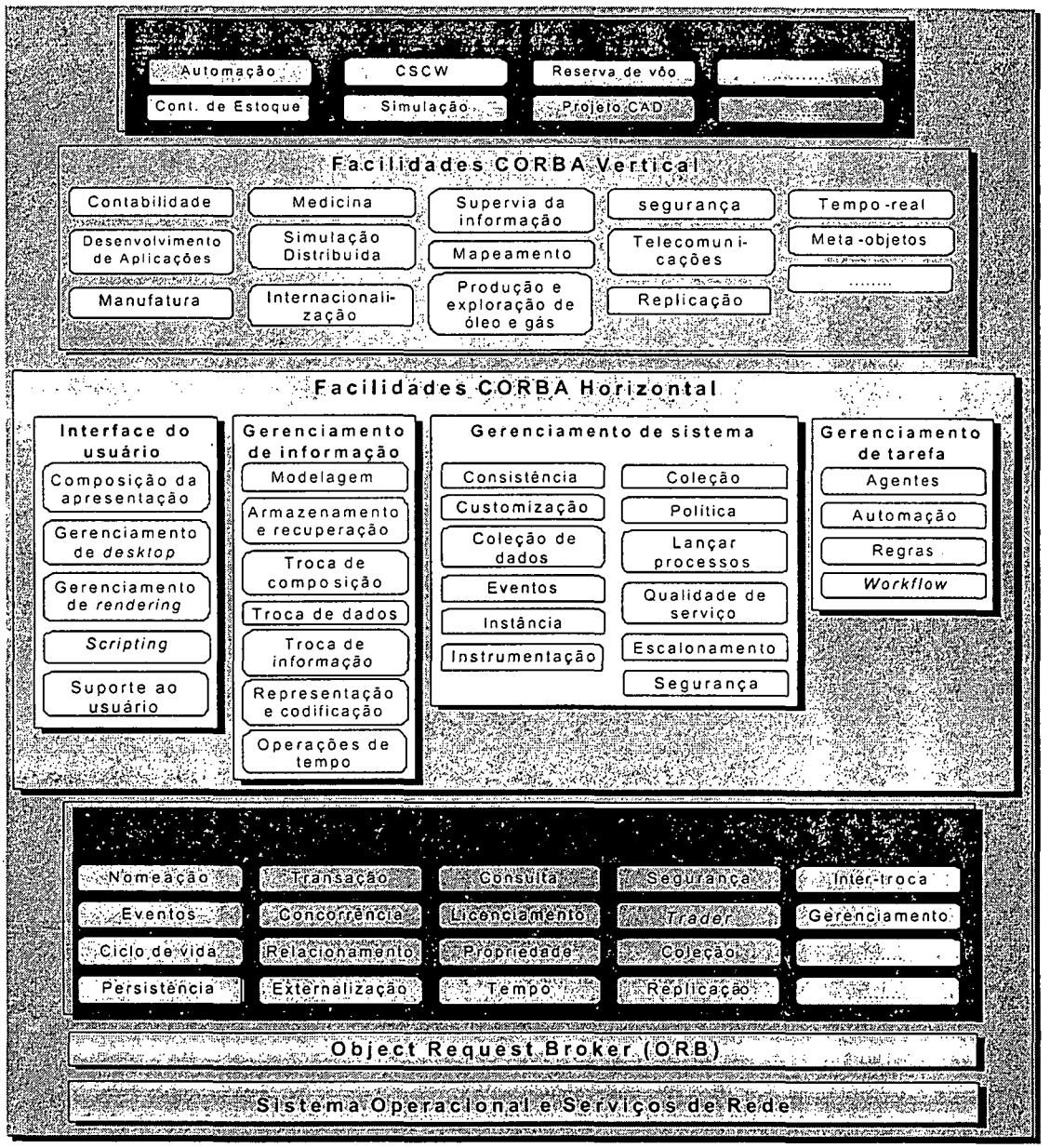


Figura 2.1. Arquitetura OMA (*Object Management Architecture*).

2.2.2.1 Estrutura do ORB

O CORBA (*Common Object Request Broker Architecture*) é constituído por um ORB que implementa abstrações e semânticas na comunicação entre objetos em um sistema distribuído (figura 2.2). O ORB, num sentido mais genérico, pode ser entendido como um canal de comunicação para que objetos heterogêneos possam interoperar. Permite que objetos invoquem, transparentemente e confiavelmente, operações em objetos remotos em um ambiente distribuído heterogêneo.

As interações no ambiente CORBA seguem o modelo cliente/servidor. Uma chamada cliente é transformada em uma requisição (mensagem) através de *stubs* particulares do CORBA (geradas pelo compilador IDL). A mensagem serializada (*marshalling*) é transmitida na rede usando as estruturas do ORB que localiza o objeto servidor e transporta os dados de acordo com a sintaxe de transferência. No servidor, a mensagem que chega passa o controle para o *Adaptador de Objeto Portável* (*Portable Object Adaptor* - POA) que, por sua vez, tem a responsabilidade de ativar a implementação de objeto correspondente. Os processos de deserialização (*unmarshalling*) e a conseqüente chamada de método na implementação do objeto são realizados através do correspondente *skeleton* do objeto servidor.

Quando o processamento do método invocado conclui, o retorno se utiliza dos mesmos mecanismos (Stubs, ORB, etc) na entrega ao cliente dos resultados.

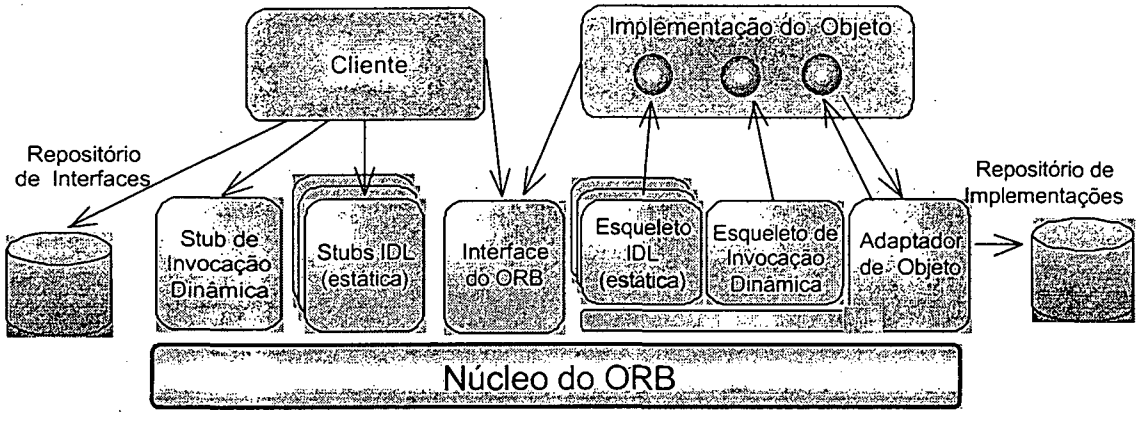


Figura 2.2. Arquitetura CORBA (*Common Object Request Broker Architecture*).

O *Adaptador de Objeto Portável* (POA), presente na figura 2.2, oferece uma maneira bastante simples para a *Implementação do Objeto* acessar serviços oferecidos pelo ORB. Alguns destes serviços ORB disponíveis através do POA são: geração e interpretação de referências de objetos, invocação de métodos, ativação e desativação de objetos, mapeamento de referências de objetos para implementações, registros de implementações. O POA é um adaptador de objeto padronizado pela OMG, mas nada impede que se desenvolvam diferentes tipos de adaptadores de objetos, de acordo com as necessidades específicas de uma implementação de objeto.

A *Interface do ORB* (figura 2.2), segundo as especificações CORBA, deve ser idêntica para as diferentes implementações CORBA. Esta interface deve ser independente

de qualquer interface de objeto ou adaptador de objeto. Devido a maioria das funcionalidades do ORB serem oferecidas através dos adaptadores de objetos, *stubs* e *skeletons*, nesta interface é oferecida apenas um pequeno número de operações que são comuns tanto para clientes como para implementações de objetos.

Na requisição de um serviço, o cliente pode realizar uma invocação no servidor de duas formas: estática através de *Stubs* IDL ou dinâmica através da interface de invocação dinâmica (DII). Em ambas abordagens a implementação do objeto (o servidor) não percebe o tipo de invocação utilizada na requisição pelo cliente. Na invocação estática, o cliente faz a invocação de um método no objeto servidor se utilizando de *Stubs* IDL apropriados, gerados no processo de compilação do arquivo de definição de interface IDL correspondente. Uma *stub* pode ser entendida como uma representação local do objeto remoto.

Para invocações dinâmicas, as interfaces de objetos CORBA devem estar disponíveis, armazenadas no sistema (*repositório de interfaces*). Uma invocação dinâmica permite ao cliente dinamicamente construir e invocar métodos no servidor usando interfaces DII. Este tipo de chamada oferece uma grande flexibilidade para sistemas altamente dinâmicos: o cliente determina o objeto a ser invocado, o método a ser executado e o conjunto de parâmetros desse método através de uma chamada ou uma seqüência de chamadas, em tempo de execução.

O *Repositório de Interfaces* contém informações de interfaces IDL em uma forma disponível em tempo de execução para as invocações dinâmicas. Usando a informação disponível no repositório é possível a um programa encontrar um objeto remoto mesmo desconhecendo sua interface, seus métodos e parâmetros, e sua forma de ativação. Já o *Repositório de Implementações* contém informações que permitem ao ORB localizar e ativar implementações de objetos.

2.2.2.2 Objetos de serviço – COSS

No desenvolvimento de aplicações de objetos distribuídos, a OMG oferece um conjunto de serviços que facilitam o trabalho do projetista da aplicação. No padrão CORBA, essas funcionalidades adicionais são providas como objetos de serviços, isto é,

objetos CORBA específicos com suas interfaces IDL bem definidas. Por exemplo, a concorrência e a persistência, que são necessárias para algumas aplicações, não são incluídas, pela OMG, como mecanismos no núcleo do ORB (figura 2.2). Ao contrário disto, a OMG define os objetos de serviço COSS (*Common Object Services Specifications*) [OMG97a] que tratam destes temas, mantendo o núcleo do ORB o mais simples possível com apenas as funcionalidades comuns a todas as aplicações. Portanto, os serviços COSS podem ser entendidos como uma extensão ou uma complementação das funcionalidades do ORB.

Os COSS formam uma coleção de serviços (interfaces e objetos) que são padronizados pela OMG dentro da arquitetura OMA (figura 2.1). Ademais, múltiplos objetos de serviços podem cooperar no sistema. Por exemplo, um serviço de suporte a grupo pode utilizar o serviço de recuperação de erro para restabelecer objetos servidores após uma falha. Além de estar de acordo com a norma CORBA, um serviço é muito mais fácil de ser padronizado do que uma extensão ao ORB ou até mesmo um novo adaptador de objetos.

Uma especificação de objeto de serviço usualmente consiste de um conjunto de interfaces e de uma descrição do comportamento do serviço, que podem ser utilizados por objetos de aplicação e outros objetos de serviço. A sintaxe usada para especificar as interfaces é a IDL/OMG. A semântica que especifica o comportamento do serviço é expressa, em geral, em termos de objetos, operações e tipos de dados padronizados [OMG96]. A OMG tem definido até agora vários serviços (figura 2.1), tais como:

- **Serviço de ciclo de vida**: define serviços e convenções para criar, deletar, copiar e mover objetos. Devido aos ambientes baseados no CORBA suportar em objetos distribuídos, este serviço define serviços e convenções que permitem ao cliente realizar operações nesse serviço remotamente;

- **Serviço de persistência**: define uma interface única para o armazenamento persistente do estado dos objetos nos diversos servidores de armazenamento, incluindo objetos base de dados (ODBMSs), base de dados relacionais (RDBMSs) e arquivos simples. O estado do objeto é considerado composto de duas partes: o *estado dinâmico*, que está tipicamente na

memória e não constitui parte do ciclo de vida do objeto como um todo (ex: informações que não seriam preservados em um evento de falha no sistema); e o *estado persistente* que o objeto usa para reconstruir o seu estado dinâmico, quando necessário;

- **Serviço de nomes**: estas especificações definem as necessidades na implementação e na administração distribuída de nomes e seus contextos em ambientes CORBA. Permite que componentes a partir de um ORB localizem outros componentes por nome, mesmo que em outros ORBs. Também, é definido suporte para nomeação de contextos federados (*federated naming contexts*). Todo um suporte hierarquizado para gestão de nomes e contextos é fornecido nesse serviço. Uma associação entre nome e referência de objeto é chamada *name binding*. Um *name binding* pode definir também um contexto de nomes (*naming context*). Neste caso, um contexto de nomes pode ser visto como um objeto que contém, na sua representação, um conjunto de *name bindings*. Em um contexto de nomes cada nome é único. Diferentes nomes podem ser designados a um objeto no mesmo ou em diferentes contextos ao mesmo tempo;

- **Serviço de notificação de eventos**: este COSS define a forma como os objetos devem registrar dinamicamente seus interesses em eventos específicos. O serviço define um objeto chamado canal de eventos que coleta e distribui eventos entre os componentes através do ORB, esse serviço desacopla as comunicações entre objetos cliente e servidor. O serviço define duas funções: a função do fornecedor do evento e a função do consumidor do evento. O fornecedor produz o evento e o consumidor processa o evento. Os eventos são transmitidos entre o fornecedor e o consumidor através de requisições especificadas segundo o CORBA. São definidas duas abordagens para este serviço: o modelo *push* e o modelo *pull*. O modelo *push* permite ao fornecedor do evento iniciar a transferência do evento ao consumidor. O modelo *pull* permite ao consumidor de eventos requisitar o evento ao fornecedor do evento.

- **Serviço de controle de concorrência**: define um gerenciador de trancas (*locks*) que garante a consistência e o controle de concorrência em objetos através do compartilhamento dos mesmos por *threads*. É utilizado, por exemplo, em aplicações que necessitam de um suporte *multi-threads*, para serializar o acesso aos recursos compartilhados.

- **Serviço de segurança**: o CORBASec é introduzido, através deste COSS, para definir serviços de segurança envolvendo autenticação, autorização, auditoria e não repudiação, usados na implementação de políticas de segurança nas interações cliente/servidor em um ambiente CORBA.

- **Serviço de tempo**: este COSS define um serviço de tempo global para ambientes normalmente distribuídos e heterogêneos. Este serviço é importante, por exemplo, na ordenação dos eventos que ocorrem no sistema. Pode ser usado também para disparar eventos em valores de tempo especificados através de mecanismos de temporização e alarmes ou ainda, para computar intervalos de tempo entre eventos. Este serviço segue o padrão UTC (*Universal Time Coordinated*).

Existem ainda outras especificações de serviço COSS, tais como o de transação, licenciamento, tolerância a faltas, propriedades que não estão descritos neste texto [OMG97].

2.2.2.3 Facilidades comuns

As facilidades comuns são um coleção de serviços de propósitos gerais utilizados pelas mais diversas aplicações. As facilidades comuns são divididas segundo dois aspectos, as facilidades comuns horizontais e as facilidades comuns verticais (figura 2.1). As facilidades horizontais podem ser utilizadas por diferentes aplicações, independente da área da aplicação, são divididas segundo quatro categorias: interface de usuário, gerenciamento de informação, gerenciamento de sistema e gerenciamento de tarefa. As facilidades verticais são utilizadas em áreas de aplicação especificas, por exemplo: gerenciamento e controle de imagens, supervias de informação, manufatura integrada por comutador, simulação distribuída, contabilidade, etc.

2.2.2.4 Interoperabilidade entre ORBs

Nas especificações CORBA 1.1 não estavam claras a forma de como garantir a interoperabilidade entre objetos pertencentes a diferentes implementações de ORBs. A política da OMG é a padronização do CORBA e seus serviços, na maioria das vezes, apenas em nível de interface, deixando em aberto aos desenvolvedores de *software* a parte de implementação. Devido a isto, ficou difícil de garantir uma interoperabilidade entre objetos de diferentes implementações de ORB sem uma padronização do protocolo de comunicação e do formato das mensagens que deveriam circular nesses ORBs. Para suprir esta necessidade, a OMG publicou o padrão CORBA 2.0, nestas especificações foi definido os seguintes protocolos (figura 2.3):

- *Protocolo Inter-ORB Geral* (GIOP): que especifica um conjunto de formatos para mensagens e dados transportados nas comunicações entre ORBs;
- *Protocolo Inter-ORB Internet* (IIOP): que especifica como mensagens GIOP são transmitidas numa rede TCP/IP (GIOP + TCP/IP = IIOP);

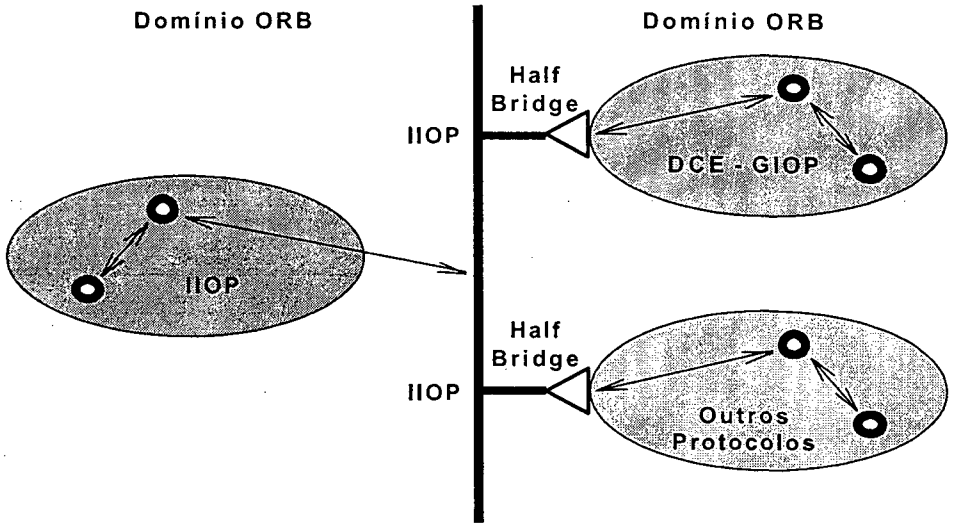


Figura 2.3. Interoperabilidade entre ORBs.

- *Protocolo Inter-ORB para Ambientes Específicos* (ESIOPs): que é uma especificação feita para permitir a interoperabilidade de um ORB com outros ambientes (ex.: DCE).

Para comunicações de objetos em ambientes CORBA com outro em ambientes não CORBA foi introduzido o *Half Bridge*. Este mecanismo faz o mapeamento que transforma uma requisição expressa em termos do modelo de domínio origem para o modelo do domínio destino. Isto é, converte o formato das mensagens, de um domínio não CORBA, para o GIOP para que possam ser transmitidas através do protocolo IIOP, e vice-versa.

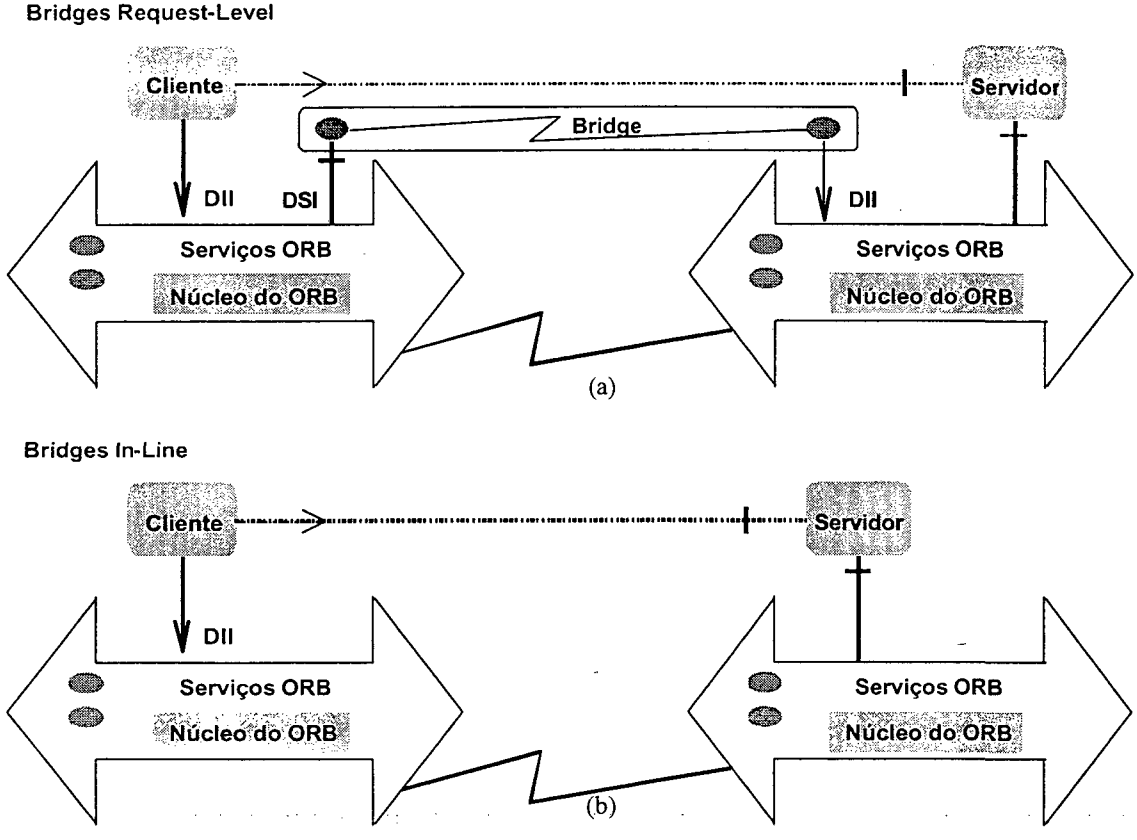


Figura 2.4. As duas formas para interoperabilidade entre ORBs.

Na versão 2.0 do CORBA foi acrescentada a interface *skeleton* dinâmica (DSI) que é utilizada para interconexão de ORBs distintos. Quando o POA recebe uma invocação para uma implementação de objeto que não tem um *skeleton* pré-compilado (abordagem estática), esta invocação é passada para a interface de *skeleton* dinâmica que ativa a DII de um outro ORB que possua este *skeleton*, onde vai ser feita a invocação; isto forma o que é identificado nas especificações como *Bridge Request-Level* (figura 2.4a). A outra forma de interoperabilidade é através da abordagem *Bridge In-Line* na qual "a ponte" é implementada dentro do núcleo do ORB (figura 2.4b).

2.3 Inclusão de suporte de grupo e tolerância a faltas no CORBA

A noção de grupo determina uma associação em que seus membros apresentam uma relação abstrata comum, uma política interna e/ou uma regra de acesso comum [Lea94]. Um grupo de objetos deve estar dentro destas condições de associação. Modelos de processamento em grupo são essenciais em aplicações distribuídas, contribuindo para requisitos de maior disponibilidade de recursos, tolerância a faltas, distribuição de carga, etc. Um grupo de objetos, como um objeto qualquer, encapsula um estado e se faz acessível através de um conjunto de métodos bem definidos. Requisitos de transparência de grupo podem ser colocados tanto em nível de cliente como de servidor: objetos simples e grupos de objetos não são distinguíveis externamente. Características como componentes reusáveis, presentes no paradigma objeto, devem se manter quando em relações de grupo.

As questões que se colocam neste contexto são: quais requisitos de serviços de grupo desejamos ? Como integrar grupos de processamento em sistemas abertos construídos a partir do uso de padrões e plataformas consagradas ? É evidente que os requisitos dependerão do modelo de processamento de grupo usado e estes terão reflexos em vários aspectos do suporte, como o gerenciamento e a comunicação (itens 3.3.1. e 3.3.2). Para que possamos usufruir as vantagens do processamento de grupo em modelos de programação orientada a objetos em ambientes abertos, é necessário definir mecanismos de suporte e avaliar a integração destes mecanismos nas estruturas destes ambientes. Este item faz um estudo preliminar de alguns requisitos, considerados gerais, para inclusão de suporte de grupo em *middleware* CORBA.

2.3.1 Gerenciamento de grupo

O gerenciamento de um grupo de objetos deve tratar com a consistência do processamento no modelo de grupo no sistema. Aspectos como criação e ativação de grupos, *membership*[1], tratamento de faltas, coordenação no processamento replicado são tópicos do gerenciamento de grupo que serão discutidos na ótica da integração destas funcionalidades ao modelo CORBA.

[1] Lista de membros ativos no grupo.

2.3.1.1 Serviço de membership

Este é um dos principais mecanismos que um suporte a grupos de objetos precisa oferecer para garantir aplicações distribuídas robustas. O mecanismo é caracterizado por um algoritmo que atualiza as entradas e saídas (normais ou por falha) de objetos no grupo, mantendo uma lista corrente de membros ativos (*membership*). Uma possível solução para a disponibilidade deste serviço no CORBA seria estender a interface *Portable Object Adapter* (POA) com operações como *addmember* e *delmember* ou, como outra alternativa, que se crie, por exemplo, uma outra interface *Group Object Adapter* (GOA) como um subtipo do POA englobando o *membership* e outras operações de gerenciamento. Uma outra possibilidade, que não seja integrar este serviço a partir do ORB, é fornecer tal funcionalidade na forma de um objeto de serviço comum (COSS).

Um *groupview* ou lista de membros é necessário que se obtenha pelo ORB para a coordenação das execuções no grupo e para implementar mapas de comunicação de grupo. Portanto, mudanças de *membership* devem ser comunicadas a todos os membros do grupo. Estas comunicações de mudanças devem ser recebidas na mesma ordem pelos componentes ativos do grupo. O uso de serviços de difusão atômica é imprescindível para tal. As informações e o serviço de *membership* certamente não precisam estar contidos no ORB, mas este pode ter mecanismos necessários para a notificação confiável destas informações nos membros do grupo.

2.3.1.2 Serviços de tratamento de faltas

Este serviço inclui a detecção de objetos com comportamento faltoso e eventualmente, o respectivo tratamento. Em grupos de objetos é necessário criar mecanismos para detectar a ocorrência de falhas em membros do grupo, prevenindo que, por exemplo, o cliente fique esperando indefinidamente a resposta. Serviços de detecção de falhas podem ser implementados fazendo uso de mecanismos de *timeout* e de *keepalive*, úteis na detecção de falhas. Um serviço deve ser responsável por coletar todas as informações de *timeout* e *keepalive* e produzir um fluxo único confiável e coerente de notificação de falhas. Na implementação deste serviço em um *middleware* CORBA é preciso decidir a classe de faltas a ser detectada (ou monitorada). Além disso, o serviço de *membership* deve se encarregar da nova lista de membros (*view*) a cada detecção de falha.

2.3.1.3 Serviços de transferência de estado

Serviços de transferência de estado são importantes no tratamento de faltas e na re-inserção de objetos dentro de um grupo. A transferência pode se dar através de uma cópia consistente do estado corrente de um objeto, de forma que um novo objeto entrando no grupo possa ao receber a cópia, tornar-se uma réplica idêntica às outras. Serviços de *checkpoint* e *log* podem periodicamente copiar em disco o estado de um objeto (*checkpoint*) e requisições subseqüentes ao último salvamento de estado no grupo, caracterizando *checkpoints* e *logs* que são usados na recuperação em situação de falhas dos membros do grupo. Serviços de transferência de estado podem ser implementados a partir da interface POA ou mesmo como um objeto de serviço COSS.

2.3.2 Comunicação de grupo

O suporte para grupo tem como um dos pontos essenciais o desempenho da comunicação. Um ORB pode ser visto como canalizando pedidos e respostas de serviços, mediando interações através de protocolos apropriados. O pedido de um cliente deve ser mapeado, transparentemente, em pedidos de operação em objetos membros do grupo. Dependendo da técnica de replicação que possa estar usando o suporte de comunicação de grupo, nem todos os membros necessitam receber ou executar a operação. Segundo os modelos de replicações usuais, alguns mapeamentos possíveis na comunicação são:

- Transmitir o pedido a um membro privilegiado ("objeto primário ou líder");
- Enviar o pedido de forma aleatória a um membro do grupo;
- Enviar o pedido a todos os membros do grupo (modelo Máquina de Estado [Schneider90])

Todos estes "mapas" são implementáveis a partir de um ORB, usando protocolos apropriados e também os serviços de *membership*. Em relação aos resultados, se muitos, provenientes dos vários membros, é necessário que sejam coletados e apresentados ao cliente como um resultado único. Outra vez, dependendo do modelo de grupo, as alternativas possíveis são:

- O primeiro resultado que chegar é passado ao cliente, os restantes são descartados;

- Concatenar os resultados em seqüências e enviar ao cliente. Esta alternativa é útil em modelos de processamento cujos membros participam de alguma forma de processamento paralelo;

- Um ajustador ou votador calcula um valor a ser enviado ao cliente a partir do conjunto de resultados recebidos.

A escolha do mapeamento no envio de pedidos e da coleta de resultados deve ser especificada como parte da implementação do processamento replicado e, portanto estar armazenada no repositório de implementações. A entidade que implementa essa funcionalidade pode consultar estes mapas quando da ativação de um grupo. A execução dos mapas de envio e a coleta de resultados pelo ORB são importantes para se obter os graus de transparência desejados.

2.3.2.1 Transparência de grupo de objetos

A transparência entre grupos de objetos é um requisito importante porque simplifica os códigos de aplicação afastando dos mesmos as complexidades de gerenciar a comunicação de grupo. Dois tipos de transparência são delineáveis nas relações de grupo:

- Transparência do servidor: os objetos membros de um grupo servidor são vistos pelos seus clientes como se fossem um objeto servidor simples que implementa a mesma interface. A implementação através de um *middleware* CORBA de um mapa apropriado de envio para requisições e também para a coleta de resultados pode definir a transparência do servidor na forma de grupo, da figura 2.5, onde o cliente não distingue entre um objeto simples e o grupo representado da figura.

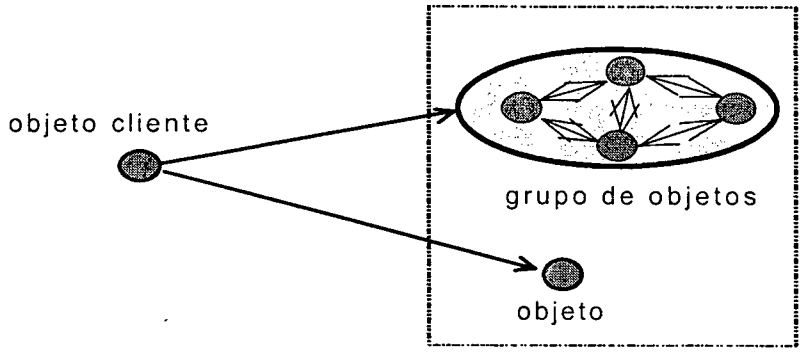


Figura 2.5. Transparência de grupo servidor.

- Transparência do cliente: as requisições enviadas de um grupo cliente ou de um objeto cliente simples para um objeto servidor devem ser equivalentes e tratadas de maneira única. Isto se consegue através de mecanismos do ORB ou do modelo de replicação utilizado no cliente. Na figura 2.6, um dos membros é o responsável pelas interações externas ao grupo; por outro lado, na figura 2.7, um mecanismo no ORB captura as múltiplas requisições replicadas de um grupo cliente e as transforma em apenas uma emissão de requisição ao servidor.

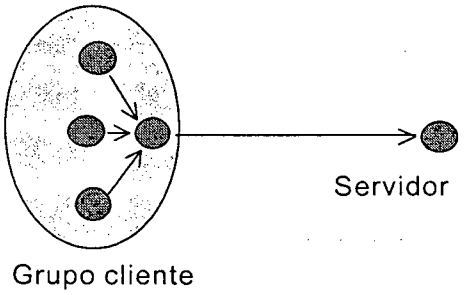


Figura 2.6. Grupo com um membro centralizador.

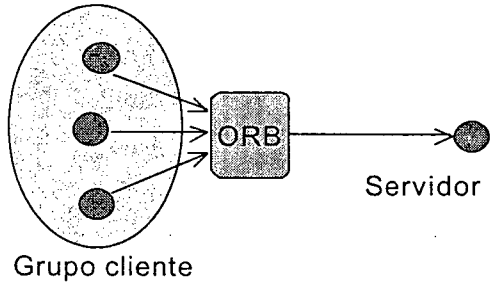


Figura 2.7. ORB responsável pela comunicação.

Estas formas de transparência nas comunicações podem ser bem suportadas utilizando mecanismos de *proxy* e *dispatcher* incluídos no ORB [Shapiro86, Hagsand92]. A figura 2.8 mostra um modelo de implementação para comunicação de grupo usando estes mecanismos. Os *stubs* e *proxies* do lado dos clientes e os *dispatchers* e *skeletons* do lado dos objetos no grupo servidor são gerados a partir da compilação de uma especificação de interface de um grupo servidor. O *stub* residente no espaço de endereçamento de cada membro cliente oferece uma interface de serviço idêntica à da ativação de um método de um objeto servidor simples local. O *proxy* é a interface local, no

lado cliente, do grupo de objetos servidor. A comunicação de um cliente com um grupo servidor é possível através de seu *proxy*. O *proxy* distribui uma requisição de serviço entre os objetos do grupo servidor usando os serviços de um protocolo de comunicação (item seguinte). Em cada sítio receptor, o *dispatcher* localiza o membro do grupo servidor e passa a requisição ao *skeleton* que, por sua vez, ativa o método.

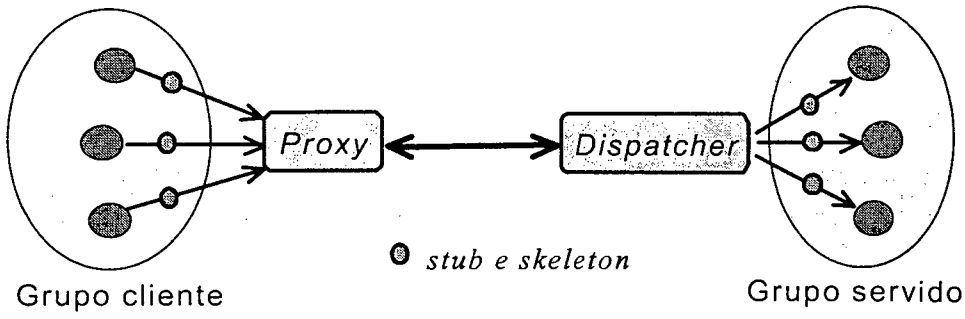


Figura 2.8. Comunicação no estilo *Proxy/Dispacher*.

2.3.2.2 Implementação das comunicações

A noção de grupo implica no uso de um identificador único para um conjunto de objetos que compartilham alguma característica em comum, de modo que o mesmo seja tratado por outros objetos da aplicação como um objeto simples. A aplicação não precisa se envolver na expansão dos endereços de grupo dentro das listas de destinos. Usando um serviço de protocolo *multicast*, de mais baixo nível, envolvendo primitivas de interação multi-ponto, em muito o desempenho das comunicações e a complexidade do ORB podem ser melhorados. Se este protocolo *multicast* garantir propriedade de acordo sobre as mensagens engajadas e fornecer diferentes propriedades de ordenação, o ORB, além da simplicidade envolvida, poderá suportar diferentes modelos de replicações. Três tipos de ordenação usualmente apresentados são ordem total, ordem causal e ordem FIFO. A propriedade de ordenação garante que todas as mensagens difundidas em um grupo serão entregues de acordo a uma propriedade de ordenação.

A implementação de mecanismos de ordenação de mensagem pode ser implementada segundo quatro abordagens: *ordenação baseada no histórico de mensagens*, *ordenação centralizada*, *commit multi-fases* e *baseada em relógios*, que são apresentadas em [Shrivastava92]. Os mecanismos de comunicação não devem ser visíveis ao cliente, e devem estar disponíveis tanto para as *stubs* de invocação estática como para as interfaces

de invocação dinâmica (DII), ou para o *proxy* gerado a partir da interface IDL do grupo através do ORB.

2.3.2.3 Descrição das interfaces de gr upo servidor

O conjunto de serviços que podem ser acessados pelos objetos clientes é especificado através da linguagem de definição de interface IDL. A IDL, então, serve para descrever o formato de cada chamada dos métodos oferecidos pelo grupo servidor e dos parâmetros necessários para efetuá-las. Em um arquivo de definição de interface de um grupo de objetos estão presentes todas as informações necessárias à geração do suporte necessário para que um cliente possa ativar um método do grupo. A IDL deve ser puramente declarativa, sem definições de variáveis ou estruturas algorítmicas. É desejável que a descrição de interface de grupo de objetos seja compatível com o modelo IDL do CORBA [OMG96] e suportada pelo ORB. Para isso, é necessário adicionar novos tipos de declarações para grupo de objetos ou considerar que qualquer implementação de objeto seja, por *default*, tratada como grupo de objetos, mesmo que seja um grupo contendo apenas um objeto membro. Assim, é possível que a IDL para grupo seja totalmente idêntica ao IDL do CORBA convencional, pois ao se compilar a IDL de grupo todas as abstrações de grupo seriam tratadas no nível de suporte, garantindo a total transparência da aplicação.

Os pontos levantados nesta seção de certa forma condicionam a inclusão ou o acesso a um suporte de grupo via ORB. Estes pontos influenciaram de forma mais acentuada a abordagem conhecida como *integração* que será apresentada a seguir.

2.4 As abordagens para suporte a grupo ou tolerância a falta no CORBA

Na literatura são encontradas diversas experiências no sentido da inclusão em *middleware* CORBA de mecanismos que suportam grupo de objetos e/ou tolerância a faltas. Essas propostas podem ser classificadas dentro de três abordagens distintas: integração (item 2.4.1), serviço (item 2.4.2) e de interceptação (item 2.4.3). Cada uma destas abordagens apresenta características que visam atender a requisitos relacionados a transparência, desempenho, conformidade com o padrão CORBA, flexibilidade e

facilidade de uso. Apresentaremos nos próximos itens uma descrição conceitual de cada uma destas abordagens seguidos dos exemplos de protótipos descritos na literatura.

2.4.1 Abordagem de integração

A abordagem de integração consiste na construção ou na modificação de um *middleware* CORBA existente, a fim de fazer com que um suporte de comunicação de grupo proprietário seja incorporado ao ORB, tornando-se parte integrante do mesmo. Nesta abordagem, o *middleware* CORBA é alterado para que as aplicações não possam distinguir objetos simples de grupos de objetos uma vez que as semânticas de invocação são as mesmas, deste modo, um alto grau de transparência é alcançado. A idéia principal nesta abordagem é que o processamento de grupo seja suportado por uma ferramenta de comunicação de grupo abaixo do núcleo do ORB. Todas as chamadas que envolvam comunicação ou gestão de grupo são repassadas pelo núcleo do ORB a este suporte de mais baixo nível. Esta abordagem já foi adotada em algumas plataformas CORBA existentes, como o Orbix+Isis (item 2.4.1.1) e o Electra (item 2.4.1.2). Para a implementação do Orbix+Isis, são utilizadas facilidades de gerenciamento de comunicação de grupo fornecidas pela ferramenta Isis [Birman91]. Já o Electra é estruturado de modo que possa ser executado sobre várias ferramentas. Atualmente, existem implementações do Electra sobre o Isis, Horus [Renesse95] e Muts [Renesse93].

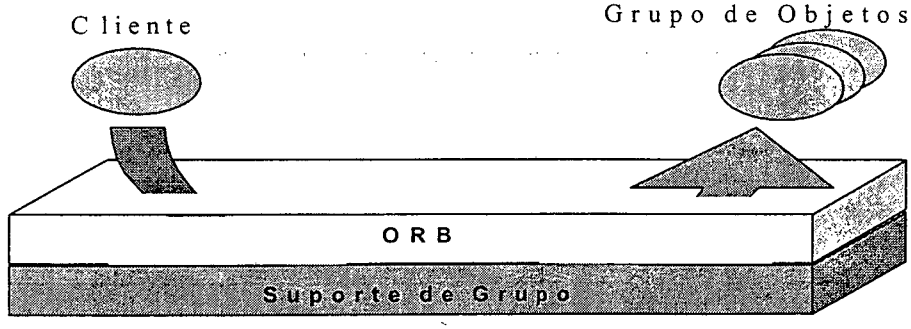


Figura 2.9. Execução de uma requisição na abordagem de integração.

Na abordagem de integração, as referências de objeto passam a poder identificar tanto um objeto único, como um grupo de objetos. O ORB é responsável por distinguir estas referências. A figura 2.9 ilustra a execução de uma requisição em um grupo de objetos. No lado do cliente, a requisição quando passada ao ORB, é reconhecida pelo mesmo como uma requisição endereçada a um grupo, sendo convertida em uma chamada

de *multicast* na ferramenta de mais baixo nível que dá suporte à comunicação de grupo. Em seguida, a operação adequada é invocada pelo ORB em cada membro do grupo de objetos servidores. Os resultados do processamento retornam pelo mesmo caminho, porém no sentido inverso. Finalmente, as repostas são coletadas no lado cliente, submetidas a alguma função de consenso e retornadas ao objeto que fez a requisição.

As implementações da abordagem de integração podem fazer uso de adaptadores de objetos disponíveis nos servidores que fornecem acesso às facilidades para a gestão de grupo. Esta possibilidade é permitida pela especificação CORBA. Todavia, nos *middlewares* Orbix+Isis e Electra, o próprio adaptador de objeto básico é estendido, permitindo que qualquer objeto em um sistema possa tornar-se membro de um grupo. Além disso, estes *middlewares* definem outras extensões ao padrão CORBA. No Orbix+Isis, as regras de mapeamento das estruturas definidas em IDL para a linguagem de implementação [OMG96] foram estendidas para gerar os códigos de gestão e comunicação de grupo. No Electra, as regras de mapeamento foram respeitadas, mas as classes de implementação do ORB foram estendidas para suportar as necessidades na gestão e comunicação de grupo.

2.4.1.1 Orbix+Isis

A ferramenta Orbix+Isis [IONA95] é um produto comercial desenvolvido em conjunto entre as empresas ISIS Distributed Systems Inc. e IONA Technologies, Ltd. É um suporte *middleware* que permite um sistema ser construído como um conjunto de objetos interagindo segundo o modelo CORBA/OMG. O Orbix+Isis simplifica o desenvolvimento e a integração de aplicações distribuídas tolerantes a faltas. Cada objeto tem uma interface bem definida e especificada em uma linguagem de definição de interface (IDL), compatível CORBA. O ORB neste caso é simplificado pelos recursos ISIS na implementação das abstrações de grupo. Mecanismos de *membership*, transferência de estado e comunicação *multicast* confiável com diferentes tipos de ordenação, fornecidos pelo Isis, são usados pelo ORB no suporte às aplicações Orbix+Isis.

Modelo de Grupo do ORBIX+ISIS

No modelo Orbix+Isis, os servidores formam um grupo de objetos C++ replicados ou associados, em que as interações cliente/servidor são concretizadas usando mecanismos de comunicação confiável. Grupos de objetos podem ser definidos em cima de dois estilos de execução: *Réplicas Ativas* e *Fluxo de Eventos* (*Event Stream*).

No estilo de *Replicação Ativa*, cada objeto membro compartilha a mesma interface e equivalente semântica de implementação, o que assegura que cada membro responde de maneira idêntica às mesmas requisições de serviço. O Orbix+Isis mantém uma classe base *Réplicas Ativas* para implementar servidores segundo este estilo de execução. Execuções no estilo *Réplicas Ativas* permitem três tipos de comunicações:

- *Multicast*: uma requisição é difundida para todos os membros de um grupo de objetos que executam e um único resultado é devolvido ao chamador, mantendo a transparência de servidor. O estilo *multicast* corresponde ao modelo de replicação Máquina de Estados. Por outro lado, o cliente pode ganhar acesso às respostas de todos os membros do grupo; para tanto é necessário que construa um *smart proxy* (objetos *proxies* fornecem, do lado do cliente, suporte para a comunicação de grupo). O *proxy* construído deve herdar o comportamento do *proxy default* e adicionar a capacidade de enviar todos os resultados ao cliente.

- *Client's Choice*: neste estilo as requisições são passadas a apenas um dos membros do grupo. Este tipo de comunicação tem o objetivo de aumentar o desempenho nas interações cliente/servidor. Foi criado apenas para operações *read-only*. Uma função *chooser* do lado do cliente determina o membro a receber a requisição.

- *Coordinator/Cohort*: neste estilo (item 2.5.1), o *coordenador* executa a operação e então envia os resultados ao cliente e a todos os *cohorts* para que estes usem os resultados para a atualização de seus estados. Uma função *chooser* determina o *coordenador* para o processamento da requisição do cliente. Se o coordenador falha a função *chooser* é ativada automaticamente para a escolha do novo *coordenador*.

Neste estilo, *Réplicas Ativas*, o Orbix+Isis oferece ainda serviços de *membership*, transferência de estado e ordenação de requisições. Transferência de estado e *membership* são serviços chave para diferentes modelos de processamento de grupo, porque define um controle sobre o número de membros livres de faltas e permite que novos objetos se juntem ao grupo, tornando-se réplicas exatas.

No estilo de execução "*Fluxo de Eventos*", os clientes usando apenas comunicações assíncronas *one-way* enviam mensagens (eventos) para objetos membros de um grupo, chamados de "*receptores de eventos*". Neste modelo, clientes enviam mensagens ao *event stream* que mantém estes eventos e direciona os mesmos aos objetos *subscribers* (receptores de evento). Os receptores de evento podem se juntar ou deixar o grupo de *subscribers* em qualquer momento. O estilo de execução *fluxo de eventos* desacopla os clientes dos servidores, pois os objetos clientes, que mandam mensagens para o *fluxo de eventos*, não se preocupam se os receptores de eventos estão ou não ativos.

Os dois estilos de execução, *Réplicas Ativas* e *fluxo de eventos*, têm suas configurações definidas em arquivos separados do código da aplicação no *Isis Repository* (item seguinte).

Arquitetura ORBIX+ISIS

A arquitetura de um sistema fazendo uso do Orbix+Isis está apresentada na figura 2.10. O Orbix+Isis consiste de biblioteca de classes C++ e um suporte de execução que implementa as funcionalidades dos estilos de execução de grupo (*processamento de grupo* e *fluxo de evento*). Nesta estrutura convivem um ORB convencional (Orbix) e o Orbix+Isis com suporte para grupo. A camada de comunicação Orbix trata de comunicações ponto a ponto. A camada de comunicação Orbix+Isis usa as facilidades Isis no suporte para grupo permitindo então que se ofereçam as bases para aplicações distribuídas tolerantes a faltas.

A linguagem IDL do Orbix+Isis estende a IDL/OMG para permitir a geração de códigos requeridos para a construção de aplicações tolerantes a faltas no Orbix+Isis. As interfaces IDL de grupo no Orbix+Isis geram na compilação as estruturas necessárias para um dos estilos de execução no grupo servidor. Usando estas estruturas (*stubs, proxy*, etc..), o cliente se conecta e se comunica com um grupo de objetos como se este fosse um objeto

Orbix simples. O repositório Isis (IsR) é uma hierarquia de arquivos do Orbix+Isis similar ao repositório de implementação do Orbix. Cada servidor Orbix+Isis tem seu arquivo no IsR, que define os procedimentos de ativação e de configuração do grupo de servidores. Os aspectos de desempenho e de estilo de execução do grupo são especificados no repositório Isis (IsR). As informações disponíveis nestes arquivos permitem, por exemplo, mudanças na aplicação envolvendo o estilo de execução de grupo, sem a necessidade de modificação em código ou de novas recompilações.

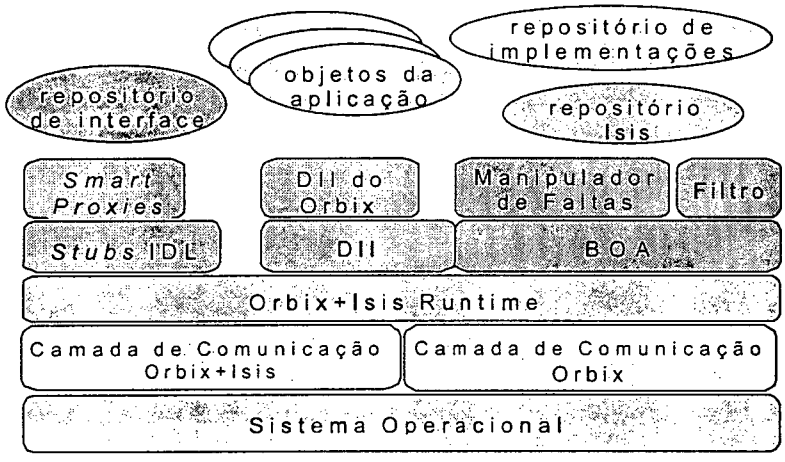


Figura 2.10. Arquitetura do Orbix+Isis.

2.4.1.2 Electra

O Electra [Maffeis95a, Maffeis95b] é um *Object Request Broker* (ORB) com algumas compatibilidades com o padrão CORBA [OMG96], cuja arquitetura suporta também mecanismos de tolerância a faltas para aplicações distribuídas, através do uso dos conceitos de grupo de objetos. Para o desenvolvimento de aplicações distribuídas, este modelo combina os benefícios do padrão CORBA com o poder de ferramentas de mais baixo nível, tais como: Isis [Birman91], Horus [Renesse95], Transis [Amir92], Consul [Mishra93], Chorus [Chorus92], entre outros. Na atual versão do Electra é possível torná-lo operacional sobre as plataformas de suporte Isis e Horus, mas, novos adaptadores podem ser desenvolvidos para outras ferramentas, o que o torna um sistema flexível.

O Electra é implementado em C++ e o mapeamento IDL para C++ está de acordo com o especificado pela OMG [OMG96]. A IDL do Electra é idêntica às especificações da OMG, pois o modelo considera as implementações de objeto como grupo de objetos.

Mecanismos de herança são possíveis desde que os membros do grupo sejam do mesmo tipo, instâncias da mesma interface ou pelo menos ter uma interface antecessora em comum, onde só as operações herdadas deste antecessor podem ser difundidas ao grupo.

Modelo de Grupo de Objetos

O modelo Electra, seguindo os padrões CORBA/OMG, permite que objetos possam ser implementados em diferentes linguagens de programação, sobre um número qualquer de máquinas. O Electra, como todo ORB, é o responsável pela implementação das semânticas de comunicação e independe das linguagens de programação ou das técnicas de implementação usadas na construção dos objetos. As comunicações no Electra podem se dar no estilo disseminação (*multicast*) confiável ou por comunicação *ponto-a-ponto*. O cliente faz uso de um mesmo modelo de invocação de método, independente se o servidor é um objeto simples ou um grupo. Estas invocações síncronas, assíncronas ou semi-síncronas[2] (*deferred-synchronously*), realizadas em interface estática ou dinâmica no estilo CORBA, podem servir para emissões de *multicasts*. Dois modos de comunicação de grupo são disponíveis no Electra:

- Transparente: um grupo é visto como um objeto simples e altamente disponível, ao qual as requisições são submetidas como num ORB convencional, ou seja, o cliente recebe só um resultado do grupo;
- Não-transparente: permite o acesso, em uma invocação, dos resultados de cada membro individual do grupo de objetos.

O Electra suporta grupos no modelo réplicas ativas, fazendo uso de ferramentas de mais baixo nível no fornecimento de serviços de *multicast* atômico e de gerenciamento de grupo (figura 2.11). A transferência de estado e o serviço *membership*, suportados por ferramentas como Isis, são disponíveis como operações Electra no *Basic Object Adapter* (BOA). Na interface do BOA do Electra foram ainda adicionados mecanismos para ativar um grupo de objetos e selecionar o protocolo de *multicast* a ser utilizado. A classe *environment* CORBA é utilizada para a seleção do estilo de invocação e principalmente, para passar exceções do servidor ao cliente.

[2] É assíncrona até o tempo *t*, após este período o chamador se bloqueia até o retorno do resultado (síncrona).

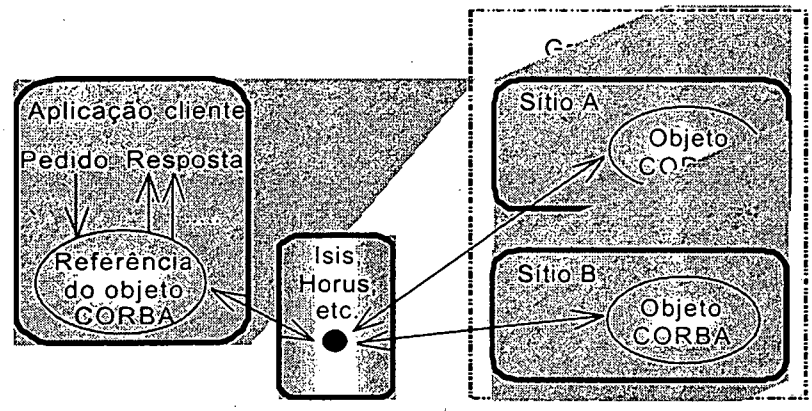


Figura 2.11. Comunicação de grupo no Electra.

Arquitetura ELECTRA

O Electra é um sistema flexível capaz de operar sobre vários tipos de ferramentas e sistemas operacionais que ofereçam algum suporte de grupo. A figura 2.12 mostra a arquitetura de um sistema fazendo uso do Electra.

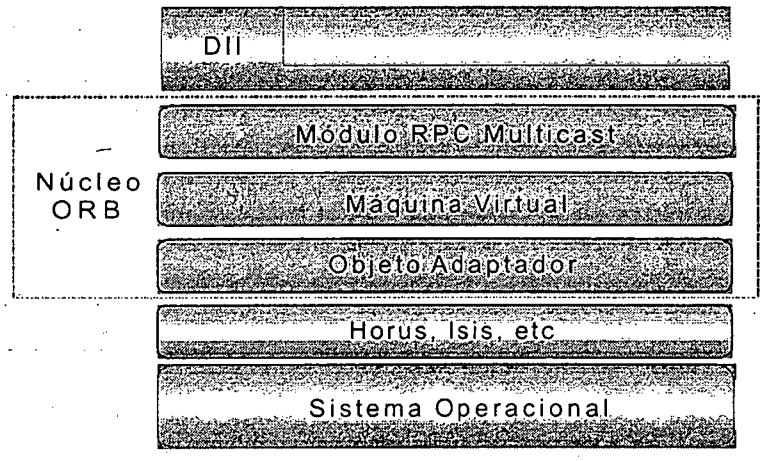


Figura 2.12. Arquitetura do Electra.

A interface de invocação estática (EII), a interface ORB (IORB), o adaptador de objeto básico (BOA) e a interface de invocação dinâmica (DII) têm as mesmas funções das interfaces CORBA apresentadas na figura 2.2. O núcleo do ORB está fundamentado sobre a interface de invocação dinâmica (DII) e o módulo RPC *multicast* que suporta RPC assíncrono para grupo ou objeto simples. A interface *máquina virtual* é inserida nesta arquitetura com o objetivo de oferecer portabilidade sobre as diversas plataformas básicas possíveis. A função desta interface é abstrair o modelo CORBA das funcionalidades tais

como, comunicação fim-a-fim, grupo de processos, *threads*, etc, oferecidas por estas plataformas de mais baixo nível. O objeto adaptador tem a função de mapear as operações da máquina virtual na interface da plataforma básica (API de baixo nível). O objeto adaptador é específico para a plataforma usada. Para que o Electra funcione em uma plataforma basta desenvolver um objeto adaptador e acrescentá-lo na arquitetura, sem a necessidade de fazer qualquer modificação aos códigos residentes acima dele.

2.4.2 Abordagem de serviço

A idéia básica desta abordagem consiste no provimento de suporte a grupo e tolerância a faltas na forma de objetos de serviço acima do ORB, e não como parte do próprio ORB. O ORB se abstrai de qualquer aspecto relacionado ao suporte para tolerância a faltas, deixando esta responsabilidade aos objetos de serviço de grupo, os quais tem sua interface definida de acordo com a IDL/CORBA. Dessa forma, por exemplo, o serviço de suporte a grupo deve ser formado por uma coleção de objetos de serviço, que podem estar localizados em diferentes estações da rede. Estes objetos devem ser responsáveis pelo gerenciamento e ainda, pela entrega de mensagens aos objetos membros do grupo, mantendo as propriedades definidas pelo tipo de comunicação de grupo utilizado. O envio de uma requisição segundo a abordagem de serviço é ilustrada na figura 2.13.

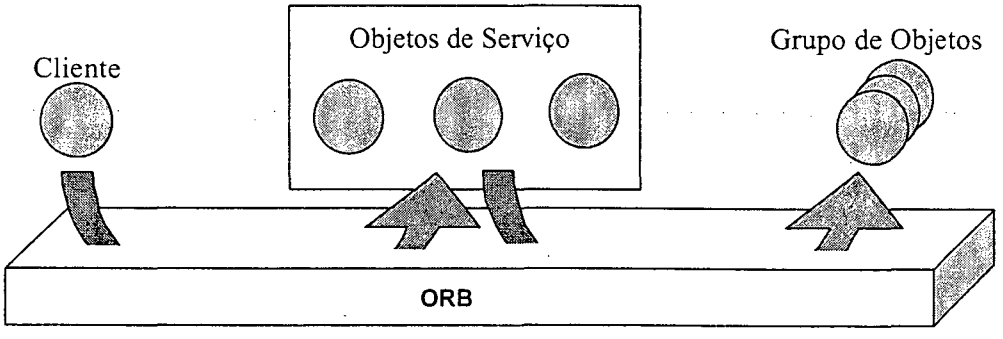


Figura 2.13. Execução de uma requisição na abordagem de serviço.

A abordagem de serviço é compatível com o padrão CORBA, já que segue a filosofia da OMG, ou seja, o ORB deve prover os mecanismos mínimos para quebrar as limitações das linguagens de programação e espaços de endereçamento, permitindo que objetos heterogêneos interoperem. Os objetos de serviço servem como *building blocks* para dar suporte ao desenvolvimento de diferentes aplicações. A plataforma básica de comunicação provida pelo CORBA, ou seja, o ORB, suporta apenas os mecanismos para a invocação de

métodos de objetos remotos. Funções mais específicas são adicionadas ao ORB na forma de objetos de serviço definidos através de especificações *COSS* (*Common Object Services Specification*) da OMG [OMG97]. Como resultado disso, os objetos de serviço de grupo podem ser reaproveitados por qualquer *middleware* que esteja em conformidade com o padrão CORBA.

Como exemplos da abordagem de serviço, pode-se citar o *Object Fault-tolerance Service* (OFS) [Liang98] e o *Object Group Service* (OGS) [Felber98]. O OFS oferece suporte a tolerância a faltas por meio de mecanismos de replicação. Todavia, não define serviços para comunicação em grupo. O OGS oferece suporte a aplicações confiáveis por meio de um serviço de grupo de objetos. Este serviço segue a filosofia da OMG, sendo implementado a partir de objetos de serviço CORBA que podem ser utilizados em outros contextos. Tanto o OFS quanto o OGS estão de acordo com o padrão CORBA. Neste contexto, estes *middlewares* não utilizam ferramentas de suporte a grupo proprietárias e as regras de mapeamento IDL são respeitadas.

2.4.2.1 OFS

A maioria dos *middlewares* CORBA para tolerância a faltas utiliza réplicas ativas. Todavia, varias pesquisas sugerem que as técnicas de replicação semi-ativa e passiva (anexo A) podem ser utilizadas em vários domínios de aplicação [Huang93]. A vantagem do uso dessas é que os protocolos de replicação são menos complexos, não requer, por exemplo, comunicação de grupo com ordenação total (*multicast* atômico). Neste contexto, o OFS é um serviço que fornece recursos para tolerância a faltas implementado sobre um *middleware* CORBA (Orbix 2.0) em uma plataforma Solaris. O OFS suporta técnicas de replicação semi-ativa e passiva. A implementação do OFS não requer nenhuma modificação no ORB ou no mapeamento da linguagem IDL. Além disso, não é assumido nenhum suporte proprietário para a comunicação de grupo.

O OFS oferece um serviço de gerenciamento de replicação baseado no protocolo apresentado em [Chen92]. Neste protocolo de replicação, um grupo é formado por objetos pertencentes a uma das três classes: uma réplica primária (ativa), réplicas passivas mornas

e réplicas passivas frias[3]. Diferente das demais, as *réplicas passivas frias* não são executadas no sistema. Por sua vez, as *réplicas passivas mornas*, também, não respondem às requisições dos clientes, mas recebem, periodicamente, o estado do primário. Finalmente, somente a réplica ativa (a primária) responde aos clientes com algum grau de sincronia. Caso ocorram falhas, réplicas passivas frias podem ser promovidas a passivas mornas; e réplicas passivas mornas podem ser promovidas a ativas. Esta abordagem mantém um alto nível de confiabilidade e disponibilidade, só apresentando uma degradação de desempenho em situações de falha. A seguir é apresentada a arquitetura do OFS, descrevendo-se seus componentes. Posteriormente é descrito em mais detalhes como ocorre o suporte a tolerância a faltas no OFS.

Arquitetura do OFS

Conforme ilustrado na figura 2.14, o OFS é formado por dois objetos: o Gerente de Replicação (*Replication Manager*) e o Anjo Guardião (*Guardian Angel*). Estes objetos devem estar presentes em cada sítio do sistema. O gerente de replicação é responsável por aceitar pedidos de registro de objetos de aplicação que requerem suporte a tolerância a faltas e manter um número suficiente de réplicas no sistema de acordo com o grau de confiabilidade desejado. Em particular, apenas dois destes gerentes (*Replication Manager*) funcionam como réplicas ativas e os demais funcionam como réplicas passivas mornas. O Anjo Guardião é um objeto interno, isto é, inacessível aos usuários do serviço, pois não possui interface IDL. As funções deste objeto incluem a monitoração dos objetos de aplicação, a detecção de falhas e a ativação de réplicas. Para tal, o Anjo Guardião periodicamente realiza comunicações com os objetos de aplicação e os gerentes de replicação ativos.

Na figura 2.15, temos um exemplo de um sistema com um objeto de aplicação, uma réplica passiva morna e uma réplica passiva fria, todos gerenciados pelo gerente de replicação OFS. Para integrar-se tolerância a faltas à aplicação, os *objetos de aplicação* devem registrar junto ao OFS uma lista de réplicas e o grau de replicação mínimo desejado (figura 2.15). Além disso, um *objeto de aplicação* deve ser capaz, com auxílio do OFS, de repassar seu estado interno para as réplicas passivas mornas. A interação entre objetos

[3] As réplicas ativas, passivas mornas e passivas frias são designadas na literatura original [Liang98] em

ativos, réplicas passivas mornas e o OFS é ilustrada na figura 2.15. As setas indicam que o objeto de origem tem uma referência possuída pelo objeto de destino. Quando o OFS é o destino de uma seta, a referência possuída pelo objeto identifica o gerente de replicação.

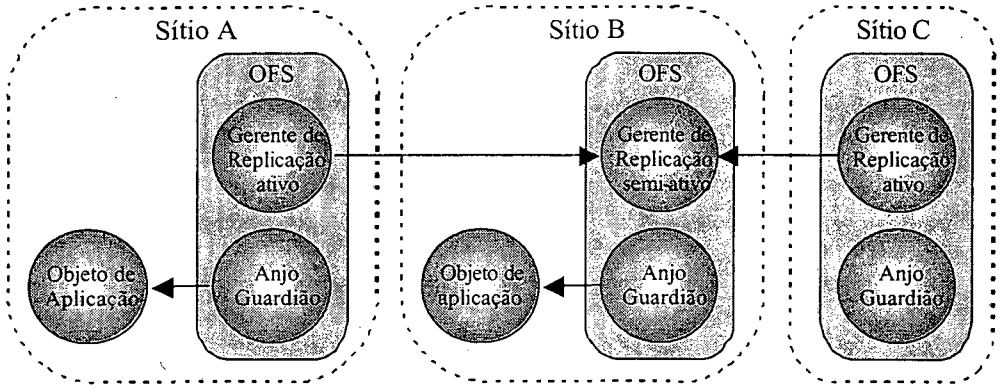


Figura 2.14. Arquitetura do OFS.

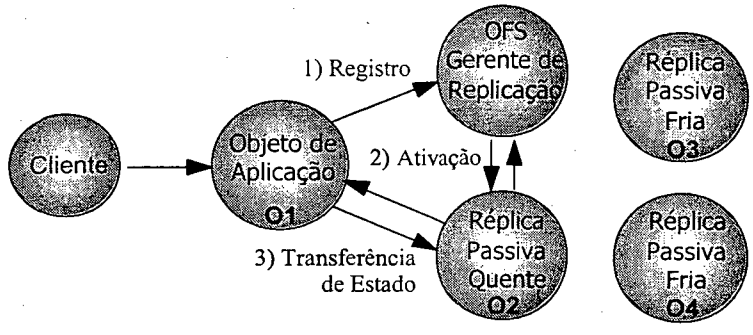


Figura 2.15. Arquitetura do OFS.

Supondo que um objeto de aplicação `O1` registre os objetos `O2`, `O3` e `O4` como suas réplicas e solicite um grau de tolerância a faltas 1. O OFS deve aleatoriamente escolher uma das réplicas e ativá-la. Desta forma, devem ser mantidas uma réplica passiva morna e duas réplicas frias. Neste sentido, o objeto de aplicação é considerado a réplica ativa. Após ser ativada, a réplica passiva morna pode requisitar o estado corrente à réplica ativa, a fim de sincronizar-se com esta. Além disso, periodicamente, a réplica ativa pode passar para a réplica passiva morna as requisições que recebeu para serem executadas.

Caso ocorra uma falha em um objeto de aplicação, o OFS deve perceber que existe apenas uma réplica passiva morna e promovê-la a réplica ativa. Além disso, o OFS deve escolher aleatoriamente uma das réplicas passivas frias e promovê-la a passiva morna. Esta

inglês como: "*hot backups*", "*warm backups*" e "*cold backups*", respectivamente.

nova réplica passiva morna deve sincronizar-se com a ativa, requisitando o seu estado. A recuperação de falhas em uma réplica passiva morna é similar ao apresentado.

2.4.2.2 OGS

O OGS é um ambiente orientado a objetos que permite a construção de aplicações distribuídas confiáveis sobre *middlewares* CORBA. O OGS não requer nenhuma mudança ou extensão ao padrão CORBA, podendo ser executado sobre qualquer ORB compatível com a especificação CORBA 2.0. Todavia, este ORB deve oferecer múltiplos fluxos de execução ou *threads*. Esta facilidade é utilizada para implementar comunicações assíncronas. O OGS é construído a partir de vários outros serviços CORBA. Todavia, nenhum destes outros serviços é exclusivo para o suporte a grupo, podendo ser utilizados em outros contextos.

A figura 2.16 ilustra os serviços utilizados na implementação do OGS. O serviço de mensagens permite realizar comunicações assíncronas ponto a ponto. O serviço de difusão provê mecanismos para a difusão de mensagens. O serviço de monitoramento é responsável pela detecção de falhas nos objetos envolvidos em comunicações de grupo. O serviço de consenso garante que mensagens difundidas por vários clientes sejam recebidas segundo um mesmo tipo de ordenação nos vários membros de grupo. Finalmente, o serviço de grupo de objetos, ou seja, o OGS propriamente dito, oferece mecanismos para a interação entre clientes e grupos de objetos.

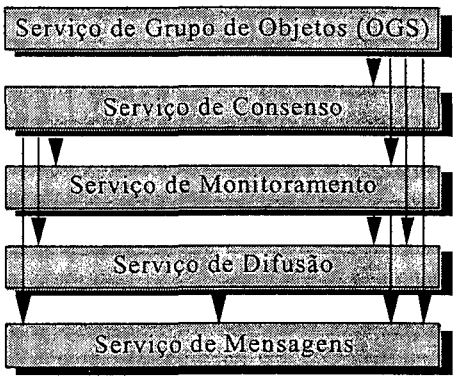


Figura 2.16. Serviços utilizados pelo OGS.

A seguir é discutida a arquitetura do OGS. São apresentados os objetos que compõem o OGS, descrevendo em detalhes a interação entre estes objetos. Posteriormente, são discutidos os tipos de comunicação suportados no OGS.

Arquitetura do OGS

O OGS é formado por uma coleção de interfaces e objetos que definem e implementam as operações necessárias para a gestão e comunicação de grupo. Em especial, as interações entre clientes e servidores são intermediadas por objetos *proxies*. A figura 2.17 ilustra a utilização de *proxies* na difusão de uma mensagem. Os *proxies* nos sítios clientes e servidores têm funções distintas. Um *proxy* cliente além de fornecer mecanismos para o *multicast*, também oferece meios para que um objeto cliente possa obter informações sobre a composição do grupo com o qual está se comunicando. Com tais informações, o cliente pode enviar mensagens ponto a ponto para determinados membros do grupo utilizando os mecanismos normais de invocação ponto a ponto do CORBA. Esta facilidade pode ser útil para algumas aplicações.

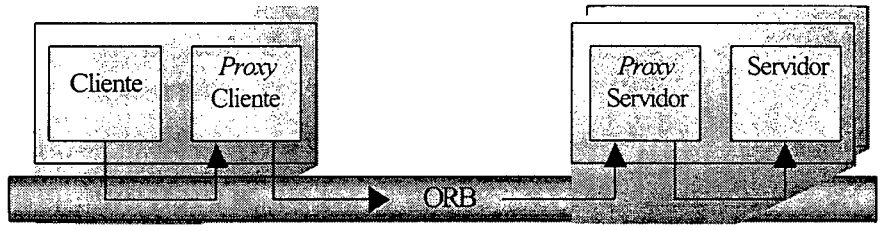


Figura 2.17. Difusão de mensagem no OGS.

As funcionalidades dos *proxies* servidores também não estão restritas a apenas a entrega de mensagens de difusão aos membros de grupo. Para juntarem-se ou deixarem um grupo, os objetos servidores invocam operações em um *proxy* servidor. Estas operações provocam mudanças da pertinência do grupo que são notificadas pelos *proxies* servidores a todos os membros do grupo. Além disso, quando um novo membro junta-se ao grupo, os *proxies* servidores iniciam o protocolo de transferência de estado. Para tal, os *proxies* servidores invocam operações de obtenção e atualização de estado. Neste contexto, todos os membros de grupo devem ser derivados de uma mesma interface que define as operações invocadas pelos *proxies* servidores.

Um *proxy* permite o acesso a um único grupo. Ou seja, um servidor necessita de um *proxy* para cada grupo de qual deseja fazer parte. Similarmente, um cliente precisa de um *proxy* distinto para cada grupo com o qual deseja comunicar-se. Neste contexto, a quantidade e a natureza dos *proxies* ativos, em um dado momento no sistema, dependem das aplicações sendo correntemente executadas. Desta forma, a princípio os *proxies* clientes e servidores não estão disponíveis para o uso. Estes *proxies* são criados a medida que são necessários. A criação de *proxies* é realizada por meio de objetos fábrica. Os objetos fábrica estão disponíveis na inicialização do sistema e podem criar *proxies* clientes e servidores para qualquer grupo.

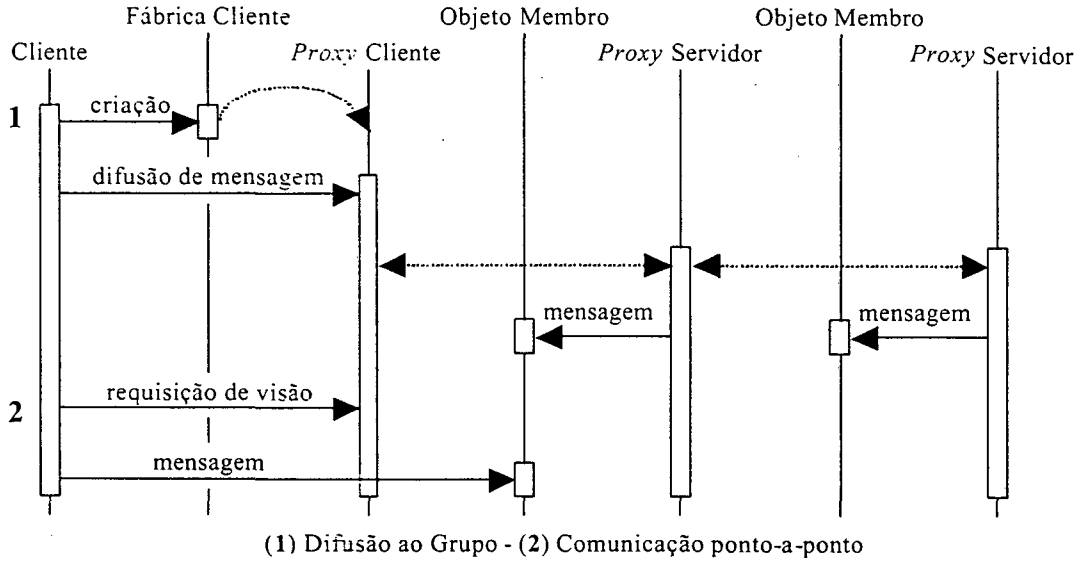


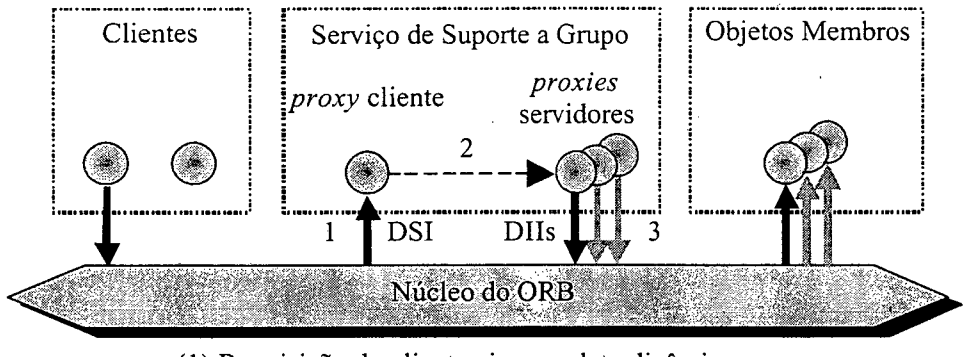
Figura 2.18. Interação entre um cliente e objetos membros de grupo

A figura 2.18 ilustra interações entre os objetos envolvidos no OGS. Uma caixa representa uma operação ativa no objeto cujo tipo é indicado no topo de cada linha vertical. O tempo flui de cima para baixo. As setas planas representam invocações, as setas em arco representam a criação de novos objetos e as setas com sentido duplo representam mensagens específicas do protocolo de serviço. A figura 2.18 expõe a interação entre um cliente e um grupo. São apresentados dois tipos de interação: uma mensagem de *multicast* para o grupo e uma comunicação ponto a ponto com um dos membros. Assume-se que o grupo, do exemplo da figura, contenha dois membros. Para executar a comunicação, o cliente primeiro requisita a criação de um *proxy* cliente. Então, o *multicast* é executado utilizando-se o objeto *proxy* cliente. Para comunicações ponto a ponto, o cliente adquire

junto a seu *proxy* referências para os objetos membros de grupo. O cliente envia uma mensagem a um dos membros, utilizando mecanismos de comunicação ponto a ponto do CORBA.

Tipos de Comunicação no OGS

O OGS permite que a interação entre clientes e membros de grupo ocorra por meio de comunicação tipificada ou não-tipificada. A comunicação não-tipificada permite que os clientes enviem apenas valores do tipo `any`[4] como mensagens. Enquanto que a comunicação não-tipificada é útil e mais eficiente em algumas situações específicas, geralmente é mais conveniente para os clientes invocarem diretamente uma operação da interface do servidor, isto é, utilizar comunicação tipificada. A comunicação tipificada é um aspecto importante do OGS, já que provê transparência de grupo para os clientes e servidores. Neste contexto, o cliente pode realizar invocações a um grupo de objetos como se estivesse invocando um servidor único. Além disso, os objetos servidores também não podem diferenciar as comunicações de grupo de mensagens comuns.



(1) Requisição do cliente via esqueleto dinâmico
(2) Chamada de difusão aos sítios servidores
(3) Invocação das operações via interfaces dinâmicas

Figura 2.19. Implementação da comunicação tipificada[5].

A comunicação tipificada é obtida por meio da utilização de duas características do padrão CORBA: os esqueletos IDL dinâmicos e as interfaces de invocação dinâmica. Os esqueletos dinâmicos permitem que os *proxies* clientes recebam requisições de clientes baseadas na interface IDL do servidor, mesmo que esta interface não seja conhecida em

[4] O tipo *any*, definido pela especificação CORBA, pode suportar valores de quais quer outros tipos [OMG96].

tempo de compilação. Então, estes *proxies* traduzem o conteúdo das requisições para um formato predefinido e realizam uma difusão para os sítios servidores. Nestes sítios, os *proxies* servidores invocam as operações adequadas nos objetos membros por meio de interfaces de invocação dinâmica. Qualquer resultado da operação é retornado para o cliente utilizando comunicação ponto a ponto. O funcionamento da comunicação tipificada é ilustrado na figura 2.19. Embora nesta figura os sítios clientes e servidores não tenham sido identificados, é importante notar que os objetos clientes e membros de grupo podem estar sendo executados em processadores diferentes e até mesmo ORBs distintos.

2.4.3 Abordagem de interceptação

A abordagem de interceptação prevê que as mensagens enviadas aos objetos servidores devam ser capturadas do ORB e redirecionadas para um sistema de comunicação de grupo separado do *middleware* CORBA. Uma vez que as mensagens são capturadas do ORB, esta abordagem garante a transparência das funcionalidades de grupo às aplicações, e não requer a modificação do *middleware* CORBA utilizado. A concretização dessa captura pode ser realizada em nível de aplicação, interfaces do sistema operacional ou mesmo mecanismos do próprio CORBA, conhecidos como interceptadores. Exemplos de sistemas que utilizam interceptação em nível de aplicação são encontrados em MetaFT [Fraga97, Lau97, Lau99c] e FT-MOP [Killijian98]. O sistema Eternal [Naras97] é um exemplo de interceptação implementado usando os mecanismos do sistema operacional. Finalmente, em Phoinix [Liang96] e GroupPac[6] [Oliveira99a, Lau00a] temos interceptação implementada usando mecanismos do ORB.

2.4.3.1 Interceptação em nível de aplicação

O uso de estruturas de linguagens de programação para a implementação de mecanismos de interceptação em nível de aplicação (figura 2.20) é uma técnica bastante conhecida na literatura [Joshi97, Fabre95, Chiba93]. A idéia consiste em desenvolver estruturas na linguagem de programação que permitam o desvio de uma invocação direcionada a um objeto para um outro. Esse desvio pode ser entendido como a

[5] A sigla DSI vêm da expressão em inglês: " *Dynamic Skeleton Interface*", sendo que referencia os esqueletos dinâmicos.
[6] O GroupPac será apresentado no próximo capítulo.

interceptação de uma requisição. Essa estrutura de linguagem pode ser implementada como objetos *proxies* [Shapiro86], mecanismos de filtros [Joshi97] ou protocolos meta-objetos [Kiczales91].

Nesta abordagem, o uso do suporte de grupo é por fora do ORB, a partir de estruturas de linguagens de programação que definem o desvio de uma requisição para um serviço de comunicação de grupo. O protocolo meta-objeto [Kiczales91] – uma forma de implementar a "reflexão computacional" [Maes87] – é um modelo de programação em que o sistema atua sobre o seu próprio comportamento. Aplicado à programação orientada a objetos o paradigma da reflexão computacional, através da abordagem de meta-objetos, estrutura os objetos em dois níveis: nível base (*objeto-base*) e nível meta (*meta-objeto*). Nesta separação, o código funcional, ou nível base, é envolvido com a algorítmica da aplicação. O código não-funcional (ou nível meta) é responsável pela execução das políticas de controle que atuam sobre o código da aplicação em tempo de execução. Técnicas de tolerância a faltas têm também sido implementadas segundo este modelo. Em [Fabre95], são apresentadas discussões sobre o uso da reflexão computacional para implementar modelos de replicação em sistemas distribuídos. No MetaFT[7] [Fraga97, Lau99c] e FT-MOP [Killijian98] esta experiência é estendida para ambientes CORBA.

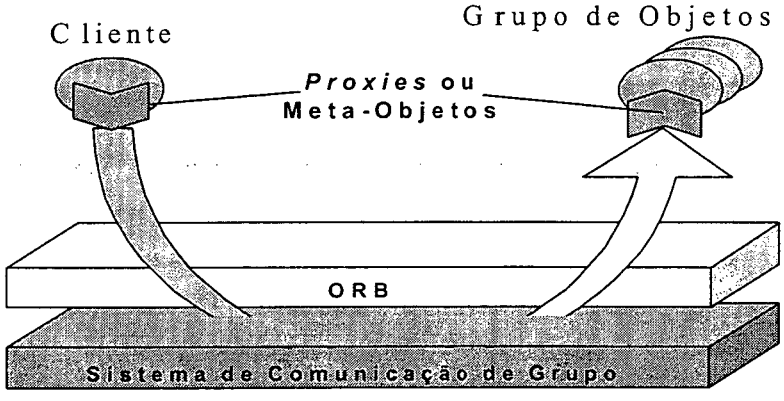


Figura 2.20. Execução de uma requisição na abordagem de interceptação.

2.4.3.1.1 FT-MOP

O FT-MOP (*Fault Tolerant – Meta Object Protocol*) é uma experiência baseada em protocolos meta-objetos para implementar aplicações tolerantes a faltas no CORBA. Os

[7] O MetaFT será apresentado no próximo capítulo.

meta-objetos do modelo FT-MOP implementam tolerância a faltas usando diferentes estratégias de replicação em sistemas distribuídos. Esses meta-objetos são responsáveis por:

- Gestão de membros: gerenciar a inserção ou remoção, normal ou por falha, das réplicas em um grupo de objetos;
- *Multicast* confiável: difusão de uma requisição em um grupo replicado com garantias de acordo e ordenação;

A arquitetura do FT-MOP consiste em associar para cada objeto CORBA da aplicação (objetos base O_1 e O_2 da figura 2.21) um meta-objeto (MO_1 e MO_2 da figura 2.21). Os meta-objetos são representados como objetos CORBA, com uma interface IDL, e implementam as estratégias de tolerância a faltas definidas para a aplicação. A associação da interface do meta-objeto com o objeto-base é alcançada através de extensões no compilador IDL/CORBA. Isto é, para cada interface de um objeto-base (objeto da aplicação) o compilador IDL gera automaticamente a interface do meta-objeto correspondente. Desta forma, o programador implementa a aplicação nos objetos-base e os mecanismos de tolerância a faltas nos meta-objetos correspondentes.

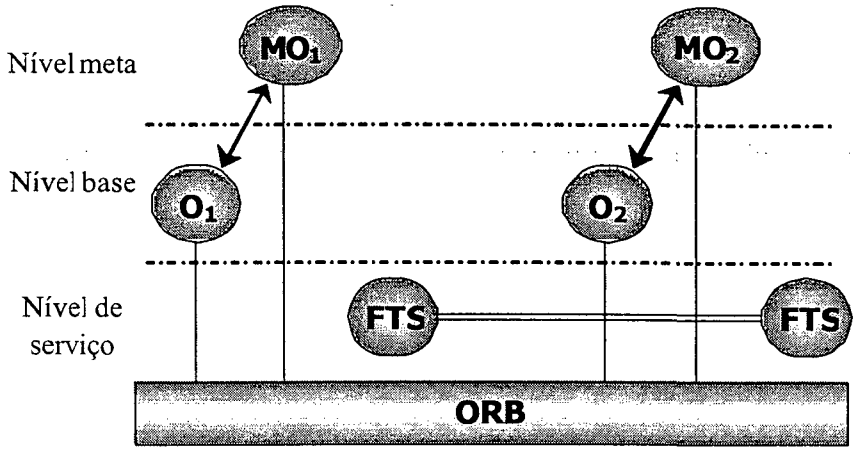


Figura 2.21. Arquitetura do FT-MOP.

Neste modelo deve estar disponibilizado um conjunto de objetos de serviço para suporte a tolerância a faltas (FTS da figura 2.21), tais como: gerenciamento de replicação, detecção de falhas, objetos fábricas e serviço de comunicação de grupo (ainda não especificado). O FT-MOP intercepta as mensagens e interage com estes serviços para

implementar as estratégias de tolerância a faltas. Conforme descrito em [Killijian98], as decisões de projeto em relação ao meta-nível para tolerância a faltas e o uso deste como MOP para implementar os mecanismos ainda estão sendo estudadas.

2.4.3.2 Interceptação usando interfaces do sistema operacional

A utilização de interfaces do sistema operacional para implementar mecanismos de interceptação é explorada no trabalho realizado no sistema Eternal [Naras97, Moser98]. Esta técnica se baseia no conceito de que é possível estender as funcionalidades de um sistema operacional, em nível de usuário, sem a necessidade de modificação do *kernel* ou das bibliotecas padrão do sistema. A técnica utilizada no sistema Eternal consiste na interceptação das chamadas de sistema realizadas pelo ORB antes que estas alcancem o *kernel* do sistema operacional. A chamada de sistema interceptada é então modificada para implementar a funcionalidade desejada – o desvio de uma requisição, que antes seria transportada pela camada TCP/IP, para um sistema de comunicação de grupo (figura 2.22). Esta interceptação é completamente transparente, tanto o ORB quanto o sistema operacional não percebem a interceptação. Esta solução é dependente das funcionalidades oferecidas pelo sistema operacional Unix, portanto apresenta problemas de interoperabilidade.

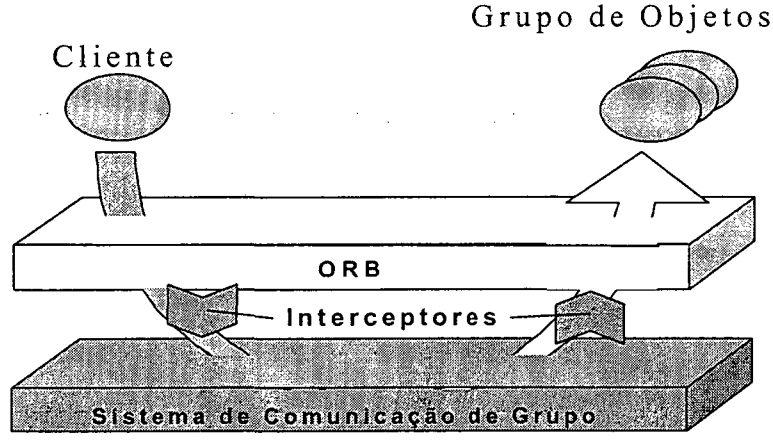


Figura 2.22. Execução de uma requisição na abordagem de interceptação em nível de S.O.

2.4.3.2.1 Eternal

O ambiente Eternal [Naras97, Moser98] aprimora qualquer implementação da especificação CORBA 2.0, provendo tolerância a faltas às aplicações sem que seja

necessária nenhuma modificação no ORB. Para prover a tolerância a faltas, o Eternal replica objetos em diferentes sítios do sistema distribuído e utiliza facilidades providas pela ferramenta Totem – um sistema para comunicação de grupo que provê mecanismos de *multicast* atômico. A vantagem do Eternal é que as operações de grupo são transparentes tanto para os objetos CORBA como para o próprio ORB. Esta funcionalidade é implementada inteiramente em nível de processos de aplicação. Os objetos de aplicação e o ORB também não precisam ser recompilados para tirar vantagem da capacidade de interceptação. Todavia, a implementação do Eternal depende de mecanismos fornecidos pelo sistema Unix.

O Eternal provê dois esquemas para a replicação de objetos: replicação ativa e passiva. Na replicação ativa, as operações são executadas por cada réplica dos objetos, requerendo a detecção e supressão de resultados duplicados. Na replicação passiva, apenas a réplica primária executa cada operação, requerendo mecanismos adicionais para garantir a consistência entre as réplicas primárias e passivas. Os objetos replicados no Eternal podem ser construídos hierarquicamente, isto é, serem compostos por outros objetos. Neste contexto, são providos mecanismos para manipular operações aninhadas. Além disso, é permitido a replicação de objetos clientes e servidores. Este modelo do Eternal permite o desenvolvimento de aplicações distribuídas como se elas fossem executadas em um único processador, mantendo a transparência da replicação e da distribuição em nível de aplicação.

Arquitetura do Eternal

O ambiente Eternal consiste em um sistema de suporte a grupo localizado entre o ORB e o sistema operacional. Este ambiente pode capturar de forma transparente uma coleção de chamadas de sistema feitas usando o ORB durante a execução do objeto. O formato das requisições interceptadas, quando das chamadas de sistema, são definidos pelos protocolos GIOP/IIOP. Depois de capturadas nas chamadas de sistema, as requisições (mensagens) são manipuladas e mapeadas no Totem.

A estrutura do Eternal é ilustrada na figura 2.23 [Naras97]. Os clientes realizam chamadas via ORB, que, por sua vez, utiliza o protocolo IIOP. O interceptador captura as chamadas IIOP, que são originalmente dirigidas ao TCP/IP, e as repassa ao gerente de

replicação. Então, este gerente é responsável por difundir as mensagens por meio do Totem. No lado servidor, cada réplica é invocada e os resultados são emitidos ao cliente pelo caminho inverso.

O gerente de recursos do Eternal, ilustrado na figura 2.23, cria as réplicas de um objeto e as distribui ao longo da rede de acordo com o nível de replicação desejado. O gerente de replicação emite as operações, em *multicast* para as réplicas, mantém a consistência entre estas, detecta e recupera falhas. Finalmente, o gerente de evolução explora a replicação para suportar atualizações de componentes de *hardware* e *software* no sistema.

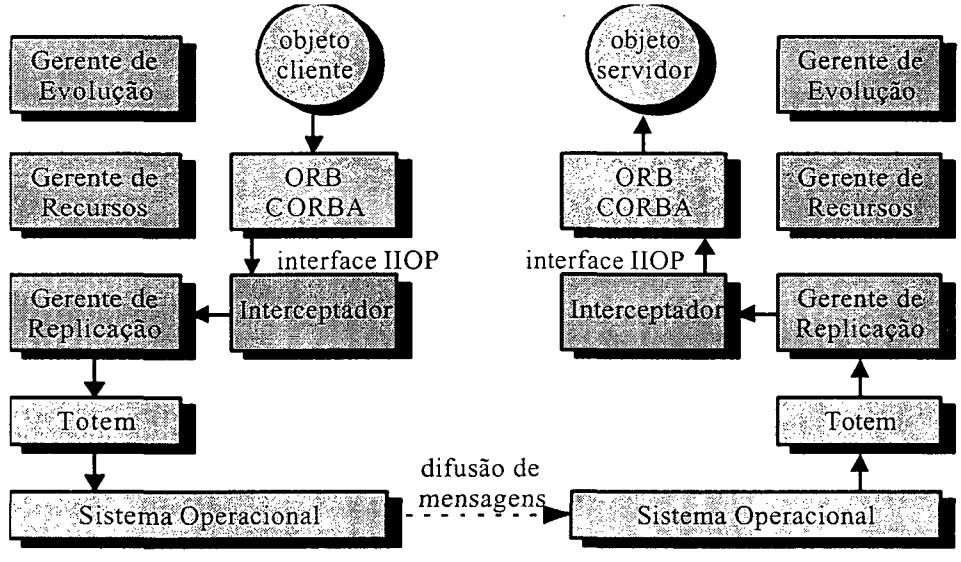


Figura 2.23. Arquitetura do Eternal.

Interceptação no Eternal

Um objeto CORBA típico invoca muitas chamadas de sistema durante seu tempo de vida. Isto inclui chamadas para requisição de memória, acesso a bibliotecas, operações de arquivo e operações de rede. Todavia, apenas uma pequena coleção destas chamadas necessita ser interceptada e manipulada pelo Eternal. As chamadas capturadas devem envolver a comunicação com outros objetos, que podem estar presentes remota ou localmente. Tais chamadas são invocadas pelo ORB, segundo o protocolo IIOP, para estabelecer conexões entre objetos e manter a interação entre estes.

open (descritor)
close (descritor)
read (descritor, <buffer de leitura>)
write (descritor, <buffer de escrita>)
poll (<lista de descritores>)

Figura 2.24. Chamadas de sistema interceptadas pelo Eternal[8].

Na figura 2.24 são ilustradas as chamadas de sistema interceptadas pelo Eternal. Além de serem utilizadas pelo protocolo IIOP, estas chamadas também são invocadas para realizar operações ordinárias do sistema operacional, devendo, neste caso, serem desconsideradas pelo interceptador. A chamada *open()* é utilizada pelo protocolo IIOP para estabelecer conexões TCP/IP. A chamada *poll()* é utilizada para associar dois objetos em uma conexão. As chamadas *read()* e *write()* são utilizadas na recepção e envio de dados. Finalmente, a chamada *close()* é utilizada para encerrar uma conexão. Todas estas chamadas recebem como parâmetro um descritor de conexão[9]. Este descritor é retornado pela chamada *open()*, sendo também utilizado para identificar quais chamadas de sistema são invocadas pelo protocolo IIOP.

2.4.3.3 Interceptação usando mecanismos do ORB

A inclusão de suporte de grupo no CORBA usando mecanismos do ORB tem seguido dois caminhos: extensão do compilador IDL/CORBA para geração de *proxies* específicos para comunicação de grupo [Liang96] e o uso de interceptadores como mecanismo oferecido pelo ORB. O Phoinix [Liang96] permite o desenvolvimento de aplicações confiáveis sobre *middlewares* CORBA por meio de funcionalidades ativadas a partir de mecanismos de interceptação. Os mecanismos de interceptação no Phoinix são produzidos pela extensão do compilador IDL/CORBA, gerando *proxies* e extensões de *skeletons* específicos para a comunicação de grupo [Liang96]. Essas extensões descaracterizam as soluções do Phoinix como solução aberta dentro das especificações CORBA, dificultando, portanto, a portabilidade do mesmo.

[8] Somente os argumentos relevantes das chamadas de sistema são mostrados.
[9] No sistema *Unix*, o descritor de uma conexão tem o mesmo formato e pode receber o mesmo tratamento de um descritor de arquivo.

O conceito de interceptador como mecanismo do ORB foi introduzido inicialmente nas especificações do serviço de segurança do CORBA [OMG00d]. Atualmente, existe uma especificação da OMG, denominada *Portable Interceptor* [OMG98b], com o objetivo da generalização deste mecanismo. Logicamente, um interceptador é interposto no caminho de invocação ou resposta entre um cliente e um objeto alvo, sendo responsável pela ativação transparente de controles ou processamento especiais às quais estariam sujeitas invocações normais no ambiente CORBA [Oliveira99a, Lau00a]. A diferença desta técnica em relação às duas anteriores é que esta é completamente compatível ao padrão da OMG. No suporte a grupo, interceptadores capturam uma requisição e a redirecionam a um sistema de comunicação de grupo (figura 2.25).

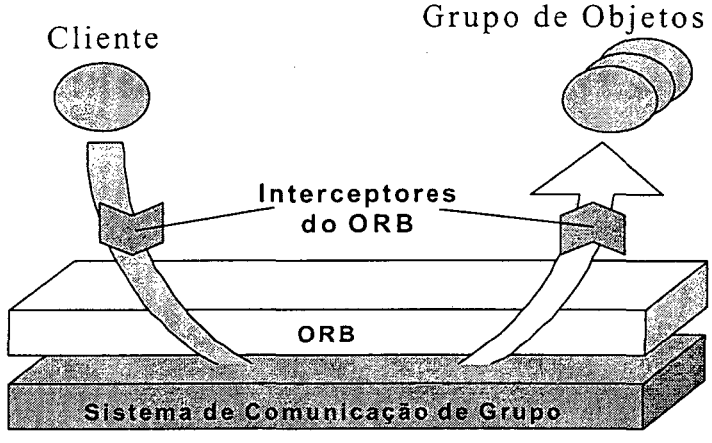


Figura 2.25. Execução de uma requisição na abordagem de interceptação.

2.4.3.3.1 Phoinix

O Phoinix [Liang96] é um ambiente de desenvolvimento de aplicações distribuídas baseadas em CORBA, no qual as implementações de objetos são construídas tolerantes a faltas de forma semi-automática. Em particular, é definida uma estratificação de três níveis de serviços de tolerância a faltas no Phoinix, os serviços de reinicialização (*restart service*), recuperação por retrocesso (*rollback-recovery service*) e replicação ativa. Sendo que a capacidade de tolerância a faltas cresce à medida em que o nível aumenta nesta estratificação. Atualmente, apenas os dois primeiros níveis de tolerância a faltas do Phoinix estão implementados, isto é, os serviços de reinicialização e recuperação por retrocesso. A versão corrente do Phoinix foi desenvolvida sobre a ferramenta Orbix 3.0 [IONA95] e o sistema operacional SunOS 4.2. Neste contexto, o Phoinix pode ser facilmente portado

para outras plataformas, já que o Orbix consiste em uma implementação completa do CORBA.

Se um objeto é chamado tolerante a faltas, o mesmo deve ser construído com os níveis definidos na estratificação. As implementações de objeto dos primeiro, segundo e terceiro níveis de tolerância a faltas do Phoinix são chamadas respectivamente objetos reinicializáveis (*restart object*), objetos registráveis (*logable objects*) e objetos replicados (*replicated objects*). Os objetos reinicializáveis, como sugere o nome, continuam as suas execuções como novos servidores depois da recuperação de um erro. Desta forma, contextos de execução anteriores ao erro são perdidos. Os objetos registráveis continuam seus serviços, após a recuperação de um erro, a partir do último ponto de recuperação já estabelecido pela execução do serviço. No caso dos objetos replicados, várias réplicas de uma mesma implementação de objeto coexistem no sistema com alguma forma de cooperação, mantendo o serviço mesmo na presença de réplicas faltosas.

O Phoinix foi projetado assumindo-se algumas hipóteses em seu ambiente de operação. Primeiramente, os sítios falhos devem operar segundo um modelo de falha controlada[10]. Neste contexto, as falhas no sistema distribuído são relativamente infreqüentes e independentes uma das outras. Além disso, é assumido que as invocações dos cliente às implementações de objetos não são aninhadas. Isto é, uma implementação de objeto invocada não pode requisitar serviços de outras implementações de objetos. Esta hipótese garante que uma implementação de objeto não possa ser suspensa devido a falhas do cliente. Em outras palavras, a detecção de falhas no cliente se torna desnecessária.

Os processos de detecção e recuperação de faltas no Phoinix são realizadas por códigos de tolerância a faltas gerados a partir de um compilador IDL proprietário. No lado cliente, *proxies* confiáveis interceptam e analisam exceções geradas pelo ORB, ativando os mecanismos de recuperação de erros quando necessário. No lado servidor, esqueletos tolerantes a faltas registram as invocações recebidas e o estado corrente das implementações de objeto. Neste contexto, a interceptação é realizada de forma transparente tanto para os objetos clientes como servidores.

[10] O modelo de falha controlada usado é o "*fail-stop model*".

A seguir é detalhada a arquitetura do Phoinix e discutido o desenvolvimento de aplicações tolerantes a faltas neste ambiente. Posteriormente, são apresentadas a detecção e recuperação de erros por meio dos serviços de reinicialização e recuperação por retrocesso do Phoinix.

Arquitetura do Phoinix

O desenvolvimento de uma aplicação tolerante a faltas no Phoinix é ilustrado na figura 2.26. Como pode ser observado, este modelo é semelhante ao desenvolvimento de uma aplicação convencional em um *middleware* CORBA. Para declarar um objeto tolerante a faltas, as palavras chaves `Restart` e `Logable` devem ser incluídas na especificação IDL de objetos reinicializáveis ou registráveis, respectivamente. Então, o compilador EIDL[11] analisa o arquivo de especificações de interfaces e produz códigos de tolerância a faltas em adição aos *stubs* clientes e esqueletos de implementação produzidos pelos compiladores IDL padrões. Para o lado cliente, o compilador EIDL produz um *proxy* confiável para cada interface IDL. No lado das implementações do objeto servidor, o compilador EIDL gera esqueletos tolerantes a faltas. A biblioteca de tolerância a faltas define as classes bases dos esqueletos tolerantes a faltas e objetos registráveis.

A figura 2.27 ilustra a arquitetura de um sistema que faz uso do Phoinix. No topo desta arquitetura encontram-se os objetos clientes e as implementações de objetos. Abaixo destes objetos encontra-se uma camada de tolerância a faltas, que consiste na ferramenta Phoinix propriamente dita. Além dos *proxies* confiáveis, esqueletos tolerantes a faltas e a biblioteca de tolerância a faltas apresentados anteriormente, estão presentes nesta camada um servidor de registros[12] e um gerente de replicação. O servidor de registros trabalha em conjunto com os esqueletos tolerantes a faltas para permitir a correta operação dos objetos registráveis. O gerente de replicação, ainda não está presente na versão atual do Phoinix, todavia deve ser responsável por coordenar as decisões entre as réplicas de uma implementação de objeto durante as requisições de clientes. Finalmente, a última camada da arquitetura Phoinix consiste no ORB.

[11] A sigla EIDL vem da expressão em inglês "*enhanced IDL*" e refere-se a uma especialização da linguagem IDL que permite definir objetos tolerantes a faltas [Liang96].
[12] O servidor de registro é referenciado em inglês na literatura original [Liang96] como "*log server*".

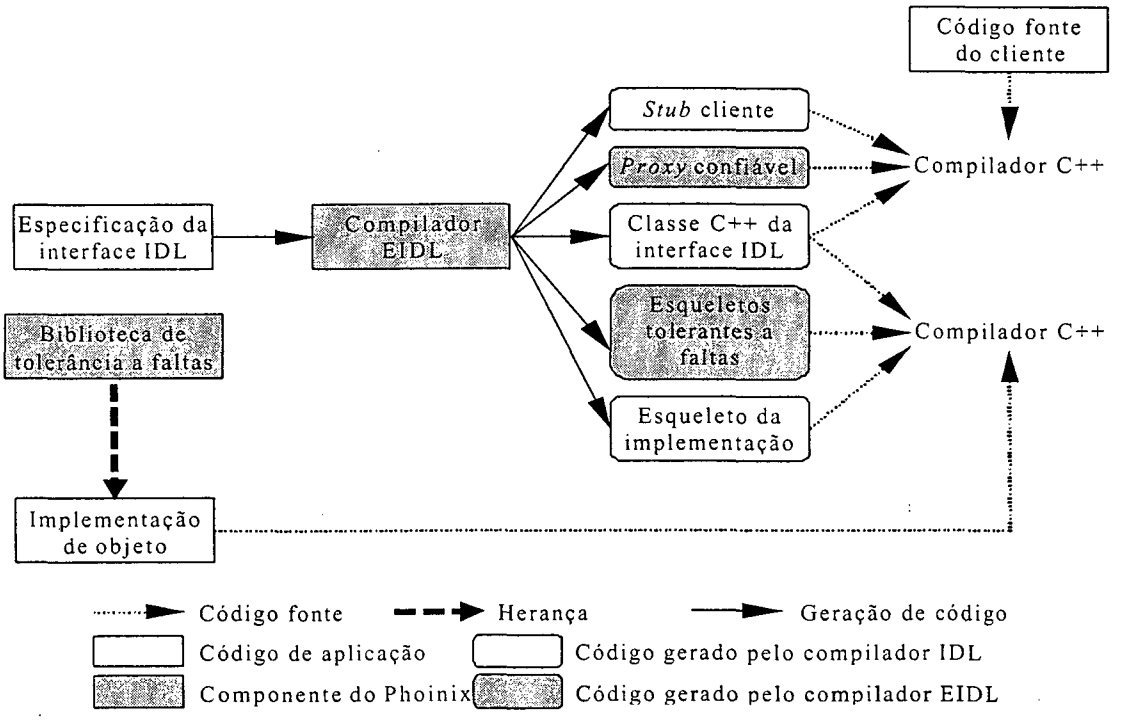


Figura 2.26. Desenvolvimento de aplicações no Phoinix.

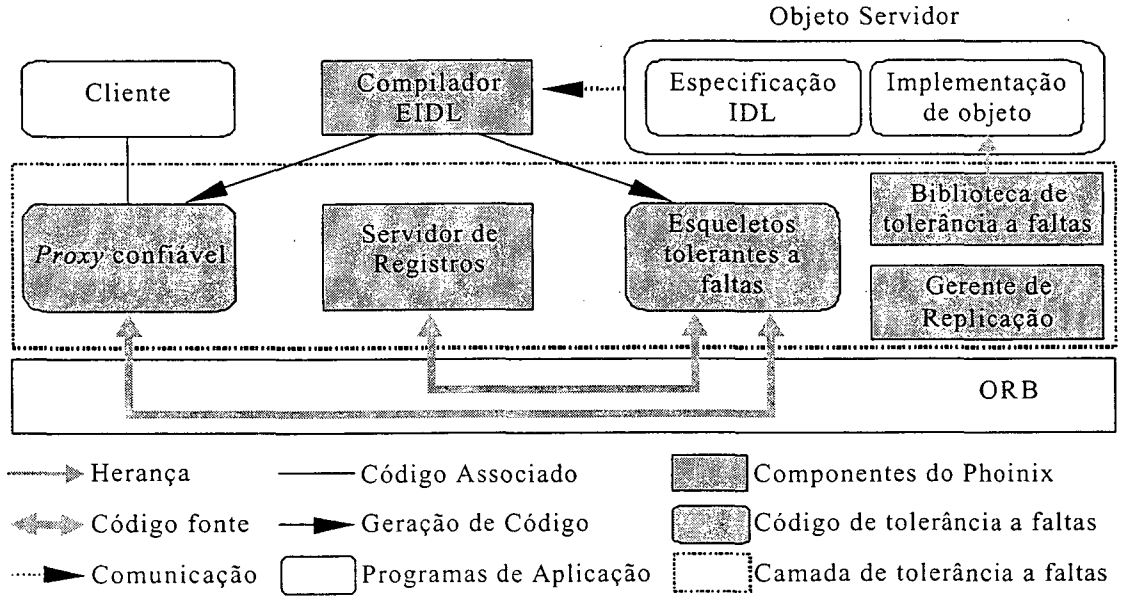


Figura 2.27. Arquitetura do Phoinix.

Detecção e recuperação de falhas no Phoinix

A detecção de falhas nas implementações de objeto no Phoinix é feita no lado cliente pelos *proxies* confiáveis. Para tal, estes *proxies* utilizam os serviços convencionais do

ORB. Neste contexto, o cliente detecta uma falha em uma implementação de objeto depois de fazer uma invocação e receber uma exceção do ORB. Assim que detectam uma falha, os *proxies* confiáveis também são responsáveis pela ativação do mecanismo de recuperação de erros.

O mecanismo de recuperação de erros varia de acordo com o nível do serviço de tolerância a faltas suportado. Em especial, a recuperação de erros na abordagem de reinicialização é extremamente simples. Assim que é detectada uma falha em uma implementação de objeto reinicializável, o *proxy* cliente apenas ativa uma implementação de objeto substituta. Este nível de tolerância a faltas apenas garante que as implementações de objetos estarão sempre disponíveis, não preocupando-se com o estado dos objetos.

A recuperação de erros no serviço de recuperação por retrocesso envolve quatro operações: o registro das requisições recebidas pelas implementações de objeto registráveis, a re-execução destas requisições, o armazenamento dos estados dos objetos registráveis e carregamento destes estados. Durante a operação normal do sistema, isto é, na ausência de erros, as requisições dos clientes enviadas às implementações de objetos registráveis são repassadas pelos esqueletos tolerantes a faltas ao servidor de registro, que é responsável por armazená-las. Os objetos registráveis tem a capacidade de decidir se uma requisição deve ou não ser salva de acordo com o domínio de aplicação. Neste contexto, nem todas as requisições são salvas. Além disso, periodicamente o servidor de registro cria pontos de recuperação, isto é, salva o estado dos objetos registráveis. Uma vez que um ponto de recuperação é criado, o servidor de registros elimina as requisições recebidas pelo objeto registrável que precedem o ponto de recuperação recém estabelecido.

Caso uma implementação de objeto falhe definitivamente, caracterizando um *crash*, o cliente recebe uma exceção como resposta à próxima requisição invocada. Então o *proxy* confiável associado ao cliente ativa uma implementação de objeto substituta. Esta implementação de objeto comunica-se com o servidor de registros para receber o estado da implementação de objeto falha e as requisições recebidas após o armazenamento deste estado. Finalmente, com o carregamento do estado e a re-execução das requisições, a nova implementação de objeto pode iniciar seus serviços.

2.5 Considerações gerais

Na literatura são encontrados diversos trabalhos de pesquisa e mesmo produtos que tratam da inclusão de mecanismos de tolerância a faltas na arquitetura CORBA. Além disso, existe um grupo de interesse especial em tolerância a faltas, formada pela OMG, que definiu um conjunto de requisitos desejáveis para o suporte as aplicações tolerantes a faltas na arquitetura CORBA (capítulo 5). Estes requisitos foram publicados na forma de um RFP (*Request for Proposal*) [OMG98a]. Muitos desses requisitos ainda não são satisfeitos nas soluções encontradas na literatura. A incompatibilidade maior está na necessidade de se manter as propriedades de sistemas abertos. Mesmo os esforços encontrados na literatura, como o Electra e o Orbix+Isis, exemplos da abordagem de integração, que apesar de oferecerem um alto grau de transparência e desempenho, não são completamente compatíveis com as especificações da OMG. O que, por exemplo, não permite a interoperabilidade com outros ORBs diferentes.

Apresentamos a seguir uma comparação informal entre as três abordagens (integração, serviço e interceptação) de provimento de tolerância a faltas e suporte de grupo, discutindo critérios como: transparência, facilidade de uso, portabilidade, interoperabilidade, conformidade com o padrão CORBA e desempenho [Felber98]. Esta comparação é baseada em implementações das abordagens presentes na literatura, isto é, o Orbix+Isis, Electra, OGS e Eternal. O OFS e o Phoinix são desconsiderados em alguns critérios, já que os mecanismos para a tolerância a faltas oferecidos não suportam a comunicação em grupo.

Transparência de replicação

A transparência de replicação oculta o processamento em grupo do programador da aplicação, dando a ilusão de que as invocações são originadas e atendidas por um único objeto. Na abordagem de integração, os clientes não precisam saber que a operação invocada é atendida por um grupo. Todavia, em situações especiais, os clientes podem se beneficiar deste conhecimento. Tomando como exemplo o OGS, a abordagem por objetos de serviço pode ser utilizada com ou sem transparência. No primeiro caso, os clientes invocam diretamente os serviços oferecidos pelo grupo servidor. Isto é realizado por meio

da utilização de esqueletos (*skeleton*) e interfaces de invocação dinâmica (DII) durante a comunicação entre clientes e servidores. No segundo caso, clientes e servidores devem, respectivamente, invocar e atender operações específicas para a comunicação em grupo. A abordagem de interceptação obriga a transparência. Os mecanismos de interceptação são responsáveis pela invocação do grupo servidor. Deste modo, diferente das outras abordagens, um cliente não pode acessar todas as respostas de uma invocação. No caso do FT-MOP e Phoenix, que fornecem suporte para a técnica de replicação passiva (anexo A), em caso de falha da réplica primária, o mecanismo de interceptação utilizado faz a função de redirecionar a requisição para a nova réplica primária. O OFS não possui transparência de replicação, o objeto cliente é avisado da falha do primário.

Facilidade de uso

A facilidade de uso é uma consideração importante, já que diminui o tempo de desenvolvimento e manutenção dos programas, tornando-os mais robustos e confiáveis. Por facilidade de uso entende-se a baixa complexidade na ativação e utilização dos mecanismos de gerenciamento e comunicação de grupo. Quanto menos construções de código para adaptar a aplicação a estes mecanismos, melhor a facilidade de uso. Neste contexto, esta característica está bastante relacionada com a transparência destes mecanismos.

As plataformas Electra, Orbix+Isis e Eternal apresentam maior facilidade de uso, já que o estabelecimento das estruturas de grupo é automático e transparente. Na abordagem de serviço, o OGS apresenta a configuração do suporte a grupo explícita, mas a comunicação em grupo pode ser transparente. No OGS, as aplicações usam os objetos fábrica e *proxies* para estabelecer explicitamente as estruturas de suporte a grupo. Neste sentido, esta abordagem combina mecanismos de configuração flexíveis com suporte a grupo transparente. Todavia, é importante ressaltar que, para as abordagens de interceptação e serviço, uma vez ativadas as estruturas de grupo, a sua utilização é transparente. Neste contexto, a comunicação em grupo não possui nenhuma complexidade adicional em relação a uma comunicação ponto a ponto convencional do CORBA.

O OFS, FT-MOP e Phoinix não oferecem suporte para comunicação em grupo, mas mecanismos para tolerância a faltas, através de replicação passiva. As aplicações em OFS

devem implementar explicitamente as comunicações com os objetos de serviço de tolerância a faltas. No FT-MOP e Phoinix, o estabelecimento das estruturas de tolerância a faltas é feito pelos meta-objetos (FT-MOP) ou pelos *proxies* (Phoinix). Em ambos os casos, o compilador IDL é estendido para gerar classes meta-objetos ou *proxies*. A diferença é que no FT-MOP o programador da aplicação deve implementar os códigos para tolerância a faltas nos meta-objetos. Enquanto que no Phoinix, os *proxies* têm seus códigos para tolerância a faltas implementados diretamente no processo de compilação da IDL (figura 2.26).

Portabilidade

A portabilidade implica na independência de ORBs específicos. Em particular, são consideradas a portabilidade do código do suporte a grupo e das aplicações desenvolvidas sobre este suporte. Na abordagem de integração, o código dos mecanismos de suporte a grupo é integrado ao ORB. Além disso, as aplicações desenvolvidas utilizam construções não padronizadas pelo CORBA. Neste sentido, os códigos do suporte e das aplicações não são portáveis. Na abordagem de serviço, tanto o código das aplicações como do suporte a grupo são portáveis, já que não dependem de características da implementação de um ORB específico. Em relação a abordagem de interceptação, mais especificamente no Eternal são utilizados mecanismos do Unix para suportar grupos. Desta forma, os mecanismos de grupo desta abordagem não são portáveis. Todavia, estes mecanismos não são referenciados no código das aplicações, tornando-as completamente portáveis.

No FT-MOP e Phoinix, tanto o código do suporte a tolerância a faltas como das aplicações desenvolvidas sobre este suporte são portáveis. Em ambos os casos, as invocações são feitas por meio de comunicação ponto a ponto convencional do CORBA, portanto não requer ORBs específicos para a execução desses códigos.

Interoperabilidade

A interoperabilidade implica na possibilidade de aplicações em ORBs distintos interagirem. Implementações que fazem uso de sistemas de comunicação proprietários não são interoperáveis. Este é o caso do Electra. O Orbix+Isis combina invocações sobre o Isis e sobre o IIOP. Desta forma, os objetos só podem interoperar por meio de invocações IIOP

ponto-a-ponto. O Eternal pode escolher quais as requisições devem ser interceptadas, também podendo interoperar sobre o IIOP. A abordagem de serviço utiliza apenas primitivas de comunicação do ORB, sendo completamente interoperável.

Conformidade com o padrão

A abordagem de integração não está em conformidade com o padrão CORBA, já que as estruturas definidas pelo padrão CORBA são estendidas pelo Orbix+Isis e Electra. A abordagem de serviço respeita a especificação CORBA. Os mecanismos de grupo na abordagem de serviço são independentes do núcleo do ORB, sendo definidos por objetos de serviço e suas interfaces IDL. A abordagem de interceptação é completamente independente do ORB, sendo baseada em estruturas do protocolo IIOP. O FT-MOP e Phoinix não estão em conformidade com padrão porque o compilador IDL é estendido para gerar mecanismos para interceptação (classes meta-objetos ou *proxies*).

Desempenho

A abordagem de integração apresenta o melhor desempenho, já que os mecanismos de gestão e comunicação em grupo são executados por uma ferramenta especializada. Apesar do suporte a grupo no Eternal também ser baseado em uma ferramenta especializada, o seu desempenho depende do mapeamento das estruturas IIOP para o sistema de grupo. O desempenho da abordagem de serviço é comprometido devido a utilização de mecanismos de comunicação ponto-a-ponto do ORB. Além disso, devido a ser estruturado como uma pilha de serviços definidos por interfaces IDL, a sua execução apresenta uma sobrecarga.

A tabela 2.1 fornece uma comparação quantitativa entre as propostas estudadas.

Plataformas	*Transparência de replicação*	*Facilidade de uso*	*Portabilidade*	*Interoperabilidade*	*Conformidade com o padrão*
Electra	Sim	Boa	Ruim	Nenhuma	Não
Orbix+Isis	Sim	Boa	Ruim	Sim	Não
OFS	Não	Ruim	Boa	Sim	Sim
OGS	Opcional	Boa	Boa	Sim	Sim
FT-MOP	Sim	Boa	Boa	Sim	Não
Phoinix	Sim	Boa	Boa	Sim	Não

Eternal	Sim	Boa	Regular	Sim	Sim

Plataformas	*Desempenho*	*Suporte de comunicação de grupo de baixo nível*	*Replicação ativa*	*Replicação passiva*
Electra	Alto	Sim	Sim	Não
Orbix+Isis	Alto	Sim	Sim	Não
OFS	-	Não	Não	Sim
OGS	Baixo	Não	Sim	Não
FT-MOP	-	Não	Não	Sim
Phoinix	-	Não	Não	Sim
Eternal	Médio	Sim	Sim	Não

Tabela 2.1. Comparação das plataformas.

2.6 Conclusão do capítulo

Este capítulo introduziu o modelo de objetos distribuídos da OMG, detalhando os principais componentes da arquitetura CORBA e os seus serviços subjacentes. Levantou um conjunto de requisitos básicos necessários ao suporte para tolerância a faltas e comunicação de grupo a partir de *middleware* CORBA.

Na seqüência, foram descritas algumas abordagens, detectadas a partir de propostas presentes na literatura, que incluem estes suportes. Essas abordagens atendem de maneira diferenciada os requisitos necessários para o suporte de grupo que foram descritos anteriormente. Uma análise é feita em relação às abordagens considerando os aspectos de transparência, facilidade de uso, portabilidade, interoperabilidade e conformidade com o padrão CORBA. É necessário ressaltar que todas essas pesquisas contribuíram, de certa forma, na definição das especificações FT-CORBA padronizadas recentemente pela OMG (capítulo 5).

CAPÍTULO 3
Comunicação de Grupo e Tolerância a Faltas no CORBA: Duas Propostas

3.1 Introdução

Este capítulo descreve duas propostas de trabalho que foram desenvolvidas em nossos laboratórios visando a introdução de mecanismos que viabilizassem a tolerância a faltas e a comunicação de grupo a partir da arquitetura CORBA. Estas duas propostas constituem-se no MetaFT [Fraga97, Lau99c] e no GroupPac [Lau99b, Lau00a].

Anterior à padronização, o projeto MetaFT [Lau96b, Fraga97, Lau97, Lau99c], a nosso conhecimento, foi uma das primeiras experiências conhecidas na literatura no sentido de introduzir abstrações de comunicação de grupo e de suporte para tolerância a faltas a partir de *middleware* CORBA. Estes trabalhos iniciais investigavam meios para se conseguir a captura de requisições de clientes direcionados a objetos replicados. O objetivo desta captura era conseguir o desvio da requisição para um suporte de comunicação de grupo garantindo deste modo, um alto grau de transparência a nível de cliente, de invocação múltipla do método desejado em todos os objetos membros da replicação. A solução encontrada, na época, foi o uso do paradigma de reflexão computacional, através de protocolos meta-objetos [Maes87], para prover comunicação de grupo e tolerância a faltas (seção 3.2). Este foi o primeiro trabalho envolvendo a abordagem de interceptação para introduzir tolerância a faltas e grupo no CORBA.

Esta primeira experiência, e os novos trabalhos relacionados que começaram a ser publicados um pouco depois na literatura [Felber98, Naras97], serviram para que evoluíssemos para uma nova proposta: GroupPac [Oliveira99b, Lau99a, Lau99b, Lau00a, Lau00d]. O modelo de introdução de mecanismos de tolerância a faltas e comunicação de grupo no GroupPac está baseado na combinação das abordagens de serviço e de

interceptação (capítulo 2). A primeira versão do GroupPac é a síntese dos nossos esforços ainda antes das especificações FT-CORBA.

Os objetivos deste capítulo são: apresentar o MetaFT – a nossa experiência usando interceptação via protocolos meta-objetos para tornar disponíveis mecanismos de tolerância a faltas para aplicações distribuídas constituídas a parir de *middlewares* CORBA; e descrever os objetos de serviço do GroupPac, introduzidos para prover tolerância a faltas no CORBA através da combinação das abordagens de serviço e de interceptação. Em cada um dos pontos apresentados neste capítulo serão discutidas nossas experiências envolvendo a implementação das propostas citadas, seguida de testes de desempenho realizados no sentido de verificar a eficiência dos mesmos.

3.2 MetaFT

O MetaFT [Fraga97, Lau97, Lau99c] define um modelo de programação que permite a introdução de técnicas de replicação em ambientes heterogêneos, tendo como plataforma de desenvolvimento o CORBA. Segue a abordagem de interceptação em nível de aplicação (capítulo 2, item 2.4.3.1). O modelo adota uma estrutura reflexiva segundo a abordagem de meta-objetos [Maes87]. A implementação das técnicas de replicação no MetaFT é apoiada no uso de ferramentas proprietárias.

3.2.1 Reflexão computacional

A essência do paradigma de reflexão computacional está no fato de que um sistema pode executar processamento sobre si mesmo, modificando então o seu comportamento. O paradigma reflexivo é introduzido na programação orientada a objetos na forma da abordagem de meta-objetos [Maes87] onde os aspectos funcionais e não-funcionais são separados com o uso de objetos-base e meta-objetos. Os objetos-base descrevem com os seus métodos as funcionalidades da aplicação, enquanto os meta-objetos associados executam as políticas de controle que determinam o comportamento de seus objetos-base correspondentes. As chamadas aos métodos de objetos-base são desviadas no sentido de ativar meta-métodos que permitem modificar o comportamento do objeto-base ou adicionar funcionalidades às correspondentes chamadas em nível base.

3.2.2 O modelo de programação MetaFT

A estrutura reflexiva proposta para incorporar os conceitos de técnicas de replicação é representada nas figura 3.1. Cada réplica foi mapeada sob a forma de um objeto-base, ao qual se associou um meta-objeto que assume funções de coordenação da técnica usada. Nos modelos de replicação que implementamos assumimos hipóteses de faltas de *crash*. Como admitimos um acoplamento forte entre controlador e réplica associada, os erros gerados serão atribuídos a ambos; em *crash*, controlador e réplica associada cessam de executar.

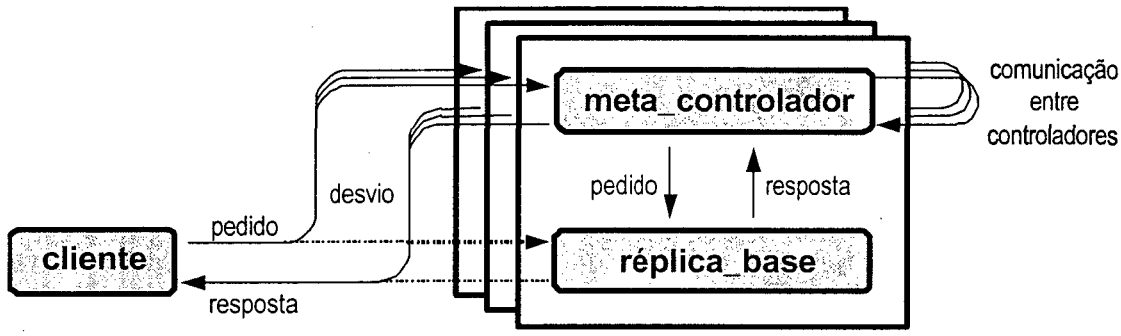


Figura 3.1. Estrutura reflexiva para o modelo de replicação ativa competitiva.

O modelo que integra técnicas de replicação no contexto CORBA é mostrado na figura 3.2. Nessa figura, o cliente apresenta-se estruturado em um cliente-base que representa o comportamento da aplicação, e um meta-cliente, que não possui função ativa em nossa implementação, mas que poderia ser usado no gerenciamento de um cliente replicado, ou para implementar mecanismos de tratamento de exceções no cliente. A estrutura de cada réplica do servidor é semelhante à do cliente: um objeto réplica base, realizando o serviço que se deseja replicar; um meta-controlador, responsável pelo protocolo de coordenação da técnica de replicação; e meta-objetos especiais, identificados genericamente como meta-comunicação, que controlam tanto no lado do cliente como no lado servidor o acesso ao suporte fornecido por uma plataforma CORBA [Fraga97]. Estas entidades concentram o conjunto de *stubs* do cliente, do servidor (as *stubs* para comunicação das réplicas) e o BOA[1] (*Basic Object Adater*) com o suporte para gerenciamento de grupo, gerados todos a partir da tradução da especificação IDL da interface do objeto servidor. O tratamento de "objeto abstrato" dado a meta-comunicação

[1] O BOA é um mecanismo similar e anterior ao POA (*Portable Object Adapter*) do padrão CORBA.

em nível de modelo segue a linha de alguns autores [Hagsand92] e tem o sentido de uma simples separação para maior clareza. Na verdade, estas interfaces são geradas como um conjunto de métodos que serão compostos com múltiplas heranças nos meta-objetos cliente e controlador.

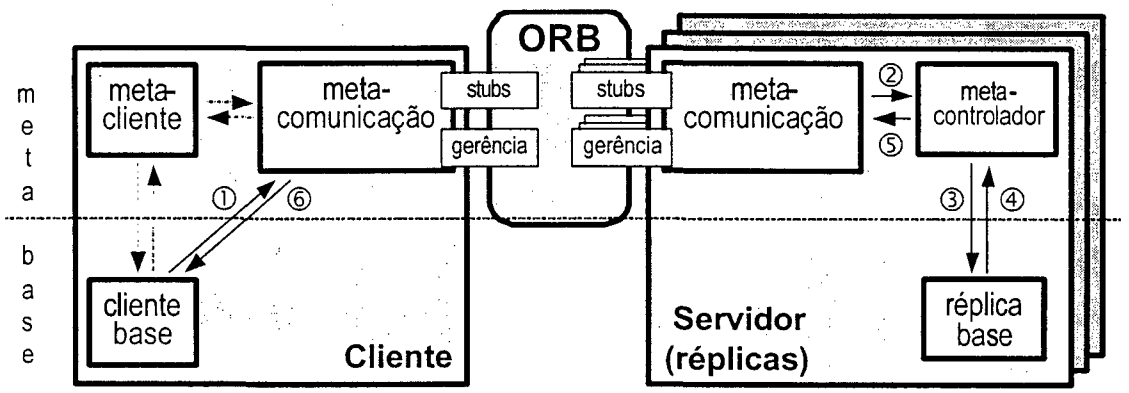


Figura 3.2. Estrutura do modelo sobre um suporte CORBA.

Na figura 3.2, as setas numeradas indicam o trajeto normal de um pedido do cliente: o pedido emitido pelo cliente base é desviado para o nível meta onde, através do *stub* apropriado no objeto meta-comunicação do cliente, o pedido é enviado pelo ORB ao serviço replicado; a seta ① descreve esse percurso. Em cada réplica, o meta-comunicação através de um *stub* local recebe o pedido e o transfere ao meta-controlador (②), que então aciona a réplica local (③). Ao receber sua resposta (④), o meta-controlador executa o protocolo de coordenação servindo-se do meta-comunicação para interagir com as outras réplicas. O processamento e as interações em nível de meta-controladores são condicionados pelo modelo de replicação utilizado. Após, a resposta é então devolvida ao cliente (⑤ e ⑥).

Este modelo pode ser usado com várias técnicas de replicação, as diferenças essencialmente se concentrarão nos meta-controladores do servidor replicado. Em algumas técnicas as entidades meta-comunicação podem ganhar funcionalidade além daquela de concentrar métodos de acesso aos serviços do suporte CORBA. Por exemplo, no uso de réplicas ativas com mecanismos de votador ou ajustador, a implementação da votação ou ajuste é programada de maneira menos custosa no lado do cliente. A transparência pode ser conseguida neste caso, implementando estes mecanismos na entidade meta-comunicação do cliente que com a adição desta funcionalidade ganha as características de um objeto real.

3.2.3 O modelo MetaFT n a programação de técnicas de replicação

No sentido de ilustrar um dos vários exemplos de técnicas de replicação desenvolvidos por nós com o modelo citado, descrevemos nesse item a nossa experiência com a técnica Líder/Seguidores. Nesta técnica de replicação todas as réplicas são ativas e executam o mesmo código, mas apenas a réplica líder é a responsável pelas interações de entrada e saída e pelas decisões que afetam a consistência das réplicas (anexo A). Ao receber um pedido de ativação o líder dissemina o mesmo entre os seguidores. O processamento do método se dá igualmente em todas as réplicas, porém só o líder envia os resultados ao cliente.

A figura 3.3 apresenta o pseudocódigo do meta-controlador referente à técnica líder/seguidores. A cada método-base de réplica é associada um meta-método no controlador (método_base_1 e meta_método_1, figura 3.3). As ações do controlador são descritas sucintamente na figura citada. Um diagrama temporal envolvendo as interações entre réplicas no processamento de uma requisição do cliente é mostrado na figura 3.4.

A indicação do líder é determinada pela réplica com `meu_id` igual a 1, a mais antiga do grupo (**`rank_system`** = 1, figura 3.3). As experiências que desenvolvemos fizeram uso de ferramentas de comunicação de grupo e os suportes utilizados (Horus e ISIS) adotam mecanismos de *rank*, o que fornece às implementações das técnicas de replicação que dependem de réplica privilegiada, a vantagem de eleger um líder sem a necessidade de qualquer troca de mensagem em nível de réplica. O uso do método `encerramento` tem como finalidade a detecção de falha da réplica privilegiada (líder) o que é simplificado pelas plataformas utilizadas. O método `encerramento` é implementado fazendo uso de listas de *membership* fornecidas pelos suportes de comunicação de grupo. Essas listas de *membership* são atualizadas pelo BOA, a cada entrada ou saída de réplicas no grupo.

```
class meta_controlador_1_im {
  // declaração de variáveis

  method meta_método_1 (parâmetros) {     // declarado na IDL
        método_base_1 (parâmetros);
        meta_controle (parâmetros);
  };
    ... // declaração dos demais meta-métodos
```

```
};
```

```
class meta_controlador_2_im {
     method meta_controle (parâmetros){
       encerrou := false;  semáforo := false;
         meu_id := rank_system();
         lider := 0;
         while not encerrou do
           lider := lider + 1;
           if (meu_id = lider) then
             // sou o líder
             grupo.resposta_lider(resposta);
           return; // retorna resposta ao cliente
           else
             wait(semáforo or timeout);
           if not encerrou then
                grupo.encerramento();
              end;
        end;
      end;
  };
  method encerramento(){              // declarado na IDL
    if (lider ∈ membership) then
         encerrou := true;
       end;
     };
     method resposta_lider(resposta){     // declarado na IDL
       if (minha_resposta ≠ resposta) then
      minha_resposta := resposta;
         semáforo := true;
       end;
     };
};
```

Figura 3.3. Pseudocódigo do Meta-controlador da replicação *líder/seguidores.*

Na figura 3.4, após o processamento do método requisitado, o líder (réplica 1) é mostrado enviando os resultados ao cliente e a seus seguidores. Uma vez satisfeita a condição do `wait` (figura 3.3 e 3.4), os seguidores executam o método `encerramento`. A réplica líder não pode sinalizar o fim de processamento aos demais membros do grupo, pois, ao retornar os resultados do processamento para o cliente a sua execução no método requisitado encerra. A ativação do método `encerramento` cabe a pelo menos uma réplica seguidora que provocará a execução do mesmo em todas as outras réplicas. Esse método tem a função de certificar se a resposta foi efetivamente enviada pelo líder (condição encerrou do laço **`while`**). A verificação do encerramento é simples: se após a difusão do método encerramento o líder ainda estiver vivo (pertencer ao *membership* do grupo) então a resposta foi efetivamente enviada. Caso contrário, um novo líder é

escolhido e o processo se repete. No algoritmo, a ativação do método `encerramento` é transmitida a todas as réplicas do grupo, de maneira totalmente ordenada.

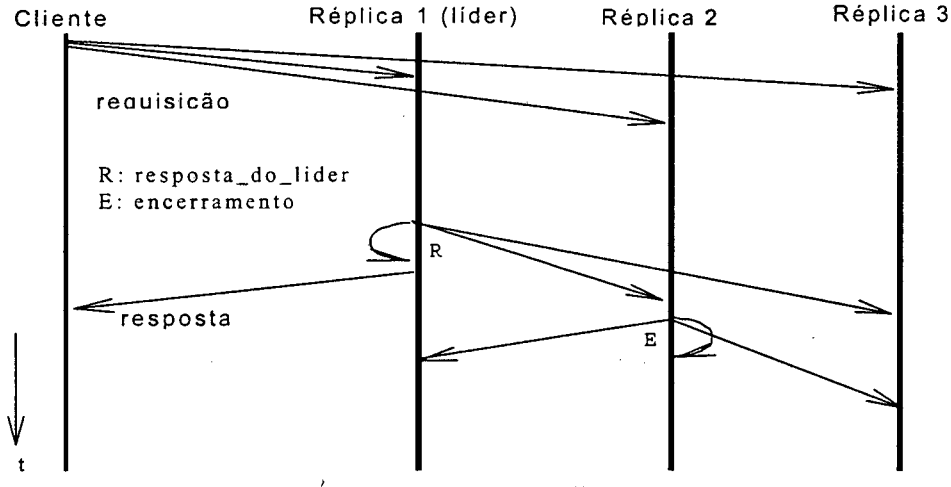


Figura 3.4. Diagrama temporal da replicação *líder/seguidores*.

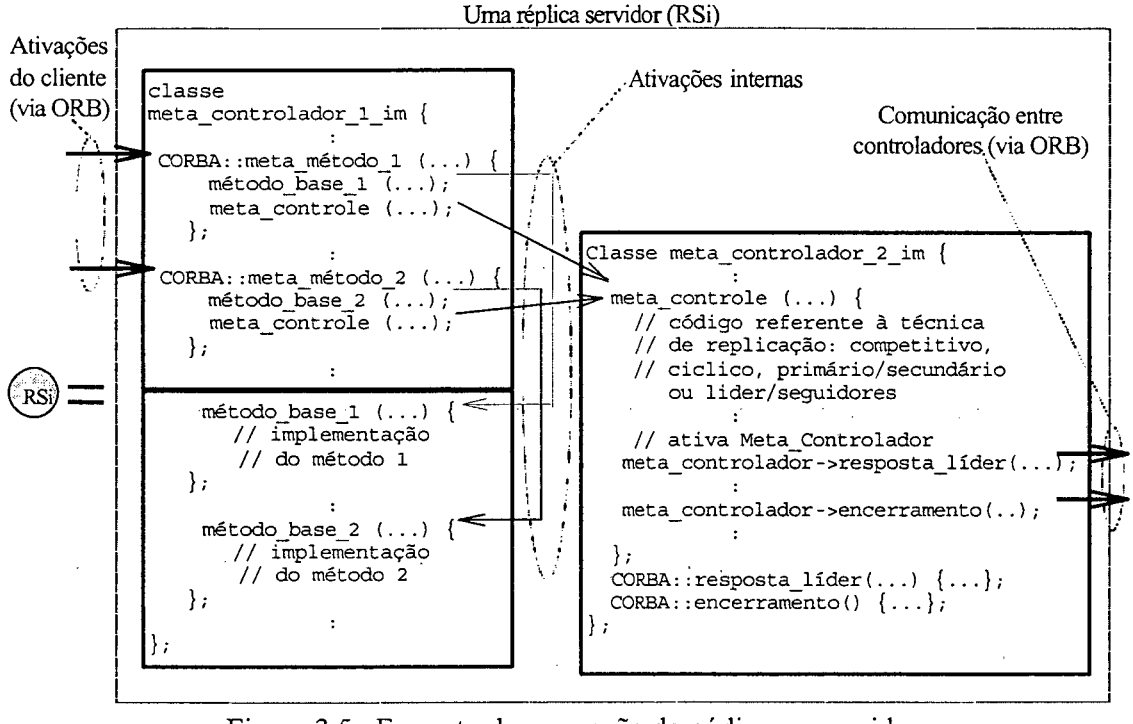


Figura 3.5. Formato da separação do código no servidor

Em nossas implementações, os métodos base das réplicas simulavam operações de uma aplicação de serviço bancário e foram desenvolvidos separados da parte da coordenação de réplicas. A figura 3.5 enfatiza essa separação que é a base do paradigma reflexivo. Em nossos experimentos diferentes meta-objetos, implementando diferentes

técnicas de replicação, eram trocados, mudando o comportamento na coordenação de réplicas do serviço, sem afetar as funcionalidades do nível base.

3.2.3.1 Implementação do modelo sobre o sistema Electra

O passo inicial para a implementação de um sistema sobre qualquer plataforma CORBA é a descrição em IDL da interface do meta-controlador. A interface consiste da declaração de cada um dos métodos oferecidos pelo servidor replicado ao cliente. Além desta, é necessário declarar uma segunda interface, composta pelos métodos que servem para o gerenciamento da replicação, nas interações entre as diferentes réplicas do serviço. O código em IDL de ambas as interfaces CORBA do servidor replicado, de acordo com as especificações descritas no item anterior, é mostrado na figura 3.6. A interface *meta_controlador_1* especifica o acesso dos clientes ao serviço replicado, enquanto a interface *meta_controlador_2* declara os métodos necessários às interações intra-réplicas.

```
interface meta_controlador_1 { // IDL
   // Descrição dos tipos de dados empregados
   // Descrição dos métodos do servidor
   boolean meta_método_1 (parâmetros);
   ...
   boolean meta_método_n (parâmetros);
};
interface meta_controlador_2
{
   // Descrição dos métodos do meta controlador
   boolean resposta_líder (in resposta);
   boolean encerramento ();
};
```

Figura 3.6. Interface IDL do servidor replicado.

Um dos primeiros experimentos com o modelo MetaFT, e que implementava a replicação semi-ativa, usava o sistema Electra. A escolha deste ORB estava ligado pela disponibilidade, via interface CORBA, de uma ferramenta de comunicação de grupo, o que facilitaria a implementação do MetaFT. A versão 1.0 do ORB Electra não suporta *threads* preemptivas, o que limita o grau de concorrência no tratamento de pedidos do cliente, além das interações entre os meta-controladores do serviço replicado. Esta restrição torna difícil a implementação de técnicas de replicação. Para contornar essa limitação, adotamos uma solução que consiste em separar as funcionalidades do meta-controlador em dois processos

UNIX. Um trata das interações cliente/servidor e da ativação do método do objeto-base e o outro é envolvido com as interações entre réplicas (entre os meta-controladores das réplicas). Cabe ressaltar que ambas as interfaces são na verdade duas facetas de um mesmo servidor (ou, em nosso caso, de um mesmo grupo de objetos). Portanto, as duas interfaces da figura 3.6 foram compiladas separadamente, e em tempo de execução o sistema se apresenta na forma de dois grupos de objetos (grupo meta_controlador_1 e meta_controlador_2).

No processo de compilação das interfaces de uma aplicação, o Electra gera automaticamente todo o suporte para a comunicação (*stubs*) entre as entidades envolvidas, incluindo também as funcionalidades para gerenciamento de grupos da classe BOA. O compilador também gera arquivos contendo "*esqueletos*" dos códigos do cliente e do servidor (declarações de variáveis e métodos). O programador fica então responsável pela descrição dos objetos-base da aplicação (meta_controlador_1) e do gerenciamento da replicação (meta_controlador_2), preenchendo os corpos dos métodos definidos nas interfaces (figura 3.6). Com esse esquema de implementação, que está ilustrado na figura 3.7, os objetos-base cliente e servidor permanecem isentos de quaisquer atividades que não estejam ligadas à aplicação propriamente dita. Todos os aspectos relativos ao gerenciamento da replicação e às interações no contexto do CORBA ficam concentrados ao nível dos meta-objetos (meta_controlador_2) [Lau96a].

3.2.3.2 Implementação do modelo sobre o Orbix+Isis

O ORB Orbix+Isis, se comparado ao Electra, é uma ferramenta já na forma de produto comercial, que apresenta mais facilidades para processamento e gerenciamento de grupos. Ou seja, também deixa disponíveis operações de uma ferramenta de grupo de mais baixo nível a partir de interfaces CORBA. Mas a principal vantagem em relação ao primeiro é o seu suporte a *multithreads*. Na implementação do modelo de integração com o uso do Orbix+Isis, cada par associado objeto-base/meta-objeto compartilha um mesmo processo UNIX, fazendo com que as interações entre ambos sejam locais, sem necessidade do ORB. As necessidades de concorrência entre objeto-base e meta-objeto no interior de um processo são satisfeitas através do uso de uma biblioteca de *threads* oferecida pela ferramenta Isis. O passo inicial da implementação é também a pré-compilação IDL das duas interfaces da figura 3.6. Porém, vale ressaltar neste caso que a geração da

implementação do servidor dá origem a um único grupo de processos que atuará em sua execução. A figura 3.7 explicita o processo de geração dos códigos do cliente e servidor a partir da IDL.

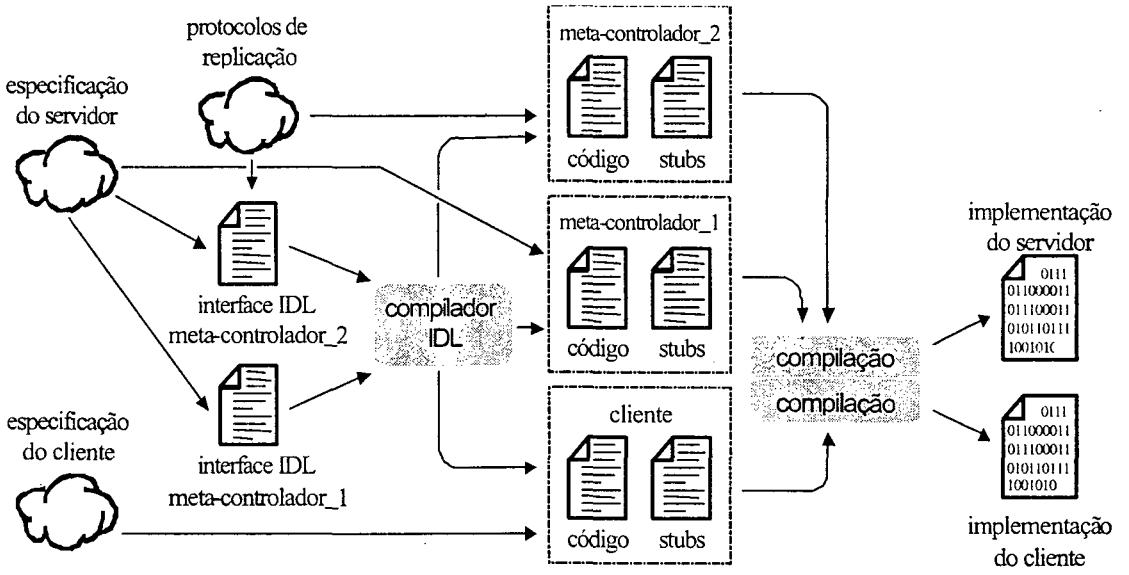


Figura 3.7. Processo de construção da aplicação.

3.2.4 Desempenho dos experimentos realizados

Neste item, apresentamos as medidas de desempenho realizadas no laboratório para checar nossas experiências, usando o Electra e o Orbix+Isis para implementar o MetaFT. Estas medidas foram feitas a partir da invocação de métodos nos servidores replicados. O desempenho do MetaFT foi medido com diferentes graus de replicação. O ambiente considerado, nestes testes de desempenho, foi composto de uma rede local (10 Mbps Ethernet) heterogênea usando duas máquinas Sun Ultra 1 com Solaris 2.5, um Axil 240 também com Solaris 2.5, uma máquina Pentium 100 e um Pentium 233 MMX, ambos com Linux, e finalmente, um Pentium 233 MMX com windows95.

Os testes conduzidos consistiram de cem requisições no grupo replicado. Na figura 3.8, o tempo de resposta (eixo Y) relacionado ao grau de replicação (eixo X) foi determinado considerando a média dessas cem invocações de métodos. A curva da operação de requisição apresenta um incremento no tempo de resposta na medida em que o grau de replicação é incrementado. Usando o Electra ou o Orbix+Isis para implementar o MetaFT o fator de crescimento do tempo de resposta por réplica adicionada era de aproximadamente 10 milissegundos (ms). No entanto, como podemos ver no gráfico da

figura 3.8, o Orbix+Isis apresenta um desempenho melhor que o Electra – aproximadamente 5 milissegundos. Medidas realizadas com as ferramentas Electra e Orbix+Isis, sem o MetaFT, mostram um desempenho de aproximadamente 50% superior, mostrando, desta forma, o custo de desempenho do nosso modelo.

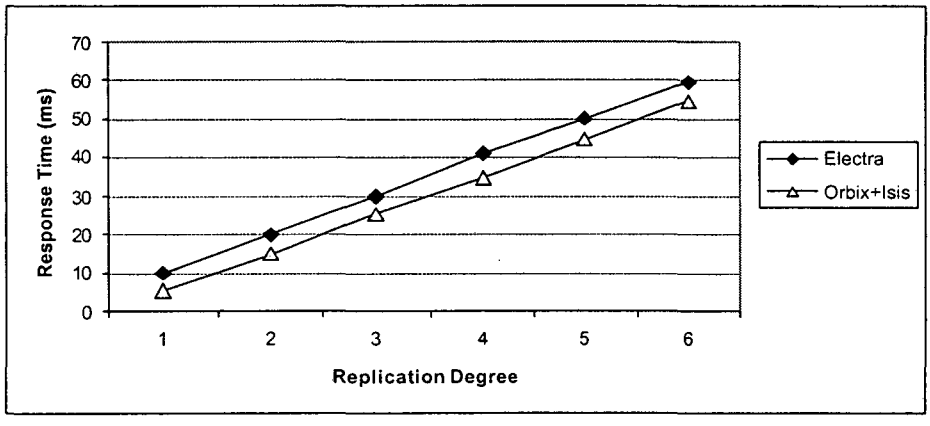


Figura 3.8. Desempenho do MetaFT usando o Electra e o Orbix+Isis.

3.2.5 Considerações sobre o modelo

Em relação às falhas que eventualmente ocorrem na evolução do sistema, o modelo de programação concentra a nível meta medidas de recuperação do grau de replicação, independente do tipo de técnica de replicação usada. Se o número de réplicas cair abaixo de um limite preestabelecido, a réplica mais antiga toma a iniciativa de lançar novas réplicas, restabelecendo a população ideal. O código referente a estes procedimentos de recuperação vale para os dois ORBs utilizados e é baseado num teste de *membership* (`view.number < quorum_mínimo`), e está inserido no corpo do método `view_change`[2] [Lau96a]. Nossa abordagem de atualização de estado (*checkpoint*) das réplicas difere daquela proposta em [Fabre95], na qual as atualizações de estado se dão através de meta-métodos fazendo *updates* em atributos públicos de suas réplicas associadas, com o uso exclusivo de protocolos de coordenação. Em nossa abordagem utilizamos mais primitivas de suporte e menos protocolos em nível de coordenação, simplificando os códigos de nossos meta-controladores. A recuperação de estado, em nosso sistema, está baseada na primitiva `join`, que deve ser oferecida pela interface BOA do suporte CORBA, e ativada através do método `view_change`.

[2] Método que trata do gerenciamento de *membership*, fornece a informação do número de réplicas no grupo (`view.number`).

Devido ao fato da linguagem usada (*C++*) não possuir construções específicas para o suporte à reflexão, como por exemplo, o desvio automático da ativação de um método no nível base para o respectivo meta-método, nas nossas implementações a reflexão é implementada artificialmente, através da ativação direta ao meta-método, a partir do código do cliente (figura 3.5). O uso de uma linguagem suportando reflexão, como é o caso de *Open C++* [Chiba93], poderia eliminar este problema, mas neste caso o compilador IDL do ambiente CORBA usado teria que suportar o mapeamento para essa linguagem.

A reflexão computacional permite a independência dos códigos das réplicas em relação aos protocolos de coordenação, conduzindo a uma grande flexibilidade no sistema: mudar de técnica ou alterá-la para atender a graus de tolerância a faltas desejados podem resultar em simplesmente trocar os protocolos de coordenação a nível meta, sem implicar na alteração de algoritmos de aplicação, ou ainda em mudanças em nível de suporte de execução, o que seria difícil em sistemas heterogêneos. O uso da reflexão na implementação de técnicas de tolerância a faltas não é novo [Fabre95], [Fraga97], [Lisbôa96] e mesmo a separação entre a coordenação e as réplicas já foi preconizada em [Brito95] e [Powell91].

Em [Fabre95], uma arquitetura R2 é usada para as divisões meta-objetos para a coordenação e objetos-base para as funcionalidades de aplicação. Os exemplos especificados no texto são simples, envolvem replicações (genéricas) ativas, passivas e semi-ativas. As coordenações a nível meta são bem mais complexas pelo uso de comunicações ponto-a-ponto.

O modelo apresentado integra conceitos de reflexão e processamento de grupo em ambientes heterogêneos. As implementações realizadas fazem uso intensivo das funcionalidades de plataformas CORBA, o que implicou numa simplificação das necessidades de coordenação nas técnicas de replicação implementadas. Além disso, o uso do CORBA permitiu a implementação de nossa aplicação sobre um ambiente de execução heterogêneo (grupo de máquinas com sistema operacional SunOS 4.X, Solaris 2.5 e Linux 2.0, interligadas por uma rede local), facilitando os aspectos de interoperabilidade. A estrutura de integração proposta se mostrou bem flexível, podendo comportar outras técnicas de replicação [Fraga97]. Até o momento, implementamos as técnicas de replicação *Primário/Secundário* [Budhiraja93], *Líder/Seguidores* [Little91], *Competitiva*

[Powell91] e de *Redundância Cíclica* [Powell91], usando o mesmo modelo de programação. As mudanças necessárias à substituição da técnica de replicação no modelo limitam-se à interface IDL dos meta-controladores e aos seus códigos, que implementam os protocolos de coordenação correspondentes.

Nas nossas implementações, foram usados ORBs que apresentavam nas interfaces CORBA operações de uma ferramenta de comunicação de grupo de mais baixo nível (ORBs que seguem a abordagem de integração, capítulo 2), o que facilitou nossas implementações. Mas a exemplo do Eternal (ORB que segue a abordagem de interceptação), o MetaFT pode ser implementado usando uma ferramenta de comunicação de grupo independente das interfaces do CORBA (figura 2.20). Neste caso, em nível meta, no cliente, as requisições interceptadas seriam difundidas usando uma ferramenta de comunicação de grupo, e em meta-nível, nos servidores replicados através de interfaces CORBA, os métodos seriam invocados. A diferença com o Eternal é que no caso do MetaFT a interceptação é feita em nível de aplicação. A nosso conhecimento o Eternal foi a primeira experiência envolvendo a abordagem de interceptação no sentido de integrar tolerância a faltas em ambientes CORBA.

3.3 GroupPac

A primeira versão do GroupPac [Lau00d], que foi desenvolvido em nosso laboratório e na universidade do Texas em Austin [Lau99b], é um sistema que fornece suporte de grupo [Oliveira99a, Oliveira99b, Lau00a] e tolerância a faltas [Lau99a, Lau99b] na arquitetura CORBA. Este suporte é provido através de um conjunto de objetos de serviço (ou componentes) que são especificados segundo os requisitos apresentados na *COSS* (*Common Object Services Specification*) do padrão OMG. De acordo com estes requisitos, os objetos de serviço do *GroupPac* servem como blocos de construção para a implementação de técnicas de tolerância a faltas ou de aplicações orientadas a grupo (ex: aplicações de trabalho cooperativo). Esses objetos de serviço podem ser combinados constituindo-se em *frameworks* que permitem, por exemplo, a implementação de diferentes esquemas de tolerância a faltas. A solução adotada no *GroupPac* consiste em combinar alguns aspectos da abordagem de serviço (item 3.4.2) e de interceptação (item 3.4.3.3) oferecendo, deste modo, uma melhor transparência dos serviços de grupo à aplicação. No

sentido de mostrar as potencialidades desse conjunto de serviços, é discutido o *framework* que implementa um suporte para replicações ativas. As soluções apresentadas nesta seção visam demonstrar que o GroupPac consegue atender aos requisitos relacionados a portabilidade, interoperabilidade, reusabilidade e conformidade com o padrão OMG, além de ser uma solução que pode não ser dependente de ferramentas de comunicação de grupo proprietário.

Os serviços do *GroupPac* devem representar soluções abertas, na forma de interfaces genéricas. Na falta dessas interfaces padronizadas, antes da padronização do FT-CORBA, tivemos que definir os objetos de serviço a partir de suas interfaces, especificadas em IDL do padrão CORBA. A opção por determinados algoritmos no desenvolvimento desses serviços ou ainda, a construção dos mesmos no topo de uma ferramenta de tolerância a faltas como o ISIS, Horus ou Totem, não é um fator muito importante na implementação desses serviços. Pois, uma vez que a OMG defina seus serviços comuns para tolerância a faltas e as suas interfaces genéricas e padronizadas, as escolhas de implementação dos mesmos não serão um problema dentro de uma perspectiva de sistemas abertos. As nossas definições em relação a esses serviços são discutidas a seguir.

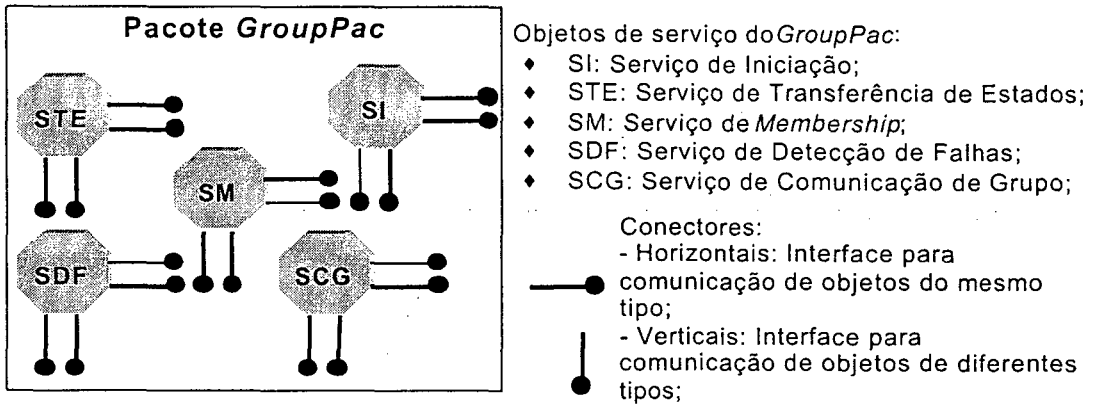


Figura 3.9. O pacote de objetos de serviço do GroupPac.

Na figura 3.9, são apresentados os objetos de serviço SI, STE, SM, SDF e SCG – todos fazendo parte do *GroupPac*. Cada um desses objetos possui uma interface horizontal e uma vertical, ambas definidas em IDL/CORBA: a primeira permite as comunicações entre objetos de serviço do mesmo tipo, e a segunda constitui-se no meio pelo qual fornecem serviços para outros objetos.

O objeto SI (Serviço de Iniciação) é responsável pela criação e iniciação dos outros objetos de serviço do pacote. É a partir desse objeto que a configuração necessária ao suporte de uma aplicação replicada é construída. O objeto STE (Serviço de Transferência de Estados) oferece funcionalidades para transferências de estados de um objeto para outro. Esse serviço pode ser usado, por exemplo, em modelos de replicação ou ainda, para migração de objetos. O objeto SM (Serviço de *Membership*) é responsável pelo serviço de gerenciamento de grupos de réplicas (grupos de objetos). Esse gerenciamento deve ser transparente à aplicação, exercendo um controle dinâmico nas entradas e saídas de objetos de um grupo pela manutenção de listas atualizadas de seus membros (*membership*). O objeto SDF (Serviço de Detecção de Falhas) envolve um conjunto de procedimentos de detecção de falhas de objetos em um grupo. O SDF opera em conjunto com o objeto SM: quando o SDF detecta um *crash* de um dos objetos do grupo, essa falha é imediatamente reportada ao objeto SM para que esse gere uma nova lista de membros. Por último, temos o objeto SCG (Serviço de Comunicação de Grupo) que oferece um conjunto de facilidades para comunicação de grupo. Este serviço fornece um conjunto de protocolos confiáveis de comunicação de grupo, com diferentes políticas de ordenação de mensagem (FIFO, Causal ou Total).

Os serviços descritos acima podem ser combinados permitindo a implementação de diferentes esquemas de tolerância a faltas. Um dos experimentos usando o *GroupPac* foi o serviço de nomes replicado - *CosNamingFT* [Lau 99a, Lau99b]. Esse serviço de nomes segue as especificações CosNaming da OMG [OMG97a]. A técnica de replicação implementada nesse serviço é o primário/*backup* (técnica de replicação passiva), construída a partir dos objetos de serviço definidos no *GroupPac*. Na seqüência deste texto, apresentamos os serviços do *GroupPac* e um *framework* que através da combinação desse serviços, forneça suporte para replicação ativa. E no próximo capítulo é apresentado nossas experiências com o CosNamingFT.

3.3.1 Descrição dos serviços GroupPac

Apresentamos neste item os objetos de serviço do *GroupPac* e uma forma de combiná-los para dar suporte a implementação da técnica de replicação ativa (anexo A). O suporte de grupo necessário para garantir o determinismo das réplicas ativas é montado a partir dos objetos de serviço do *GroupPac*. A nossa proposta de suporte de grupo combina

características das abordagens de serviço e de interceptação, apresentadas no capítulo 2 dessa tese. Deste modo, os aspectos de gerenciamento de réplicas são providos por um conjunto de objetos de serviço em nível do ORB, segundo a abordagem de serviço. Aspectos relacionados com a comunicação de grupo seguem a abordagem de interceptação onde conceitos como o de interceptador (item 3.4.3.3), definido pela OMG [OMG98b], são usados.

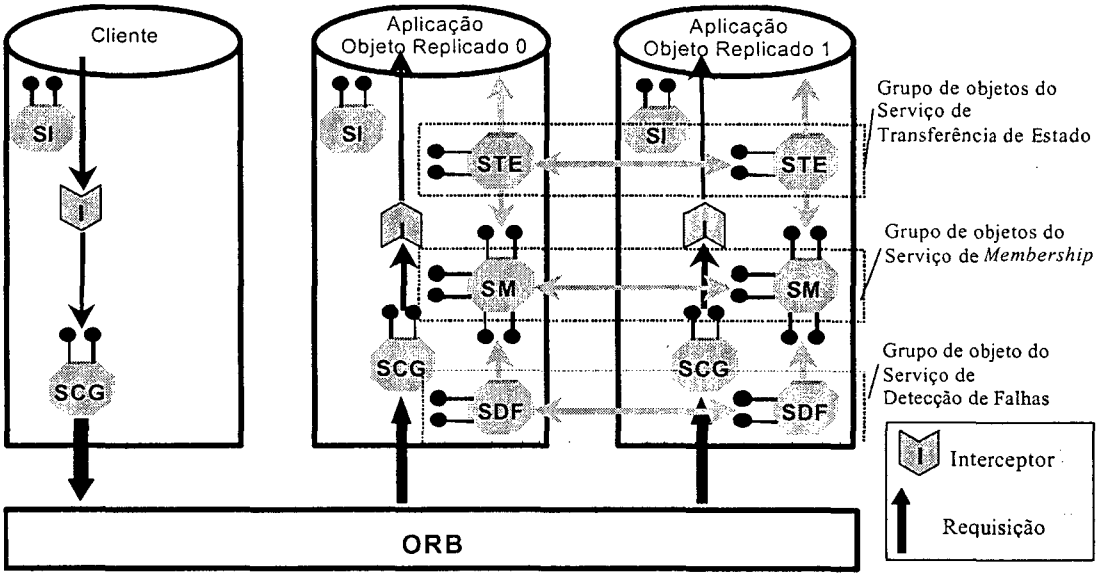


Figura 3.10. Suporte a grupo usando objetos de serviço do GroupPac.

A vantagem nessa combinação de aspectos dessas abordagens é que, uma vez configurado o suporte de comunicação, os grupos ficam totalmente transparentes para seus clientes. Na essência poderemos dizer que todo o suporte a grupo estará disponível na forma de objetos de serviço comum e que alguns desses serviços serão ativados de forma transparente, através do uso de interceptadores.

Os objetos de serviço do *GroupPac* que vão formar o suporte de grupo em que cada objeto replicado da aplicação ocupam junto com as réplicas dos objetos SI, STE, SM, SDF e SCG do nosso *framework* o mesmo espaço de endereçamento (por exemplo, um único processo UNIX). A figura 3.10 ilustra o uso dos objetos do *GroupPac* na técnica de replicação considerada. A falha de um destes objetos é considerado como falha da réplica como um todo (*crash* do processo). Os serviços STE, SM, SDF e SCG, são dispostos na forma de grupos de objetos de serviço. Por exemplo, levando em conta as conexões

horizontais da figura 3.10, os objetos SM de cada réplica formam o grupo que fornece o serviço de *membership*.

3.3.1.1 Serviço de gerenciamento de grupo

Esse serviço controla as entradas e saídas dinâmicas das réplicas do grupo, fornecendo a informação de quais réplicas estão operacionais no sistema. A combinação dos grupos de serviço SM e SDF implementam essa funcionalidade. O serviço de detecção de falhas implementado pelo grupo SDF tem a finalidade de 'detectar' falhas causadas por *crash* de réplicas. Esse serviço implementa um protocolo [Ricciardi91] com premissas de comunicações ponto-a-ponto, confiáveis e assíncronas entre os pares comunicantes. Portanto, a falta de uma mensagem de resposta pode não ser devido a sua perda no canal de comunicação, mas pode indicar que, ao invés disso, alguma coisa errada está acontecendo com o servidor. É assumido que os detectores são sempre perfeitos, isto é, se uma réplica não responde, após o *timeout* pré-estabelecido, é considerada falha.

Na figura 3.11, cada objeto do grupo SDF envia mensagens do tipo "você está ativo ?", periodicamente (período T(s)), ao membro anterior da ordenação em anel virtual do grupo. Ao não receber resposta de seu antecessor dentro de um prazo pré-definido, uma réplica R_i considera a possibilidade do *crash* de seu antecessor R_{i-1}, devendo então ativar o método `leave_group` do objeto SM da réplica R0. Essa ativação indica a suspeita em relação a réplica R_{i-1}.

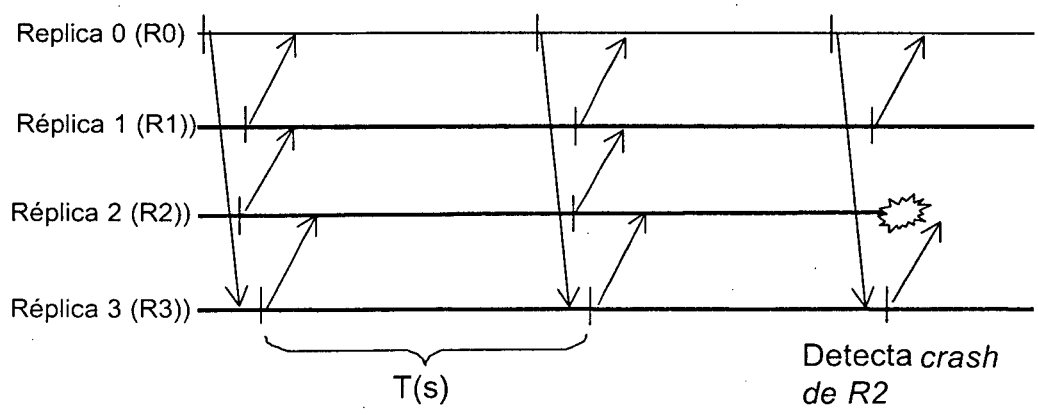


Figura 3.11. O serviço de detecção de falhas.

O serviço de *membership* é então executado (ou seja, os objetos do grupo SM são ativados). O protocolo implementado nesse serviço é de decisão centralizada envolvendo

interações entre as réplicas em duas fases (três fases no pior caso) para alcançar acordo com as outras réplicas SM sobre a nova composição do grupo [Ricciardi91]. A coordenação na obtenção de uma nova lista de membros (*new view*) é sempre centralizada no objeto SM da réplica de menor *rank* (no caso acima, o R0). A figura 3.12 mostra uma instancia bastante simples de funcionamento desse protocolo. O objeto SDF da réplica R3 ao detectar o *crash* da réplica R2 envia a mensagem "suspeite de R2" para o objeto SM da réplica R0. Neste ponto, o protocolo de *membership* é ativado e a réplica R0 difunde uma mensagem *commit* para verificar quem ainda continua no grupo, as réplicas que estão ativas devem dar um reconhecimento a esta mensagem (*Ack*). Após um prazo de espera pré-definido, o primário produz uma nova lista de membros a partir dos *Ack* recebidos [Renesse95] [Ricciardi91].

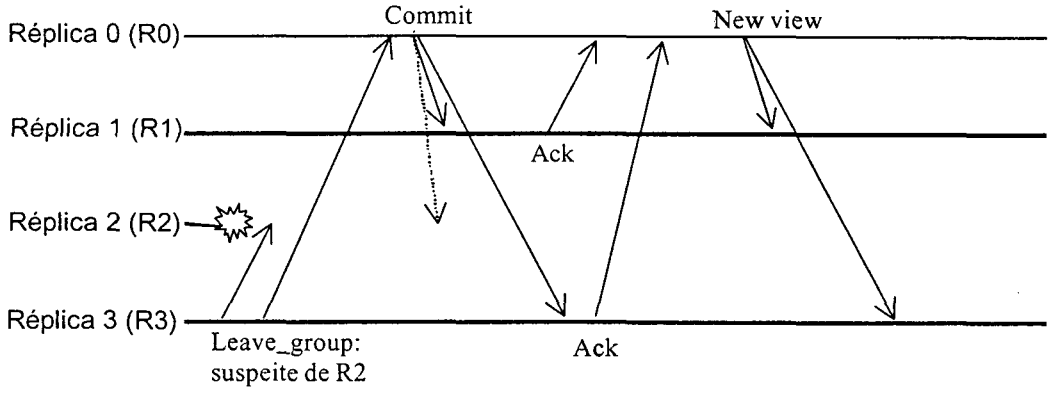


Figura 3.12. Situação: crash da réplica R2

3.3.1.2 Serviço de transferência de estado

O serviço de transferência de estado (STE) do *GroupPac* é também incorporado ao espaço de endereçamento das réplicas da aplicação, para garantir que cada nova réplica no grupo tenha seu estado consistente em relação às outras. Na figura 3.13, são apresentados os passos executados através desse serviço na transferência de estados. Uma nova réplica ao tentar se juntar a um grupo deve invocar o método `join_group` na réplica R0. A referência de objeto da réplica R0 é obtida através do método `resolve` do objeto *CosNaming* – servidor de nomes do sistema (item 3.2.2.2). A partir desta invocação os objetos SM de cada réplica do grupo, que compõem o serviço de *membership* em si, interagem para decidir pela entrada dessa nova réplica (*commits* e *acks*). Se a nova réplica é aceita, o serviço de transferência de estado é ativado. Esse serviço consiste na obtenção

do estado da réplica mais antiga do grupo (R0), usando o método `get_state`, e transferi-lo para a nova réplica usando o método `set_state` (figura 3.13). Só assim, a nova lista de membros (new view) é instalada.

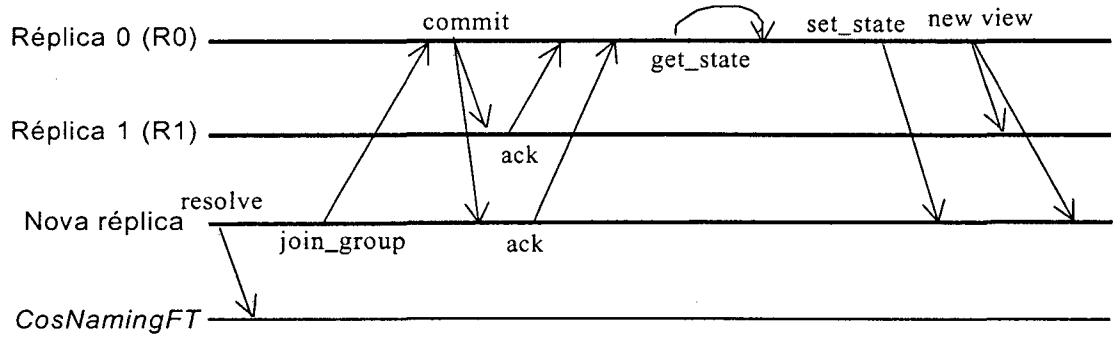


Figura 3.13. Réplica que se junta a um grupo usando os serviços SM e STE.

3.3.1.3 Serviço de comunicação de grupo no GroupPac

Esse serviço [Oliveira99a, Oliveira99b, Lau00a] deve prover, através de suas interfaces, acesso a comunicação de grupo confiável, com diferentes políticas de ordenação de mensagem (FIFO, Causal ou Total). A construção desse serviço pode ser feita a partir de outros objetos de serviço do *GroupPac* (por exemplo, o SM e o SDF), com comunicações ponto-a-ponto através do ORB ou com ferramentas proprietárias de comunicação de grupo.

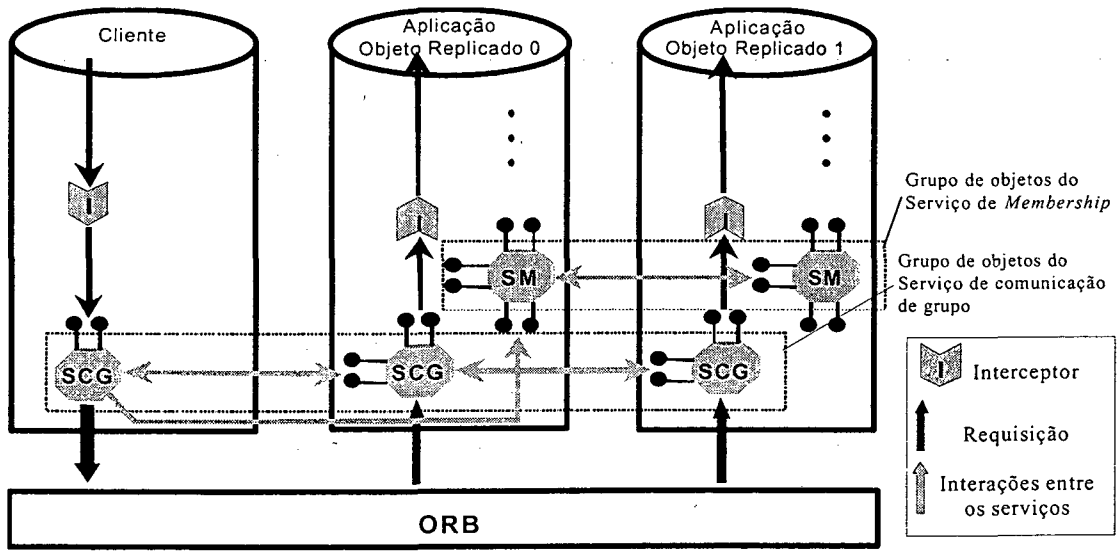


Figura 3.14. O serviço de comunicação de grupo.

A implementação da abordagem de interceptação, usando mecanismo do ORB (item 3.4.3.3), no que se refere à comunicação de grupo, permite que os mecanismos necessários sejam acionados, de forma transparente, não envolvendo as aplicações cliente e servidor. Esta transparência é alcançada a partir de mecanismos que desviem as requisições do cliente. O uso de interceptador, mecanismo definido nas especificações CORBA, e introduzido no capítulo anterior (item 2.4.3.3), permite este desvio transparente, adequado às recomendações da OMG. O interceptador (interceptador I da figura 3.14) captura a requisição de cliente e a redireciona para o grupo de objetos de serviço SCG (serviço de comunicação de grupo, figura 3.14). Os SCGs que dão suporte a cada objeto de aplicação (cliente e grupo de servidores replicados) formam um grupo e interagem de forma a alcançar o acordo e a ordem das mensagens, antes de entregá-las aos interceptadores que por fim, enviam-na a sua respectiva réplica.

O objeto SCG (Serviço de Comunicação de Grupo), que oferece facilidades para comunicação de grupo, pode disponibilizar um conjunto de protocolos de comunicação de grupo, com diferentes políticas de ordenação de mensagem (FIFO, Causal e Total), dependendo do *framework* que faça parte.

3.3.2 Aspectos de implementação do GroupPac

Discutiremos nesta seção alguns aspectos relacionados com a implementação do *GroupPac*. Todos os serviços deste pacote são implementados na linguagem Java, o que traz a vantagem da portabilidade, e suas interfaces especificadas segundo o padrão IDL/OMG. As implementações foram realizadas na plataforma de desenvolvimento *OrbixWeb* [IONA97]. Esse *middleware* foi completamente desenvolvido na linguagem Java, seguindo as especificações do CORBA 2.0. Os objetos do *OrbixWeb* se comunicam usando o protocolo IIOP, também padronizada pela OMG.

Todos objetos de serviço, apresentados nos itens anteriores, podem implementar algoritmos específicos de suas funcionalidades e interagirem com os seus pares de grupo usando comunicações ponto a ponto. Uma segunda possibilidade é estes objetos de serviço, a partir de suas interfaces, mapearem operações em ferramentas de mais baixo nível. Estas duas abordagens de implementação foram concretizadas em nossos experimentos:

1) Os objetos de serviço do GroupPac (figura 3.9) são mapeados em

funcionalidades do Isis [Birman91], ou de uma outra ferramenta de comunicação de grupo (ex: Horus e Transis). Esta solução, proposta na dissertação de mestrado [Oliveira99a, Oliveria99b, Lau00a], introduz um adaptador de grupo que permite este mapeamento nas APIs[3] do Isis. Esta alternativa será apresentada na seqüência deste capítulo;

2) Consiste na implementação dos objetos de serviço do GroupPac sem o uso do Isis, ou de uma outra ferramenta qualquer de mais baixo nível. Apresentaremos esta solução no capítulo 4, na forma de um exemplo de aplicação (um serviço de nomes tolerante a faltas – CosNamingFT [Lau99a, Lau99b]) usando os objetos de serviço do GroupPac.

3.3.3 Implementação do GroupPac fazendo uso de ferramenta de comunicação de grupo

A estrutura proposta como plataforma para o desenvolvimento das implementações é apresentada na figura 3.15. Esta estrutura prevê a justaposição de dois ambientes: o ORB e o suporte a grupo (objetos de serviço do GroupPac). Enquanto as comunicações ponto-a-ponto convencionais trafegam pelo ORB, as comunicações de grupo são implementadas usando estruturas de comunicação separadas. Estas estruturas separadas constituem o suporte a grupo, o adaptador de grupo e a ferramenta proprietária (Isis [Birman91]).

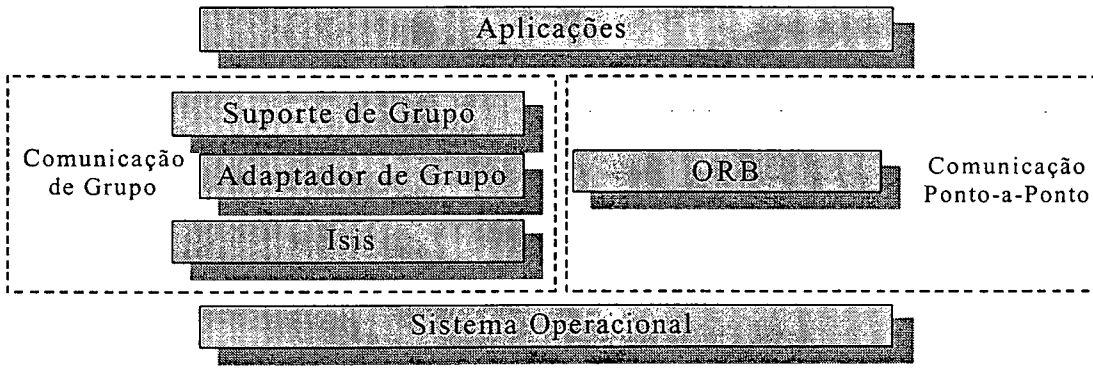


Figura 3.15. O serviço de comunicação de grupo.

A ferramenta Isis oferece suporte de mais baixo nível para implementar os serviços de gerenciamento e de comunicação de grupo. O Adaptador de Grupo visa livrar os objetos de serviço do GroupPac em suas implementações da dependência de ferramentas específicas. O adaptador define uma interface IDL padrão para o acesso aos serviços do

[3] Interface de programação da aplicação (*Application Programming Interface*).

Isis, desacoplando as implementações do suporte a grupo de detalhes do Isis. Finalmente, o Suporte de Grupo que consiste de um conjunto de objetos de serviço que fornecem às aplicações uma interface para a utilização dos serviços de grupo.

Conforme apresentado no inicio da seção 3.3.1, a invocação dos objetos de serviço do GroupPac (fornecido pelo Suporte de Grupo da figura 3.15) é realizada pelos mecanismos de interceptadores fornecidos pelo ORB (abordagem de interceptação, item 2.4.3). No entanto, a fim de possibilitar uma análise dessa proposta, foi realizada, também, uma implementação segundo a abordagem de serviço (item 2.4.2), na qual a invocação dos serviços é feita de forma não transparente, sem o uso de interceptadores. A seguir são discutidos aspectos de implementação e avaliações de desempenho de ambas soluções.

As duas abordagens avaliadas se utilizam da ferramenta Isis. Desta forma, os mecanismos de comunicação e gerenciamento de grupo apresentam uma sobrecarga semelhante, permitindo que sejam avaliados apenas os processos de ativação dos serviços nas duas abordagens. Em particular, o desempenho das implementações das abordagens de serviço e de interceptação deve variar somente durante o estabelecimento das estruturas de grupo. Na abordagem de serviço as próprias aplicações são responsáveis por solicitar a criação dessas estruturas. Na abordagem de interceptação, esta criação é ativada transparentemente.

3.3.3.1 Implementação pela abordagem de serviço

Na figura 3.16, são ilustradas a utilização de objetos *proxies* e adaptadores na difusão de uma mensagem. Somente as comunicações entre adaptadores clientes e servidores utilizam o Isis. Os demais objetos do modelo se utilizam do ORB em comunicações ponto-a-ponto. Devem ser criados objetos *proxies* e interfaces SCG para cada grupo do qual um servidor deseja participar ou com o qual um cliente deseja comunicar-se. Estes objetos são criados a partir de fábricas[4]. O uso de fábricas é obrigatório, já que a linguagem IDL não define construtores. Na figura 3.17 é ilustrada a criação de *proxies* e adaptadores servidores. Inicialmente, o servidor da aplicação requisita a uma fábrica a criação de um *proxy*. Em seguida, este *proxy* transparentemente requisita a outra fábrica a criação de um

[4] Os objetos fábrica, referenciados em inglês como "*factory objects*", são definidos pela especificação do COSS de ciclo de vida. Em particular, um objeto fábrica é um objeto CORBA comum que é utilizado para criar outros objetos.

adaptador. A partir de então, o *proxy* e o adaptador criados podem ser utilizados normalmente pela aplicação. A criação destas estruturas no lado cliente é análoga.

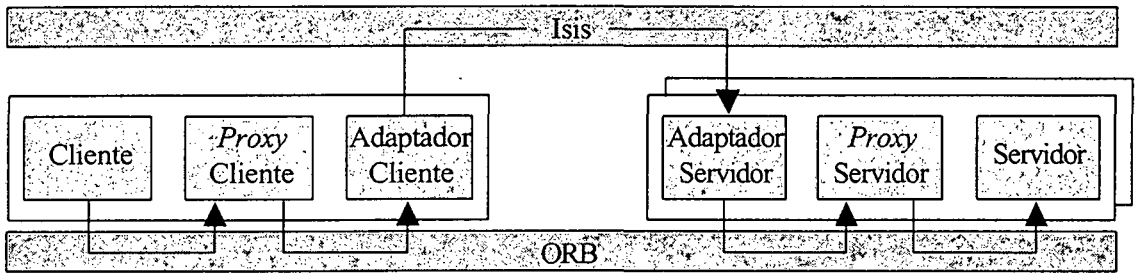


Figura 3.16. Difusão de mensagem na implementação da abordagem de serviço.

Os *proxies* e adaptadores devem ser criados em processos distintos, isto é, eles não devem residir no mesmo processo da fábrica. Cada fábrica é responsável pela criação de apenas um único tipo de objeto. Além disso, cada fábrica deve responder por um domínio. Em particular, espera-se que esteja disponível uma fábrica para cada máquina da rede, o que garante que *proxies* e adaptadores não sejam objetos remotos, oferecendo um maior desempenho e confiabilidade às aplicações.

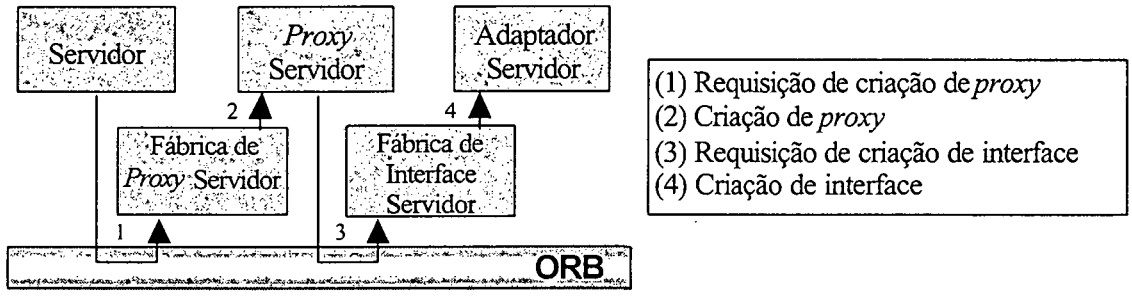


Figura 3.17. Criação de proxies e interfaces de servidores.

A função principal dos *proxies* é permitir a difusão de requisições dos clientes aos servidores. Esta tarefa é realizada por meio de esqueletos e interfaces de invocação dinâmicas. Em particular, os esqueletos dinâmicos permitem que os *proxies* clientes atendam requisições baseadas na interface do servidor. Por sua vez, as interfaces de invocação dinâmica (*DII: Dynamic Invocation Interface*) permitem que os *proxies* servidores invoquem operações definidas nos membros de grupo. O uso destas estruturas é necessário, já que em tempo de compilação, não é possível determinar as operações do grupo ao qual os *proxies* serão associados.

No lado servidor, os *proxies* muitas vezes devem invocar operações nos membros de grupo. Isto acontece durante as mudanças de *view* do grupo e durante as transferências de estado. Neste contexto, os membros de grupo devem ser derivados de uma interface IDL padrão que define estas operações de gerenciamento. Na figura 3.18 é ilustrada esta interface, além das interfaces dos *proxies* clientes e servidores e suas fábricas.

```
module GroupPac {
  // Group server member
  interface GroupServerMember {
    any getState ();
    void setState (in any state);
    void changeView (in GroupView view);
  };
  // Group server proxy
  interface GroupServerProxy {
    void joinGroup ();
    void leaveGroup ();
    GroupView getView ();
  };
  // Group server proxy factory
  interface GroupServerProxyFactory {
    GroupServerProxy create (in GroupServerMember
                    groupServerMember, in string groupName);
    void destroy (in GroupServerProxy groupServerProxy);
  };
  // Group client proxy
  interface GroupClientProxy {
    GroupView getView ();
  };
  // Group client proxy factory
  interface GroupClientProxyFactory {
    GroupClientProxy create (in string groupName);
    void destroy (in GroupClientProxy groupClientProxy);
  };
```

Figura 3.18. Criação de *proxies* e interfaces de servidores.

A única operação disponível em um *proxy* cliente retorna a composição do grupo. Com esta informação as aplicações clientes podem executar comunicações ponto-a-ponto com cada membro do grupo. Além desta funcionalidade, os *proxies* servidores oferecem operações para a entrada e saída de membros de um grupo. A interface das fábricas permite apenas a criação e destruição de *proxies*. Finalmente, a interface dos membros de grupo permite a obtenção e configuração de seus estados, além da notificação de alterações na composição do grupo.

3.3.3.2 Implementação pela abordagem de interceptação

A implementação da abordagem de interceptação, através de mecanismos do ORB (item 3.4.3.3), permite que os mecanismos de grupo sejam de forma transparente adicionados às aplicações clientes e servidores. Esta transparência é alcançada a partir do uso de interceptadores[5] do OrbixWeb. Os interceptadores (no caso, filtros) são associados às aplicações clientes e servidores e se responsabilizam pela comunicação com as fábricas para a criação de *proxies*. Além disso, os filtros fazem com que as entradas e saídas de membros de grupo sejam automáticas. No OrbixWeb foram usados filtros de processo que interceptam todas as requisições e respostas chegando ou partindo de um processo cliente ou servidor, independentemente dos objetos de origem e de destino.

Na abordagem de interceptação implementada, a utilização dos filtros é prevista somente durante a associação de um cliente ou um servidor a um grupo. Neste contexto, os filtros devem interceptar apenas as comunicações entre as aplicações e o serviço de nomes. Segundo a especificação CORBA, cada referência de objeto é associada a um ou mais nomes. O serviço de nomes é utilizado para registrar e recuperar estas referências. A abordagem de interceptação aproveita esta característica do padrão CORBA para definir nomes de grupo. Um nome de grupo identifica um serviço que é prestado por um grupo de servidores. Cada nome de grupo é prefixado pela *string* "grp://". Esta característica visa facilitar a sua identificação. Em especial, um mesmo servidor, deve poder cadastrar-se individualmente ou como um membro de grupo. Desta forma, todas as requisições ao serviço de nomes que especificarem um nome de grupo devem ser interceptadas e manipuladas pelos filtros. Esta solução caracterizou nossa preocupação em definir meios de referenciar grupos de objetos. Conforme veremos no capítulo 5, a OMG solucionou esse problema através da padronização do formato de referência de grupo de objetos (IOGR).

Para obter uma referência a um objeto servidor, os clientes enviam uma requisição ao serviço de nomes. Caso esta requisição referencie um nome de grupo, um filtro deve interceptá-la. Então, este filtro deve entrar em contato com uma fábrica para criar um *proxy* de grupo. Em seguida uma referência deste *proxy* é retornada ao cliente. Neste contexto, o

[5] Nas implementações desenvolvidas foram usados como interceptador mecanismos conhecidos como filtros no OrbixWeb, cujas funcionalidades são similares às que se espera de um interceptador.

cliente tem a ilusão de que recebeu uma referência para um objeto comum, quando na verdade recebeu uma referência para um *proxy*. Esta situação é ilustrada na figura 3.19.

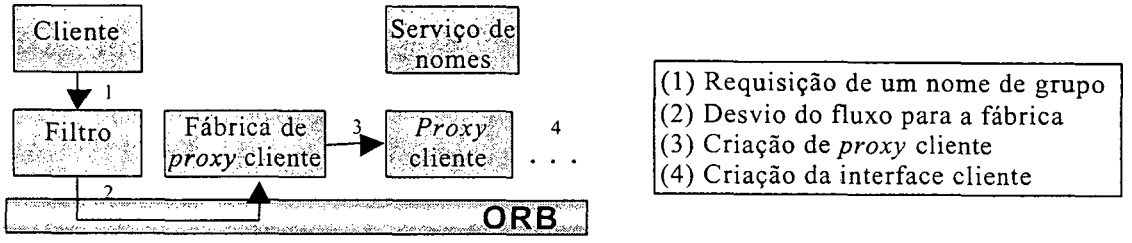


Figura 3.19. Ativação das estruturas de grupo na abordagem de interceptação.

Para cadastrarem-se, os servidores também devem realizar uma requisição ao serviço de nomes. Caso esta requisição especifique um nome de grupo, um filtro deve interceptá-la. De modo similar ao lado cliente, este filtro deve entrar em contato com uma fábrica para criar um *proxy* de grupo. Além disso, o filtro deve invocar a operação de junção ao grupo no *proxy*. A partir de então, o servidor passa a receber requisições de seu *proxy* como se ele fosse um objeto cliente qualquer.

3.3.3.3 Comparação dos desempenhos das abordagens implementadas

O desempenho das abordagens implementadas foi também confrontado com a do Orbix+Isis que é representante da abordagem de integração (item 2.4.1). Os ORBs da abordagem de integração embora falhem em outros requisitos, normalmente são os que apresentam melhores desempenhos (tabela 2.1).

As avaliações de desempenho foram realizadas com base em testes envolvendo três estações Ultra Sparc rodando o sistema operacional Solaris 2.5 sobre uma rede Ethernet 10 Mbs. O limite no número de estações se deve a condições da licença da ferramenta Isis disponível. Nestas avaliações de desempenho também deve ser considerado que enquanto no Orbix+Isis, as aplicações são desenvolvidas em C++, nas implementações da arquitetura proposta, as aplicações são desenvolvidas em Java. A linguagem Java 1.1 apresenta um desempenho cerca de dez vezes menor que a linguagem C. Todavia, o impacto do uso da linguagem Java é bem menor, já que os mecanismos de difusão e gerenciamento de grupo são providos pelo Isis. Além disso, um dos fatores mais importantes para o desempenho é o tempo de transmissão na rede.

Na figura 3.20 são apresentados os tempos médios de resposta de uma invocação sobre um grupo de objetos servidores, considerando o Orbix+Isis (abordagem de integração) e as implementações do SCG do GroupPac segundo as abordagens de serviço e de interceptação. A média destes tempos foi obtida a partir do tempo de resposta de mil invocações. Além disso, nesta mesma figura, é ilustrada a evolução dos tempos de resposta em função do número de servidores presentes no grupo. Cada servidor é executado em uma máquina distinta. Como pode ser percebido, o Orbix+Isis possui um desempenho superior. Embora com o aumento do número de servidores presentes no grupo exista um aumento no tempo de resposta, a diferença de desempenho entre as abordagens estudadas permanece constante. Neste contexto, caso o número de servidores seja muito grande, espera-se que a diferença de desempenho seja desprezível. A justificativa para a diferença de desempenho entre estas abordagens é a utilização de comunicações ponto-a-ponto entre objetos intermediários nas implementações das abordagens de serviço e de interceptação (figura 3.21).

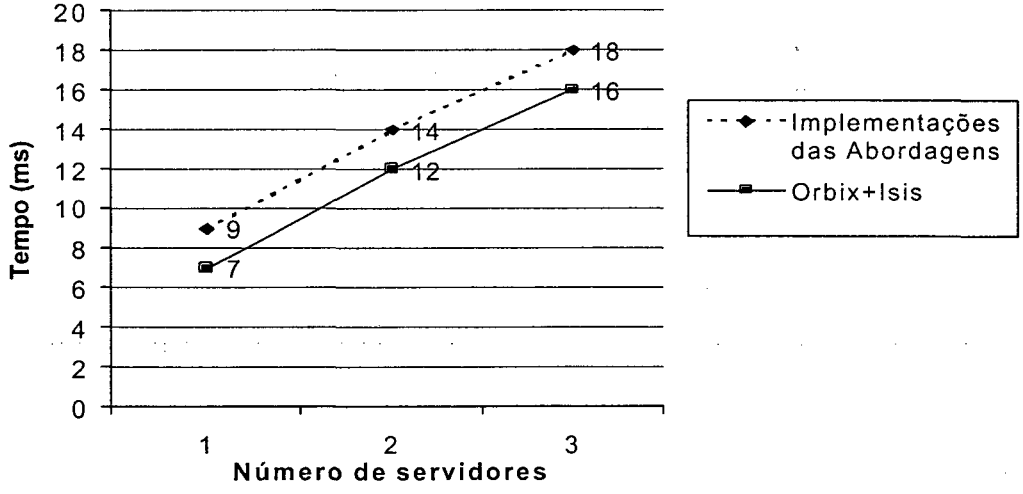


Figura 3.20. IDL do serviço de comunicação de grupo

A fim de avaliar melhor o desempenho das abordagens, foram levantados tempos de resposta parciais a uma invocação de um serviço prestado por um grupo de objetos. Conforme ilustrado na figura 3.21, os tempos de resposta parciais envolvem toda a trajetória entre os objetos de aplicação (cliente e servidor). Nesta trajetória são considerados os tempos parciais para o envio de uma requisição e o retorno de seu resultado.

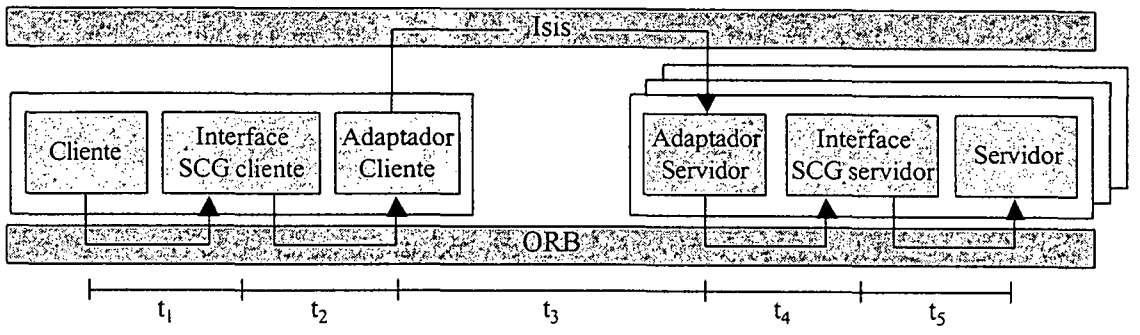


Figura 3.21. Pontos onde os tempos de resposta parciais foram medidos.

Na figura 3.22(a) é apresentado o impacto de cada tempo parcial na execução de uma invocação a um serviço prestado por um grupo de servidores. Pode ser percebido que o aumento do número de servidores implica apenas no aumento no tempo de tráfego entre as interfaces clientes e servidores. Esta característica é esperada, já que a comunicação entre as aplicações e os *proxies* e entre os *proxies* e as interfaces, envolve apenas tráfego local. Além disso, é somente na comunicação entre as interfaces que ocorre a execução da difusão em grupo. A seguir são apresentadas alternativas de implementação da arquitetura proposta que visam o aumento de desempenho. Em particular, estas alternativas buscam a diminuição do tempo envolvido nas comunicações locais. Neste contexto, estas comunicações deixariam de ser intermediadas pelo ORB.

	1	2	3
t_1	0,5	0,5	0,5
t_2	0,5	0,5	0,5
t_3	7	12	16
t_4	0,5	0,5	0,5
t_5	0,5	0,5	0,5
t_{total}	*9*	*14*	*18*

(a) Implementações propostas

	1	2	3
t_1	0,5	0,5	0,5
t_{234}	7,5	12,5	16,5
t_5	0,5	0,5	0,5
t_{total}	*8,5*	*13,5*	*17,5*

(b) Ausência de fábrica de interfaces

	1	2	3
t_{total}	*8*	*13*	*17*

(c) Ausência de fábricas

Figura 3.22. Tempos médios (ms) de resposta parciais em variações da arquitetura proposta

A primeira alternativa de implementação da arquitetura proposta envolve a eliminação dos processos onde residem os adaptadores. Neste contexto, ao invés de entrar em contato com uma fábrica, os *proxies* instanciam diretamente os adaptadores. Desta forma, os *proxies* e adaptadores deixam de residir em processos distintos, fazendo com que a interação entre estes objetos passe a ser executada em um mesmo espaço de

endereçamento. Esta alternativa não causa nenhum impacto sobre as aplicações clientes e servidores, já que estas aplicações mantêm a mesma interface com os *proxies*. Neste contexto, esta alternativa pode ser implementada segundo as abordagens de serviço e interceptação. Na figura 3.22(b) são apresentados tempos médios de resposta desta alternativa. Em particular, o tempo t_{234} envolve as comunicações do *proxy* cliente ao *proxy* servidor e vice-versa. Segundo estas avaliações, pode-se perceber que houve uma considerável diminuição nos tempos de resposta.

A segunda alternativa de implementação da arquitetura proposta envolve tanto a eliminação dos processos onde residem os adaptadores, como a eliminação dos processos onde os *proxies* (interface SCG) residem. De forma semelhante à alternativa anterior, as aplicações cliente e servidor passam a instanciar diretamente os *proxies*. Esta alternativa, embora também apresente um considerável aumento de desempenho, conforme ilustrado na figura 3.22(c), aproximando-se do alcançado pelo Orbix+Isis, possui algumas desvantagens. Esta alternativa não consiste em uma solução CORBA, já que o código dos *proxies* é disponibilizado às aplicações como parte de uma biblioteca – o ORB foi suprimido. Além disso, a sua implementação não consiste nem na abordagem de serviço, nem na abordagem de interceptação.

Outro fator de desempenho que deve ser observado é o tempo necessário para o estabelecimento das estruturas de grupo, isto é, os *proxies* e adaptadores. Na implementação da abordagem de serviço este tempo é de 2 ms, já na abordagem de interceptação são necessários 3 ms. Esta diferença se deve ao uso de filtros na abordagem de interceptação. No caso do Orbix+Isis, as estruturas de grupo são inerentes a qualquer aplicação. Desta forma, não é necessário tempo para a sua criação.

3.3.4 Considerações gerais sobre GroupPac

Abordagens que se baseiam em objetos de serviço comum (COSS) para implementar mecanismos e suportes de tolerância a faltas estão mais de acordo com os princípios da OMG. As funcionalidades necessárias são adicionadas como objetos de serviço e não representam em modificações do ORB, não apresentando qualquer modificação semântica na comunicação normal entre clientes e servidores. As soluções sendo apresentadas como objetos de serviço com interfaces genéricas definidas pela OMG devem atender às

expectativas de soluções abertas. O *GroupPac* segue essa linha de soluções abertas. Esse pacote de serviços, conceitualmente, provê objetos de serviço como um conjunto de blocos de construção que podem ser arranjados de diferentes formas no sentido de compor diferentes esquemas ou arquiteturas de serviços de aplicação com propriedades de tolerância a faltas. É importante ressaltar que esse *framework* foi proposto no sentido de atender aos requisitos apresentados no RFP da OMG e que refletiu, na seqüência, a evolução dos trabalhos do grupo de interesse criado na OMG para tratar com as padronizações referentes a tolerância a faltas nas especificações CORBA.

Os exemplos presentes na literatura, que seguem essa mesma linha (abordagem de serviço), são de alguma forma limitados. O OFS [Liang98] oferece meios na forma de objetos de serviço comum para que se implemente serviços de aplicação replicados tomando como base técnicas de replicação passiva. Porém nenhum suporte para comunicação de grupo necessário na implementação de técnicas de replicação ativa é apresentado no OFS. No OFS os objetos de serviço são definidos com um fim específico, não apresentando a possibilidade de uso dos mesmos em diferentes esquemas de tolerância a faltas. Se tomarmos o OGS como exemplo, em termos de suporte a grupo, esta proposta está dentro da filosofia do CORBA. No entanto, como exposto em [Felber98], aspectos de tolerância a faltas não foram devidamente tratados, as preocupações se concentram somente na inclusão de objetos de serviço que permitam a comunicação de grupo.

Por último, na abordagem de interceptação, o sistema Eternal apresenta uma solução em que é dependente das funcionalidades do Unix e de um suporte de grupo proprietário o que dificulta aspectos de portabilidade. Já no Phoinix é proposta uma extensão da IDL do CORBA para geração de *proxys* para comunicação de grupo, o que de fato afeta o aspecto de portabilidade e interoperabilidade com outras plataformas.

Resgatando a tabela 3.1 do capítulo 2, temos uma comparação desses trabalhos relacionados com o MetaFT e GroupPac. Nossas duas propostas oferecem *transparência de replicação* uma vez que seguem a abordagem de interceptação nas invocações de grupo. Em termos de *facilidade de uso*, no MetaFT, o programador deve selecionar um meta-objeto, que implementa uma determinada técnica de replicação, na hora de construir a aplicação. No GroupPac, a aplicação deve instanciar o objeto SI (figura 3.9) para que este

faça a composição dos objetos de serviço do GroupPac para uma determinada técnica de replicação. Em ambos os casos, a aplicação não percebe a existência do suporte de grupo.

Plataformas	*Transparência de replicação*	*Facilidade de uso*	*Portabilidade*	*Interoperabilidade*	*Conformidade com o padrão*[6]
Electra	Sim	Boa	Ruim	Não	Não
Orbix+Isis	Sim	Boa	Ruim	Sim	Não
OFS	Não	Ruim	Boa	Sim	Sim
OGS	Opcional	Boa	Boa	Sim	Sim
FT-MOP	Sim	Boa	Boa	Sim	Não
Phoinix	Sim	Boa	Boa	Sim	Não
Eternal	Sim	Boa	Regular	Sim	Sim
MetaFT	Sim	Boa	Ruim	Sim	Sim
GroupPac	Sim	Boa	Boa	Sim	Sim

Plataformas	*Desempenho*	*Suporte de comunicação de grupo de baixo nível*	*Replicação ativa*	*Replicação passiva*
Electra	Alto	Sim	Sim	Não
Orbix+Isis	Alto	Sim	Sim	Não
OFS	-	Não	Não	Sim
OGS	Baixo	Não	Sim	Não
FT-MOP	-	Não	Não	Sim
Phoinix	-	Não	Não	Sim
Eternal	Médio	Sim	Sim	Não
MetaFT	Alto	Sim	Sim	Sim
GroupPac	Alto ou baixo	Opcional	Sim	Sim

Tabela 3.1. Comparação das propostas.

O exemplo de implementação do MetaFT apresentado neste capítulo não é portável, pois usa o Electra e o Orbix+Isis, que também são soluções não portáveis. Conforme comentamos (item 3.2.5), o correto seria o MetaFT ser implementado tal como a figura 2.20, sem um ORB integrado a uma ferramenta de grupo, o que traria o benefício da portabilidade. O GroupPac apresenta uma boa portabilidade porque tanto a aplicação como seus objetos de serviço podem ser executados em outros ORBs, desde que estes ofereçam mecanismos de interceptadores.

[6] Sem alteração das especificações.

Tal como o Orbix+Isis e o Eternal, o MetaFT e o GroupPac permitem tanto comunicações através de uma ferramenta proprietária de comunicação de grupo como também, através do IIOP (figura 3.15), para fins de interoperabilidade. Além disso, o GroupPac pode ser inteiramente independente dessas ferramentas, desde que os protocolos de grupo e de tolerância a faltas sejam implementados como objetos de serviço se comunicando, exclusivamente, através do ORB (capítulo 4).

Em termos da conformidade com o padrão, nossas propostas não alteram o ORB e nem o compilador de IDL para gerar estruturas adicionais para tolerância a faltas, tal como ocorre com o FT-MOP e com o Phoinix.

3.4 Conclusão do capítulo

O MetaFT foi concebido com o propósito de desviar, de forma transparente, requisições clientes para um suporte de comunicação de grupo [Fraga97, Lau97, Lau99c]. O objetivo disso era obter um alto grau de transparência – o cliente não precisaria interagir diretamente com os mecanismos do suporte de grupo nas invocações para grupo de servidores. No MetaFT, os mecanismos de tolerância a faltas eram implementados segundo os conceitos da reflexão computacional. O paradigma reflexivo usado seguia o paradigma de meta-objetos [Kiczales91]. O gerenciamento do modelo de replicação definido nos protótipos não era implementado usando facilidades ou serviços de objeto CORBA, mas sim, faziam parte da aplicação na forma de meta-objetos. O modelo de implementação de replicações se mostrou bastante flexível, mudar de técnica de replicação se resumia a simples troca de meta-objetos.

Os objetos de serviço do *GroupPac* são arranjados no sentido de formar um suporte para tolerância a faltas e comunicação de grupo em ambientes abertos. A nossa proposta de grupo combina características das abordagens de serviço e de interceptação, apresentadas no capítulo 2. Todo o suporte a grupo está disponível na forma de objetos de serviço comum sobre o ORB e alguns desses serviços, envolvidos com a comunicação de grupo, são ativados de forma transparente, através do uso de interceptadores do ORB. Isto é, todas as invocações clientes para os objetos da aplicação são desviadas pelos interceptadores para um serviço de comunicação de grupo (na forma de objeto de serviço ou ferramenta de comunicação proprietária). O uso de interceptadores garante um alto grau de transparência

em relação aos serviços de grupo do *GroupPac* e um alto grau de portabilidade, uma vez que os interceptadores utilizados são mecanismos do próprio ORB. Em relação à interoperabilidade, o *framework* proposto pode se utilizar apenas das primitivas de comunicação do ORB (IIOP), o que o torna completamente interoperável com outras plataformas CORBA (capítulo 4).

Em um outro projeto, denominado de JaCoWeb (http://www.lcmi.ufsc.br/jacoweb), relacionado com segurança de sistema e CORBA, aplicamos as experiências alcançadas com o GroupPac para integrar tecnologias de segurança na arquitetura CORBA [Wangham01]. Novamente, os conceitos de interceptação, objetos de serviço e ferramenta de comunicação segura de baixo nível foram combinados, tal com no GroupPac, para prover comunicações seguras aos objetos CORBA de aplicação.

CAPÍTULO 4
CosNamingFT: Um Serviço de Nomes CORBA Tolerante a Faltas

4.1 Introdução

Conforme antecipado no capítulo anterior, apresentamos neste capítulo uma experiência no uso dos serviços do GroupPac para implementar um serviço de nomes tolerante a faltas, chamado de CosNamingFT [Lau99a, Lau99b]. O exemplo apresentado consiste em combinar os objetos de serviço do *GroupPac* para implementar a técnica de replicação primário/*backup*. A solução adotada segue a abordagem de interceptação usando mecanismos de ORB (item 2.4.3.3), e sem o uso de um suporte de comunicação de grupo (item 3.3.2). Neste caso, os mecanismos de interceptação do ORB são utilizados para invocar, de forma transparente para aplicação, os objetos de serviço GroupPac. Os objetos de serviço do GroupPac neste exemplo implementam algoritmos específicos e não mapeiam em ferramentas de mais baixo nível.

O serviço de nomes tem um papel muito importante em sistemas distribuídos. O seu principal objetivo é facilitar o acesso a recursos compartilhados. O serviço de nomes, se considerarmos o modelo de objetos distribuídos definido através das especificações CORBA, pode ser tomado como "um portão de acesso" aos diversos objetos em um sistema distribuído. Todos os objetos distribuídos têm acesso ao serviço de nomes pelo menos uma vez durante o seu ciclo de vida: ou para registrar sua referência de objeto ou para obter a referência de outros objetos do sistema. No contexto da OMG [OMG96], foi definido um *COSS* (capítulo 2) padronizando o serviço de nomes para o acesso via CORBA a objetos distribuídos. As especificações do serviço de nomes (*CosNaming*) [OMG97a] foram padronizadas de forma a considerar aspectos de portabilidade, interoperabilidade e reusabilidade que são conceitos importantes em sistemas abertos.

Essas especificações, entretanto, não contemplam requisitos de tolerância a faltas, fundamentais no caso de sistemas distribuídos.

Uma possível solução para garantir a tolerância a faltas do serviço de nomes seria a utilização do COSS *serviço de persistência* [OMG97a]. Essa especificação define uma interface única para serviços de armazenamento persistente de estados de objetos. A decisão de não seguir uma abordagem fundamentada no serviço de persistência é devido a fatores de desempenho. Um serviço de nomes com copias persistentes, quando da ocorrência de *crash,* necessita ser reiniciado e fazer com que recupere estados a partir de um arquivo, por exemplo.

Este capítulo, de certa forma, demonstra a utilidade de alguns dos objetos de serviços do GroupPac e ainda, de que é possível se conseguir soluções abertas para o problema de replicação de objetos distribuídos. Na seqüência desta seção apresentaremos: o padrão *CosNaming* da OMG para o serviço de nomes; uma descrição da arquitetura *CosNamingFT*; alguns aspectos de implementação e testes de desempenho realizados sobre o *CosNamingFT*; uma comparação do *CosNamingFT* com trabalhos relacionados presentes na literatura. Por fim, as conclusões deste capítulo.

4.2 Serviço de nomes do padrão CORBA

Um objeto pode ser localizado fazendo uso de seu *nome* e da sua *referência de objeto.* O nome de um objeto corresponde a um conjunto de caracteres, normalmente expressando uma qualidade ou característica associada ao mesmo. Na referência de objeto estão contidas as informações necessárias para a localização do objeto em um sistema distribuído. Nas especificações *CosNaming*, uma associação entre nome e referência de objeto é chamada de *name binding.* Um *name binding* é sempre definido dentro de um contexto de nomes (*name context*). Cada objeto CORBA *NamingContext* manipula um conjunto próprio de *name bindings* em que cada nome é único, mas que não impede, caso desejado, a associação de diferentes nomes a uma única referência de objeto. O termo *NamingContext* será usado no decorrer desse texto como sendo o objeto que implementa as funcionalidades do serviço de nomes definidas nas especificações *CosNaming.*

Na OMG, a referência de objeto é definida numa forma padrão conhecida como IOR (*Interoperable Object Reference*). A IOR [OMG96] é uma seqüência de caracteres que quando convertida para o formato adequado fornece as informações de endereço IP do sítio, a porta, um ponteiro para o objeto a ser acessado e algumas informações de controle relacionadas à própria IOR. Cada objeto distribuído deve ter sua IOR disponível, registrada em um *NamingContext*. Com isso, um cliente necessitando da IOR de um determinado objeto servidor deve, através do envio do nome do mesmo, obtê-lo no *NamingContext* desse objeto.

A figura 4.1 apresenta um diagrama temporal separando em fases os procedimentos no suporte dado pelo serviço de nomes (*NamingContext*) às interações cliente/servidor. Essas fases compreendem à iniciação do *NamingContext*, ao registro de uma IOR de objeto servidor no *NamingContext* (*binding*), e por último, à obtenção, de uma IOR de objeto servidor por parte de um cliente.

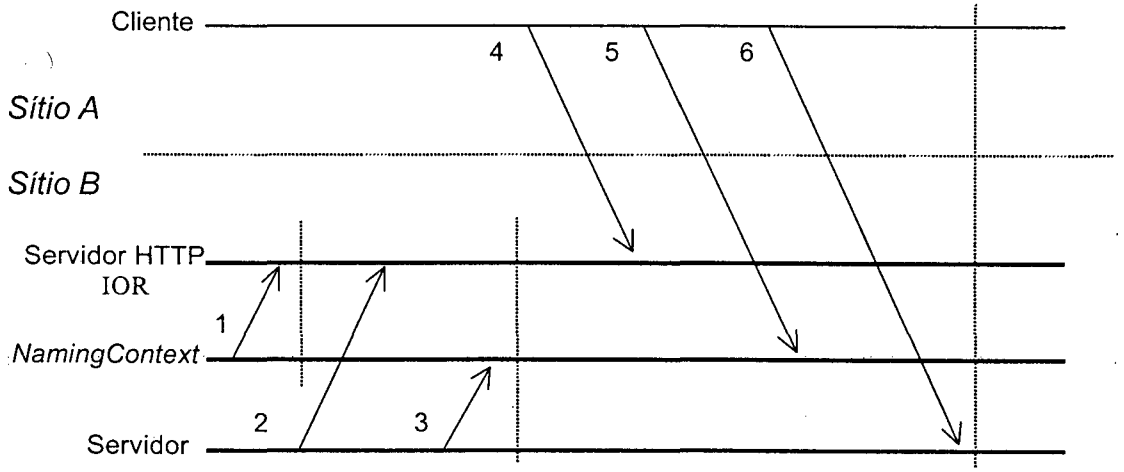


Figura 4.1. CosNaming e o suporte às interações cliente/servidor.

Um objeto *NamingContext* na sua iniciação (passo 1, figura 4.1) deve colocar sua própria IOR num repositório (por exemplo, um arquivo em um sistema de arquivos), tornando-a disponível a objetos CORBA em geral para que o possam acessar. O serviço de nomes, descrito nesse texto, em sua implementação, tem a sua IOR colocada num diretório com acesso através de um servidor HTTP (ex: http://www.lcmi.ufsc.br/~lau/ior) para facilitar invocações sobre esse serviço via *internet*. Uma vez o *NamingContext* em execução, os objetos CORBA podem registrar suas IORs e se tornarem disponíveis no sistema.

```
public class FooServer {
  public static void main(String args[]) {
    try {
      // obtendo o IOR do servidor de nomes
      NamingContext ncRef =

(NamingContext)ORB.resolve_initial_references("NameService");
                :
```

Figura 4.2. Obtendo o IOR do NamingContext.

Todo objeto servidor quando de sua iniciação registra sua IOR em um *NamingContext*. Para isso, o servidor precisa primeiro obter a IOR do *NamingContext* o que é feito através do seu método `resolve_initial_reference`, que tem a função de acessar o repositório (servidor de HTTP no nosso caso) para obter a IOR do *NamingContext* (passo 2 da figura 4.1). O código em Java que implementa este passo é apresentado na figura 4.2.

```
          :
      // ref - referência do objeto servidor Foo
      Foo ref = new FooServant();
      NameComponent path[] = {new NameComponent("Foo", "App")};
      // bind a referência de objeto no NamingContext
      ncRef.bind(path, ref);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
```

Figura 4.3. Registrando-se no NamingContext.

```
public class Client {
  public static void main(String args[]) {
    try{
      NamingContext ncRef =

(NamingContext)ORB.resolve_initial_references("NameService");
       // obtém a referência do objeto Foo (ref) a partir do seu
nome.
      NameComponent path[] = {new NameComponent("Foo", "App")};
      Foo ref = FooHelper.narrow(ncRef.resolve(path));
      // invoca o método m0 do objeto Foo
      ref.m0();
      System.exit(0);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
```

Figura 4.4. Interações do cliente para viabilizar invocações sobre um objeto servidor.

Após obter a IOR do *NamingContext*, o servidor da aplicação pode invocá-lo para se registrar e se tornar disponível no sistema (passo 3 da figura 4.1). O código da figura 4.3 mostra o novo nome do objeto a ser registrado na estrutura *NameComponent* consistindo de dois atributos: um identificador (ex: "Foo") e um tipo (ex: "App"). Uma vez definido o *NameComponent* o servidor chama o *NamingContext* para registrar o par nome e referência do objeto usando o método `bind`.

Os procedimentos de um cliente no sentido de viabilizar a sua comunicação com um objeto servidor são mostrados na figura 4.1, nas interações indicadas como 4, 5 e 6. Uma vez que um objeto servidor está disponível no sistema, o cliente para ter suas requisições de métodos executadas sobre esse servidor deve primeiro obter a IOR do *NamingContext* usando o seu método `resolve_initial_reference` (passo 4 da figura 4.1). Com a posse dessa IOR (disponível a partir do servidor de HTTP), o cliente usando o nome do objeto servidor a ser acessado monta o `NameComponent("Foo", "App")` e invoca o *NamingContext* para resolver esse nome (passo 5 da figura 4.1). Se o nome existir dentro do contexto indicado, o serviço retorna a IOR associada a esse nome enviado pelo cliente. A partir daí o cliente pode desempenhar as invocações no servidor (passo 6 da figura 4.1). A figura 4.4 apresenta o código que implementa as interações do cliente para viabilizar a sua comunicação com o objeto servidor.

4.3 Arquitetura do CosNamingFT

Os serviços oferecidos pelo *CosNamingFT* seguem às especificações do padrão COSS de serviço de nomes definido pela OMG [OMG97a], adicionando a estes atributos e características de tolerância a faltas. Na replicação de um objeto *NamingContext* os serviços do GroupPac são usados implementando as funcionalidades da técnica primário/*backup* [Budhiraja93]. A implementação do *NamingContext* replicado (*CosNamingFT*) usando esse modelo de replicação passiva, define um objeto *primário* (um *NamingContext* ativo) e as réplicas restantes como *backups* (objetos *NamingContext* passivos). As comunicações dos clientes com os serviços do *CosNamingFT* são todas realizadas com a réplica primária. Por razões de desempenho, a replicação usada é não bloqueante [Budhiraja93] no sentido em que o primário atualiza os objetos *backups* (*checkpoint*) após ter respondido ao cliente. Obviamente, isto admite a possibilidade de

estado inconsistente entre o primário e seus *backups*. No pior caso, um novo serviço pode ter sido registrado com o primário e, imediatamente após isto, acontece o *crash* do primário, sem que este tenha atualizado os *backups*. Em relação a este caso, assumimos um sistema em que a configuração dos serviços oferecidos à aplicação (quando todos os serviços se registram no servidor de nomes) é realizada no momento da sua iniciação. Desta forma, qualquer falha de registro de serviço pode ser facilmente detectada e corrigida na iniciação.

Os *backups* podem assumir o papel de primário em caso de falha desse último. Após o processamento de cada requisição de cliente e as respectivas atualizações de estado, todas as réplicas corretas, que formam o *CosNamingFT*, devem se manter com os mesmos *name bindings* (o estado do serviço).

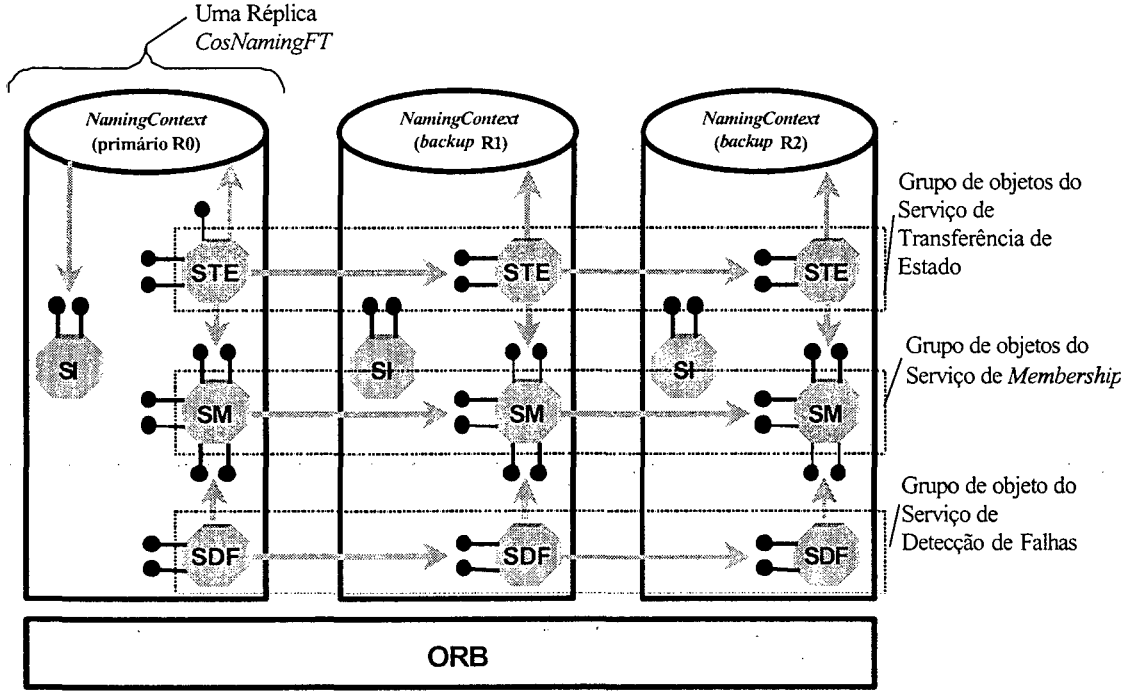


Figura 4.5. Composição dos blocos de serviço para formar a arquitetura do *CosNamingFT*.

Uma requisição do cliente, para obter a referência de um objeto servidor (IOR), no *NamingContext* primário é, em essência, uma operação *stateless*, o que significa que não haverá significante diferença entre o estado do *NamingContext* primário com os de seus *backups*. Portanto, se o primário falhar antes de responder aos seus clientes é indistinto tanto para abordagem bloqueante como não bloqueante. Neste caso, o cliente recebe uma

exceção e tenta se conectar ao novo primário para obter a informação desejada. A seguir é dada uma descrição de como cada objeto de serviço é incorporado na arquitetura *CosNamingFT* no sentido de fornecer o suporte necessário à replicação primário/*backups*.

Os objetos de serviço do *GroupPac* são incorporados no *CosNamingFT* atuando de forma transparente para a aplicação (figura 4.5). Cada réplica do *CosNamingFT* é composta pelos objetos *NamingContext*, SI, STE, SM e SDF e, em termos de modelo de execução, são conformados em um único processo. A falha de um destes objetos (*NamingContext*, SI, STE, SM ou SDF) é considerado como falha da réplica do *CosNamingFT* como um todo (*crash* da réplica). Os serviços STE, SM e SDF são compostos na forma de grupos de objetos de serviço. Por exemplo, levando em conta as conexões horizontais da figura 4.5, os objetos SM de cada réplica *CosNamingFT* formam o grupo que fornece serviço de *membership*.

O gerenciamento do grupo de réplicas *CosNamingFT* ocorre tal como descrito no item 4.3.1.1 do capítulo anterior, combinando os serviços SM e SDF. Quando ocorre o *crash* da réplica primária do *CosNamingFT*, o novo primário é escolhido entre as réplicas *backups*. A definição do novo primário é feita no anel virtual, recaindo sempre sobre o membro mais antigo do grupo (o sucessor do antigo primário). Nesse modelo de replicação passiva, apenas a réplica primária é ativa e interage com os objetos clientes. Portanto, somente a IOR da réplica primária do serviço de nomes precisa estar disponível, a partir de um repositório do sistema (no nosso caso o servidor HTTP, item 4.2), a todos os objetos do sistema. Na substituição do primário, o novo primário deve, na sua iniciação, registrar a sua IOR no servidor HTTP. A figura 4.6 exemplifica a detecção da falha de uma réplica primária e a sua substituição por uma das réplicas *Backup*. Nesse exemplo, R3 suspeita de R0 e ativa o método `leave_group` do objeto SM na réplica R1. Essa última réplica passa então a centralizar a execução do protocolo de *membership*.

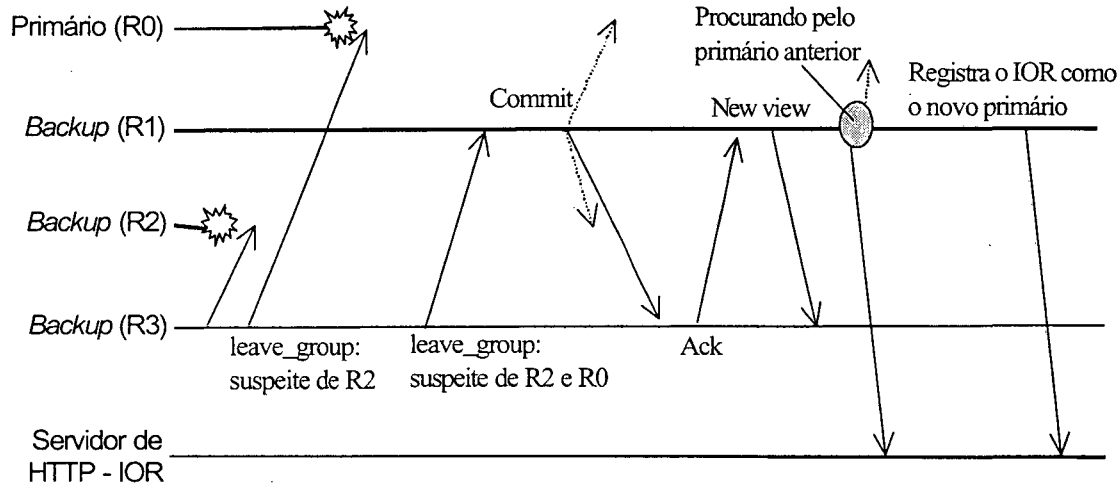


Figura 4.6. Situação: crash da réplica R2 e R0 e a escolha do novo primário.

A réplica primária é sempre aquela que tem a sua IOR registrada no repositório (servidor HTTP). Quando um novo primário é eleito, esse na sua iniciação lê o IOR disponível no servidor HTTP antes de registrar a sua referência de objeto (apagando a antiga IOR). De posse dessa IOR, a réplica tenta se comunicar com o primário anterior (aquele que supostamente está em *crash*). Caso receba resposta, a réplica que está tentando se registrar como novo primário, deve cessar todo o seu processamento de iniciação e se reintegrar ao grupo como uma réplica Backup. No caso de não haver resposta, a réplica se registra no servidor de HTTP se tornando efetivamente o primário do grupo (R1 da figura 4.6). O procedimento descrito evita a possibilidade de se ter duas réplicas primárias em um dado instante.

4.3.1 Serviço de transferência de estado

O serviço de transferência de estado (STE) do *GroupPac* é também incorporado às réplicas que formam o *CosNamingFT*, para garantir que cada *backup* mantenha seu estado consistente em relação ao primário. As transferências de estados do primário não são bloqueantes. Na figura 4.7, são apresentados os passos executados por esse serviço quando da transferência do estado do primário para os *backups* na invocação do método `bind` (operação de escrita).

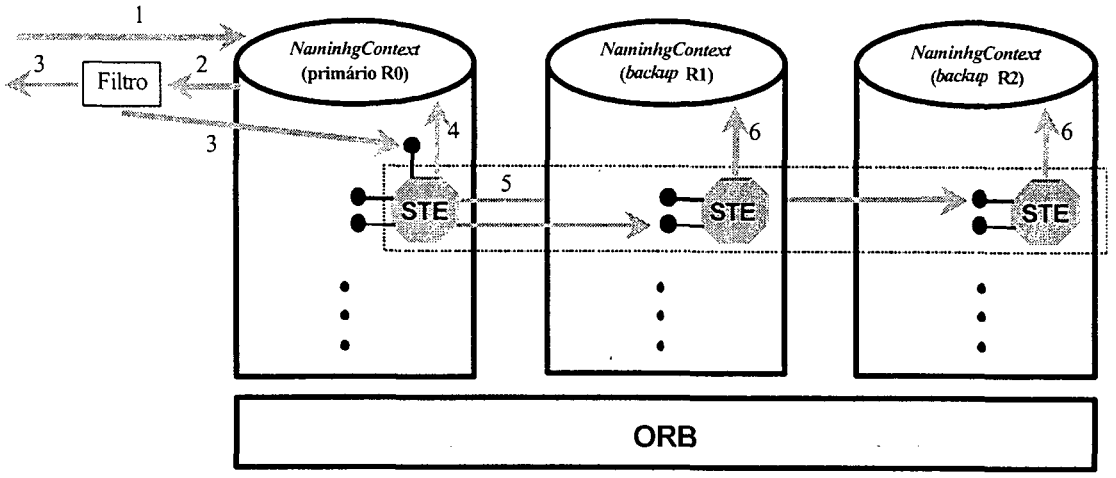


Figura 4.7. O serviço de transferência de estados.

Passos:

1. Invocação de um cliente sobre o serviço *NamingContextFT*, que deve ser executada sobre a réplica primária desse serviço;
2. A resposta correspondente à invocação é interceptada;
3. A referência do objeto interceptado é enviada ao serviço de transferência de estados para que este possa invocar o objeto *NamingContext* primário.
4. A resposta interceptada retoma ao seu caminho natural de envio ao cliente;
5. Com a referência do objeto *NamingContext* primário, o STE invoca esse objeto obtendo o seu estado (get_state);
6. Com as informações de estado, o objeto STE local difunde o estado do primário para os seus pares de grupo nas réplicas *backups*;
7. Cada objeto STE local ao receber o estado do primário executa o *upcall* set_state no sentido de atualizar o objeto *NamingContext* de sua réplica.

4.4 Implementação do CosNamingFT

Discutiremos nesta seção alguns aspectos relativos à implementação de cada um desses blocos de construção que compõem a arquitetura do *CosNamingFT*. A implementação do *CosNamingFT* adota as especificações COSS/OMG referente a serviço de nomes (*CosNaming*). Nenhuma extensão ou modificação é feita na interface padrão, como mostrado na figura 4.8. A inclusão dos serviços do *GroupPac* também em nada

altera as especificações COSS/OMG. Todos os serviços deste pacote são implementados em *Java* e suas interfaces especificadas segundo o padrão IDL/OMG.

4.4.1 Estrutura geral

A implementação do *CosNamingFT* envolve dois componentes de *software* codificados em Java: o *NamingContextServant.java* e o *NamingContextServer.java.* O componente *NamingContextServant.java,* por sua vez, implementa as operações do *CosNaming* descritas nas especificações IDL. A interface do *NamingContext,* implementada nesse componente de *software,* é apresentada na figura 4.8, sendo basicamente constituída pelas operações de conectar (*bind*), reconectar (*rebind*) ou desconectar (*unbind*) um nome a um objeto, operações de recuperar um objeto através do seu nome (*resolve*), e ainda de operações para criar, destruir ou listar um contexto de nomes. A essas operações implementadas no *NamingContextServant.java,* são adicionadas ainda operações de *upcall* como o `get_state` e o `set_state`, necessárias para o fluxo de informações entre os objetos de serviço CORBA do *GroupPac* e o serviço de nomes.

O componente *NamingContextServer.java* implementa as funcionalidades que envolvem a criação e iniciação dos objetos que compõem as réplicas no *CosNamingFT.* A implementação do *NamingContextServer.java* segue de uma forma geral as especificações, adicionando ainda códigos para o registro da IOR do objeto primário no servidor HTTP e, também, para a iniciação do objeto SI (serviço de iniciação) para que este crie e inicie os objetos de serviço do *GroupPac* na réplica correspondente.

```
module CosNamingFT {
   typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
   typedef sequence <NameComponent> Name;
   enum BindingType { nobject, ncontext };
         :
   interface NamingContext {
         :
      void bind(in Name n, in Object obj)
       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind_context(in Name n, in NamingContext nc)
```

```
            raises(NotFound, CannotProceed, InvalidName);
        Object resolve(in Name n)
            raises(NotFound, CannotProceed, Inva lidName);
        void unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context(in Name n)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void destroy()
            raises(NotEmpty);
             :
};
```

Figura 4.8. A IDL do CosNaming

4.4.2 A IDL dos serviços do GroupPac

As IDLs dos serviços do *GroupPac* são apresentadas na figura 4.9. Cada uma dessas interfaces será representada por um objeto de serviço em tempo de execução. Cada réplica do objeto de aplicação é mapeada juntamente com os objetos de serviço necessários para o suporte a grupo. Como cada serviço do *GroupPac* é um conjunto de objetos CORBA, as comunicações entre os mesmos são também através do ORB. Na nossa implementação, no sentido de facilitar, o gerenciamento das referências de objeto que envolve esses serviços, é definido como repositório das mesmas o serviço de *membership* (cada objeto SM que compõe este grupo mantém uma cópia do conjunto dessas referências na estrutura `AllServRefs`). O acesso a essas referências de objeto é local e garantido através de herança da interface *MembershipService* (figura 4.10). Com isso, cada objeto de serviço desejando realizar uma interação com seus pares complementares de grupo pode fazer através da obtenção da *IOR* (*Interoperable Object Reference*) junto ao objeto SM localizado no seu processo réplica.

A estrutura `ServiceRefs` é utilizada pelo objeto SI no momento em que uma nova réplica solicita a entrada no grupo replicado. Após iniciar os objetos de serviço, o objeto SI invoca o método `join_group` do objeto SM enviando a estrutura `ServiceRefs`, que contém as referências de cada objeto de serviço (fd, st e ms da figura 4.10), pertencente ao processo solicitante (nova réplica). Após o protocolo de *membership* ter sido executado e decidido que o novo processo pode ser inserido no grupo, o objeto SM da réplica R0 difunde o novo `AllServRefs` para todos os objetos SM do grupo invocando o método `view_change`. O método `leave_group` é invocado no SM da réplica R0 pelos objetos do grupo de detecção de falhas (SDF) na detecção de um *crash* de uma réplica do grupo.

```
module GroupPac {
  typedef sequence<any> State;
  typedef sequence<string> MembersCrash_seq;
  // Membership service IDL interface - SM
  interface MembershipService {
    struct ServiceRefs {
      Object fd;
      Object st;
      MembershipService ms;
    };
    struct AllServRefs {
      sequence<string> id_seq; // id: Identifier
      sequence<Object> fd_seq; // fd: Failure Detector
                               //     Reference
      sequence<Object> st_seq; // st: State Transfer Reference
      sequence<MembershipService> ms_seq; // ms: Membership
                                          // Reference
    };
    boolean join_group(in ServiceRefs servRefs,
                  out string rank);
    boolean leave_group(in MembersCrash_seq membersCrashId);
    void view_change(in short newRank, in short
                new_view_number);
    boolean ack(in string memberAck);
  };
  // Failure Detector IDL interface - SDF
  interface FailureDetector: MembershipService {
    void keep_alive();
  };
  // State Transfer IDL interface - STE
  interface StateTransfer: MembershipService {
    void get_state(in Object obj, in string stg);
    void set_state(in short id, in State state);
  };
};
```

Figura 4.9. A IDL dos serviços utilizados do GroupPac.

A implementação dos objetos detectores de falhas (SDF) se fundamenta basicamente no método `keep_alive`. Cada membro do grupo SDF invoca esse método no membro anterior da seqüência de anel virtual do grupo (item 3.3.1.1). A detecção de *crash* ocorre quando uma exceção do CORBA indicando falha de comunicação é sinalizada.

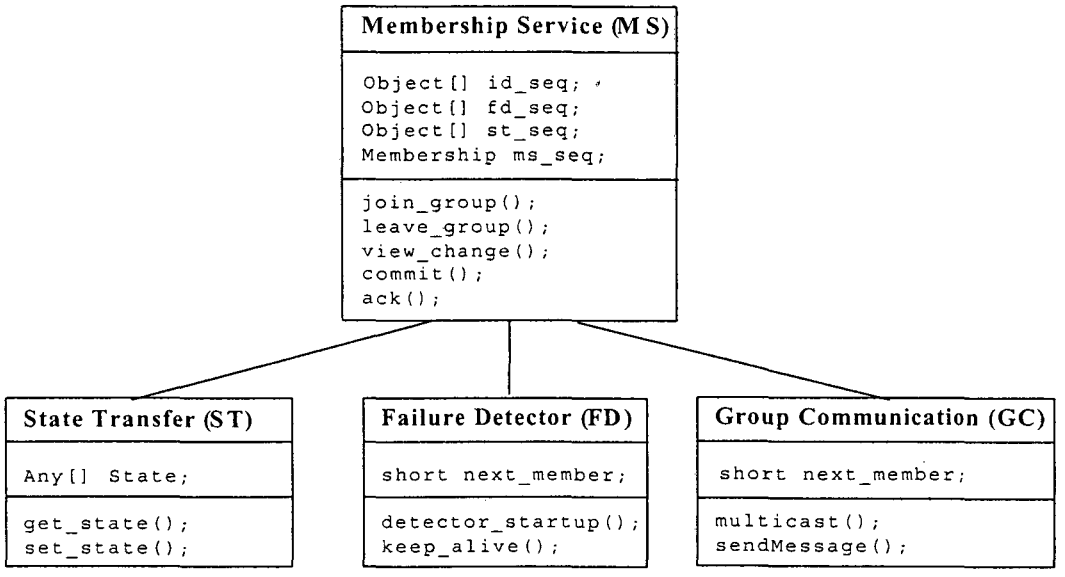


Figura 4.10. A hierarquia de interfaces no GroupPac.

4.4.3 Detalhes de implementação do STE

Em relação às transferências de estado para os *backups*, executadas através dos objetos STEs, é preciso que se definam quais as informações da réplica primário são necessárias para manter a consistência entre réplicas no modelo *primário/backups*. Em cada objeto *NamingContext*, os pares *name binding*, formados pela associação de cada nome (*NameComponent*) à referência de objeto (IOR) correspondente, são guardados na variável `context`. Essa variável `context`, implementada usando o *hashtable* do pacote java.util.hashtable, constitui o estado da réplica primária definido para as transferências de estados nos casos em que novos *backups NamingContext* são inseridos dinamicamente. Transferências do primário após processamentos referentes a operações que correspondem a atualizações do contexto de nomes (como *bind*, *rebind* e *unbind*, por exemplo), resultam em informações de estado que se resumem a apenas um *name binding*.

Os métodos `get_state` e `set_state` (implementados no *NamingContext-ServantFT*) têm a função de converter/desconverter o estado de um objeto para/de um formato de dados padrão definido para transferência de estado através dos objetos STEs. Devido ao fato de `context`[1] ser um tipo *hashtable*, não definido no CORBA, a solução adotada foi convertê-lo em uma seqüência de *bytes*. Essa conversão do `context` para

[1] Estrutura de dados que contém um conjunto de *naming binding* que implementa o estado de um *NamingContext*.

uma seqüência de *bytes* possibilita a manipulação do mesmo como um tipo `Any` (tipo suportado pelo CORBA). Ou seja, quando o objeto STE invoca em *upcall* o método `get_state` do objeto *NamingContext* (passo 5 da figura 4.7), esse método pega o `context` e separa cada *name binding* contido nele, convertendo-os em seqüências de *bytes*. Na figura 4.11 é dado um exemplo onde um `context`, com dois *name bindings*, é mostrado em formato de seqüência de *bytes*. Essa seqüência de *bytes* é definida da seguinte maneira: o primeiro *byte* descreve o número total de *name bindings* (*2*), o *byte* seguinte define o número de *bytes* do primeiro *NameComponent* (*120*), em seguida os *bytes* do *NameComponent* (NC1). Depois, um outro *byte* define o número de *bytes* da primeira referência de objeto (*150*), em seguida os *bytes* da referência de objeto (RefObj1) e assim sucessivamente para os *name binding* subseqüentes.

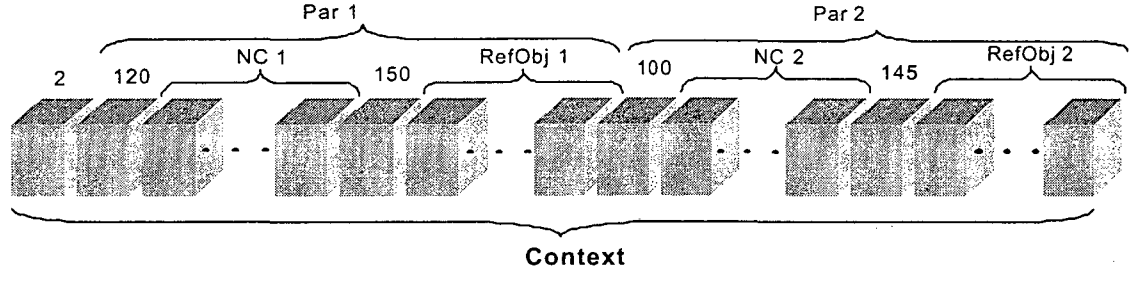


Figura 4.11. Exemplo de um context em uma seqüência de bytes.

Para a implementação da interceptação descrita na figura 4.7 (passo 2) é usado o mecanismo de filtro fornecido pelo *OrbixWeb*. O filtro é um dispositivo que provoca o desvio transparente do fluxo normal de uma invocação de método. Esse desvio pode ser usado para adicionar funcionalidades ou controles a uma chamada de método. O serviço de transferência de estados (grupo de STEs) do *CosNamingFT* é ativado usando o filtro em uma interceptação *post-marshaling* para pegar a resposta do objeto servidor e extrair a referência do objeto *CosNaming* da réplica primária, isso tudo de maneira transparente.

4.5 Desempenho do CosNamingFT

Neste item apresentamos as medidas de desempenho do *CosNamingFT*, realizadas em nosso laboratório, com o sentido de validar nossa proposta. Essas medidas foram realizadas a partir das invocações dos métodos *bind* e *resolve*, portanto, registrando e obtendo a referência de objeto, respectivamente. O desempenho do *CosNamingFT* foi

levantado considerando diferentes graus de replicação. O ambiente considerado nesses testes de desempenho foi constituído de uma rede local (com 10 Mbps *Ethernet*) heterogênea usando duas máquinas Sun Ultra 1 com Solaris 2.5, uma Axil 240 também com Solaris 2.5, um Pentium 100 e um Pentium 233 MMX, ambos usando Linux e por fim, um Pentium 233 MMX com Windows 95.

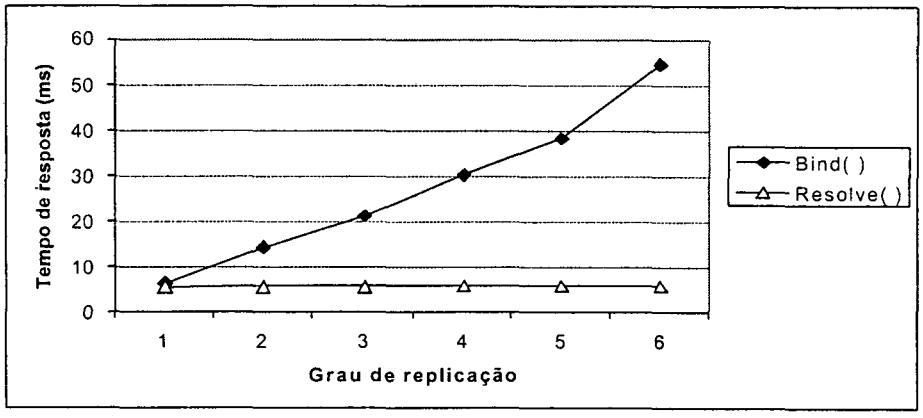


Figura 4.12. Desempenho do CosNamingFT.

Os testes realizados consistiram de cem (100) chamadas da operação *bind* e também o mesmo número de chamadas da operação *resolve*, todas executadas sobre o *CosNamingFT*. Na figura 4.12, os tempos de resposta (eixo Y) pelo grau de replicação (eixo X) foram determinados considerando a média dessas cem invocações, nos dois tipos de chamadas. A curva referente à operação *bind* representa um aumento no tempo de resposta à medida que se cresce o grau de replicação do *CosNamingFT*. Esse resultado é explicado pelo fato de que essa operação altera o estado do objeto *NamingContext* primário e que torna necessário, portanto, o envio da atualização de estado às réplicas *backups*. A operação *resolve* apresenta um desempenho praticamente constante em função do grau de replicação. Por ser uma operação de apenas leitura, a operação *resolve* não envolve necessidade de envio de atualizações de estado aos *backups* e, portanto tem o desempenho inalterado mesmo quando o número de réplicas cresce.

Nos experimentos realizados com a operação *bind*, o fator de crescimento do tempo de resposta se manteve praticamente constante e de valor aproximado a 8 milissegundos por réplica adicionada ao *CosNamingFT*. Esse fator de crescimento na curva do *bind* é válido para até cinco réplicas. Para um grau de replicação envolvendo seis réplicas, o tempo de resposta medido foi de 54.6 milissegundos. No caso, a taxa de crescimento maior

referente a essa sexta réplica foi creditada ao fato que essa sexta réplica foi implementada em uma plataforma bem mais lenta que as demais (um Pentium 100). Se considerarmos um serviço de nomes usual, um grau de replicação considerado razoável para esse serviço seria de três réplicas [Maffeis96]. Necessidades de graus maiores ou ainda reposições nesse serviço poderiam usar o suporte que permite uma variação dinâmica no número de réplicas – novas réplicas podem ser inseridas dentro do grupo durante a execução do serviço. Considerando a natureza dessa aplicação, o tempo de resposta de 21.2 milissegundos obtido neste experimento (*bind*) para um grau de replicação três pode ser tomado como bastante aceitável.

4.6 Trabalhos relacionados ao CosNamingFT

Em [Maffeis96] é apresentada uma proposta de um serviço de nomes tolerante a faltas usando conceitos CORBA. A técnica de replicação, usada nesse trabalho, é baseada na abordagem de replicação ativa: todas as réplicas processam as requisições e a réplica mais rápida retorna a resposta ao cliente. A solução apresentada nesse artigo não oferece interoperabilidade. É dependente de um suporte de comunicação de grupo de baixo nível tal como Isis [Birman91] ou Horus [Renesse95], constituindo-se, dessa maneira, em uma solução baseada em suportes proprietários.

Em [Maffeis96] são também apresentadas medidas de desempenho para a abordagem de replicação usada no serviço de nomes. Nos experimentos citados, a operação *bind* apresenta uma taxa de crescimento do tempo de resposta por réplica adicionada em torno 6 milissegundos, quando o suporte de baixo nível usado é o Isis. Se o Horus for utilizado como serviço de baixo nível essa taxa fica em torno de 1 milissegundo por réplica por réplica adicionada. O conjunto de máquinas usadas nessas medidas foram cinco *SPARCstation* 10 e quatro *SPARCstation* 20, tendo todas o sistema operacional SUNOS 4.1.3 e considerando as mesmas sobre uma rede de 10 Mbps *Ethernet*. Apesar das dificuldades para efetuar as comparações devido as diferentes opções de máquinas, sistema operacional, técnica de replicação etc., adotados em [Maffeis96], podemos afirmar que o uso da abordagem de replicação ativa baseada num suporte de comunicação de grupo otimizado é certamente uma justificativa para o melhor desempenho em relação ao nosso trabalho. Mas, por outro lado, podemos citar como uma das vantagens do *CosNamingFT* a

sua conformidade com as recomendações de objetos de serviço COSS da OMA. As soluções adotadas no *CosNamingFT* estão baseadas nos conceitos de sistemas abertos.

4.7 Conclusão do capítulo

As extensões ao *CosNaming* propostas neste trabalho para torná-lo tolerante a faltas em nada altera o padrão deste serviço definido pela OMG. Os serviços do *GroupPac* são inseridos de forma separada e transparente no *CosNamingFT*, a utilização ou não desses objetos de serviço é definido no código que implementa o objeto SI. Isto é, a utilização ou não desses serviços não implica na necessidade de qualquer modificação no código que implementa o serviço de nomes em si. Também, seguindo as recomendações da OMG, todas as comunicações entre os objetos de serviço são através do ORB (segundo o protocolo IIOP) sem o uso de qualquer mecanismo de comunicação que não esteja em conformidade com esse padrão.

As implementações apresentadas neste capítulo vieram também no sentido de verificar a viabilidade de se implementar mecanismos de tolerância a faltas em nível de aplicação sobre um *middleware* CORBA baseada em objetos de serviço. A abordagem implementada segue todas as recomendações da OMG no documento *Request for Proposal (RFP)* para tolerância a faltas, no sentido da definição de serviços e facilidades para a construção de aplicações tolerantes a faltas.

O *OrbixWeb* foi usado nessas implementações, mas isso também não implica na perda de transportabilidade dos códigos da aplicação para outras plataformas CORBA/Java. Além disso, a utilização do *Java* como linguagem de programação garante a portabilidade dos serviços desenvolvidos, permitindo que o mesmo conjunto de *byte-code* gerado do *CosNamingFT* pudesse ser utilizado em diferentes plataformas de máquina e sistemas operacionais como foi feito efetivamente neste trabalho. As medidas de desempenho vieram no sentido de reforçar a aplicabilidade do *CosNamingFT* como uma arquitetura para prover um serviço de nomes tolerante a faltas.

O *CosNamingFT* foi desenvolvido dentro do contexto do projeto Sistema Nile[2], da universidade do Texas - Austin, com o objetivo de oferecer um serviço de nomes tolerante a faltas para os objetos que compõe esse sistema. A implementação deste serviço pode ser utilizada sem qualquer restrição sobre qualquer plataforma CORBA permitindo, por exemplo, que objetos CORBA desenvolvidos na linguagem C++ ou outras possam também utilizar este serviço.

Com este capítulo finalizamos a primeira parte desta tese de doutorado. Até agora, apresentamos todas nossas experiências, antes da padronização do FT-CORBA, no sentido de propor soluções para integrar suporte de grupo e tolerância a faltas na arquitetura CORBA. As implementações apresentadas no capítulo anterior e neste vieram no sentido de verificar a viabilidade de se implementar mecanismos de tolerância a faltas e de comunicação de grupo, em nível da aplicação, sobre a plataforma aberta CORBA. Conforme apresentado na introdução desta tese, aspectos relacionados com escalabilidade serão tratados na segunda parte desta tese, já considerando as especificações FT-CORBA padrão OMG.

[2] O Nile (*National Challenge Computing Project*) é uma plataforma baseada em CORBA para processamento distribuído e paralelo em redes de larga escala (www.nile .utexas.edu).

CAPÍTULO 5
As Especificações FT-CORBA e a Adequação do GroupPac a estes novos Padrões

5.1 Introdução

Por parte da OMG, a preocupação em introduzir mecanismos de tolerância a faltas no CORBA começou, de fato, em Outubro de 1998, a partir da publicação de um documento RFP (*Request for Proposal*) no sentido de convidar empresas e instituições de pesquisa a proporem soluções para introduzir suportes de tolerância a falta no CORBA. Portanto, os trabalhos de introdução da tolerância a faltas nos padrões CORBA são recentes e, seguindo os calendários da OMG, as especificações FT-CORBA foram publicadas em Abril de 2000. Estas especificações definem um conjunto de serviços essenciais para a construção de aplicações confiáveis em ambientes abertos. Os serviços especificados correspondem basicamente: ao de gerenciamento de réplicas (*membership*), ao de detecção de falhas, ao de notificação de falhas, ao de *logging* e *checkpoint*, e ao serviço de recuperação de réplicas falhas. Esses serviços são disponíveis na forma de objetos de serviço COSS (*Common Object Services Specificaction*), possuindo suas interfaces definidas em IDL/CORBA (*Interface Definition Language*). Entretanto, essas especificações deixam em aberto alguns pontos importantes relacionados à tolerância a faltas, permitindo então, extensões proprietárias.

Este capítulo apresenta um estudo sobre as especificações de tolerância a faltas no CORBA. De inicio, são descritos os principais requisitos definidos no RFP (*Request for Proposal*) FT-CORBA. São apresentadas, na seqüência, as especificações finais do FT-CORBA, fruto do esforço do comitê de tolerância a falta da OMG no sentido de alcançar a padronização do FT-CORBA. Em seguida, apresentamos uma discussão sobre a adaptação

do GroupPac a essas novas especificações, juntamente com alguns aspectos e decisões feitas durante o processo de implementação. Por último, as conclusões deste capítulo.

5.2 Requisitos FT-CORBA

O documento RFP (*Request for Proposal*) publicado pela OMG definia um conjunto de requisitos delineando as propostas que foram submetidas à OMG com o objetivo de introduzir tolerância a faltas na arquitetura CORBA. Segundo essa RFP, os serviços de tolerância a faltas devem apresentar funcionalidades para o gerenciamento de replicações, mascaramento de falhas e o tratamento de componentes faltosos. No caso de aplicações requererem controles adicionais no gerenciamento de faltas, suportes para notificação de falhas, de configuração e de controle em operações de restauração, propostas seriam também aceitas.

Os requisitos desta RFP se dividiam em gerais e específicos para tolerância a faltas. Os primeiros são adotados em todas as especificações CORBA, tratando principalmente aspectos relacionados a sistemas abertos, tais como portabilidade, reusabilidade e interoperabilidade. Dentre estes requisitos, podemos citar alguns aspectos impostos sobre serviços CORBA:

- As propostas devem ser apresentadas na forma de interfaces em conformidade com o padrão IDL/OMG e no estilo de programação CORBA;

- As propostas devem ser precisas e funcionalmente completas. Não devem ter interfaces, operações ou funções implícitas ou escondidas para habilitar a implementação de uma especificação proposta;

- As propostas devem, sempre que possível, reutilizar especificações OMG existentes, o que inclui o CORBA, os serviços de objetos e as facilidades comuns, para evitar definir novas interfaces que desempenham funções similares;

- As propostas devem justificar e especificar completamente qualquer mudança ou extensão que for necessário nas especificações existentes. O

que inclui mudanças e extensões nos protocolos Inter-ORB necessários para interoperabilidade. Em geral, a OMG é a favor que as propostas apresentem o mínimo de mudanças e extensões nas especificações existentes;

- As propostas devem considerar o aspecto de flexibilidade de implementação. Não devem ser incluídas descrições de implementação, porém podem especificar restrições nos procedimentos do objeto que a implementação precisa levar em conta sobre aquelas definidas pela semântica da interface;

- As propostas devem ser substituíveis e interoperáveis, permitindo assim que implementações independentes sejam possíveis. Uma implementação deve ser substituível por uma implementação alternativa sem requerer mudanças na implementação de qualquer cliente.

Embora o OMG adote especificações de interface, será levada em conta a viabilidade técnica da implementação durante o processo de avaliação das propostas. Nesse processo, critérios de desempenho, portabilidade, segurança e conformidade também devem ser considerados.

Em relação aos requisitos específicos para tolerância a faltas, no RFP são apresentados os requisitos obrigatórios e os opcionais. Os requisitos obrigatórios são os seguintes:

- As propostas definirão a unidade de redundância como sendo o objeto. Os objetos são replicados, e as réplicas de um objeto formam um grupo de objetos;

- As propostas especificarão interfaces para permitir a constituição de replicação passiva;

- As propostas especificarão interfaces para auxiliar na sincronização de estado dentro de um grupo de réplicas durante operação normal, ou em processo de restauração após uma falha;

- As propostas especificarão interfaces para a admissão de um novo membro e assegurar que os estados de todas as réplicas sejam equivalentes antes de retomar o processamento do serviço;

- As propostas especificarão interfaces em que a replicação seja transparente aos clientes, o que inclui mascarar para o cliente as falhas e a recuperação de membros no grupo replicado;

- As propostas especificarão interfaces para permitir que os grupos de réplicas determinem quais membros são operacionais, falhos e, quando aplicável, o primário (réplica ativa principal na replicação);

- Propostas especificarão as interfaces necessárias para controlar a recuperação de réplicas faltosas.

Conforme citado, os requisitos obrigatórios definem propostas voltadas para modelos de replicação passiva. O modelo de replicação ativa é solicitado na RFP na forma de requisitos opcionais:

- As propostas podem especificar interfaces para permitir a constituição de replicação ativa;

- As propostas podem especificar interfaces para coletar dados estatísticos de falhas e prover relatórios destas estatísticas;

- As propostas podem especificar interfaces para detecção de falhas e notificação, permitindo aos membros restantes, um gerente ou o cliente, executar as operações necessárias para o gerenciamento de falhas;

- As propostas podem especificar interfaces que permitam suprimir respostas múltiplas de um grupo replicado para uma única requisição, ou simplesmente deixem o cliente coletar todas as respostas;

- As propostas podem especificar interfaces para permitir a supressão de requisições redundantes de um grupo de replicação ativa para outro grupo de replicação;

- Propostas podem especificar interfaces para assegurar que uma réplica faltosa é terminada efetivamente sem causar efeitos colaterais prejudiciais.

Além destes requisitos, na RFP foram apresentados alguns aspectos a serem discutidos durante o processo de padronização. Essas discussões se referem aos impactos que as propostas possam causar nas especificações OMG como um todo. Alguns aspectos de discussão são:

- Como serão usados os objetos de serviço existentes para satisfazer as exigências;
- Quais os objetos de serviço existentes poderiam se tornar tolerante a faltas usando a proposta;
- As propostas devem discutir como os *bindings* serão criados e usados, como a transparência de operação será alcançada, e como a notificação de falhas será feita aos clientes registrados e terceiros;
- As propostas devem discutir como os clientes e servidores podem residir em ORBs heterogêneos, o que inclui o aspecto da interoperabilidade;
- Discutir os impactos na transparência de localização das réplicas;
- Discutir como as interfaces propostas resolvem o problema de particionamento de grupo;
- As propostas que apresentem qualquer interface de QoS discutirão a relação destas interfaces com a interface de QoS do serviço de mensagem CORBA;
- Se for o caso, discutir como um gerenciamento de falhas proprietário é usado para implementar as funcionalidades da interface proposta;

Critérios de avaliação determinariam o quanto as propostas satisfazem às exigências especificadas. A evolução da padronização desses serviços de tolerância a faltas seguiu um calendário preciso da OMG.

5.3 As especificações FT CORBA

Diante dos requisitos apresentados na seção anterior, várias empresas e centros de pesquisas atenderam ao RFP (*Request for Proposal*) [OMG98a]. E, como resultado desses esforços, recentemente, a OMG publicou uma especificação padrão para introduzir tolerância a faltas na arquitetura CORBA (FT-CORBA) [OMG00a]. As especificações FT-CORBA definem um conjunto de serviços essenciais para o desenvolvimento de aplicações confiáveis em sistemas abertos. A tolerância a faltas no CORBA é obtida através da replicação de objetos, técnicas de detecção e recuperação de falhas. Nessa primeira especificação foi apresentado um conjunto de interfaces e protocolos que podem ser separados em três módulos: Gerenciamento de Replicação (SGR, item 5.3.1.3); Gerenciamento de Falhas (SGF, item 5.3.1.4); e Gerenciamento de Recuperação e *Logging* (SRL, item 5.3.1.5); além das definições para Interoperabilidade (item 5.3.1.2). O *Serviço de Gerenciamento de Replicação* é constituído pelos serviços de gerenciamento de propriedades, gerenciamento de grupo de objeto e fábrica genérica. Para o *Serviço de Gerenciamento de Falhas* são definidas as interfaces de detecção de falhas, notificação de falhas e análise de falhas. Finalmente, no *Serviço de Gerenciamento de Recuperação e Logging* são definidos os mecanismos para transferência de estado de objeto e recuperação de réplicas faltosas. Os objetos de serviço de gerenciamento de replicação, notificador de falhas e detector de falhas de *host* são replicados.

5.3.1 Arquitetura FT-CORBA

Na figura 5.1, é apresentada a arquitetura de tolerância a faltas do CORBA. Nela são apresentados os objetos de serviço que fornecem, no *middleware*, as funcionalidades básicas para a construção de aplicações tolerantes a faltas. Segundo as especificações, o *Serviço de Gerenciamento de Replicação* é o responsável pelo gerenciamento de grupo. Esta tarefa é delegada ao seu *Serviço de Gerenciamento de Grupo de Objetos*, exercendo um controle dinâmico, nas entradas e saídas (normal ou por falha) de objetos replicados, de um grupo através da manutenção de uma lista atualizada de seus membros (*membership*). O objeto *Serviço de Gerenciamento de Replicação* utiliza o objeto *Fábrica Genérica* no processo de criação ou remoção de replicas de um grupo. Nesse processo, o objeto *Fábrica Genérica* interage com os objetos fábrica locais (representantes locais do primeiro) para a

criação ou remoção de réplicas nas diferentes estações de um sistema distribuído. Além disso, temos o *Serviço de Gerenciamento de Propriedades* onde são definidas e manipuladas as propriedades de tolerância a faltas de cada grupo de objetos sob controle do *Serviço de Gerenciamento de Replicação.* As propriedades definem, basicamente, como cada grupo deve ser gerenciado e controlado pelos serviços do FT-CORBA. Por exemplo, uma das informações mantidas no gerenciamento de propriedades é a definição da técnica de replicação implementada em um grupo, que registra se o mesmo pode funcionar como: *Replicação Passiva Fria*, *Replicação Passiva Morna* e *Replicação Ativa* [OMG00a].

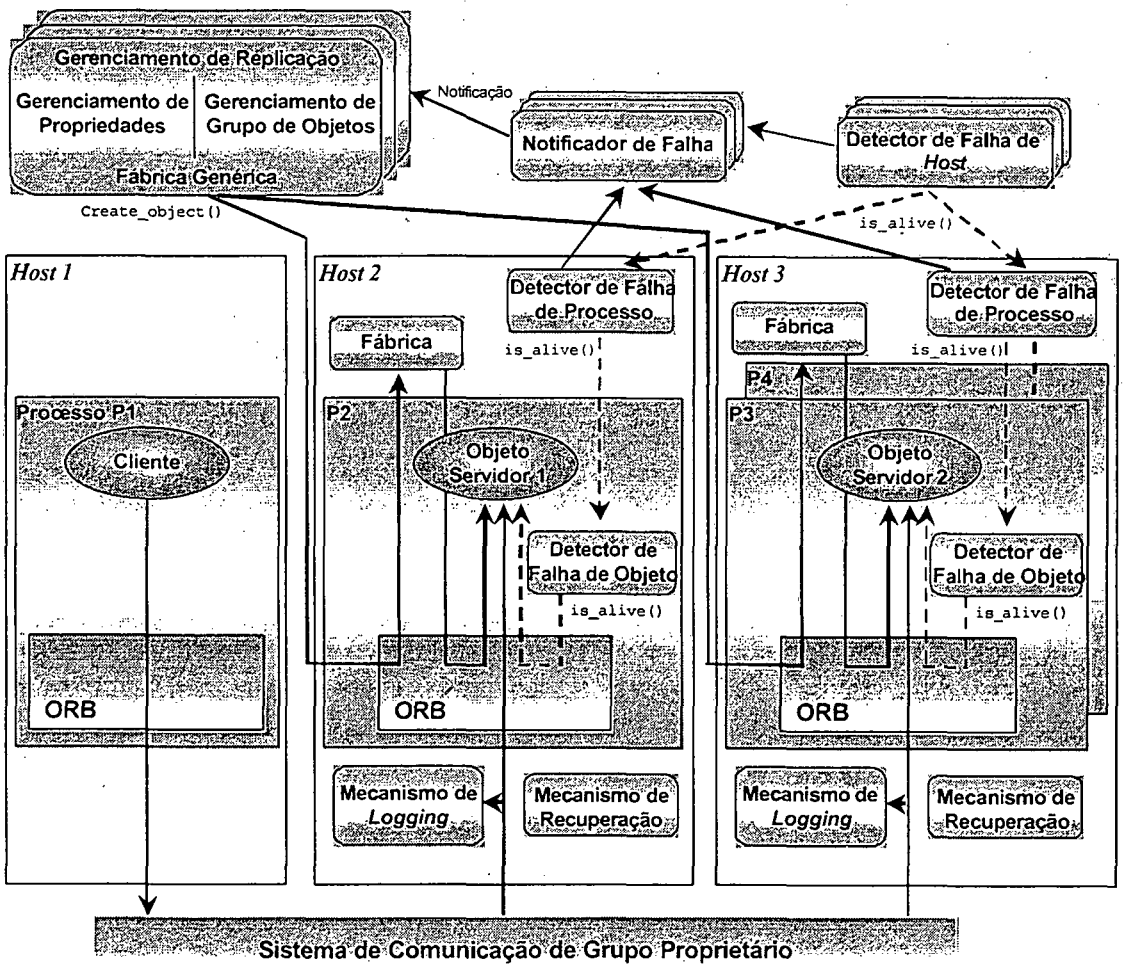


Figura 5.1. Arquitetura de tolerância a faltas CORBA.

Todas as comunicações entre os objetos são através do ORB e, caso necessário, com o auxílio de uma ferramenta de comunicação de grupo. O sistema de comunicação de grupo fornece ao *middleware* CORBA os mecanismos de comunicação de grupo confiável para o suporte às técnicas de replicação – oferecendo alguma garantia de atomicidade e

ordem na entrega das mensagens. Nos itens subseqüentes, é apresentado, em mais detalhes, cada um desses objetos de serviço da arquitetura de tolerância a faltas CORBA.

No *Serviço de Gerenciamento de Falhas*, a detecção (ou monitoramento) de falhas é dividida em três níveis: objeto, processo e *host*. Os detectores de falhas nos três níveis citados são baseados em mecanismos de *timeout*. O detector de falhas em nível de *host* é mantido replicado no sentido de garantir a continuidade do serviço mesmo em presença de falhas. O *Serviço Notificador de Falhas* é também replicado e tem a função de enviar mensagens de notificação ao gerente de replicação, a partir dos registros de falhas enviados pelos detectores de falhas. Esses registros são necessários ao *Serviço de Gerenciamento de Replicação* para atualização das listas de membros.

O *Serviço de Logging*[1] e *Recuperação* inclui, basicamente, três funções: registrar (fazer uma cópia) as requisições recebidas pelo primário (objeto servidor 1), atualizar o estado das réplicas que se juntam ao grupo e recuperar réplicas faltosas. Este serviço atua nos eventos de falha do objeto primário e na inclusão de uma nova réplica no grupo. Por exemplo, em um evento de falha do primário o *Serviço de Recuperação* envia ao novo primário as requisições enviadas ao primário anterior a partir do último *checkpoint*[2].

5.3.1.1 Domínios de tolerância a falta

As especificações FT-CORBA introduziram o conceito de domínio de tolerância a faltas com o objetivo de possibilitar o uso otimizado dos serviços de tolerância a faltas. Conceitualmente, um domínio de tolerância a faltas é constituído de um conjunto de grupos de objetos replicados, ou seja, é composto de grupos, que se apresentam dispostos em *hosts*, formados por processos e objetos. Segundo as especificações FT-CORBA, um objeto não pode pertencer a mais de um grupo e domínio de tolerância a faltas. Por outro lado, não há limitação quanto a *hosts* pertencerem a diferentes domínios (figura 5.2). Além disso, cada domínio deve ter, obrigatoriamente, o conjunto de serviços FT-CORBA (SGR, SGF e SLR, figura 5.1) atuando sobre cada grupo de objetos dentro deste domínio de tolerância a falta.

[1] *Logging* é o mecanismo pelo qual as invocações efetuadas em um membro de um grupo são armazenadas para que sejam re-executadas em outro membro em caso de falha.
[2] *Checkpoint* é o termo usado para definir uma cópia do estado de um objeto.

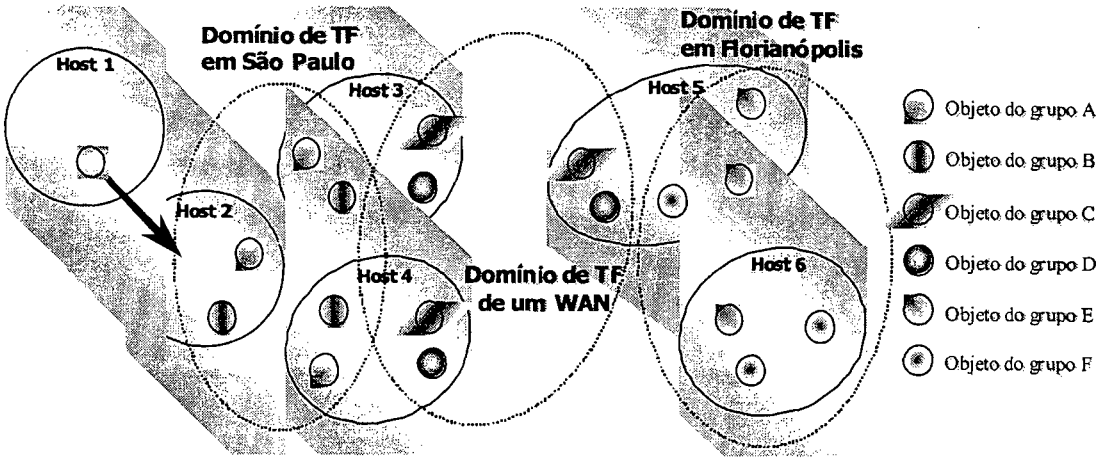


Figura 5.2. Diferentes domínios de tolerância a faltas do FT-CORBA.

5.3.1.2 Interoperabilidade

A OMG, como é de sua característica, se limita em definir interfaces de serviço genéricas, tentando atender diferentes abordagens de tolerância a faltas. Por exemplo, protocolos de comunicação de grupo confiável, base fundamental para a implementação de técnicas de replicação em sistemas distribuídos, não são padronizados – como era de se esperar, pela complexidade dos mesmos e pela quantidade de algoritmos envolvidos. A OMG enfatiza, neste caso, o uso de soluções proprietárias (figura 5.1). Todavia, para garantir um mínimo grau de interoperabilidade no tratamento com grupos, os ORBs devem adotar a IOGR (*Interoperable Object Group Reference*): uma referência interoperável de grupo de objetos definida em [OMG00a]. A IOGR corresponde a uma extensão da IOR (*Interoperable Object Reference*), referência a um objeto simples. Uma IOGR permite a um cliente referenciar um grupo de objetos como uma entidade única. De forma genérica, esta referência de grupo contém o conjunto de IORs de cada membros de um grupo de objeto em seus campos.

Na figura 5.3, é apresentada a estrutura de uma IOGR, a qual contém múltiplas etiquetas, chamadas de TAG_INTERNET_IOP *Component*. Essa etiqueta, que tem a mesma estrutura de uma IOR convencional para um objeto único, contém as informações de versão, endereço IP do *host*, a porta, um ponteiro para o objeto membro a ser acessado, e algumas informações de controle relacionadas à própria etiqueta, chamada de *components*. Cada objeto membro definido na IOGR pertence a um grupo de objetos e essa indicação é dada em *components*. A estrutura da *components* apresenta uma etiqueta

TAG_PRIMARY *component*, definida em apenas um membro do grupo na IOGR, e que determina quem é o objeto primário do grupo. Uma etiqueta TAG_FT_GROUP *Component* contém um conjunto de informações do grupo de objetos – fornece um perfil do grupo de objetos referenciado. As informações contidas nessa etiqueta são o identificador do domínio de tolerância falha (FTDomainID), o identificador do grupo de objeto (ObjectGroupId) e o número de versão da IOGR (ObjectGroupRefVersion) – de maneira a identificar unicamente um grupo dentro do domínio de tolerância a faltas. Finalmente, no *Multiple Components Profiles* é definida de forma direta a TAG_FT_GROUP que a IOGR representa.

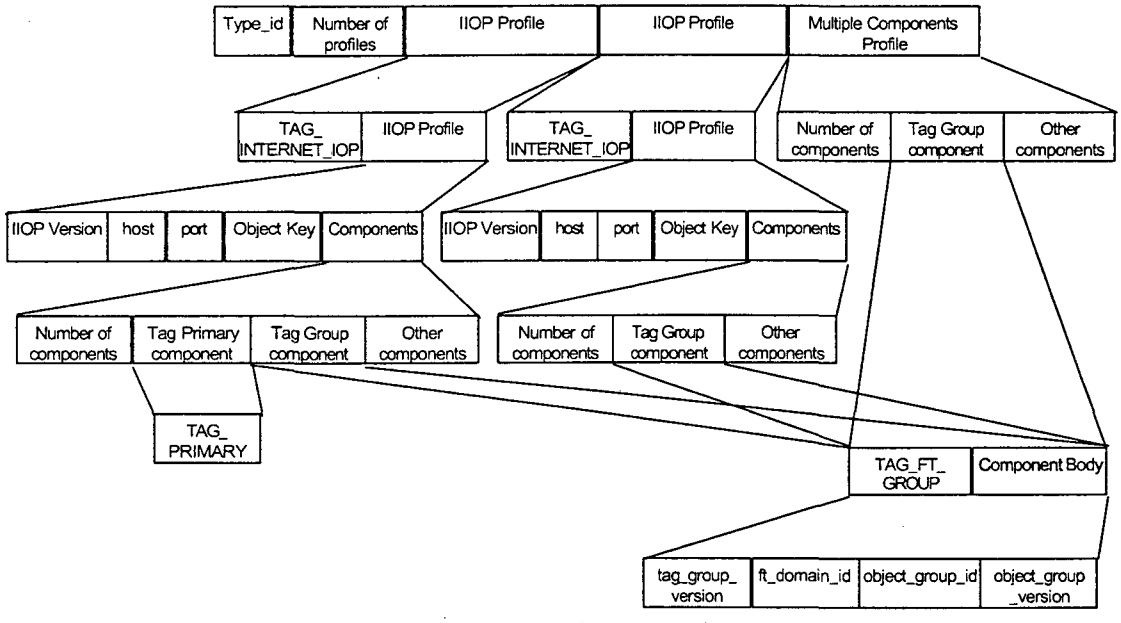


Figura 5.3. Estrutura de uma referência de grupo de objetos interoperável (IOGR).

O suporte da implementação deve permitir a construção de uma IOGR a partir de um conjunto de referências de membros (IOR) e a modificação dos parâmetros da TAG_FT_GROUP.

5.3.1.3 Gerenciamento de replicação

O objeto de Serviço Gerenciamento de Replicação (SGR) é um importante componente da arquitetura de tolerância a falta (figura 5.1). Herda as funcionalidades dos objetos de Serviço de Gerenciamento de Propriedades (SGP), Serviço de Gerenciamento de Grupo de Objetos (SGG) e Serviço Fábrica Genérica (SFG) para oferecer um completo suporte para gerenciamento de replicação de objeto. No entanto, o SGR, em si, oferece

apenas operações relacionadas com o notificador de falha. Basicamente, gerencia a referência de grupo de objetos (IOGR) do notificador de falhas para que esta possa ser obtida pela aplicação ou pelos outros serviços da arquitetura. Contudo, as especificações permitem que a interface do SGR possa ser estendida, com soluções proprietárias, com métodos similares para os outros componentes da arquitetura FT-CORBA.

Gerenciamento de propriedades

O SGR herda as funcionalidades do Serviço de Gerenciamento de Propriedades (SGP), o que torna possível ao SGR definir as propriedades de tolerância a faltas dos grupos de objetos criados. Segundo as especificações, o serviço de gerenciamento de propriedade mantém um conjunto de propriedades de tolerância a faltas associada a cada grupo de objetos, assim definidas:

- *ReplicationStyle*: define a técnica de replicação utilizada pelo grupo de objeto, assim classificada:

  - *Stateless*: estilo de replicação em que o estado do grupo de objetos não é alterado pelas requisições clientes. Por exemplo, um servidor que oferece acesso apenas para leitura a uma base de dados;

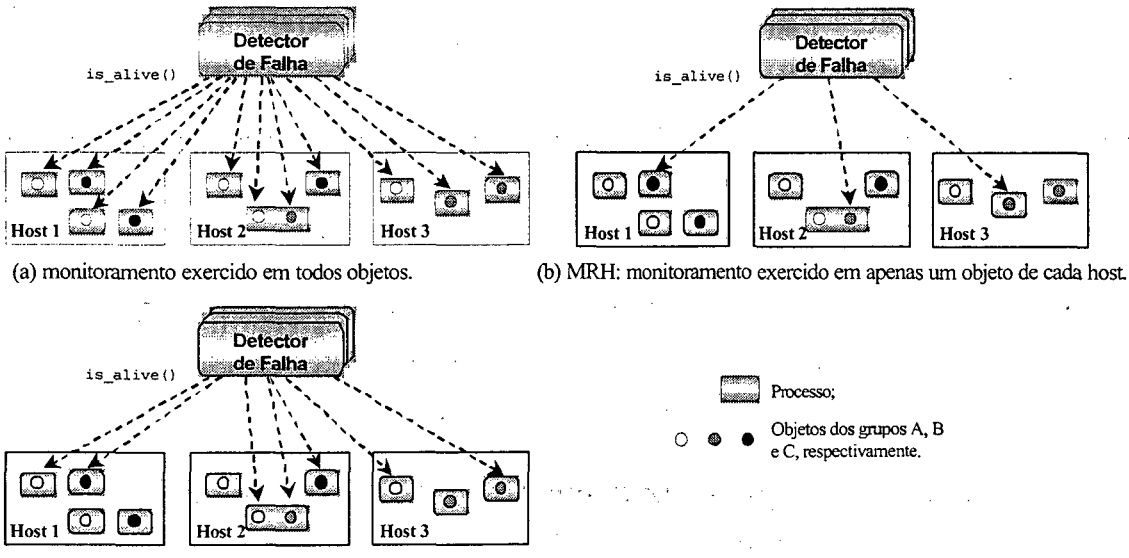
  - Replicação passiva morna: apenas um membro (o primário) executa o método invocado na interface de serviço. As outras réplicas operam como *backups* em que os métodos invocados não são executados. No entanto, o estado do primário é transferido (*checkpoint*), periodicamente, para os *backups*. Na falha do primário, um dos *backups* é escolhido para ser o novo primário, este tem seu estado sincronizado através da execução de requisições extraídas do *log*, na ordem dos seus registros de salvamento, após o último *checkpoint*.

  - Replicação passiva fria: apenas um membro (o primário) da replicação executa o método requisitado. As réplicas *backups* não recebem nenhum tipo de mensagem ou *checkpoint* – ficam em modo espera (*stand by*). Em caso de falha do primário, uma réplica *backup* é ativada para assumir o papel de nova primária. A réplica ativada

tem o seu estado atualizado a partir do último *checkpoint* e com a execução de requisições extraídas do *log*, na ordem dos seus registros de salvamento, após o este último *checkpoint*;

- Replicação ativa: todos os membros executam, independentemente, o mesmo método invocado na interface do serviço. Neste caso, falhas eventuais são mascaradas – as réplicas não faltosas produzem o resultado requerido adaptado à técnica de ajuste usada na coleta das respostas do servidor replicado – esta técnica de replicação ainda não está especificada no FT-CORBA;

➢ *FaultMonitoringGranularity*: define o tipo de monitoramento de falha, se sobre cada membro individual de um grupo (figura 5.4a); se sobre um membro representativo de um *host H* (*MRH* – figura 5.4b) contendo mais de um objeto – a falha do membro representativo é considerado como a falha de todos os objetos pertencente ao *host H*, independente se é membro do mesmo grupo ou não. E finalmente, o *MRGH* em que o monitoramento é exercido sobre um membro representativo de um grupo *G* no *host H* (figura 5.4c);



(a) monitoramento exercido em todos objetos.

(b) MRH: monitoramento exercido em apenas um objeto de cada host.

(c) MRGH: monitoramento exercido em apenas um objeto membro de um grupo em um host.

Figura 5.4. Tipos de monitoramento de falha no FT-CORBA.

- *MembershipStyle*: define para o gerenciamento de replicação (*membership*) se a criação de novas réplicas é feita em nível de aplicação (MEMB_APP_CTRL) de forma não transparente ou em nível de suporte (MEMB_INF_CTRL), usando os objetos fábrica (figura 5.1);

- *ConsistencyStyle*: define se os procedimentos de transferência de estado (*checkpoint*), *logging* e recuperação são realizados em nível de aplicação (CONS_APP_CTRL) ou através dos serviços oferecidos na arquitetura de tolerância a faltas no CORBA (CONS_INF_CTRL);

- *FaultMonitoringStyle*: são definidos dois atributos, o *PULL* em que o objeto detector de falhas envia periodicamente mensagens para o objeto monitorado para verificar se este está ativo. E o *PUSH* em que o próprio objeto envia mensagens periodicamente ao detector de falhas para indicar que está ativo;

- *Factories*: fornece as informações para criar outros objetos no domínio de TF. Esta propriedade contém os seguintes parâmetros: a IOR de uma fábrica local, a localização onde deve criar, remotamente, um membro de um grupo de objetos e os critérios usados para criar esse membro;

- *InitialNumberReplicas*: define o número de réplicas de um objeto a ser criado inicialmente;

- *MinimumNumberReplicas*: define o número mínimo de réplicas em um grupo para manter o grau de tolerância a falta desejado;

- *FaultMonitoringIntervalAndTimeout*: define o intervalo de monitoramento (*ping*) e o tempo de resposta (*timeout*) do objeto monitorado para determinar se está faltoso;

- *CheckpointInterval*: determina o intervalo de tempo entre cada atualização de estados.

Essas propriedades para grupos de objetos devem ser estabelecidas estaticamente em cada domínio de tolerância as falhas. No entanto, algumas delas podem ser

estabelecidas e modificadas dinamicamente durante a execução da aplicação (por exemplo, as três últimas propriedades apresentadas na lista).

Gerenciamento de grupo de objetos

O serviço de gerenciamento de grupo (SGG) oferece métodos que permitem criar e registrar as referências de grupo de objeto (IOGR) e procedimentos para disponibilizar essas IOGRs para a aplicação ou para os outros serviços da arquitetura FT-CORBA. O SGG gerencia, de forma dinâmica, todos grupos de objeto dentro de um domínio de tolerância a faltas. Esse gerenciamento dinâmico significa que o número de membros de um grupo (*membership*) pode mudar dinamicamente. Essas mudanças podem ocorrer pela entrada de um novo membro ou pela saída de um membro do grupo (normal ou por falha). Na criação de novas réplicas, o objeto de serviço gerente de replicação (SGR) usa o objeto fábrica genérica (um COSS) para criar novas cópias do objeto servidor da aplicação. O serviço de gerenciamento de grupo de objetos se encarrega de gerenciar as diferentes versões de uma IOGR, modificadas a cada atualização de *membership*. Além disso, o SGG fornece métodos para a localização de membros em um grupo de objetos.

Fábrica genérica

Este serviço permite ao SGR criar e remover grupos de objetos, réplicas (membros de grupo de objetos) e objetos não replicados. A interface da fábrica genérica é implementada pelos objetos fábricas bcais situados um em cada *host* (figura 5.1). Desta forma, o SGR pode invocar os objetos fábricas locais para criar membros individuais de um grupo de objetos e, também, permitir à aplicação invocar os objetos fábricas locais para criar objetos simples (não replicados).

5.3.1.4 Gerenciamento de falhas

O serviço de gerenciamento de falhas (SGF) é responsável pela atividade de detecção de falhas, notificação de falhas e, análise e diagnose de falhas (figura 5.1). Os detectores de falhas detectam falhas em objetos e reportam para um notificador de falhas. O notificador de falhas recebe os registros de falhas dos detectores, filtram as informações contidas nos registros, e propaga, como eventos, os registros filtrados para os consumidores que estão subscritos no notificador de falhas. O analisador de falhas faz a atividade de filtragem,

consultando os registros de falhas mantidos pelo notificador de falhas, gerando um registro filtrado para enviar de volta ao notificador de falhas (figura 5.5).

Segundo as especificações, o serviço de detecção a falhas é dividido em três níveis: detectores de falhas em nível de objeto, de processo e em nível de *host* (figura 5.1). Os detectores de falhas nos diferentes níveis são baseados em mecanismos de *timeout*. São dois os estilos de monitoramento de falha disponíveis: o *Pull* e o *Push* (este ainda não especificado) – que diferem na direção em que as informações de falha transitam no sistema.

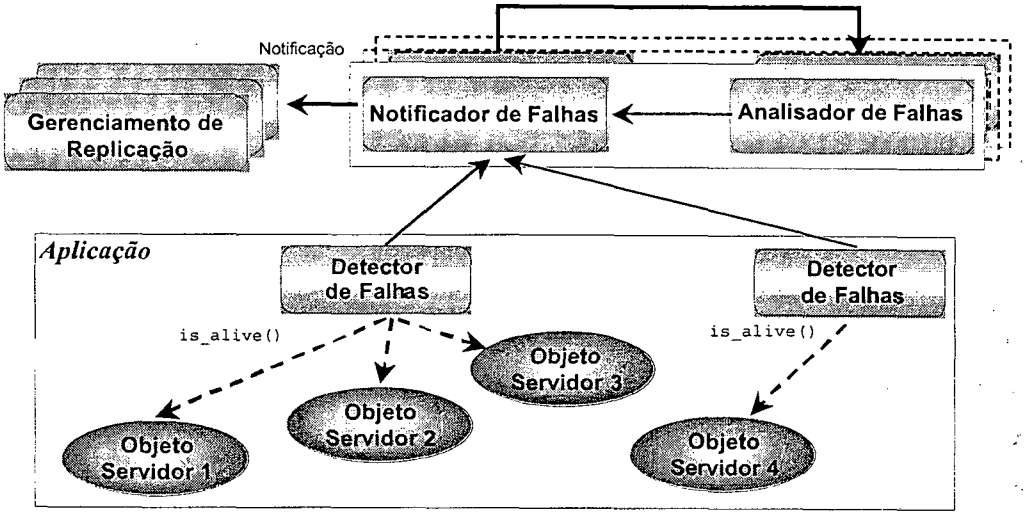


Figura 5.5. Interação entre detectores de falha, notificador de falhas e analisador de falha.

A detecção de falhas em nível de *host* se utiliza de replicações, ou seja, possibilita que vários detectores monitorem um mesmo *host*. Isto se explica pela complexidade na reposição de serviços de uma estação. Se for para restaurar funcionalidades atribuídas a uma estação – o que sempre representa altos custos, é necessário que haja uma boa probabilidade na correção de uma detecção. Em termos de sistemas de larga escala, as especificações aconselham um arranjo de detectores de falhas estruturados em forma hierárquica, com diferentes domínios de detecção.

O serviço notificador de falhas tem a função de enviar mensagens de notificação ao serviço de gerenciamento de replicação (SGR), a partir dos registros de falhas enviados pelos detectores de falhas dos três níveis. Com isso, o SGR, a partir destas notificações, pode atualizar as IOGRs de grupo. Além disso, as notificações de falhas podem ser propagadas, além do gerenciador de replicação, para qualquer objeto de serviço ou da

aplicação. Basta que estes se subscrevam como consumidores de eventos do serviço de notificador de falhas.

A função do analisador de falhas vai além da atividade de descartar registros duplicados e desnecessários. Cada analisador de falhas pode, também, coletar registros de falhas, fornecidos pelo notificador de falhas, e desempenhar a função de verificar se há uma correlação entre os registros. Desta forma, o analisador de falhas pode condensar um número grande de registros de falhas relacionados em um único registro de falha. Por exemplo, o *crash* de um *host* pode provocar registros de falhas de todos os objetos naquele *host*, bem como um registro da falha do *host* em si.

5.3.1.5 Gerenciamento de logging e recuperação

O serviço de gerenciamento de *logging* e recuperação (SLR) deve garantir a transferência de estado em situações de recuperação de objetos faltosos, atualização de estado de objetos *backup* (nos modelos de replicação passiva) e a transferência de estado para os novos membros que se juntam à replicação. Em um grupo de replicação passiva, durante uma operação normal, o mecanismo de *logging* registra o estado (*checkpoint*) e as ações (os métodos executados) da réplica primária em um *log*[3] (figura 5.6). O *log* deve preservar a ordem em que as requisições (mensagens) foram recebidas pela réplica primária. Deste modo, as requisições contidas dentro do *log* podem ser re-executadas, na ordem correta, durante o processo de recuperação. O mecanismo de *logging* mantém um *log* distinto para cada grupo de objeto. No entanto, os *logs* dos diferentes grupos podem ser registrados num mesmo *log* físico (ex: um disco físico). O mecanismo de *logging* pode ser disposto de forma distribuída, um para cada objeto replicado (figura 5.1).

Em caso de falha do objeto primário, o mecanismo de recuperação é utilizado para reaver os registros guardados no *log* para restaurar o estado de um objeto membro *backup* (figura 5.8), de tal modo que este possa substituir o objeto primário faltoso. Segundo as especificações, nenhuma interface é definida para este serviço porque estes mecanismos não são acessíveis diretamente pela aplicação. São definidas duas interfaces (*Checkpointable* e *Updateable*) que os objetos da aplicação devem herdar. A primeira classe permite a transferência de *checkpoint* (o estado mais recente do primário que foi salvo no *log*) para ser usado: em replicação passiva fria, na obtenção do estado de um

[3] Mecanismos que registra o conjunto de requisições (mensagens) que chegam no servidor.

primário faltoso por parte de um *backup* (o novo primário, figura 5.8); e em replicação passiva morna e replicação ativa quando da entrada de um novo membro no grupo. A interface *Updateable* é utilizada para transferência de estado parcial na replicação passiva morna – em situações em que um *backup* já possui um estado inicial e é atualizado periodicamente, por exemplo, a cada mudança de estado do objeto primário (figura 5.7).

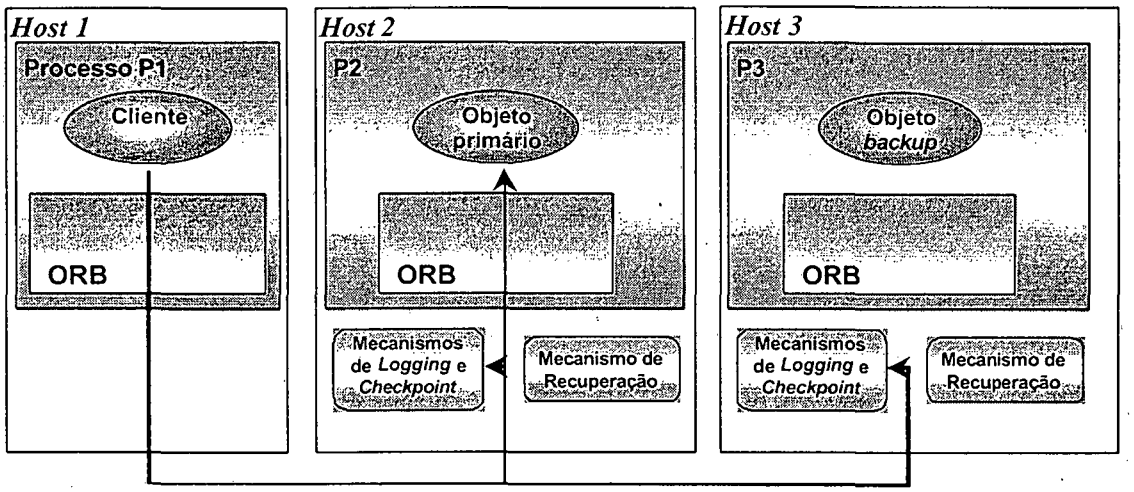


Figura 5.6. Operação normal: registro das requisições em um log.

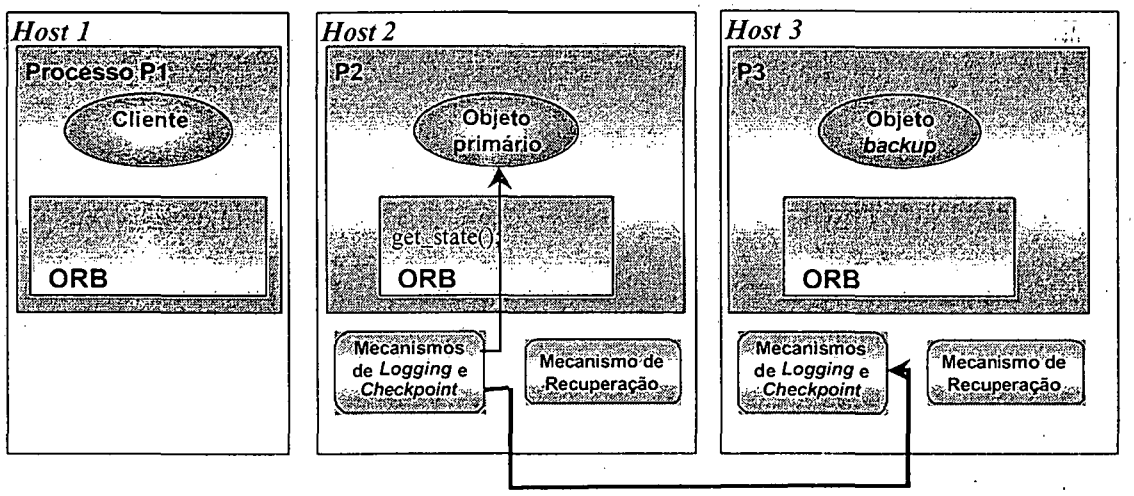


Figura 5.7. Registro de *checkpoint*: atualização de log.

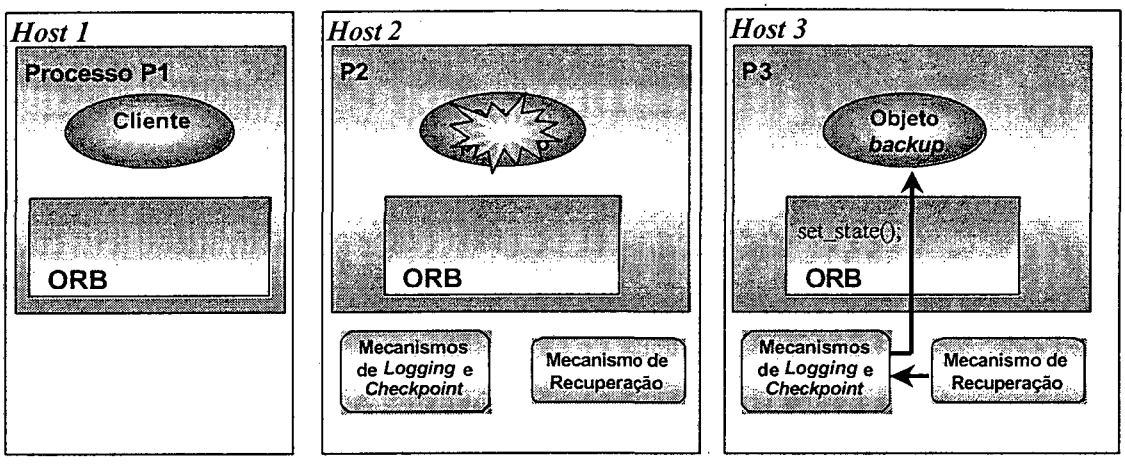


Figura 5.8. Recuperação: atualização do novo primário.

5.3.2 Considerações sobre o FT -CORBA

Um padrão que tenta atender a todos os requisitos de um amplo espectro de aplicações (diversificação) pode satisfazer essas necessidades pobremente ou, pode ser muito complexo para implementar [OMG00a]. Portanto, o comitê da OMG responsável pela padronização do FT-CORBA optou, para esse primeiro momento, por assumir apenas alguns compromissos. Em particular, tratar alguns aspectos de interoperabilidade entre implementações de ORB, com suporte para tolerância a faltas, de diferentes fabricantes de software. Todavia, é previsto pelo grupo responsável pelo FT-CORBA que, ao longo de um processo de amadurecimento, novas extensões para tolerância a faltas (como interfaces e protocolos) e melhor interoperabilidade sejam agregadas a esta especificação presente [OMG00a]. Além disso, a OMG aconselha que outras necessidades, além daquelas definidas nas especificações FT-CORBA, podem ser atendidas, no momento, por soluções proprietárias.

Do ponto de vista da "flexibilidade versus interoperabilidade" temos que a máxima flexibilidade é alcançada quando o número de interfaces e protocolos padronizados é o menor possível, garantindo maior liberdade aos desenvolvedores para definir outros mecanismos de acordo com suas necessidades. De modo contrário, a máxima interoperabilidade é alcançada quando se tem um grande número de interfaces e protocolos padronizados. Uma vez que todas as plataformas sigam o mesmo conjunto de especificações, teremos maior probabilidade de aplicações, desenvolvidas sobre estas diferentes plataformas, interoperarem entre si.

Portanto, as especificações FT-CORBA, assim como os outros serviços COSS, tiveram sempre a preocupação de ponderar as relações "diversificação versus implementação" e "flexibilidade versus interoperabilidade". Isto justifica a simplicidade das especificações FT-CORBA, descritas neste capítulo.

5.4 Adequação do GroupPac de acordo ao FT-CORBA

Conforme apresentamos nos capítulos 3 e 4, o GroupPac, na sua primeira versão, foi concebido bem antes da padronização do FT-CORBA, bem como as outras propostas apresentadas no capítulo 2. Fazendo uma comparação do GroupPac com o FT-CORBA foi concluído que os serviços oferecidos, por ambos, são similares – os serviços do GroupPac de *Membership* (SM), Detecção de falhas (SDF) e Transferência de Estado (STE) são equivalentes aos serviços SGR, SGF, SLR do FT-CORBA, respectivamente. O Serviço de Comunicação de Grupo (SCG) não possui equivalência porque ainda não foi especificado um serviço deste porte pelo comitê de padronização do FT-CORBA.

Entretanto, a plena adequação do GroupPac às especificações FT-CORBA encontrou diversos problemas. A principal dificuldade está na abordagem de fornecer os serviços de tolerância a faltas definida pela OMG. No GroupPac os objetos de serviços conviviam com os objetos da aplicação num mesmo processo (mesmo espaço de endereçamento), e os mecanismos de tolerância a faltas eram executados em cada réplica de um grupo (abordagem simétrica). A vantagem disso, conforme apresentado no capítulo 3, era a flexibilidade, que permitia combinar diferentes serviços para compor diferentes *frameworks*. De maneira oposta, os objetos de serviço do FT-CORBA operam segundo a abordagem assimétrica, em que objetos de serviço (dedicados) e objetos de aplicação estão em espaços de endereçamento distintos. Isto é, um objeto de serviço que pode estar em outro processo, ou mesmo, em outro *host* (figura 5.1). A abordagem adotada pelo FT-CORBA tem a vantagem de otimizar o uso dos serviços de tolerância a faltas – um único conjunto de serviços atuando sobre todos os grupos do domínio, deste modo, minimizando o uso dos recursos do sistema (memória, processamento, comunicação, etc).

Diante destas considerações, a nova versão do GroupPac foi quase que completamente re-implementada, na sua segunda versão seguindo as especificações FT-CORBA. Basicamente, o que foi reaproveitado foram alguns códigos de detecção de falhas e de transferência de estado usados no suporte às interfaces do SGF e SLR do FT-CORBA,

respectivamente. Além disso, o Serviço de Comunicação de Grupo (SCG) foi o único que teve seu modelo de implementação mantido, tal como apresentado no capítulo anterior, usando interceptadores e uma ferramenta proprietária na comunicação de grupo. Nos próximos itens, apresentamos alguns aspectos dessas experiências de implementação do FT-CORBA.

5.4.1 Implementação do FT -CORBA

As especificações apresentam apenas as interfaces do FT-CORBA sem apresentar maiores detalhes sobre a implementação, permitindo interpretações diferentes da especificação. As dificuldades de, em um primeiro momento, interpretar a especificação e, num segundo momento, complementar a especificação nos aspectos onde haviam pontos mal definidos, para enfim definir as linhas gerais (decisões de projeto) pelas quais a implementação da mesma seria realizada constituíram a maior dificuldade desta fase da tese e sua contribuição. Tanto quanto possível, o trabalho realizado (como parte do projeto GroupPac), foi no sentido de realizar uma implementação tão completa quanto possível seguindo as especificações. Até onde sabemos, somos o primeiro grupo de pesquisa a implementar a especificação com nível de conformidade para *Replicação Passiva* (morna e fria) [OMG00a] e disponibilizá-las, com código aberto, via Internet (http://www.lcmi.ufsc.br/grouppac).

5.4.1.1 Infra-estrutura básica de suporte à replicação

As especificações FT-CORBA definem as linhas gerais dos serviços para tolerância a faltas em *middleware* CORBA. Para prover estes serviços numa condição mínima, uma infra-estrutura básica indispensável é definida nas especificações. Esta infra-estrutura é formada: pelo suporte às IOGRs usados em referencias a grupos de objetos; e pelos interceptadores tanto no lado cliente como no servidor. Apesar das especificações definirem que, para fins de interoperabilidade, esta infra-estrutura não seja indispensável, tal situação é classificada como provendo uma tolerância a faltas apenas parcial.

Para permitir o uso efetivo da IOGR, foram especificadas duas classes de suporte: IOGRFactory e ParsedIOGR. Estas classes fornecem toda a funcionalidade necessária para manipulação de IOGRs, por exemplo, a criação, modificação e a verificação das IOGRs. O suporte a IOGRs deve permitir:

- A construção de uma IOGR a partir de um conjunto de referências de membros;
- A modificação dos parâmetros do TAG_FT_GROUP, como a versão da referência;
- Que os diferentes *profiles*[4] contidos dentro de uma IOGR possam ser obtidos isoladamente, para que cada um dos membros possa ser invocado separadamente;

O uso dos interceptadores é necessário, por exemplo, nas técnicas de replicação passiva (morna e fria). Nestes casos, para que a informação contida nas IOGRs seja usada corretamente, a especificação prevê o uso de interceptadores quando das transferências de requisições, redirecionando e/ou atualizando referências usadas pelo cliente em caso de problemas nas comunicações com a réplica primária. Isto é, o interceptador do cliente pode redirecionar uma requisição para um novo primário, no caso do anterior falhar e, atualizar a IOGR contendo a nova composição do grupo servidor.

Quando do problema da réplica primária, no lado cliente a especificação define que este deve tentar todos os *profiles* disponíveis dentro de uma IOGR antes de abandonar a invocação, segundo a técnica de replicação passiva. Isto foi traduzido, em nossa implementação, pela seguinte seqüência de passos, implementada na classe `ClientIOGRSupport`:

1. Verificar se existe mais de um *profile* (membro) na referência de grupo (IOGR);
2. Se houver apenas um membro, não interferir na invocação;
3. Se houver mais de um, tentar, sucessivamente, fazendo um *ping* em cada um dos membros até que algum deles responda;
4. Quando um deste responder, redirecionar a invocação para aquele membro em particular, que deverá ser a nova réplica primária do grupo.

Do lado do servidor, a especificação define que nenhuma ativação de método deve ser permitida sem antes verificar que a referência usada pelo cliente é realmente a versão mais recente disponível e que, se o grupo tiver o estilo de replicação passiva (fria ou

[4] Contém uma referência (IOR) de um membro de um grupo de objeto.

morna), o objeto alvo é o primário de seu grupo. Isto foi traduzido, em nossas implementações, pelo seguinte procedimento:

1. Contatar o *NameService* para obter a última referência do grupo. Se a referência fornecida pelo cliente é mais antiga, redirecionar o cliente para a nova referência. Se o cliente possui uma referência ainda mais nova (o que indica um erro), cancelar a invocação;
2. Contatar o *Property Manager* para obter o estilo de replicação do grupo. Caso seja replicação passiva (morna ou fria) e o alvo não for o primário, redirecionar o cliente para o primário;
3. Se foram cumpridas as condições (1) e (2), permitir a invocação.

A classe que implementa este comportamento é a `ServerIOGRSupport`.

5.4.1.2 O serviço de gerenciamento de replicação

A implementação do *Serviço de Gerenciamento de Replicação* (SGR) se baseia fortemente nos *use-cases* definidos nas especificações FT-CORBA. Os *use-cases* definem o comportamento esperado para o SGR no gerenciamento dos grupos de objetos de um domínio de tolerância a faltas.

A diferença entre a nossa experiência no GroupPac e o gerenciamento de replicação definido pela FT-CORBA, é a redefinição do comportamento em relação às fábricas disponíveis para a criação de objetos. Nas especificações FT-CORBA, a propriedade *Factories* lista e associa fábricas a um só grupo. Na nossa implementação todas as fábricas são compartilhadas por todos os grupos de objetos. Deste modo, propriedade *Factories* é substituída por um grupo de objetos *Fábrica Genérica* que são então gerenciadas diretamente através do *Serviço de Gerenciamento de Grupo de Objetos*.

Com o intuito de tornar a implementação menos complexa, foi definida a seguinte estrutura para a implementação do *Replication Manager* (Figura 5.9): foram criados quatro grupos básicos de objetos que agrupam todos os objetos da infra-estrutura necessária e que são administrados a partir do *Replication Manager*. Todos os objetos possuem as mesmas interfaces definidas no FT-CORBA.

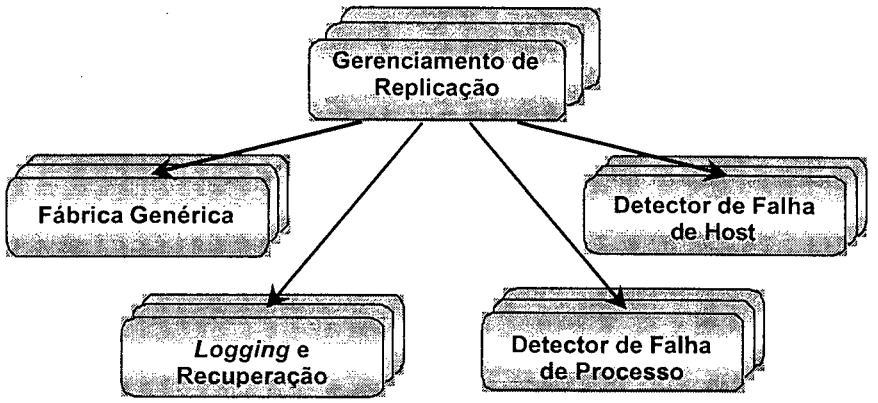


Figura 5.9. Estrutura de grupos da implementação do Gerenciamento de Replicação (*Replication Manager*).

O grupo das *Fábricas Genéricas* agrupa então, todas as fábricas cadastradas no domínio de tolerância a faltas. Os membros deste grupo são igualmente compartilhados por todos os outros grupos que necessitam de fábricas para criar réplicas em seus grupos. O grupo de *Logging & Recuperação* contém todos os objetos de *Logging* e Recuperação no domínio. O grupo serve de referência para que os membros saibam quais *Loggings* devem ser notificados em caso de mudança de estado ou invocação. Os grupos de *Detector de falhas de Host* e *de Processo* reúnem os objetos que implementam as respectivas interfaces.

Na seqüência são descritos os *Use Cases* para o SGR e das classes de serviços herdados pelo mesmo.

a) *Use Cases para o SGR*

A nossa implementação do *SGR* se baseia, em grande parte, nos *Use Cases*[5] definidos na especificação FT-CORBA. As especificações definem duas formas para o gerenciamento de grupo (*membership*): Automático ou Manual[6]. A primeira permite que todo o gerenciamento de grupo (criação e remoção de réplicas) seja realizado, de forma automática, pela infra-estrutura FT-CORBA, através do SGR. Na abordagem manual, a aplicação cria as réplicas de um grupo e solicita ao SGR para atualizar a IOGR. Ou ainda, a aplicação pode solicitar ao SGR para criar uma réplica em uma determinada locação.

[5] Define uma seqüência de procedimentos que um sistema deve executar para verificar sua conformidade com a especificação.

[6] Na verdade, as duas maneiras identificadas pelas especificações como correspondente à execução das funções de gerenciamento nas formas transparente e não transparente, respectivamente.

Grupo com membership automático

Caso o grupo possua controle de *membership* automático, os passos a seguir para a criação de um grupo de objetos são:

1. A aplicação, invocando o método `resolve_initial_references()` (ofercido pelo ORB), obtém a referência do SGR;
2. Para criar um grupo, a aplicação invoca a operação `create_object()` da interface *Fábrica Genérica* herdada pelo *SGR*, fornecendo o `type_id` e `the_criteria`. O *SGR* retorna um `factory_creation_id` que identifica o novo grupo e permite, caso necessário, a remoção do grupo no futuro;
3. O *SGR* obtém as propriedades de tolerância a faltas para o novo grupo a partir do *Gerenciador de Propriedades* usando o `type_id` fornecido. Propriedades adicionais (identificadas pelo nome "`org.omg.ft.FTProperties`") podem ser especificadas a partir do parâmetro `the_criteria`;
4. Usando as propriedades `InitialNumberReplicas` e `Factories`, (item 5.3.1.3) o *SGR* decide onde criar os membros do grupo;
5. Para cada membro, o *SGR* invoca a operação `create_object()` da *Fábrica Genérica* local, passando o `type_id` e `the_criteria` obtidos da propriedade `Factories`. A *Fábrica Genérica* retorna um `factory_creation_id` que identifica o novo membro e permite, caso necessário, a remoção do membro no futuro;
6. O *SGR* define um identificador para o grupo de objetos e constrói o componente TAG_FT_GROUP contendo o identificador do domínio de tolerância a faltas, do grupo de objetos e da versão da referência do grupo de objetos. A partir deste componente é criada a referência do grupo;

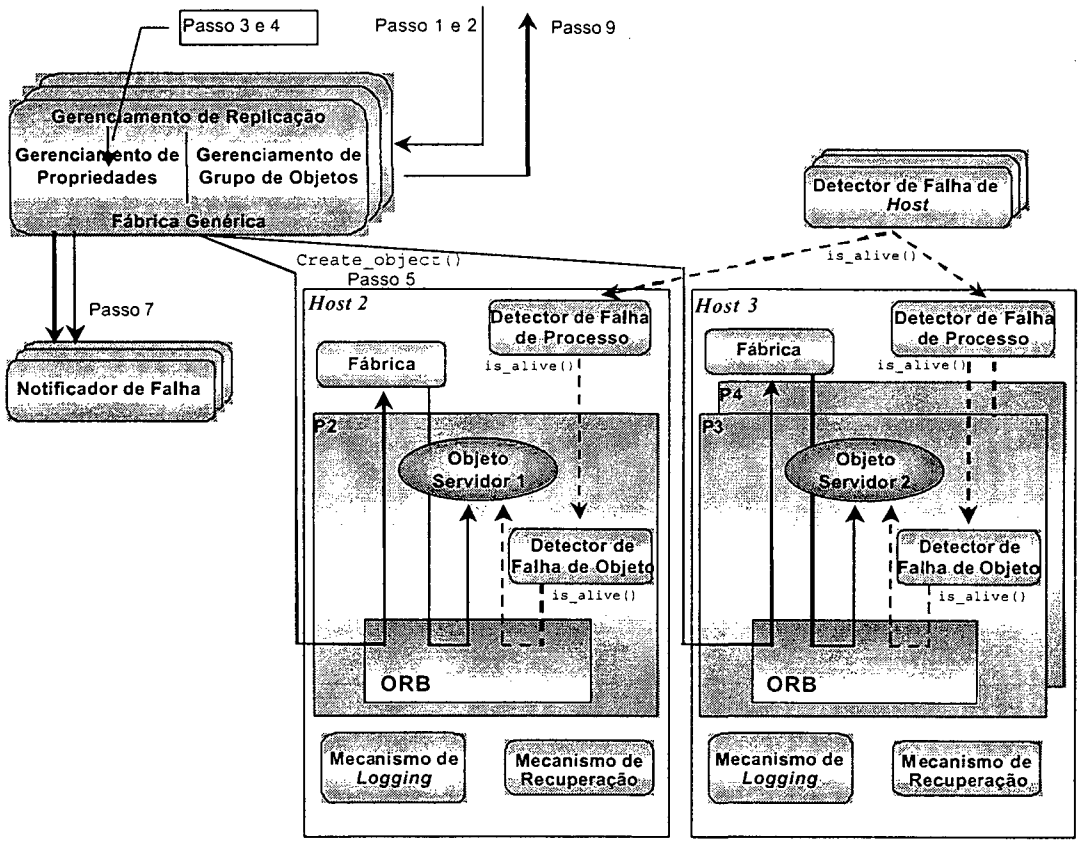


Figura 5.10: Criação de um grupo de objetos com membership automático.

7. Para cada membro criado, o *SGR*: adiciona o membro ao grupo; ativa o membro dependendo do estilo de replicação; inicia a detecção de falhas dependendo do estilo e granularidade da monitoração; registra o mesmo com o *Notificador de falhas* para receber notificações de falhas do membro;
8. Para os estilos de replicação COLD_PASSIVE (passiva fria) e WARM_PASSIVE (passiva morna), o *SGR* determina o membro primário e inclui o *component* TAG_FT_PRIMARY em seu *profile* (figura 5.3);
9. O *SGR* retorna para a aplicação a referência de grupo para o novo grupo criado (obtida no passo 6);

Grupo com membership manual

1. A aplicação, invocando o método `resolve_initial_references()` (ofercido pelo ORB), obtém a referência do SGR;

2. A aplicação obtém as propriedades de tolerância a faltas para o novo grupo a partir do *Gerenciador de Propriedades* usando o `type_id` desejado, incluindo a propriedade `InitialNumberReplicas`;

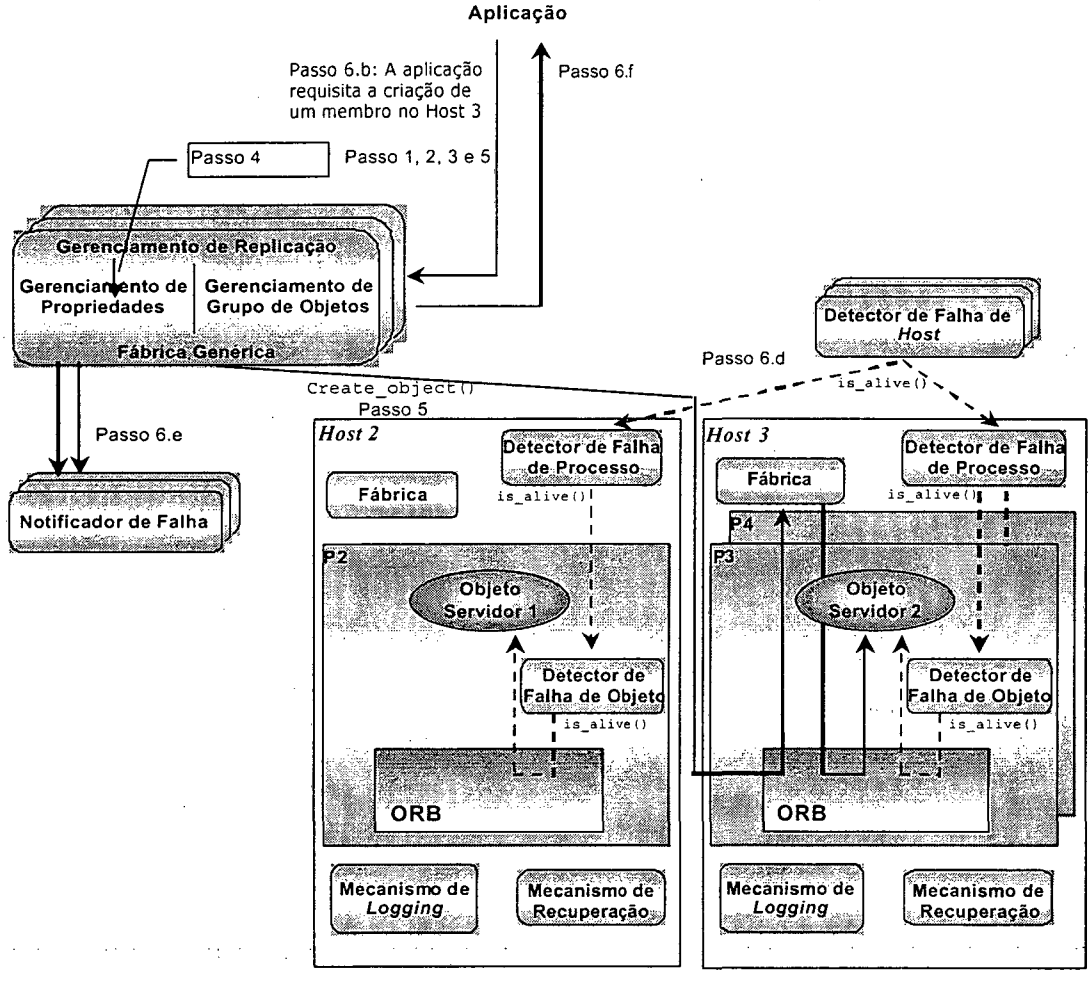


Figura 5.11: Criação de um grupo de objetos com membership manual.

3. Para criar um grupo, a aplicação invoca a operação `create_object()` da interface *Fábrica Genérica* herdada pelo *SGR*, fornecendo o `type_id` e `the_criteria`. O *SGR* retorna um `factory_creation_id` que identifica o novo grupo e permite, se necessário, a remoção do grupo no futuro;
4. O SGR determina o identificador do grupo de objetos e constrói o componente TAG_FT_GROUP contendo o identificador do domínio de tolerância a faltas, do grupo de objetos e a versão da referência do grupo de objetos. Este componente é colocado no TAG_MULTIPLE_COMPONENTS da referência do grupo;

5. O *SGR* retorna a referência do grupo para a aplicação e também um `factory_creation_id` que permite a remoção do grupo no futuro;
6. Para cada membro:
    a. Se a aplicação já criou o objeto que vai se tornar membro, invoca a operação `add_member()` da interface *Gerenciador de Grupo de Objetos*, fornecendo a referência do grupo, do objeto e a locação do objeto;
    b. Se a aplicação deseja usar a infra-estrutura de replicação, invoca a operação `create_member()` passando a locação desejada para o novo membro. O `factory_creation_id` retornado pela *Fábrica Genérica* é guardado para referência futura pelo *SGR;*
    c. O *SGR* constrói uma nova referência do grupo levando em conta o novo membro;
    d. Após verificar as propriedades de estilo, granularidade e intervalo de monitoração, o *SGR* inicia a detecção de falhas no membro;
    e. O *SGR* registra um representante no *Notificador de falhas* para receber notificações de falhas do membro;
    f. O *SGR* retorna a nova referência do grupo para a aplicação;
7. Para os estilos de replicação COLD_PASSIVE e WARM_PASSIVE, o *SGR* determina qual membro será o primário e invoca a operação `set_primary_member()` da interface *Gerenciador de Grupo de Objetos*, o que inclui o component TAG_FT_PRIMARY no *profile* escolhido.

A implementação do *Serviço de Gerenciamento de Replicação* só foi considerada satisfatória quando os testes realizados mostraram que o funcionamento do SGR estava de acordo com as especificações dos dois *use-cases* apresentados.

b) *Gerenciamento de Grupo de Objetos*

A classe responsável pelo controle dos dados associados a um grupo é a classe `ObjectGroupManagerImpl`. Esta classe provê a funcionalidade necessária para a criação, adição, remoção, listagem dos membros do grupo assim como a criação e atualização da referência do grupo no *Serviço de Nomes*.

A atualização da referência do grupo (IOGR) no *Serviço de Nomes* é feita usando o nome da interface da aplicação como chave de cadastro para compor o *name binding* (item 5.2). Isso causa o problema de termos apenas um grupo para cada interface da aplicação em um domínio de tolerância a faltas. Isto é, não podemos ter portanto, vários grupos instanciando a mesma interface dentro de um domínio de tolerância a faltas. Entretanto, esta limitação permitiu a automatização do processo de atualização da referência no *Serviço de Nomes*.

c) *Fábrica Genérica*

A interface *Fábrica Genérica* proposta pela especificação FT-CORBA deveria ser implementada pelo programador da aplicação. Ou seja, cada fábrica de objetos deveria ser implementada especificamente para uma determinada aplicação. Foi decidido que tal comportamento seria desnecessário para uma implementação em Java, pois usando os recursos de Reflexão Computacional, disponíveis na linguagem, é possível implementar uma fábrica realmente genérica, capaz de instanciar qualquer classe disponível dentro da máquina virtual onde se encontra a fábrica.

Para implementar este conceito foi necessário apenas criar uma nova propriedade que indica o nome da classe da implementação a ser instanciada no momento da criação do objeto. Graças a esta escolha a implementação, tanto do SGR como das próprias fábricas genéricas, foi extremamente simplificada, pois a manutenção da consistência dos dados das propriedades *Factories* dos grupos de objetos se tornou ainda mais simples.

É importante ressaltar que esta escolha de implementação para a interface *Fábrica Genérica* é permitida pela especificação, que deixa em aberto o comportamento das implementações ou os parâmetros que devem ser passados para elas.

5.4.1.3 O serviço de gerenciamento de falhas

A figura 5.12 apresenta a interface `PullMonitorable`, a qual fornece a operação para monitoramento de falhas. Qualquer objeto CORBA que queira ser monitorado pelo *Detector de Falhas* (SDF), segundo a especificação, deve herdar esta interface. O SDF invoca a operação `Is_alive()` no sentido de detectar uma possível falha (*crash*) do objeto monitorado. Esta é a única interface definida nas especificações FT-CORBA para detecção de falhas. O objeto *Detector de Falhas* que implementa o monitoramento não tem uma

especificação definida na forma de interface CORBA, apenas o seu procedimento, que é detectar uma falha e avisar ao notificador de falhas (figura 5.5).

```
module CORBA {
  module FT {
    Interface PullMonitorable {
      Boolean Is_alive();
    };
            :
```

Figura 5.12. Interface `PullMonitorable`.

Deste modo, tivemos que definir uma interface para o Detector de Falhas (SDF) de maneira que os métodos propostos na IDL do detector fossem no sentido de garantir a comunicação do *SGR* com os *SDF*. O SGR necessita dessa comunicação para poder adicionar, remover e atualizar referências de grupos de objetos (IOGRs) monitorados pelos detectores de falhas.

O Notificador de Falhas tem sua interface bem definida na especificação FT-CORBA. Por isso, sua implementação foi simples e está totalmente de acordo com a especificação. O Analisador de Falhas, além de não ser indispensável para a infra-estrutura de tolerância a faltas, também não possui uma interface definida. Portanto, apesar da implementação de uma política de decisão sobre a ocorrência de falhas fazer parte do projeto, esta ainda não pôde ser incluída devido à falta dessa especificação no FT-CORBA.

5.4.1.4 Gerenciamento de estados

Como a especificação do *Serviço de Logging e Recuperação* é bastante vaga e não define as interfaces para os objetos descritos, uma das tarefas deste trabalho foi propor a definição em IDL das mesmas de acordo com os procedimentos desses serviços definido nas especificações. Optou-se por reunir os serviços de *Logging* e *Recuperação* (figura 5.1) num único serviço, chamado *LoggingRecovery*, buscando minimizar o tráfego de rede, além de evitar uma grande quantidade de informações redundantes.

Para garantir que todas as mensagens enviadas ao objeto primário sejam capturadas, foram utilizados interceptadores do lado do servidor, chamados *LoggingInterceptor*. A função destes interceptadores é fazer uma cópia dos parâmetros das requisições e passar para o *LoggingRecovery* que se encontra na máquina do objeto primário.

A definição da interface do *LoggingRecovery* foi feita de forma que os métodos propostos garantam a comunicação entre o *LoggingRecovery* e seu grupo, o Gerenciador de Replicação e o *LoggingInterceptor*.

```
module FT {
  typedef sequence<octet> State;
  typedef sequence<Dynamic::Parameter> ParameterList;
  exception NoStateAvailable {};
  interface LoggingRecovery {
    // obtém o estado do primário
    State get_state(in string updateable_type_id) raises(NoStateAvailable);
    // transfere o estado para o novo primário
    void set_state(in string updateable_type_id, in State new_state);
    // envia uma cópia da requisição para o logging
    void set_update(in string operation, in ParameterList parameters,
                    in any result);
    // atualiza o estado de uma réplica morna (logging e checkpoint)
    void recovery_state(in string updateable_type_id);
    void add_updateable_iogr(in string new_updateable_type_id);
    void update_updateable_iogr(in stringupdateable_type_id);
    void update_logging_iogr();
```

Figura 5.13: Interface IDL do LoggingRecovery.

5.4.1.5 A interface gráfica FTAdmin

Para facilitar o gerenciamento por parte do usuário do *Serviço de Gerenciamento de Replicação* (SGR), foi implementada uma interface gráfica de administração para o *SGR*, o FTAdmin (figura 5.14).

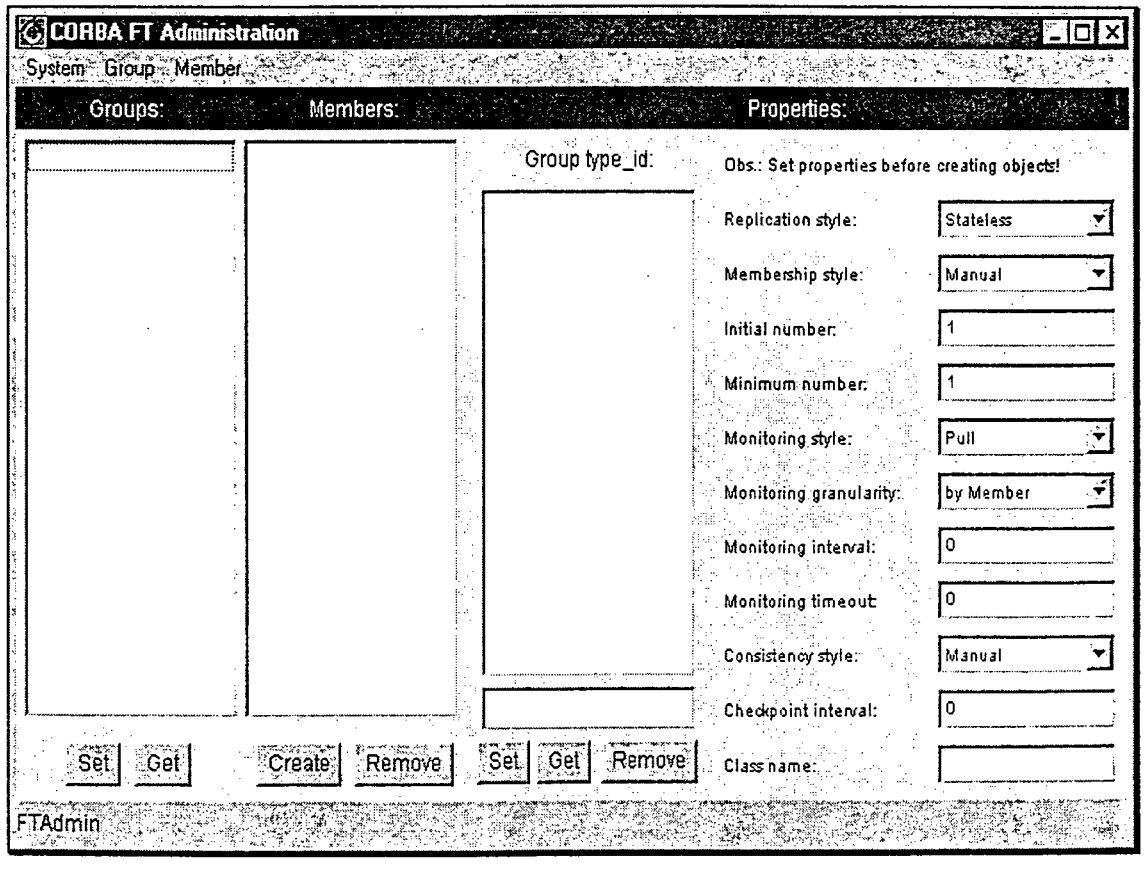
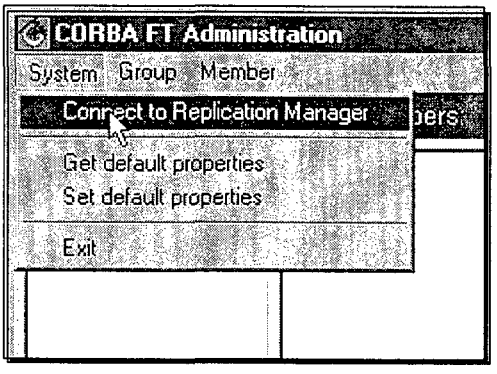


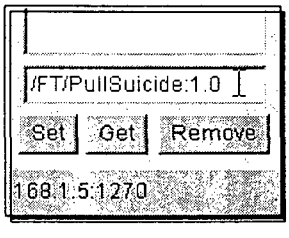
Figura 5.14: Janela principal da interface de administração FTAdmin.

A interface FTAdmin é a melhor maneira de controlar a infraestrutura do GroupPac. Sua janela principal é constituída por: uma lista de grupos (à esquerda); uma lista de membros (no centro); e um painel de edição de propriedades (à direita). A interface foi orientada no sentido de simplificar ao máximo o controle de um grupo de objetos replicados. Esta interface permite iniciar o sistema (lançando todos os objetos de serviço do FT-CORBA da figura 5.1), conectar com o *Serviço* de *Gerenciamento de Replicação*, definir as propriedades de grupo, criar um grupo de objetos replicados e criar membros (réplicas) nos grupos existentes.

Seguindo os cinco passos listados abaixo é possível criar um grupo de objetos a partir desta interface de administração. O exemplo descrito usa a interface IDL *PullSuicide* de um objeto servidor.



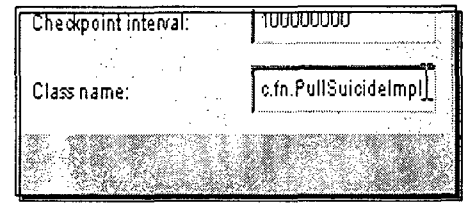
1. Iniciar o sistema:
   - Inicie o *Servidor de Nomes*, o *Gerenciador de Replicação*, uma *Fábrica Genérica* e um *Detector de falhas de Host*.
2. Conectar-se com o *Gerenciador de Replicação*:
   - Para isso, selecione a opção "Connect to Replication Manager" no menu "System".
   - Após o estabelecimento da conexão, a lista de grupos será automaticamente atualizada.
   - Ao clicar em algum dos itens da lista de grupos, o identificador da interface dos objetos do grupo será mostrado na barra de status. Ao clicar em algum dos itens da lista de membros, serão mostrados o identificador da interface do objeto e sua locação.
3. Definir as propriedades do grupo:
   - Antes de criar um novo grupo, você deve definir as propriedades de "interface". Estas são propriedades (item 5.3.1.3) que serão usadas para a criação de



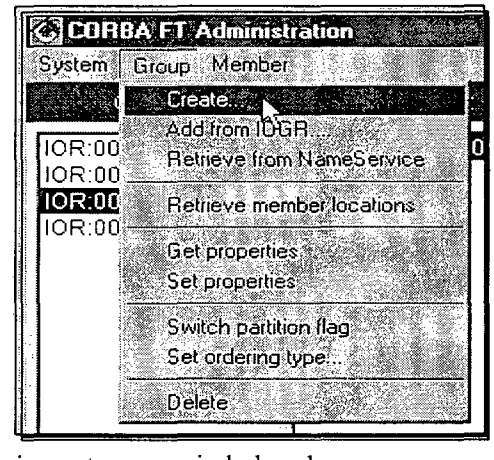
todos os grupos de objetos que implementam aquela interface.

- Como vamos criar um grupo de objetos da interface *PullSuicide*, digite na caixa de texto acima dos botões do painel de propriedade: `IDL:omg.org/CORBA/FT/PullSuicide:1.0`

- Para simplificar as operações mais tarde, digite (na caixa de texto indicada por "*Class name:*") o nome da implementação da interface que deseja usar. Neste caso vamos usar uma implementação que já vem com o pacote *GroupPac*: `edu.lcmi.grouppac.fn.PullSuicideImpl`



- Para definir as propriedades, clique no botão "*Set*" do painel de propriedades. As novas propriedades serão indicadas por um elemento sendo adicionado à lista de propriedades de grupo.



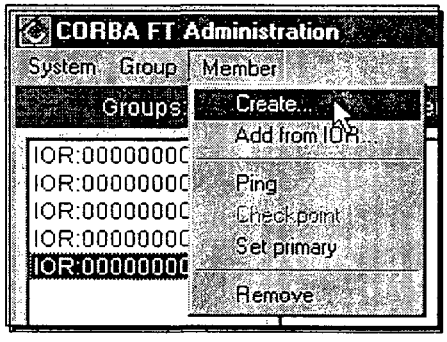
4. Criar um grupo de objetos:
   - Certifique-se de ter definido as propriedades de interface primeiro, pois certas propriedades do grupo só podem ser definidas antes de sua criação, através das propriedades de interface.

   - Para criar um grupo, clique na opção "*Create...*" do menu "*Group*". Uma janela de entrada de dados vai confirmar seu pedido. Se você tiver selecionado o identificador do grupo na lista de propriedades, este será automaticamente usado como valor padrão para a janela de entrada. Clique em OK para criar o grupo. O novo grupo vai aparecer na lista de grupos.

5. Criar membros (réplicas) nos grupos:

- Antes de criar membros nos grupos, tenha certeza de que existem *Fábricas Genéricas* suficientes. É necessário ter uma *Fábrica Genérica* para cada membro.

- Para criar um objeto (réplica) em um grupo, basta escolher um grupo na lista de grupos e selecionar a opção "*Create...*" no menu "*Member*". Uma janela de entrada de dados vai confirmar seu pedido. Se você tiver definido a propriedade "*Class name*" no passo 3, o nome da classe digitado vai ser usado como valor padrão na janela de entrada. Clique em OK para criar o membro. O novo membro vai ser adicionado à lista de membros.



- Para criar mais réplicas, clique novamente na opção "*Create...*" no menu "*Member*".

Para mais informações sobre o funcionamento da interface FTAdmin, acesse o endereço http://www.lcmi.ufsc.br/grouppac/ e siga o *link* "FTAdmin".

5.5 Considerações sobre a implementação do GroupPac segundo o FT-CORBA

A análise aprofundada da especificação *Fault Tolerant* CORBA nos permitiu detectar várias pequenas incoerências que dificultaram a implementação da mesma. Estas incoerências são devidas principalmente à abrangência da especificação. A especificação também se mostra insuficiente na definição de aspectos de tolerância a faltas orientados para grupos de objetos distribuídos em sistemas de larga escala – este assunto será explorado no próximo capítulo.

Suportes de comunicação de grupo são fundamentais para a implementação de técnicas de replicação em sistemas distribuídos. As especificações atuais do FT-CORBA ainda não definem um serviço de comunicação de grupo. O problema atual está na grande variedade de protocolos de comunicação de grupo. Se um padrão fosse definido, na atual

especificação, para esse serviço, provavelmente não seria uma solução capaz de atender aos requisitos de toda a gama de aplicações distribuídas que podem fazer uso de comunicação de grupo. A OMG ainda está tratando desse problema cuidadosamente. Por exemplo, foi lançada recentemente uma RFP no sentido de especificar um padrão para difusão não confiável de mensagem (*Unreliable Multicast* [OMG00b]) a ser incluída no CORBA. Considerando as experiências a serem obtidas nessa força tarefa, essa RFP seria, de certa forma, o primeiro passo para futuras RFPs, no sentido da concepção de um serviço de comunicação de grupo com garantias mais restritivas de acordo e ordenação.

A implementação realizada, segundo as especificações FT-CORBA, se comportou de maneira compatível com a especificação, inclusive nos requisitos de interoperabilidade, quando colocada em conjunto com ORBs sem suporte a tolerância a faltas. Todos os *use cases* definidos nas especificações FT-CORBA foram utilizados, com sucesso, para testar as implementações realizadas (http://www.das.ufsc.br/grouppac).

5.6 Conclusão

Neste capítulo, apresentamos, de forma sucinta, os objetos de serviço que compõem a arquitetura de tolerância a faltas FT-CORBA e nossas experiências no sentido de implementa-las. Verificamos que as especificações atuais do FT-CORBA apresentam diversos pontos ainda não tratados, principalmente a falta de algumas interfaces. Além disso, as interfaces da especificação são bastante genéricas, o que nos motivou a propor novas extensões no sentido de tratar aspectos de escalabilidade, conforme será apresentado no próximo capítulo desta tese.

CAPÍTULO 6
Adaptando as Especificações FT-CORBA para Redes de Larga Escala

6.1 Introdução

Fazendo uma análise detalhada das especificações FT-CORBA é verificado que muitos requisitos necessários para tolerância a faltas em sistemas de larga escala[1] (tal como a Internet) ainda não são abordados. As principais dificuldades são devidas aos aspectos de escalabilidade e as características assíncronas[2] desses sistemas – as especificações ainda não apresentam abstrações para tratar esses problemas [OMG00a]. Especificamente, ainda não foram discutidas soluções para: (1) detecção de falhas em sistemas com características assíncronas; (2) gerenciamento de replicação[3] (*membership*) e a comunicação de grupo em um contexto de redes de larga escala; (3) e, principalmente, como estas soluções podem ser integradas ou construídas a partir da estrutura do FT-CORBA, sem que isto represente modificar qualquer interface já padronizada pela OMG.

Portanto, neste capítulo é apresentada uma proposta de extensão e adequação das especificações dos serviços de detecção de falhas e gerenciamento de replicação do FT-CORBA para atender aos requisitos de redes de larga escala. Esta proposta está fundamentada em um estudo realizado que envolveu a definição de um modelo combinando o serviço de gerenciamento de replicação com um serviço de nomes hierarquizado. Este modelo nos permitiu, então, a definição de um protocolo de detecção

[1] Os sistemas de larga escala, inerentemente assíncronos, são caracterizados pela grande distribuição espacial, com um caráter aberto, integrando quantidades significativas de recursos computacionais, assinalados pela sua heterogeneidade.

[2] Esses sistemas são não deterministas no sentido em que os parâmetros temporais, tais como padrões de tráfego, taxas de transferência, atrasos de comunicação e diferentes velocidades de processamento dos processadores, são dependentes das condições de carga e evolução dinâmica desses sistemas e, portanto, não são conhecidos *a priori*.

[3] Ou gestão de pertinência.

de falhas assimétrico (serviço dedicado) e de um suporte de comunicação de grupo, próprios para sistemas de larga escala. A proposta visa apresentar soluções, baseadas nas especificações FT-CORBA, para o gerenciamento de replicação e a difusão atômica de mensagens – seriamente dificultados pelos custos e pela falta de escalabilidade das soluções usuais destes problemas quando tratamos com redes de larga escala.

Este capítulo está dividido da seguinte maneira. Na próxima seção, é feita uma discussão das especificações FT-CORBA (apresentadas no capítulo 5) considerando aspectos de escalabilidade. Na seção 6.3 é apresentada a versão mais recente do *GroupPac*, e a sua proposta para detecção de falhas, gerenciamento de replicação e comunicação de grupo em redes de larga escala. Por fim, são levantadas as conclusões deste capítulo.

6.2 Limitações de escalabilidade nas especificações fault-tolerant CORBA

O objetivo desta seção é verificar nos serviços de gerenciamento de replicação e de detecção de falhas, das especificações FT-CORBA, quais os pontos que ainda não atendem adequadamente os requisitos de escalabilidade. Além disso, são discutidos aspectos relacionados ao suporte de comunicação de grupo, que ainda não é tratado, claramente, nas especificações atuais do FT-CORBA. Neste caso, o enfoque sobre este aspecto será também para sistemas de larga escala.

6.2.1 Gerenciando replicações em domínios de tolerância a faltas

Nas especificações FT-CORBA foi introduzido o conceito de *Domínios de Tolerância a faltas* para facilitar o gerenciamento de grupos de objetos distribuídos em uma rede de computadores. Cada domínio pode conter diversos grupos de objetos (figura 6.1), cada grupo possuindo suas próprias propriedades de tolerância a faltas. É definido, também, que cada domínio tem disponível um conjunto de serviços do FT-CORBA (gerenciamento de replicação, gerenciamento de falhas e gerenciamento de recuperação e *logging*), os quais atuam, exclusivamente, dentro deste domínio de tolerância a faltas.

Os objetos pertencentes a um domínio não são, necessariamente, limitados a uma rede local (LAN – *Local Area Network*), ao invés disso, podem estar distribuídos em uma rede metropolitana (MAN – *Metropolitan Area Network*) ou ainda, em uma rede de longa

distância (WAN – *Wide Area Network*). Essas diferentes opções de rede em que um domínio pode se estender implica, certamente, na definição de diferentes propriedades de tolerância a falta.

As especificações FT-CORBA, ao limitar grupos a um domínio de TF específico, não se mostram, pelas suas abstrações, apropriadas para sistemas distribuídos de larga escala. É claro que seria sempre possível definir um domínio de tolerância a faltas único que comporte todos as réplicas de um grupo em um ambiente de larga escala. No entanto, devido às características assíncronas nestes sistemas, envolvendo grandes distâncias geográficas, as dificuldades para gerenciar grupos seriam enormes.

Uma das funções do serviço de gerenciamento de replicação (SGR, capitulo 5) é criar as referências de grupos de objetos (IOGR - *Interoperable Object Group Reference*) do domínio de TF correspondente. No entanto, estas IOGRs não estão disponíveis a partir deste serviço. A alternativa plausível é o uso do serviço de nomes (*CosNaming*) padrão OMG [OMG97a] para guardar todas as IOGRs do domínio geradas pelo serviço de gerenciamento de replicação – cada domínio de TF teria, então, o seu próprio serviço de nomes. A especificação do *CosNaming* foi recentemente revisada com o objetivo de tratar aspectos de interoperabilidade, definir um formato URL para nomes e estender as funcionalidades para ligar contextos de nomes. O serviço de nomes seria, portanto, um "portão de acesso" para todos objetos e grupos de objetos de um domínio. Contudo, segundo a especificação *CosNaming*, o serviço de nomes não é tolerante a falhas. Em outras palavras, o serviço de nomes, sendo um simples ponto de falha, pode comprometer toda a estrutura do FT-CORBA: se falhar, todos os objetos e grupos de objetos ficam inacessíveis.

Em relação ao aspecto da localização de objetos, as especificações FT-CORBA não definem como objetos (clientes) de um domínio de tolerância a faltas podem acessar objetos (servidores replicados ou não) de outros domínios. O problema está em como disponibilizar as IORs e IOGRs de um domínio para outro. O *CosNaming* fornece suporte para construção de conjuntos de servidores de nomes dispostos de forma hierárquica, através da ligação de contextos de nomes (*bind context*) [OMG97a]. Portanto, podemos atribuir a cada domínio um serviço de nomes local (contendo o contexto de nomes do domínio) ligado a um serviço de nomes global, o qual contêm o contexto de nomes de

todos os serviços de nomes locais. Todavia, a necessidade de tornar esse serviço de nomes global tolerante a falhas é também crucial pelas mesmas razões dos serviços de nomes locais.

Tolerância a faltas no serviço de nomes pode ser alcançada de duas maneiras [Lau99a]: usando o serviço de persistência padrão OMG [OMG97a] ou através de técnicas de replicação. A decisão de não seguir uma abordagem fundamentada no serviço de persistência é devido a fatores de desempenho. Um serviço de nomes com copias persistentes, quando da ocorrência de *crash,* necessita do administrador para reiniciar o mesmo e fazer com que esse recupere seu estado, por exemplo, a partir de um arquivo. Então, a questão é analisar quais técnicas de replicação são mais adequadas para os serviços de nomes local e global – considerando a disposição geográfica dos mesmos e as dimensões de larga escala do sistema.

6.2.2 O serviço de detecção de falhas

O serviço de gerenciamento de grupo de objetos (SGG, parte do serviço de gerenciamento de replicação) depende do serviço de gerenciamento de falhas (SGF) para detectar *crashes* de objetos. Nas especificações dos detectores de falhas são definidos três granularidades de monitoramento (detecção) de falhas, conforme apresentado no item 5.3.1.3. Segundo o FT-CORBA, todos os *hosts*, contendo objetos de grupos de um domínio de TF, devem ser monitorados por pelo menos um detector de falhas. Os detectores de falhas de um domínio podem ser replicados atuando sobre os mesmos *hosts* ou os mesmos objetos. Mas situações diversas podem ser admitidas: detectores podem não monitorar, necessariamente, os mesmos conjuntos de *hosts* ou objetos do domínio. Isto é, dentro de um domínio de tolerância a faltas, podemos ter detectores monitorando diferentes subconjuntos de *hosts* que compõe o domínio. Por exemplo, é admitido um ou mais *hosts* serem monitorados por um único detector de falhas em um domínio de TF. Neste caso não há tolerância a possíveis falhas de detectores no domínio.

Além disso, detecção de falhas em sistemas assíncronos é uma classe de problemas onde só são permitidas soluções probabilistas [Fisher85, Ricciardi91, Chandra96]. As especificações FT-CORBA fornecem alguns recursos, que nas suas definições, não são bem adequados para esta classe de problemas. É o caso do monitoramento de um *host* a

partir de um conjunto de detectores de falhas. Neste caso, o *host* é assumido como *crash* quando não responde, dentro de *timeouts* estipulados, a pelo menos um dos detectores do domínio. Se considerarmos as características assíncronas de um sistema de larga escala este tipo de procedimento é extremamente penalizante. No entanto, se associarmos algumas semânticas a estes detectores de falhas, as condições de detecção podem ainda ser melhoradas na precisão[4].

6.2.3 Comunicação de grupo no modelo CORBA

As especificações atuais do FT-CORBA ainda não definem um serviço de comunicação de grupo, fundamental na implementação de técnica de replicação ativa em sistemas distribuídos. O problema atual está na grande variedade de protocolos de comunicação de grupo, com diferentes propósitos e características.

Se um padrão fosse definido, na atual especificação, para esse serviço, provavelmente não seria uma solução capaz de atender aos requisitos de toda gama de aplicações distribuídas que podem fazer uso de comunicação de grupo. A OMG está tratando desse problema cuidadosamente, tanto que a maior parte dos membros do grupo responsável pela padronização do FT-CORBA está participando de outras RFPs que estão diretamente relacionadas com tolerância a faltas no CORBA. Por exemplo, recentemente, foi lançada uma RFP (*Request for Proposal*) no sentido de especificar um padrão para difusão não confiável de mensagem (*Unreliable Multicast*) [OMG00b] para ser incluída no CORBA. Considerando as experiências a serem obtidas nessa força tarefa, essa RFP seria, de certa forma, o primeiro passo para futuras RFPs, no sentido da concepção de um serviço de comunicação de grupo com garantias mais restritivas de acordo e ordenação.

Em termos de pesquisas envolvendo a parte de integração de suporte de comunicação de grupo à arquitetura CORBA existem diversos trabalhos tratando desse tema. Essas soluções podem ser classificadas, basicamente, em três abordagens (capítulo 2): integração [Maffeis95, ISIS95], serviço [Felber98] e interceptação [Fraga97, Moser98, Lau00a]. Em geral, estas abordagens se preocupam, basicamente, na disponibilidade de suporte de

[4] Em [Chandra96], são definidas várias semânticas para detectores de falhas não confiáveis a partir de duas propriedades: *completude* (*"completeness"*) e a *precisão* (*"accuracy"*). A completude especifica que um detector de falhas terminará por suspeitar de todos os processos que realmente estiverem falhos. E a precisão restringe os enganos (suspeitas errôneas) que um detector possa cometer.

comunicação de grupo na arquitetura CORBA, considerando as características de desempenho, interoperabilidade e portabilidade. No entanto, ainda não existem propostas voltadas, especificamente, para um contexto de larga escala ou mesmo uma discussão mais detalhada da adaptação de conceitos de comunicação de grupo no CORBA para estes ambientes.

Por outro lado, em sistemas de larga escala, grupos podem ser formados por uma quantidade considerável de objetos espalhados geograficamente. Um exemplo típico desse tipo de sistema seria uma base de dados de uma corporação com réplicas espalhadas por todo o planeta no sentido de facilitar o acesso às suas informações. Um novo dado inserido em uma réplica da base de dado teria que ser difundida atomicamente para as outras réplicas do grupo para assegurar a consistência. Embora nas especificações atuais sejam recomendadas ferramentas proprietárias para comunicação de grupo, o FT-CORBA, através das IOGR, limitam grupos em um domínio de tolerância a faltas específico, dificultando o uso destas ferramentas em uma rede de larga escala.

6.3 GroupPac

A nova versão do *GroupPac* [Lau00a, Lau00b, Lau01a] consiste de um conjunto de serviços específicos para o desenvolvimento de aplicações tolerantes a falhas. Adere as recomendações da OMG como objetos de serviços, tais como as especificações COSS [OMG97a]. Os serviços do *GroupPac* têm suas interfaces baseadas nas especificações FT-CORBA (ou seja, nas especificações dos serviços de gerenciamento de replicação, gerenciamento de falha, *logging* e recuperação). Esta nova versão do GroupPac, em relação ao padrão, foi adotada ainda um conjunto de extensões e adaptações no sentido de atender aos requisitos de tolerância a faltas em sistemas de larga escala.

De acordo com a figura 5.1, as soluções, propostas neste trabalho, envolvem a definição de um modelo de estruturação de sistemas distribuídos onde a composição de domínios de TF e de serviços de nome aporta a escalabilidade necessária em redes de larga escala. A adoção deste modelo tem implicações nos serviços de gerenciamento de replicação e de gerenciamento de falhas e na comunicação de grupos. A comunicação de grupo é disponibilizada no *GroupPac* usando o conceito de interceptadores e encapsulando

ferramentas proprietárias na forma de serviços de objeto[5] [Oliveira99a, Lau00a]. Propriedades de ordenação nas comunicações dependem da ferramenta usada – no caso foi usado o ISIS [ISIS95].

Estas soluções são adaptadas ao CORBA de forma transparente para a aplicação, e sem que isto represente qualquer modificação das interfaces do padrão. É também assumido nestas soluções, em nível de transporte, canais de comunicação ponto-a-ponto confiáveis, servindo de suporte aos objetos se comunicando via CORBA por troca de mensagens. Por sua vez, como hipótese de falhas é assumido que objetos (ou *hosts*) podem falhar por *crash*, sendo que os serviços da aplicação podem ser restaurados através do Serviço de *Logging* e Recuperação do FT-CORBA (figura 5.1).

6.3.1 Hierarquia de domínios de tolerância a faltas e a escalabilidade

A abstração de domínios de tolerância a faltas do FT-CORBA é bastante genérica, não existem limitações na constituição de um domínio de tolerância a faltas (item 6.2.1). No entanto, se considerarmos a complexidade em gerenciar um grupo de objetos em larga escala, com centenas ou milhares de objetos, e distribuídos geograficamente, veremos que a definição de um domínio de tolerância a faltas para comportar grupos e objetos em um ambiente deste tipo, tal como a especificação aconselha, não é adequada. Gerenciar um grupo com réplicas ativas em um domínio nestas condições seria uma tarefa bastante complexa e com custo elevado em termos de trocas de mensagens.

A literatura indica que a forma mais adequada para tratar as complexidades dos sistemas de larga escala é a decomposição hierárquica do problema [Rodrigues96, Fritzke98]. A idéia de estabelecer uma hierarquia de domínios de tolerância a faltas vem no sentido de facilitar o gerenciamento e a comunicação de grupos de objetos. Um grupo, que tenha componentes dispersos em uma rede de larga escala, teria estes componentes compondo subgrupos distribuídos entre vários domínios dessa hierarquia. Isto é, ao invés de termos um único domínio para gerenciar um grupo largamente distribuído, teremos então, um conjunto de domínios para gerenciar partes deste grupo (seus subgrupos) de

[5] Um interceptador é um mecanismo de ORB interposto entre um cliente e um objeto servidor (no nosso caso um grupo de objetos) cujo propósito é desviar de forma transparente a requisição de serviço. O interceptador então ativa o objeto que encapsula a ferramenta de comunicação de grupo, provocando a difusão da requisição de serviço no grupo de objetos servidores.

forma separada. Esta separação permite, por exemplo, que cada domínio tenha seus próprios protocolos de gerenciamento e comunicação de grupo – estabelecidos de acordo com as características do ambiente sobre o qual está disposto.

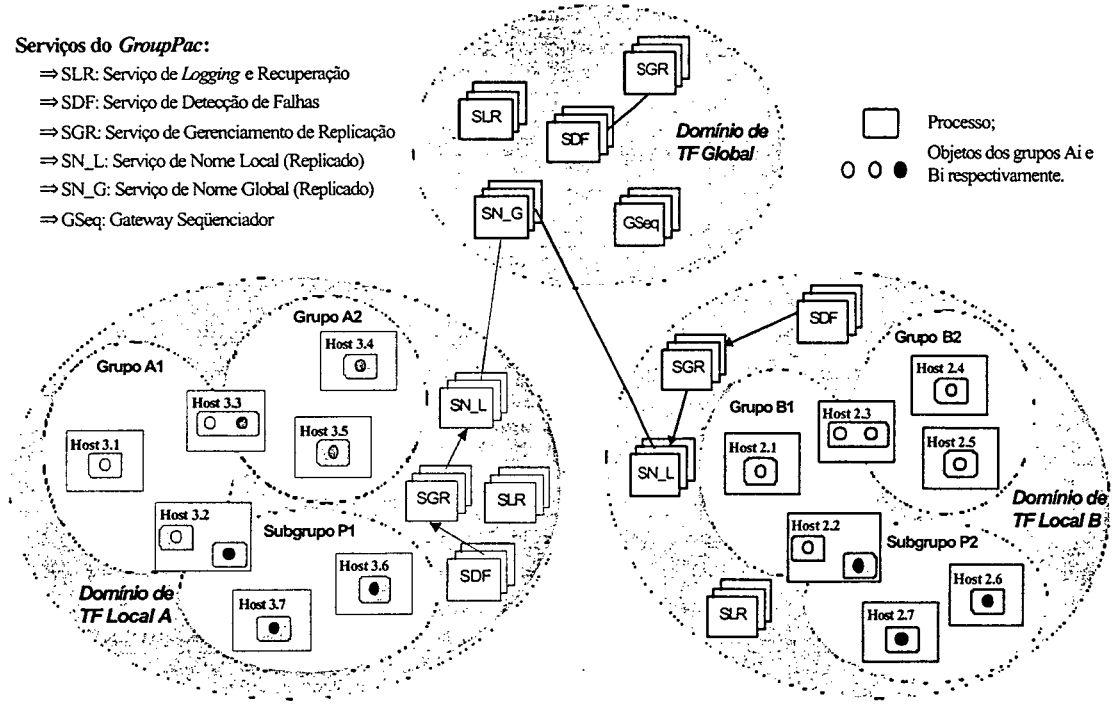


Figura 6.1. Modelo hierárquico para sistema de larga escala.

Neste modelo, os domínios podem conter grupos onde todas as suas réplicas são gerenciadas e se comunicam dentro do próprio domínio. Estes grupos são identificados no *GroupPac* como *Grupos Intradomínio*. Um grupo, que tenha componentes dispersos em uma rede de larga escala, teria estes componentes compondo subgrupos distribuídos entre vários domínios. Estes grupos envolvendo membros em vários domínios são identificados como *Grupos Interdomínios*. Nestes grupos interdomínios, membros são gerenciados separadamente, de acordo com o domínio de tolerância a faltas a que pertencem.

Na figura 6.1, os grupos *A1* e *A2* contidos no domínio *A* são exemplos de grupos intradomínios. O grupo *P*, formado pelos subgrupos *P1* (domínio *A*) e *P2* (domínio *B*), corresponde a um grupo interdomínios. A hierarquia de domínios de tolerância a faltas (TF) da figura 6.1 é apresentada em dois níveis de gerenciamento:

- *Domínios de TF Local ou inferior*: pode conter e gerenciar grupos locais que não envolvam grandes distribuições físicas (grupos intradomínios) e

também subgrupos de grupos interdomínios. Na figura 6.1, o domínio local A contém e faz a gestão do grupo *A1* e do subgrupo P1;

- *Domínio de TF Global*: é o domínio de gerenciamento global. Comporta, exclusivamente, o conjunto de serviços responsáveis por garantir a interação entre os domínios de tolerância a faltas locais. Esse domínio não comporta grupos ou subgrupos de objetos de aplicação.

De forma sucinta, cada domínio possui sua própria infraestrutura para tolerância a faltas (constituído pelo SLR, SDF, SGR, SN_L, SN_G e GSeq, ver figura 6.1) suportada pelo *GroupPac*. Serviços *GroupPac* de um domínio operam somente sobre os grupos e subgrupos contidos neste domínio de tolerância a faltas. O serviço de detecção de falhas (SDF) é o responsável por monitorar os *hosts* do seu domínio de tolerância a faltas. Eventuais falhas de *host* são detectadas pelo SDF que notifica ao serviço de gerenciamento de replicação (SGR) do domínio para que estabeleça uma nova IOGR (com uma nova lista de membros). As IOGRs dos grupos replicados do domínio são, então, enviadas ao serviço de nome local replicado (SN_L) para que este as disponibilize no sistema. O SN_L contém as IOGRs (e IORs) do domínio ao qual pertence. Os grupos SDF, SGR e SN_L podem ser replicados em qualquer número e estar localizados em qualquer *host* do domínio. Finalmente, o *Serviço de Nomes Global* (SN_G) faz a ligação de todos os SN_L – permitindo, por exemplo, que objetos de um domínio possam localizar grupos de objetos de outros domínios de tolerância a faltas. No decorrer desta seção, serão apresentados mais detalhes sobre cada um destes serviços.

Cada grupo de objetos pode ser uma aplicação replicada distinta, disponível como entidade lógica única, segundo o modelo de objetos CORBA (cliente/servidor). Os grupos contidos dentro de um domínio serão regidos de acordo com as propriedades de tolerância a faltas definidas no serviço de gerenciamento de propriedades de cada domínio de tolerância a faltas.

Grupos interdomínios, por apresentarem seus subgrupos dispostos em diferentes domínios de TF, implicam em um gerenciamento mais simples e adequado às necessidades de escalabilidade. Esta separação em subgrupos e domínios de TF está de acordo com as propriedades de minimidade [Guerraoui97] e localidade [Fritzke98]. Esta divisão permite,

por exemplo, que cada domínio tenha seus próprios protocolos de gerenciamento e comunicação de grupo – estabelecidos de acordo com as características do ambiente sobre o qual está disposto. As mudanças de *membership* são mais fáceis de tratar com esta decomposição em domínios com subgrupos. No entanto, esta decomposição não é uma solução trivial, implicando na definição de um conjunto de suportes e extensões nas especificações FT-CORBA que mostraremos ao longo desta seção.

6.3.2 O serviço de detecção de falhas do GroupPac

Conforme discutido, no item 6.2.2, a detecção de falhas do FT-CORBA, na forma como foi especificada, não é adequada para sistemas com características assíncronas e de larga escala. Para tratar com as dificuldades destes ambientes, estendemos a noção de detectores de falhas do FT-CORBA, sem que isto represente qualquer modificação das interfaces dessa especificação. Assumimos então que um *host* é considerado em *crash* se não responder a um certo número de detectores de falhas dentro de *timeouts* específicos. É necessário então um procedimento de consenso para ser executado por um conjunto de detectores na determinação de falhas. O monitoramento de um *host* a partir de um grupo de detectores minimiza a probabilidade de erros de detecção.

A solução que adotamos neste trabalho foi especificar um protocolo baseado em voto majoritário para alcançar consenso na decisão sobre as falhas de objetos. Diferente de [Chandra96], em que o módulo detector é agregado a cada processo da aplicação, a nossa solução, para melhor se adequar à especificação do serviço de detecção de falhas do FT-CORBA, propõem um conjunto de objetos detectores dedicados, dispostos no domínio, podendo ou não estar nos mesmos *hosts* de objetos de aplicação. No nosso esquema todos os detectores realizam as mesmas funções, ou seja, monitoram todos os *hosts* dentro de seu domínio de TF. A figura 6.2 exemplifica este esquema de detecção. Os detectores de falhas que se utilizam do algoritmo definido em [Ricciardi91, Renesse95] para o consenso podem ser classificados como *Perfeitos* (classe *P* [Chandra96]). Assumimos, portanto, que os detectores são sempre completos, após um certo tempo suspeitará de todo processo permanentemente faltoso ou desconectado dentro do domínio de tolerância a faltas.

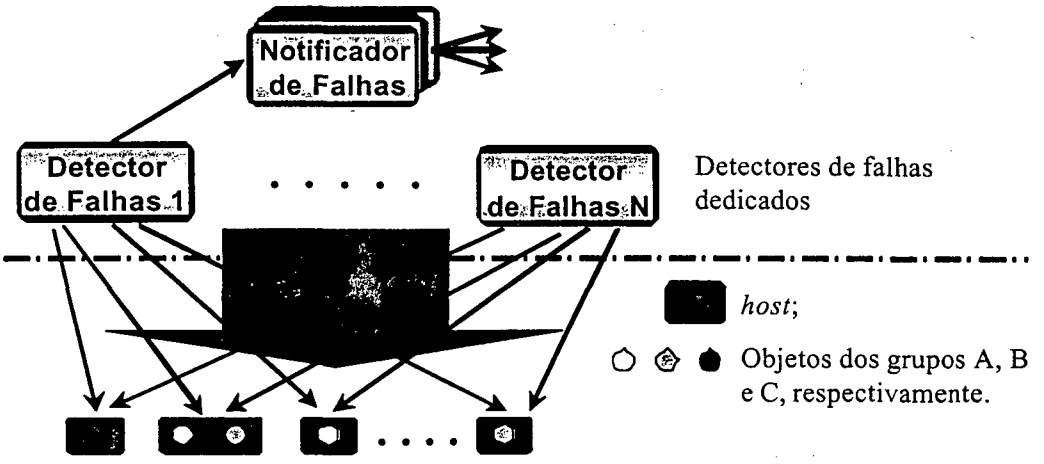


Figura 6.2. Monitoramento de falhas no GroupPac.

O serviço de detecção de falhas do *GroupPac* é concretizado em dois níveis de monitoramento: nível de detectores e nível de *hosts*. No primeiro, os detectores de falhas de *host* formam um grupo autogerenciável. Isto é, controlam as entradas e saídas (normal ou por falha) dos detectores membros do grupo. O monitoramento dos detectores é necessário para indicar o número de membros no grupo para, assim, definir a maioria de detectores durante um processo de votação.

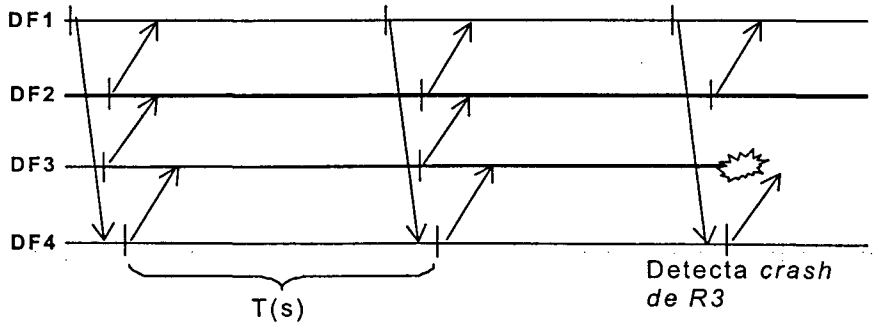


Figura 6.3. O serviço de detecção de falhas.

O monitoramento em nível de detectores usa um protocolo de *commit* centralizado (no detector primário), de duas fases (três fases no pior caso), para alcançar acordo (*agreement*) com os outros membros em relação à nova composição do grupo de detectores de falhas (DF). Este protocolo é baseado em [Ricciardi91] e foi descrito em [Lau99b]. Os membros deste grupo compõem um anel virtual e, periodicamente, cada membro monitora o parceiro imediatamente anterior da seqüência do anel. A coordenação na obtenção de uma nova lista de membros (*new view*) é centralizada no objeto detector de falhas primário DF1. A figura 6.4 mostra um exemplo de funcionamento desse protocolo. O objeto DF4 ao detectar um possível *crash* de DF3 envia a mensagem "suspeite de DF3" para o objeto

primário DF1. Neste ponto, o protocolo de *membership* é ativado e o primário difunde uma mensagem *commit* para detectar quem ainda continua no grupo, os detectores que estão ativos devem dar um reconhecimento a esta mensagem (*Ack*). Após um prazo de espera pré-definido, o primário DF1 produz uma nova lista de membros a partir dos *Ack* recebidos (que deve constituir uma maioria em relação ao *view* anterior). Caso ocorra vários particionamentos no grupo, a partição majoritária permanece ativa. Quando o detector de falhas primário (DF1) é suspeito de ter falhado; o candidato para substitui-lo segue a ordem implementada pelo anel virtual [Lau99a]. Assumimos um canal de comunicação ponto-a-ponto confiável e não bloqueante entre os membros do grupo SDF. Portanto, a falta de uma mensagem esperada não é percebida como devido à sua perda no meio de comunicação, mas, ao invés disso, como um indicativo de que algo errado está ocorrendo com o emissor.

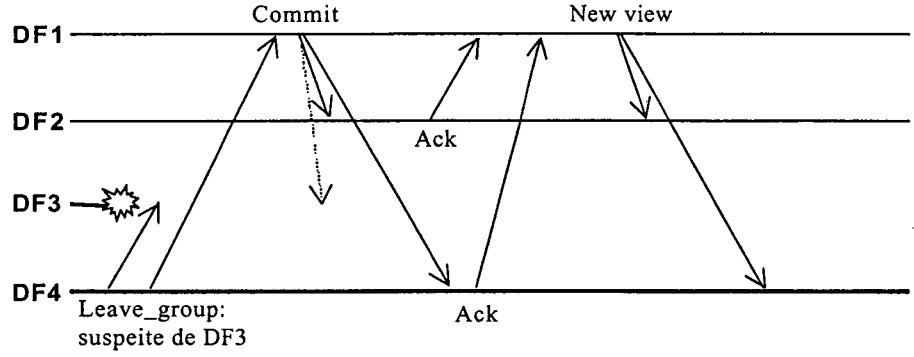


Figura 6.4. Situação: crash da réplica R2.

O outro nível de detecção de falhas no *GroupPac*, nível de *hosts*, tem então o grupo de detectores de falhas (DF_1, DF_2 DF_3 e DF_4 da figura 6.3) atuando nos *hosts* do domínio considerado. Na falha de um *host* são declarados como falhos todos os processos e objetos deste *host*. Os DFs monitoram periodicamente, dentro de um intervalo T(s), os mesmos *hosts* de um domínio de tolerância a faltas. São eles que decidem por maioria se um determinado *host* é faltoso (*crash*) ou não. Caso qualquer um dos detectores suspeite de uma falha de um *host*, o protocolo executa um procedimento, baseado em voto majoritário, para alcançar consenso. A coordenação deste procedimento de consenso é também centrada no detector *primário* na ordenação do anel.

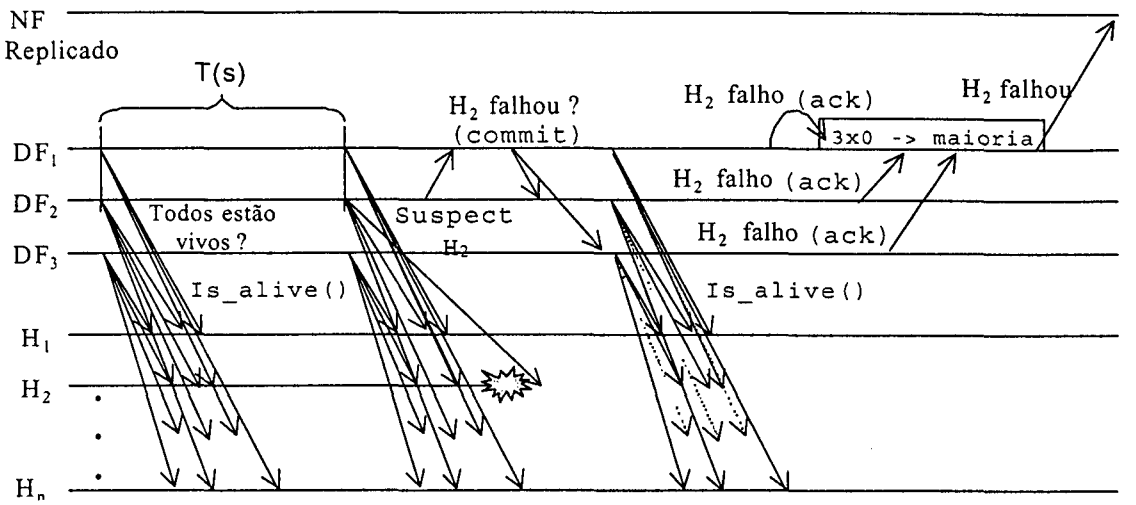


Figura 6.5. Uma instância do protocolo de detecção de falha.

A figura 6.5 apresenta uma instância desse protocolo, o DF2 suspeita do *host* H2 e avisa ao DF1 (o primário) sobre esta suspeita. A partir daí é iniciado o protocolo, em três passos, para o consenso e a sinalização da falha de H2. No primeiro passo, o DF1 solicita aos outros detectores (DF2 e DF3 da figura 6.5) para que no próximo intervalo de monitoração (T(s)) o informe sobre o status (vivo ou falho) do *host* H2. No segundo passo, após a nova monitoração, cada detector informa ao DF1 sobre sua posição em relação ao status de H2. Finalmente, no passo três, o detector de falhas primário (DF1), a partir dos resultados, decide sobre o status de H2. Então, o DF1 primário, através do notificador de falhas (NF da figura 6.5), informa ao gerenciador de replicação para que este gere uma nova IOGR (item 5.3.1.2), removendo H2 da lista. A nova IOGR é enviada ao grupo de detectores para que estes atualizem suas listas de *hosts* a serem monitorados.

6.3.3 Localizando grupos de objetos na hierarquia de domínios no GroupPac

Para que um objeto possa acessar um grupo de um outro domínio de tolerância a faltas, o *GroupPac* adapta o serviço de nomes CORBA em termos de tolerância a faltas e escalabilidade. Hierarquiza o serviço de nomes introduzindo um SN_G replicado (figura 6.6), que faz parte do domínio de tolerância global (figura 6.1). No modelo convencional, quando um objeto CORBA necessita acessar um outro objeto ou grupo de objeto, primeiramente, obtém a IOR do serviço de nomes que contém a referência do objeto (IOR) ou grupo de objetos (IOGR) que necessita ter acesso. Deste modo, todos os objetos, durante o seu ciclo de vida, acessam por pelo menos uma vez o serviço de nomes – ou para

registrar a sua IOR, ou para obter a IOR de um objeto remoto. Portanto, se o serviço de nomes falha todos os objetos do sistema ficam inacessíveis.

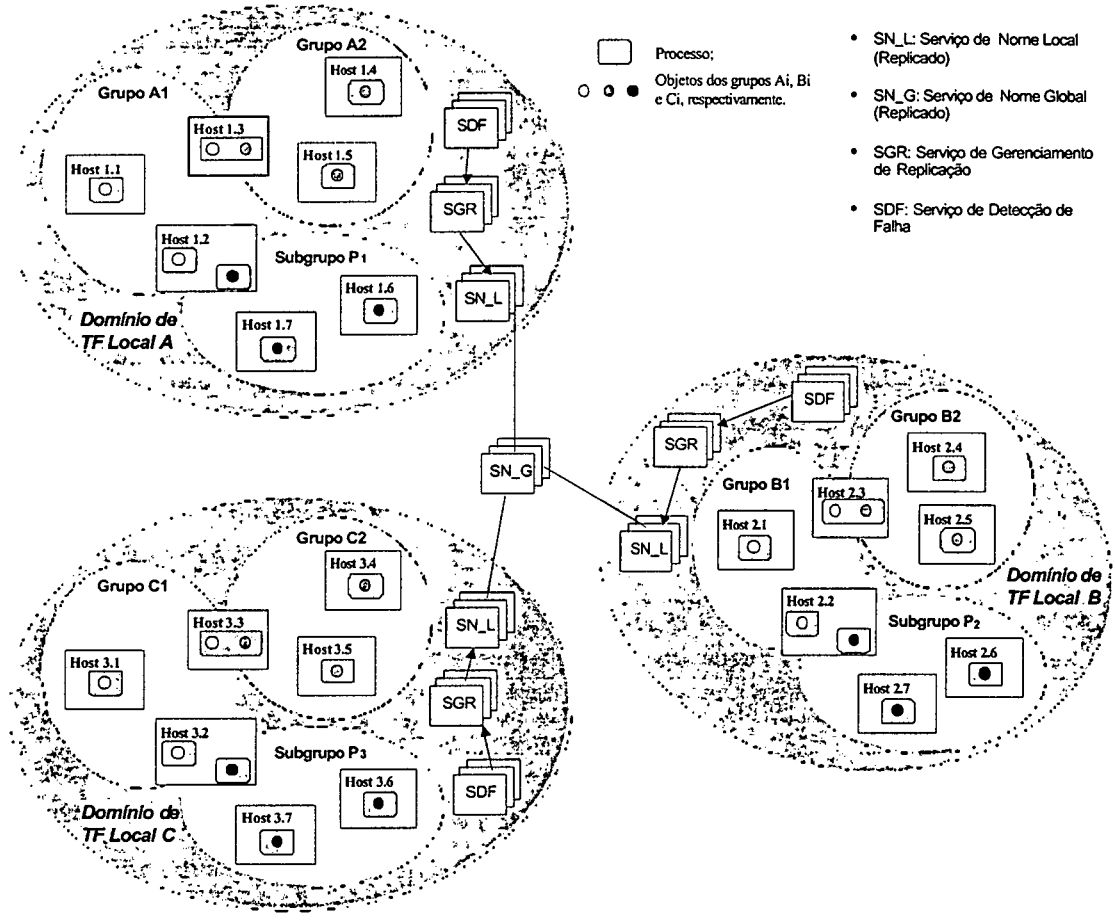


Figura 6.6. Suporte de tolerância a faltas do GroupPac em um sistema de larga escala.

Conforme apresentado no capitulo 5, a referência de objeto (IOR) do padrão CORBA foi estendida para IOGR para suportar grupo de objetos. Essa extensão no formato da referência de objeto não requereu nenhuma alteração no Serviço de Nomes (SN) do GroupPac. Uma referência de objeto para ser registrada no SN precisa ser convertida em uma seqüência de caracteres. Além disso, a essa seqüência é adicionado um cabeçalho contendo, entre outros parâmetros de controle, o tamanho da referência de objeto. Portanto, esta solução proporciona ao SN um controle flexível sobre as referências de objetos (IORs e IOGRs), permitindo que sejam salvos em diferentes tamanhos.

As soluções propostas no *GroupPac* são direcionadas no sentido de facilitar a escalabilidade. Buscamos uma forma de permitir que objetos, dos diferentes domínios de TF, pudessem ser localizados e se juntar ou sair de um grupo dinamicamente, durante o

ciclo de vida da replicação. Isto é conseguido através da associação de domínios de TF com o serviço de nomes – é importante ressaltar que a infraestrutura do FT-CORBA não possui uma interface para deixar disponível IOGRs a objetos clientes [OMG00a]. O serviço de nomes é então importante na localização destas IOGRs de grupos intradomínios (grupos *A1*, *A2*, *B1*, *B2*, *C1* e *C2* da figura 6.6) como de grupos interdomínios (subgrupos P_1, P_2 e P_3 do grupo *P*).

A composição do serviço de nomes mais domínios de TF, proposta no *GroupPac*, permite que objetos de um domínio localizem grupos e subgrupos de outros domínios de tolerância a faltas. O serviço de nomes no *GroupPac* é também baseado em uma hierarquia para facilitar o trato da complexidade de sistemas de larga escala. Esta hierarquia é formada com a introdução de um serviço de nomes global, replicado por questões de disponibilidade e de tolerância a faltas (grupo SN_G, na figura 6.6). Esta hierarquia, apresentada na figura 6.6, divide o serviço de nomes em dois níveis:

- *Serviço de Nomes Replicado Local* (SN_L): responsável por gerenciar todas as IOGRs (e IORs) dos objetos da aplicação de um domínio de tolerância a faltas. O SN_L possui também, em seu contexto de nomes, uma cópia da IOGR do SN_G;

- *Serviço de Nomes Replicado Global* (SN_G): responsável por gerenciar a IOGR de cada SN_L de domínio – possui cópias atualizadas das IOGRs dos SN_L de todos os domínios registrados.

O grupo SN_G que faz parte do domínio de TF global permite o acesso a grupos ou subgrupos de um domínio de tolerância a faltas a partir de outros domínios. No modelo apresentado na figura 6.6, a principal função do serviço de nomes global (grupo SN_G) é registrar as IOGRs dos SN_Ls de cada domínio. Além disso, sobre o serviço de nomes em si, tanto os SN_Ls como o SN_G seguem as especificações *CosNaming* [OMG97a] definidas pela OMG, e podem ser tolerantes a faltas com o CosNamingFT (capítulo 4). No modelo convencional quando um objeto CORBA necessita acesso a um outro objeto ou grupo é, primeiramente, necessário que este obtenha a IOR do serviço de nomes que contém a referência do serviço de aplicação desejado (IOR ou IOGR). Depois então, de posse desta última referência, o acesso desejado é efetuado.

A estruturação dos nomes e a associação de contextos de nomes (por exemplo, entre o SN_L e o SN_G) estão padronizadas nas especificações do serviço de nomes (CosNaming) [OMG97a]. Um nome estruturado para alcançar um objeto é formado por um conjunto de componentes (*NameComponent*) que formam um caminho de nomes. Tal como diretórios em sistemas de arquivo, contextos de nomes podem conter outros contextos de nomes, permitindo, deste modo, formar uma hierarquia estruturada em vários níveis. Por exemplo, se o nome de um objeto é estruturado com três componentes, o primeiro referenciaria o contexto raiz (no nosso caso, o SN_G), o segundo o subcontexto (SN_L), e o terceiro o objeto alvo (o objeto da aplicação). No *GroupPac*, em acessos interdomínios, há três níveis de resolução de nomes para que IORs de objetos de aplicação sejam obtidas.

6.3.3.1 Tolerância a faltas no serviço de nomes do GroupPac

O serviço de nomes, essencial na ligação entre objetos no sistema, deve atender a requisitos de tolerância a faltas. Devido à disposição e função dos SN_Ls e SN_G no sistema, temos diferentes propriedades de tolerância a faltas atribuídas a cada um destes grupos de serviços.

Tolerância a faltas nos Serviços de Nomes Locais

O serviço nomes local (SN_L) em cada domínio é implementado utilizando os serviços do *GroupPac* e a técnica de replicação primário/*backups* [Lau99b]. Tal como os outros grupos da aplicação, cada SN_L replicado é gerenciado como um grupo de seu domínio de TF (item 6.3). O primário do grupo SN_L mantém atualizados os *backups* (através de atualizações periódicas), utilizando o serviço de recuperação e *Logging*. O SN_L primário é responsável por registrar e tornar disponíveis todas as referências de objetos e grupos do domínio. No entanto, no sentido de melhorar o desempenho, requisições de operação de obtenção de referência (operação do tipo *stateless*) são atendidas por qualquer réplica *backup* [Lau99a]. Para melhor acessibilidade, as réplicas do SN_L devem estar espalhadas em diferentes pontos do domínio de TF. Todavia, é importante ressaltar que só o SN_L primário registra a IOGR de seu grupo no serviço de nomes global (SN_G). O SN_L primário faz também esta IOGR disponível em servidores HTTP como forma de redundância e para facilitar o acesso a objetos do seu domínio.

Quando a falha do SN_L primário (*crash*) é detectada (através do protocolo apresentado no item 6.3.2), é definido então, um novo primário entre seus *backups* – o primeiro da ordem definida na IOGR – e homologado pelo SGR do domínio. Com isso, o novo SN_L primário deve se registrar como o novo representante do domínio, tornando disponível a nova IOGR do grupo, nos servidores HTTP e no SN_G.

Tolerância a falta no Serviço de Nomes Global

Outras soluções foram adotadas no *GroupPac* na replicação do serviço de nomes global (SN_G). Por conter todas as IOGRs dos serviços contidos no domínio de TF Global e também dos grupos SN_Ls, o SN_G deve ser de fácil acesso para todos os objetos de cada domínio de TF Local. Portanto, as réplicas do SN_G, em termos geográficos e para melhor acessibilidade, devem ser bem distribuídas no sistema. A característica principal do SN_G é, portanto, de ser um grupo geograficamente distribuído com poucos elementos.

Adotar a técnica de replicação primário/*backup*, com um elemento primário centralizador, não é uma solução adequada exatamente por comprometer o acesso e o desempenho em um contexto de larga escala. A solução proposta indica que as réplicas do SN_G sejam distribuídas no sistema, com pelo menos uma réplica em algum *host* de cada domínio de TF Local, sendo todas ativas, exercendo as mesmas funções (registrar e tornar disponíveis IOGRs). A solução proposta implica no uso de propriedades da técnica de replicação ativa onde todas as réplicas não faltosas do grupo recebem, executam e respondem a todas requisições dos clientes [Schneider90]. Por ter todas as réplicas ativas executando o mesmo conjunto de requisições é necessário que o grupo seja determinista. O determinismo de réplicas é conseguido assegurando as regras de acordo e ordenação.

No entanto, se considerarmos as operações em um serviço de nomes global, uma requisição para obter a IOGR de um SN_L de um determinado domínio é, em essência, uma operação *stateless* – não altera o estado do SN_G. Além disso, as requisições de registro de novas IOGRs, de diferentes grupos, podem ser tratadas de forma independente – não existe uma relação de ordem entre as mesmas – o que torna desnecessário estabelecer uma ordem total no conjunto de requisições de registro de IOGRs. É considerado, também, que a carga de acesso para registros de IOGRs dos SN_L no SN_G é bem menor que os registros em um SN_L de um domínio qualquer. Diante disso, podemos

utilizar protocolos mais flexíveis, considerando que o único requisito necessário para dar suporte ao SN_G é a garantia na ordem do emissor nas atualizações das IOGRs dos SN_L. Portanto, para diminuir os custos e as complicações de um protocolo de difusão atômica no domínio de TF global que envolve grandes dimensões, propomos uma solução menos custosa. Um protocolo de difusão confiável com propriedade de ordenação FIFO (*multicast* FIFO [Hadzilacos93]) pode ser usado, o que atende perfeitamente aos propósitos de operações de registro de IOGR. Operações de solicitação de IOGR não precisam ser difundidas para todas réplicas do SN_G. Uma das réplicas, preferivelmente a mais próxima, pode atender a essa requisição.

Diante dessas considerações, apresentamos, na figura 6.7, o serviço de nomes global (SN_G). Diferentes de objetos, que segundo as especificações FT-CORBA não podem pertencer a mais de um domínio de tolerância a faltas, os *hosts*, ao contrário, não possuem tais restrições (item 6.2.1). Apesar das réplicas do SN_G fazerem parte, exclusivamente, do domínio de TF global, as mesmas podem ser instaladas em *hosts* de domínios locais, no sentido tornar o acesso dos objetos da aplicação mais eficiente.

As réplicas do SN_G formam um grupo que é, tal como SN_L nos domínios locais, gerenciado pelos serviços do *GroupPac* pertencentes ao domínio de tolerância a faltas global. Devido ao maior distanciamentos das réplicas do SN_G, intervalos de *timeout* mais longos ou protocolos de detecção de falhas [Chandra96, Ricciardi91] mais adequados às características do grupo devem ser considerados. O SDF e o SGR deste domínio realizam a tarefa de manter, através da IOGR, uma lista atualizada dos membros (*membership*) do SN_G. Em qualquer alteração na composição dos membros do grupo SN_G é gerada uma nova IOGR e enviada pelo SGR para os SN_L de cada domínio (as setas numeradas da figura 6.7a indicam o movimento destas informações). De forma similar, quando uma nova IOGR de grupo SN_L é gerada, o SGR do mesmo domínio local se encarrega em enviar uma cópia para uma réplica do SN_G, a mais próxima do domínio local. Esta réplica SN_G registra e difunde a IOGR (usando *multicast* confiável FIFO) para as outras réplicas SN_G do grupo. As setas indicadas na figura 6.7b descrevem estas ações envolvendo o registro de uma nova IOGR de grupo SN_L.

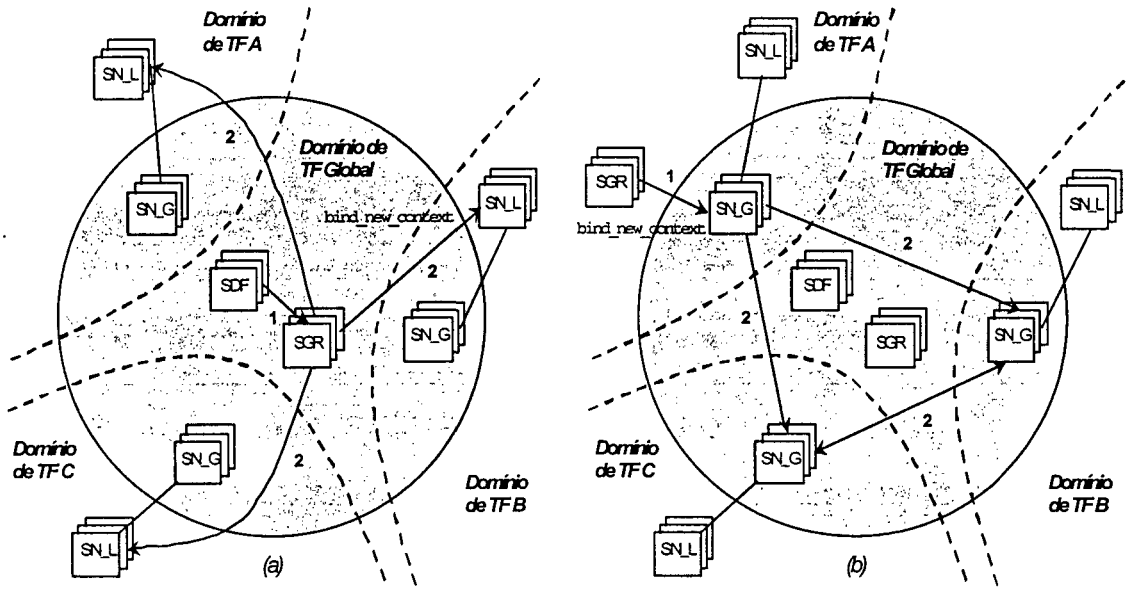


Figura 6.7. Serviço de nomes para suporte ao gerenciamento de grupo no GroupPac.

Se o desempenho do grupo SN_G se tornar fraco devido às dimensões geográficas e o número de domínios de TF locais em um sistema distribuído, a solução mais adequada é não mais localizar as réplicas membros do SN_G em *hosts* dos vários domínios de TF locais. O domínio de TF global teria o grupo SN_G distribuídos em um conjunto de *hosts* próprios deste domínio em uma grande distribuição geográfica. Mesmo assim, a associação de determinadas réplicas de SN_G com um SN_L de um domínio de TF local pode ser mantida, usando as informações disponíveis na IOGR do SN_G, disponível no contexto de nomes do domínio local (no SN_L do domínio). As figuras 6.7a e 6.7b, com os seus exemplos, são válidas também para esta situação.

6.3.4 Comunicação de grupo no modelo hierárquico do GroupPac

A partir do modelo de hierarquia de domínios adotado pelo *GroupPac* são detectados três tipos de comunicações de grupos: comunicações com grupos intradomínios do mesmo domínio de TF do cliente; comunicações de cliente com grupos intradomínios de domínios de TF diferentes daquele ao qual pertence e, comunicações com grupos interdomínios.

O primeiro tipo, envolvendo comunicações entre clientes e grupos pertencentes ao mesmo domínio, é bastante simples. Corresponde a uma requisição inicial do cliente no SN_L para liberar a IOGR e com isto, conseguir na seqüência a ligação com o grupo desejado. A comunicação de grupo se dará usando interceptador e ferramenta proprietária

disponível na forma de objeto de serviço (SCG, capítulo 3) do domínio considerado (item 6.3).

Quando um cliente e um grupo pertencem a domínios de TF diferentes, a comunicação entre ambos deve passar por uma resolução de nomes mais complicada. O objeto cliente, em um domínio *A*, ao desejar invocar um grupo de um domínio *B*, deve executar as seguintes ações (figura 6.1):

1. Acessar o serviço de nomes local (SN_L) de seu domínio para obter a referência (IOGR) do serviço de nomes global (SN_G);

2. Acessar o SN_G para obter a IOGR do SN_L do domínio desejado, o qual contém o grupo que deseja invocar;

3. Finalmente, acessar o SN_L do domínio B para obter a referência de grupo servidor (IOGR) para a ligação desejada.

A comunicação de grupo deste caso também se dará usando interceptador e uma ferramenta proprietária disponível na forma de objeto de serviço (capítulo 3). Só que este objeto de serviço e a correspondente ferramenta que implementa a comunicação de grupo pertencem ao domínio de TF do grupo servidor.

Os dois primeiros tipos de comunicações citados acima envolvem algumas trocas que depende essencialmente dos níveis de resolução necessários para estabelecer a ligação entre o objeto cliente e o grupo servidor. O terceiro tipo de comunicação citado acima envolve grupos interdomínios e é discutido nas subseções subseqüentes.

6.3.4.1 Comunicação de grupo interdomínio

A comunicação com grupos escaláveis que tem suas réplicas distribuídas em diferentes domínios de tolerância a faltas (grupos interdomínios), necessita de um suporte mais complexo do que os tipos apresentados anteriormente. Na literatura são encontradas soluções algorítmicas para comunicação de grupo (inclusive com propriedades de difusão atômica) onde os grupos se caracterizam pela dispersão de seus membros em redes de larga escala. Estes trabalhos normalmente resolvem a comunicação com estes grupos através do particionamento em vários subgrupos onde as propriedades de comunicação seriam

tratadas de maneira mais viável, localmente, em cada subgrupo [Rodrigues96, Fritzke98]. Ou seja, o particionamento do grupo deve favorecer a propriedade de minimidade [Guerraoui97] e de localidade [Fritzke98].

Se considerarmos as especificações FT-CORBA e as proposições no *GroupPac*, um grupo interdomínio teria, portanto, seus subgrupos distribuídos em diferentes domínios de tolerância a faltas locais. Cada domínio, que contém um subgrupo, poderia, portanto, ter um serviço de comunicação de grupo (SCG) usando protocolos de comunicação de grupo distintos.

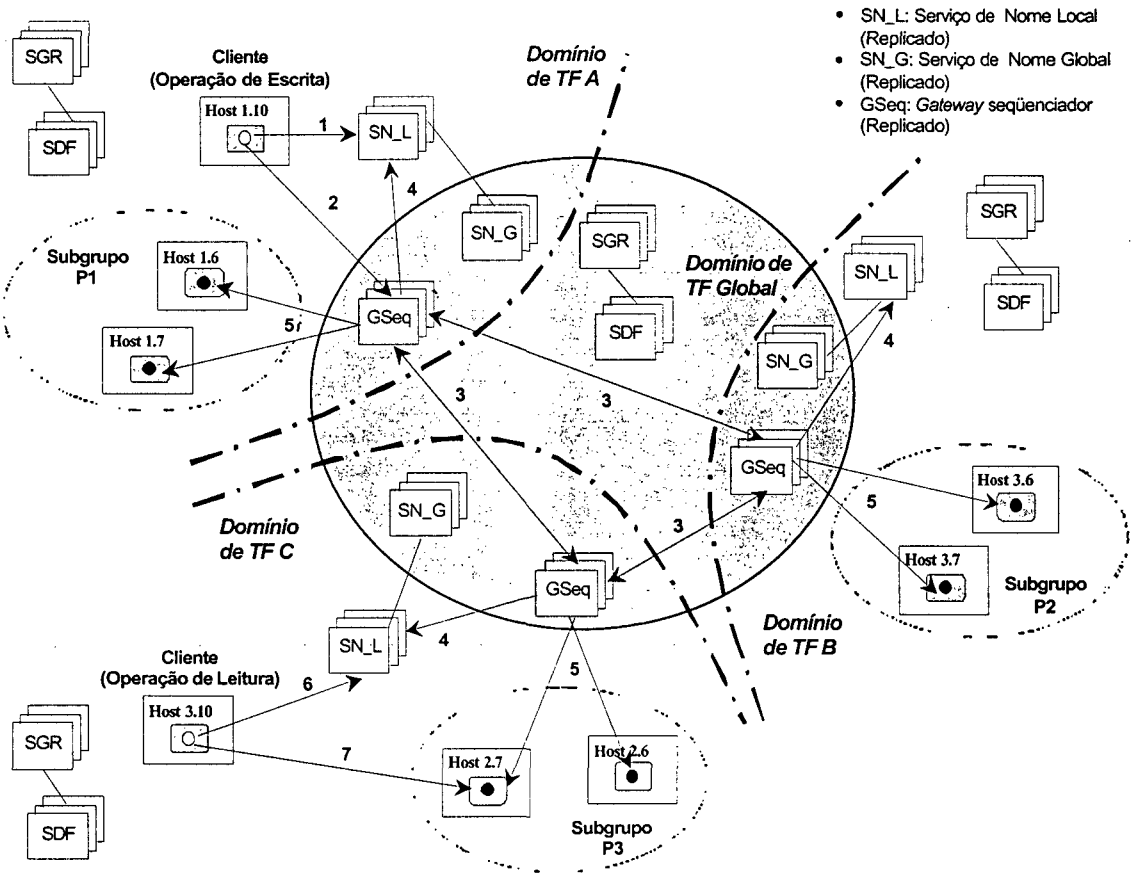


Figura 6.8. Comunicação de grupo para sistemas de larga escala usando o GSeq.

A figura 6.8 ilustra um exemplo destes grupos interdomínios que vai nos permitir entender melhor a dinâmica que envolve estes grupos. O grupo de objetos *P* desta figura é formado por um conjunto de subgrupos *P1*, *P2* e *P3*, distribuídos nos domínios locais *A*, *B* e *C*, respectivamente. Cada subgrupo é gerenciado e controlado, separadamente, pelos serviços do *GroupPac* de cada domínio (item 6.3.1). Além disso, quando uma nova réplica

é inserida em um dos subgrupos de *P*, pelo SGR do domínio de TF local correspondente, esta nova réplica tem seu estado atualizado através do Serviço de *Logging* e Recuperação [OMG00a] do *GroupPac*. Portanto, a lista de membros (*view*) de um grupo particionado em vários domínios neste modelo pode ser dinâmica.

Domínio de TF	Nome	Referência de Grupo (Subgrupos de *P*)
A	"HAL"	IOGR1 do subgrupo P1
B	"HAL"	IOGR2 do subgrupo P2
C	"HAL"	IOGR3 do subgrupo P3
D	"HAL"	IOGR4 do subgrupo P4

Tabela 6.1. Registro do Grupo P particionado em quatros domínios de TF.

Para que o modelo da figura 6.8 possa ser plausível, estendemos a noção de nome e referência de grupo de objetos (IOGR) do serviço de nomes para suportar grupos interdomínios. Todo grupo que for criado e que tiver em sua especificação a possibilidade de ter suas réplicas distribuídas em vários domínios deve assumir um mesmo nome para cada IOGR de subgrupo registrado no SN_L de cada domínio de TF. Isto é, as IOGRs de cada subgrupo (*P1, P2 e P2*), que compõe o grupo *P*, devem ter o mesmo nome registrado nos SN_Ls dos domínios locais do grupo. Se um novo subgrupo de *P* for introduzido em um outro domínio (por exemplo, em um domínio de TF *D*), este subgrupo terá uma IOGR gerada pelo SGR e deverá, também, ser registrado no SN_L de domínio *D* com o mesmo nome definido para o grupo de objeto *P* (por exemplo, "HAL" na tabela 6.1). Logicamente, apesar de ter o mesmo nome, as três IOGRs da tabela são diferentes entre si. Uma vez que cada domínio tem o seu próprio SN_L não há problema em se registrar várias IOGRs diferentes com o mesmo nome.

No sentido de viabilizar a comunicação de grupos interdomínios no *GroupPac*, são propostos na seqüência: (i) duas novas propriedades de tolerância a faltas a serem introduzidas no FT-CORBA; (ii) uma extensão na estrutura das IOGRs; e (iii) um serviço que permite a comunicação com grupos interdomínios seguindo a hierarquia de domínios proposta.

6.3.4.1.1 Propriedades de TF para dar suporte a escalabilidade

As propriedades de tolerância a faltas, definidas no FT-CORBA, são associadas para cada grupo de objeto em um domínio de tolerância a faltas. Estas propriedades são

gerenciadas pelo serviço de *gerenciamento de propriedades* (figura 5.1) e podem ser estabelecidas como valor padrão (*default*) para o domínio de tolerância a faltas, pelo tipo de objeto (aplicação), quando um grupo de objetos é criado, ou enquanto o grupo está executando. Dentre estas propriedades estão o tipo de replicação (*ReplicationStyle*), estilo de monitoramento de falhas (*FaultMonitoringStyle*), número inicial de réplicas (*InitialNumberReplicas*), intervalo de monitoramento de falhas (*FaultMonitoringInterval*) e intervalo de atualização de estado (*CheckpointInterval*). De acordo com as especificações, estas propriedades, e seus valores (parâmetros), podem ser estendidos de acordo com as necessidades da aplicação – as propriedades de TF são informações adicionais que, dependendo da aplicação, podem ser utilizadas ou não durante a execução do sistema, e não representam um ônus nas especificações FT-CORBA. Portanto, para dar suporte aos grupos que podem ser particionados em diferentes domínios, introduzimos duas novas propriedades:

- *InterdomainGroup*: determina se um grupo contém membros em diferentes domínios (interdomínio) ou não;
- *OrderingType*: determina o tipo de ordenação a ser utilizada pelos *Gateway Seqüenciadores* (item 6.3.4.1.3), se *Unreliable*, *Reliable*, *Causal* ou *Total*.

Esta última propriedade depende do suporte à ordenação de mensagens provido pela ferramenta proprietária usada na comunicação de grupo no domínio através do SCG. A necessidade de adicionar essas duas novas propriedades é devido a IOGR (item 5.3.1.2) que, conforme a especificação no FT-CORBA, limita um grupo em um único domínio de tolerância a faltas.

6.3.4.1.2 Extensão da IOGR

As especificações CORBA definem que a referência de objeto ou de grupo de objetos, além de carregar informações sobre a localização do objeto no sistema, pode agregar algumas informações adicionais referentes ao tipo de aplicação. As informações adicionais são mapeadas através de uma estrutura de dados, conhecida como etiqueta (TAG). As etiquetas podem ser adicionadas ao final da estrutura de uma referência de objeto (IOR ou IOGR). Qualquer informação contida numa TAG pode, também dependendo da aplicação, ser lida ou não durante a execução. No *GroupPac* estendemos a

IOGR com a introdução de uma etiqueta. Esta etiqueta, chamada de TAG_FTProperties, conterá todas as propriedades de tolerância a faltas definidas para o grupo correspondente – inclusive as propostas no item 6.3.4.1.1. É importante ressaltar que a inclusão desta etiqueta é uma extensão da IOGR, mas que não afeta a interoperabilidade prevista no FT-CORBA. Esta extensão objetiva permitir ao cliente (através do seu interceptador de mensagem), quando da obtenção de uma IOGR, conhecer, de antemão, todas as propriedades de tolerância a faltas do grupo – inclusive se este é ou não interdomínio.

6.3.4.1.3 GSeq – Gateway Seqüenciador

No modelo da figura 6.8 é introduzido um mecanismo de ordenação de mensagem, chamado de *Gateway Seqüenciador* (GSeq), que é parte importante na comunicação de grupo interdomínios. O GSeq é um grupo de objetos, tal como o SN_G, faz parte do domínio de TF Global – regido, portanto, pelos mesmos serviços e protocolos definidos nesse domínio (item 6.3.1). Se uma nova réplica do GSeq é adicionada no grupo pelo serviço de gerenciamento de replicação (SGR), o Serviço de *Logging* e Recuperação se encarrega de atualizar esta nova réplica, tornando-a uma cópia idêntica. O GSeq tem uma única IOGR gerada pelo SGR do domínio global. Cada réplica do GSeq pode se comunicar com as outras réplicas de seu grupo. Qualquer alteração na lista de membros do grupo GSeq é gerenciada pelo SGR do domínio global, o qual realiza a função de gerar e atualizar a lista (a IOGR) das réplicas do GSeq.

O grupo GSeq tem como principal função: servir de ponte para a difusão de uma mensagem entre os diferentes subgrupos de um grupo interdomínio. Qualquer domínio de TF local deve estar sempre associado a uma ou mais réplicas do GSeq. Quanto mais réplicas do GSeq estiverem associadas a um domínio de TF local, maior é o grau de tolerância a faltas do sistema. Apesar de fazerem parte apenas do domínio de TF global, as réplicas do GSeq podem ser posicionadas em *hosts* de diferentes domínios locais de TF no sentido de aumentar a disponibilidade e o desempenho no sistema. Neste modelo, o grupo GSeq fornece seus serviços para qualquer grupo interdomínio do sistema.

A figura 6.8 apresenta a seqüência de passos, realizada pelo GSeq, na difusão de uma requisição de cliente em um grupo interdomínio. Quando um cliente obtém a IOGR de um grupo (seta 1 da figura 6.8), no SN_L de um domínio de TF qualquer, o mecanismo de

interceptação de mensagens [Lau00a] no cliente verifica, primeiramente, através da TAG_FTProperties da IOGR (item 6.3.4.1.2), se o grupo é interdomínio ou não. Caso não seja, é feita a invocação direta para o grupo (intradomínio), usando o SCG de acordo com o tipo de comunicação adequado apresentado no início do item 6.3.4. Em caso afirmativo, o interceptador do cliente desvia a requisição, incluindo o nome do grupo (ex: "HAL"), para uma das réplicas do GSeq associada ao domínio do cliente (seta 2 da figura 6.8). Como o GSeq é replicado e distribuído nos domínios de tolerância a faltas locais, o interceptador faz a função de localizar um dos GSeq associados ao seu domínio. Neste ponto, as réplicas do GSeq usam um serviço de *multicast* (SCG do domínio global) para a disseminação da requisição no grupo GSeq, segundo a ordem definida pelo parâmetro *OrderingType* (seta 3 da figura 6.8). Uma vez executado este protocolo, as réplicas do GSeq entram em contato com o SN_L correspondente ao domínio local associado (seta 4 da figura 6.8), enviando o nome do grupo ("HAL") e obtendo a IOGR do subgrupo de *P* do domínio ao qual pertence. Após isto, as réplicas do GSeq enviam a mensagem para o subgrupo do domínio em que estão localizados (seta 5 da figura 6.8). Neste passo, é importante observar que a entrega da mensagem *m*, do GSeq para o subgrupo do domínio, está vinculada ao estilo de replicação definida na etiqueta TAG_FTProperties (item 6.3.4.1.1). O estilo de replicação deve ser o mesmo para todos subgrupos de *P*. Se o estilo for replicação ativa, a mensagem *m* é enviada para todos os membros do subgrupo, com garantia de ordem FIFO[6] – mensagens repetidas são descartadas pelos interceptadores de mensagem. Se for definida a técnica de replicação passiva (fria ou morna) ou semi-ativa (ainda não definida na especificação), em que há a figura de um elemento réplica primária, a entrega da mensagem *m* é direcionada para esta réplica, de acordo com a especificação definida no FT-CORBA. Lembrando que as especificações FT-CORBA definem, claramente, como o suporte para a técnica de replicação passiva morna e fria deve ser implementado. O que não é o caso para as técnicas de replicação ativa e semi-ativa (anexo A).

Caso uma requisição de um cliente tenha tanto parâmetros de saída (pedido – operação de escrita) como de entrada (resposta – operação de leitura), o que implica em uma resposta por parte dos membros do grupo *P*, as respostas percorrem o mesmo caminho da requisição e a que chegar primeiro no cliente é entregue, o resto pode, opcionalmente, ser descartado pelo interceptador.

[6] Como, logicamente, há apenas um emissor a garantia de ordenação FIFO é suficiente.

Em uma operação de apenas leitura (*stateless*), o cliente obtém a IOGR do subgrupo de *P* (seta 6 da figura 6.8) e o interceptador de mensagem do cliente verifica esse status de leitura. Uma alternativa para melhor desempenho, a requisição pode ser direcionada para uma das réplicas (a mais próxima) do domínio. Caso a réplica selecionada falhe durante a requisição de leitura, o interceptador busca a próxima réplica contida na IOGR. Esse procedimento prossegue até a obtenção da resposta.

O grupo GSeq que pertence ao domínio de TF global, nas comunicações de grupo usa o SCG que pode ser implementado por objetos de serviços encapsulando as funcionalidades de uma ferramenta de comunicação de grupo (tal como Isis [Birman91], Consul [Mishra93], Horus [Renesse95], Newtop [Ezhilchevan95], etc.) ou ter seu próprio protocolo, implementado em nível de aplicação como objeto de serviço CORBA. No item seguinte é discutida, em mais detalhes, a difusão atômica de uma requisição em um grupo interdomínios.

Difusão atômica em grupos interdomínios

Como exemplo, na figura 6.9 apresentamos o uso do mecanismo GSeq na difusão atômica de uma requisição em um grupo interdomínios. Em termos de suporte para a comunicação de grupo interdomínios, as funções do grupo GSeq estão divididas em três fases. As fases 1 e 3 já foram explicadas acima. De forma geral, a fase 1 representa a ativação do GSeq que deve ordenar a mensagem no sistema global. A fase 3 representa a entrega da requisição nos objetos dos subgrupos. Estas duas fases definem relações entre o serviço de gerenciamento de replicações (SGR), o serviço de nomes, nos seus diversos níveis (SN_L e SN_G), com as réplicas do GSeq (figura 6.8). A fase 2 envolve uma difusão atômica no grupo GSeq e a fase 3 uma difusão confiável nos subgrupos nos vários domínios de TF locais. A fase 2 garante a ordem total e a seqüência das difusões confiáveis sobre os subgrupos do grupo interdomínios.

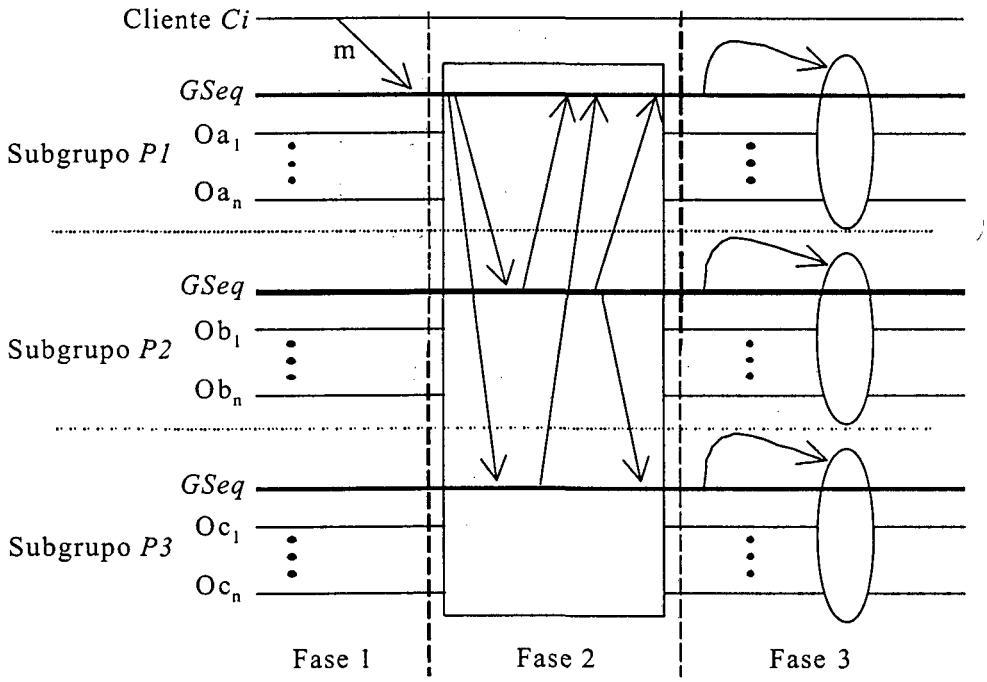


Figura 6.9. Fases do protocolo GSeq.

A difusão atômica em grupos interdomínios não implica em grupos fechados, ou seja, os clientes não necessariamente pertencem ao grupo. Outro aspecto em relação ao GSeq é que seus membros estão sujeitos aos serviços de detecção de falhas (SDF) e de *membership* fornecido pelo SGR do domínio global. A lista de membros (*view*) do grupo GSeq pode ser extraída a partir da sua IOGR enviada pelo SGR a cada réplica do GSeq. A IOGR possui um campo que identifica o seu número de versão. Na figura 6.9, quando um objeto cliente (Ci) deseja enviar uma mensagem *m* a um grupo interdomínios *P*, envia *m* a um dos *Gateway Seqüenciadores* (GSeq) associados ao seu domínio. À mensagem *m* (requisição), é adicionada as seguintes informações:

- *Ci*: identificador do objeto emissor;
- *Nd*: nome do domínio de tolerância a faltas ao qual o objeto emissor pertence (o mesmo nome registrado no SN_G);
- *Ng*: Nome do grupo interdomínio (ex: "HAL", ver tabela 6.1, o mesmo registrado nos SN_L) – necessário na determinação das IOGRs dos subgrupos ao final da ordenação, para usar nas difusões confiáveis (fase 3).

6.4 Implementação do GroupPac

Discutiremos nesta seção aspectos de implementação do *GroupPac* para redes de larga escala. Detalhes de implementação estritamente ligados ao FT-CORBA foram descritos no capítulo 5. O objetivo, desta seção, é mostrar as interfaces dos serviços que compõem o *GroupPac* para dar suporte à escalabilidade e alguns aspectos de implementação. A implementação do *GroupPac* segue o modelo de objeto de serviço COSS/OMG. Todos os serviços propostos foram implementados em Java, o que contribui com o aspecto da portabilidade de linguagem de programação, e suas interfaces especificadas segundo o padrão IDL/OMG. As implementações foram desenvolvidas e testadas na plataforma JacORB [Brose97], uma implementação, em Java, da especificação CORBA 2.3 [OMG96]. Esse *middleware* corresponde a um conjunto de suportes e ferramentas de desenvolvimento que permitem construir e integrar aplicações orientadas a objetos em sistemas abertos.

6.4.1 Implementação do serviço de detecção de falhas

Para implementar as funcionalidades descritas na subseção 6.3.2 introduzimos, como uma extensão do FT-CORBA, as interfaces `FaultDetector` e `FaultDetector_FDLevel` – observe que ambas herdam a interface `PullMonitorable` (item 5.4.1.3). Conforme descrito no item 4.2, a detecção de falhas é realizada em dois níveis de monitoramento: em nível de *host* (`FaultDetector`) e em nível detectores (`FaultDetector_FDLevel`). Os detectores de falhas realizam a tarefa de monitoramento através da referência de grupo (IOGR) que contém o endereço de cada membro de um grupo a ser monitorado. A figura 6.5 mostra o monitoramento em nível *host* e relaciona cada operação apresentada na interface `FaultDetector` (figura 6.10).

A interface `FaultDetector` (figura 6.10, linha 7) herda a interface `PullMonitorable` exatamente para permitir o monitoramento em nível de *host*. E a interface `FaultDetector_FDLevel` (figura 6.10, linha 19) fornece as operações para os detectores de falhas realizarem o monitoramento entre seus pares (em nível de detectores), conforme descrito no item 6.3.2.

```
1.  module CORBA {
2.    module FT {
i.                    :
3.    typedef long FaultDetectorStyleValue;
4.    const FaultDetectorStyleValue HOST=0;
5.    const FaultDetectorStyleValue PROCESS=1;
6.    const FaultDetectorStyleValue OBJECTS=2;
                      :
7.      interface FaultDetector:PullMonitorable {
8.        // Obtém a IOGR do grupo a ser monitorado
9.        void add_iogr(in IOP::IOR new_iogr);
10.       // Atualiza a IOGR (uma nova lista dos membros a serem monitorados)
11.       void update_iogr(in IOP::IOR new_iogr);
12.       // Recebe a IOR do objeto suspeito
13.       void suspect(Object reference);
14.       // Pergunta a todos sobre o status do objeto suspeito
15.       void commit(Object reference);
16.       // Recebe o status do objeto suspeito e decide
17.       void ack(boolean is_failure);
18.     };
19.     interface FaultDetector_FDLevel:PullMonitorable {
20.       // Adiciona a IOGR do grupo FD a ser monitorado
21.       void add_iogr(in IOP::IOR new_iogr);
22.     // Atualiza a IOGR (uma nova lista dos membros de FD a serem
    monitorados)
23.       void update_iogr(in IOP::IOR new_iogr);
24.       // Recebe a IOR do FD suspeito
25.       void suspect(Object reference);
26.       // Pergunta a todos sobre o status do FD suspeito
27.       void commit(Object reference);
28.       // Recebe o status do FD suspeito e decide
29.       void ack(boolean is_failure);
30.     };
31.   };
32. };
```

Figura 6.10. Interface FaultDetector.

6.4.2 SGR

O *Serviço de Gerenciamento de Replicação* (SGR) faz a função de gerar a IOGR de um grupo de objetos e gerenciar as propriedades de tolerância a falhas deste, através do *Serviço de Gerenciamento de Propriedades*. Portanto, as extensões realizadas no SGR se limitam na inclusão de duas novas propriedades de tolerância a falhas e na inclusão dessas propriedades, na forma de TAG, dentro da estrutura da IOGR.

6.4.2.1 Adicionando Novas Propriedades de TF

A interface `PropertyManager`, da especificação FT-CORBA, fornece um conjunto de operações para manipular (atribuir, obter, remover, etc) as propriedades de tolerância a falhas de um grupo de objetos. Para incluir as propriedades de tolerância a falhas proposta no item 6.3.4.1.1 não foram necessárias quaisquer modificações ou extensões da interface `PropertyManager`. A figura 6.11 apresenta um fragmento dos tipos de dados do módulo

de tolerância a falhas CORBA que irão compor uma IOGR. Nela foram incluídas as propriedades `ParticionedGroupValue` e `OrderingTypeValue` (linha 17 em diante). Essas duas propriedades serão lidas pelo interceptador de mensagem para iniciar o procedimento de comunicação de grupo através do grupo GSeq.

```
1.  module CORBA {
2.    module FT {
i.                  :
3.      typedef CosNaming::Name Name;
4.      typedef any Value;
5.      struct Property {
6.        Name nam;
7.        Value val;
8.      };
9.      typedef sequence<Property> Properties;
i.                  :
10.     typedef sequence<FactoryInfo> FactoryInfos;
11.     typedef long ReplicationStyleValue;
12.     const ReplicationStyleValue STATELESS = 0;
13.     const ReplicationStyleValue COLD_PASSIVE = 1;
14.     const ReplicationStyleValue WARM_PASSIVE = 2;
15.     const ReplicationStyleValue ACTIVE = 3;
16.     const ReplicationStyleValue ACTIVE_WITH_VOTING = 4;
i.                  :
-------------------------------------------------------------------
17.     typedef long ParticionedGroupValue;
18.     const ParticionedGroupValue NOT = 0;
19.     const ParticionedGroupValue YES = 1;
20.     typedef long OrderingTypeValue;
21.     const OrderingTypeValue UNRELIABLE = 0;
22.     const OrderingTypeValue RELIABLE = 1;
23.     const OrderingTypeValue CAUSAL = 2;
24.     const OrderingTypeValue TOTAL = 3;
i.                  :
```

Extensão das Propriedades de TF.

Figura 6.11. Propriedades de tolerância a falhas.

6.4.2.2 IOGR

As propriedades de tolerância a falhas de um grupo de objeto são adicionadas dentro da estrutura da IOGR na forma de etiqueta (TAG). Na figura 6.12 (linha 17), cada propriedade é uma estrutura contendo um nome (`nam`) e um valor (`val`). O tipo de dado `Name` é definido na especificação *CosNaming*, usado pelo serviço de nomes para associar com uma referência de objeto. O `Name` é formado por uma seqüência de `NameComponent`, cada um contendo um identificador (`id`) e um tipo (`kind`) (figura 6.13). O tipo `Value` é definido como `any`. O conjunto de propriedades de tolerância a falhas de um grupo é seqüenciada num tipo de dado definido como `Properties`. Portanto, a etiqueta inserida na IOGR, chamada de `TagFTPropertiesTaggedComponent`, é uma estrutura que contém essa seqüência de propriedade (`FTProperties`, linha 22).

```
1.  module CORBA {
2.  module FT {
3.    // Especificação da referência de grupo de objetos (IOGR)
4.    typedef string FTDomainId;
5.    typedef unsigned long long ObjectGroupId;
6.    typedef unsigned long ObjectGroupRefVersion;
7.    struct TagGroupTaggedComponent { // tag = TAG_GROUP;
8.      GIOP::Version version;
9.      FTDomainId ft_domain_id;
10.     ObjectGroupId object_group_id;
11.     ObjectGroupRefVersion object_group_ref_version;
12.   };
13.   struct TagPrimaryTaggedComponent { // tag = TAG_PRIMARY;
14.     boolean primary;
i.                  :
15.   typedef CosNaming::Name Name;
16.   typedef any Value;
17.   struct Property {  // struct de uma property: name e value
18.     Name nam;
19.     Value val;
20.   };
21.   typedef sequence<Property> Properties; // seqüência de propriedade
22.   typedef Properties FTProperties;
23.    // TAG contendo todas propriedades de um grupo dentro da estrutura da
    IOGR
24.   struct TagFTPropertiesTaggedComponent { // tag = TAG_FTProperties;
25.     FTProperties ft_properties;
26.   };
i.                  :
```

Figura 6.12. A etiqueta TAG_FTProperties.

```
typedef string Istring;
struct NameComponent {
  Istring id;
  Istring kind;
};
typedef sequence <NameComponent> Name;
```

Figura 6.13. A estrutura do NameComponent.

6.4.3 Serviço de Nomes Local (SN_L) e Global (SN_G)

O serviço de nomes *CosNaming* [OMG97a] faz a associação de um nome com uma referência de objeto (IOR ou IOGR). Um *contexto de nomes* é definido como um objeto serviço de nomes que contém um conjunto de pares: nome e referência de objeto. Portanto, é possível configurar um sistema com diversos contextos de nomes. Para dar suporte à ligação de contextos de nomes, a especificação *CosNaming* fornece uma interface com um conjunto de operações, além das convencionais, para permitir ligar um contexto de nomes a outro (`bind_new_context` da figura 6.14, linha 16).

Um tipo de dado `Name` é, na verdade, uma estrutura formada por uma seqüência ordenada de componentes. Um componente consiste de dois atributos: o atributo *identificador* e o atributo *tipo*, ambos representados como String IDL. O atributo

identificador é um nome qualquer que identifica um objeto. O atributo *tipo* permite classificar o objeto de acordo com a sua aplicação. Quando um `Name` é formado por apenas um componente é chamado de *nome simples*. E quando formado por múltiplos (seqüência) componentes é chamado de *nome composto*. Cada componente de um nome composto, exceto o último, é usado para nomear um contexto de nomes. O último componente denota, deste modo, o objeto da aplicação. Por exemplo, a notação <component1; component2; component3> denota que component1 e componente2 estão associados, cada um, com um contexto de nomes distinto. O component3, o último, está associado com a referência de objeto da aplicação. Portanto, uma seqüência de componentes define um caminho de nomes (*path name*), tal como diretórios em sistemas de arquivos.

```
1.  module CosNamingFT {
2.    typedef string Istring;
3.    struct NameComponent {
4.      Istring id;
5.      Istring kind;
6.    };
7.    typedef sequencce <NameComponent> Name;
8.    enum BindingType { nobject, ncontext };
9.      :
10.   interface NamingContext {
11.     :
12.     void bind(in Name n, in Object obj)
13.       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
14.     void rebind(in Name n, in Object obj)
15.       raises(NotFound, CannotProceed, InvalidName);
16.     void bind_context(in Name n, in NamingContext nc)
17.       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
18.     void rebind_context(in Name n, in NamingContext nc)
19.       raises(NotFound, CannotProceed, InvalidName);
20.     Object resolve(in Name n)
21.       raises(NotFound, CannotProceed, InvalidName);
22.     void unbind(in Name n)
23.       raises(NotFound, CannotProceed, InvalidName);
24.     NamingContext new_context();
25.     NamingContext bind_new_context(in Name n)
26.       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
27.     void destroy()
28.       raises(NotEmpty);
29.     :
30. };
```

Figura 6.14. A IDL do CosNaming

Para estabelecer a hierarquia de nomes, tal como o modelo descrito na seção 6.3.3, é necessário que cada domínio de tolerância a falhas, exceto o Global, esteja com sua configuração já estabelecida (ver o domínio de TF A da figura 6.8). Isto é, com todos os serviços do *GroupPac* ativados. O SGR de cada domínio cria o grupo SN_L e salva a IOGR em um sistema de arquivo distribuído (ex: servido HTTP). Uma vez que cada domínio local já esteja estabelecido, é criado, então, o domínio de TF Global (figura 6.8).

O próximo passo é fazer a ligação dos contextos de nomes (SN_G com os SN_L). O SGR do domínio de TF Global cria o grupo SN_G e registra, usando `bind_new_context`, a IOGR nos SN_L de cada domínio local de tolerância a falhas (figura 6.7a). O nome dado ao SN_G é um componente definido como `NameComponent("SNGlobal_Domain", "SN_G")`, onde `"SNGlobal_Domain"` é nome dado ao contexto de nomes global e `"SN_G"` define o tipo do objeto como sendo um serviço de nomes de nível global. Deste modo, é feita a ligação do contexto de nomes SN_G com os contextos de nomes SN_L, a ligação dos contextos de nomes é sempre feita primeiro de cima para baixo – o de cima se registra nos de hierarquia inferior.

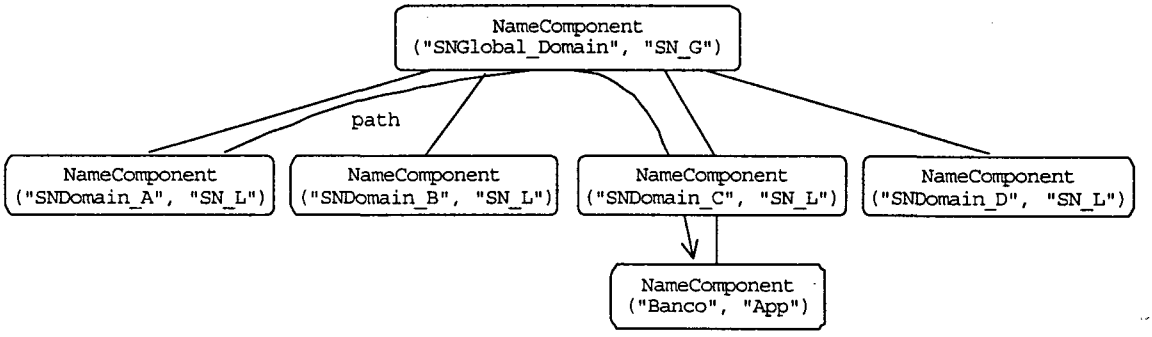


Figura 6.15. Interação Direta - obtenção da IOGR localizado em um outro domínio de TF.

Para que um objeto de um domínio A possa localizar um outro objeto, ou grupo, de um domínio C é necessário que o SN_G tenha a IOGR dos contextos de nomes locais (SN_L) dos domínios envolvidos (figura 6.15). Portanto, o SGR de cada domínio local deve registrar, também usando `bind_new_context`, a IOGR do SN_L do seu domínio no SN_G (figura 6.7b). Como descrito no item 6.3.3.1, um protocolo de difusão é executado no sentido de garantir que este registro seja realizado em todas as réplicas do grupo SN_G.

```
1.  public class Client {
2.    public static void main(String args[]) {
3.      Foo ref = null;
4.      org.omg.CORBA.Object obj = null;
5.      try{
6.         // Obter a IOGR do serviço de nomes local (SN_L), com o suporte do
    ORB
7.        NamingContext SNLA_Ref =
8.            (NamingContext)ORB.resolve_initial_references("NameService");
9.        NameComponent   path[]   =   {new   NameComponent("SNGlobal_Domain",
    "SN_G"),
10.                                  new NameComponent("SNDomain_B", "SN_L"),
11.                                  new NameComponent("Banco", "App")};
12.       obj = SNLB_Ref.resolve(path);
```

Figura 6.16. Interação Indireta - obtenção da IOGR localizada em outro domínio de TF.

Em nível de aplicação, quando um cliente (ex: em um domínio de TF *A*) necessita obter a IOGR de um grupo interdomínio ou pertencente a um outro domínio (ex: domínio de TF *C*), pode realizar esta obtenção de duas maneiras. A primeira é deixar que o contexto de nomes local do domínio de TF A realize esta obtenção – *interação indireta* (código da figura 6.16). A obtenção da IOGR, de forma indireta, é possível de acordo com a especificação do serviço de nomes. Uma vez definido o `NameComponent` com o caminho completo (`path`) da localização da IOGR desejada, o método `resolve` do contexto de nomes local (do domínio TF *A*) é implementado de forma que haja a interação entre contextos de nomes, seguindo a ordem do `path`, até a localização da IOGR desejada pelo cliente. Na segunda, a obtenção pode ser realizada através da *interação direta* do cliente com cada contexto de nomes (na ordem: com o domínio TF local *A*, domínio global e domínio TF local *B*, ver os passos no item 6.3.4 e código da figura 6.17).

```
1.  public class Client {
2.    public static void main(String args[]) {
3.      Foo ref = null;
4.      org.omg.CORBA.Object obj = null;
5.    try{
6.      // Obter a IOGR do serviço de nomes local (SN_L), com o suporte do ORB
7.      NamingContext SNLA_Ref =
          (NamingContext)ORB.resolve_initial_references("NameService");
8.  // 1) Acessa o SNLA_Ref para obter a IOGR do serviço de nomes global
    (SN_G)
9.      NameComponent path0[] = {new NameComponent("SNGlobal_Domain",
    "SN_G")};
10.     NamingContext SNG_Ref = (NamingContext)SNLA_Ref.resolve(path0);;

11. // 2) Acessa o SN_G para obter a IOGR (SNLB_Ref) do SN_L do domínio de TF
    B
12.     NameComponent path1[] = {new NameComponent("SNDomain_B", "SN_L")};
13.     NamingContext SNLB_Ref = (NamingContext)SNG_Ref.resolve(path1);
14. // 3) Finalmente, obter a IOGR do grupo replicado da aplicação registrado
15. // no serviço de nomes (SN_L) do domínio de TF B
16.     NameComponent path2[] = {new NameComponent("Banco", "App")};
17.     obj = SNLB_Ref.resolve(path);
i.                  :
```

Figura 6.17. Código do procedimento de Interação Direta para obtenção de IOGR.

Ambas soluções são similares e, portanto, produz o mesmo resultado. A primeira solução é utilizada, normalmente, em nível de aplicação, e a segunda é utilizada pelo GSeq para localizar a IOGR de um determinado subgrupo (conforme veremos no próximo item). O GSeq necessita apenas do último componente do `path`, lugar em que está o nome associado à IOGR do subgrupo de um grupo interdomínio.

6.4.4 GSeq – Gateway Seqüenciador

Conforme apresentado no item 6.3.4.1.3, o *Gateway Seqüenciador* (GSeq) é utilizado para difundir mensagens, de forma atômica, em um grupo interdomínio. Esse serviço é ativado quando um cliente obtém a IOGR desse grupo e o interceptador de mensagem identifica, através das propriedades de TF, como sendo um grupo interdomínio. Portanto, discutiremos neste item como é feito o desvio da requisição através do interceptador de mensagem e alguns aspectos de implementação do GSeq.

6.4.4.1 Usando o interceptador

O conceito de interceptadores foi introduzido inicialmente nas especificações do serviço de segurança do CORBA [OMG00d]. Atualmente, uma especificação padrão, chamada de *Portable Interceptor*[7], foi emitida pela OMG com o objetivo da generalização deste mecanismo [OMG98b]. Esse mecanismo de interceptação de mensagem é utilizado no *GroupPac* para desviar uma requisição para o GSeq (passo 2 da figura 6.8). A interface do interceptador do lado cliente (`ClientRequestInterceptor`), é apresentada na figura 6.18. Observe que esta interface herda a interface `Interceptor` (linha 29). As interfaces `RequestInfo` e `ClientRequestInfo`, sendo que a segunda herda a primeira (figura 6.18, linha 17), permitem, respectivamente, obter os atributos da requisição feita pelo cliente e a IOGR do grupo. Para permitir a montagem da requisição do cliente, a interface `RequestInfo` (linha 1) fornece, por exemplo, os atributos que identificam a requisição, a operação (método) a ser invocada no lado servidor, a lista de argumentos, etc. A interface `ClientRequestInfo` (linha 17) fornece um conjunto de operações para a obtenção das informações contidas dentro de uma IOGR. Para o nosso propósito, utilizamos a operação `get_effective_component` (linha 23) para ler a etiqueta `FTProperties`. Deste modo, o interceptador é capaz de remontar uma requisição (através dos atributos da interface `RequestInfo`) e redirecionar ao GSeq, caso a etiqueta `FTProperties` indique que o grupo é do tipo interdomínio.

[7] Logicamente, um interceptador é interposto no caminho de invocação ou resposta entre um cliente e um objeto alvo, sendo responsável pela ativação transparente de controles ou processamentos especiais às quais estariam sujeitas invocações normais no ambiente CORBA.

```
1.  local interface RequestInfo {
2.    readonly attribute unsigned long request_id;
3.    readonly attribute string operation;
4.    readonly attribute Dynamic::ParameterList arguments;
5.    readonly attribute Dynamic::ExceptionList exceptions;
6.    readonly attribute Dynamic::ContextList contexts;
7.    readonly attribute Dynamic::RequestContext operation_context;
8.    readonly attribute any result;
9.    readonly attribute boolean response_expected;
10.   readonly attribute Messaging::SyncScope sync_scope;
11.   readonly attribute ReplyStatus reply_status;
12.   readonly attribute Object forward_reference;
13.   any get_slot (in SlotId id) raises (InvalidSlot);
14.   IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
15.   IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
16. };
i.                    :
17. Local interface ClientRequestInfo: RequestInfo {
18.   readonly attribute Object target;
19.   readonly attribute Object effective_target;
20.   readonly attribute IOP::TaggedProfile effective_profile;
21.   readonly attribute any receive_exception;
22.   readonly attribute CORBA::RepositoryId received_exception_id;
23.   IOP::TaggedComponent get_effective_component(in IOP::ComponentId id);
24.   IOP_N::TaggedComponentSeq  get_effective_components(in  IOP::ComponentId
    id);
25.   CORBA::Policy get_request_policy(in CORBA::PolicyType type);
26.   Void add_request_service_context(in IOP::ServiceContext service_context,
27.                                    in Boolean replace);
28. };
i.                    :
29. Local interface ClientRequestInterceptor: Interceptor {
30.   void send_request(in ClientRequestInfo ri) raises(ForwardRequest);
31.   void send_poll(in ClientRequestInfo ri)
32.   void recive_reply(in ClientRequestInfo ri)
33.   void receive_exception(in ClientRequestInfo ri) raises(ForwardRequest);
34.   void receive_other(in ClientRequestInfo ri) raises(ForwardRequest);
35. };
```

Figura 6.18. As interfaces para interceptação de mensagem.

6.4.4.2 Implementação do GSeq

Para cada requisição emitida por um cliente, o GSeq deve realizar três etapas para realizar a comunicação interdomínios: (1) receber a requisição enviada pelo interceptador do cliente, (2) executar o protocolo de difusão e (3) entregar a requisição (ordenada) para os subgrupos do grupo interdomínio (figura 6.19). Apresentaremos a seguir como estas três etapas são implementadas.

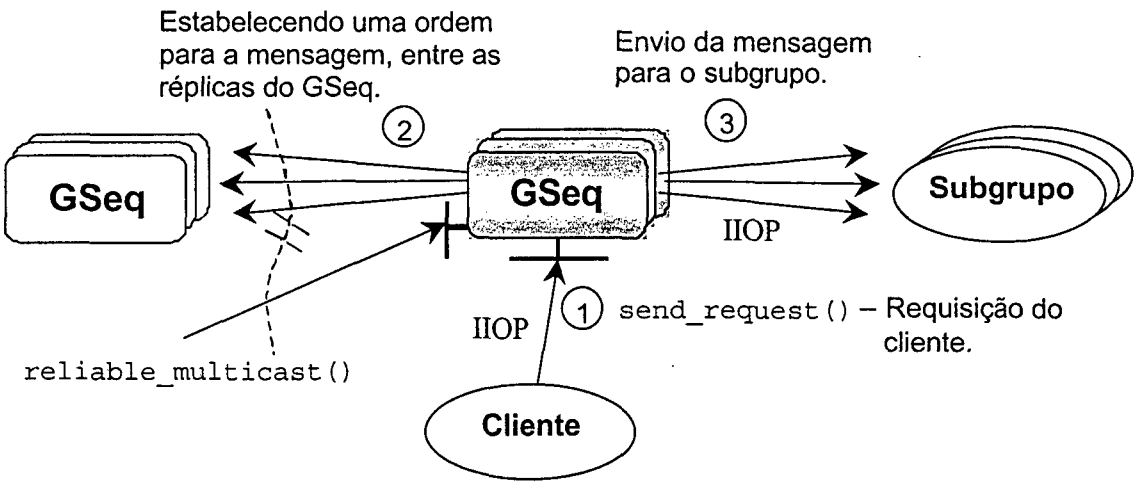


Figura 6.19. As funções do *Gateway* Seqüenciador GSeq.

Etapa 1: comunicação entre o cliente e o GSeq

Após o cliente obter a IOGR (item 6.3.4.1.3) de um grupo interdomínio, utiliza esta referência para se comunicar com o grupo, esta ativação é transparente para o cliente. No momento da invocação, o interceptador do cliente verifica que a IOGR é de um grupo interdomínio e redireciona a chamada para uma réplica do GSeq (passo 2, figura 6.8) – normalmente a mais próxima. Se a réplica escolhida falha durante esta conexão, segundo a especificação CORBA, uma exceção é retornada pelo ORB. A especificação do *Portable Interceptor* [OMG98b] permite que este trate exceções, e uma das funcionalidades é o redirecionamento da invocação para uma outra réplica do GSeq – o método que implementa esta funcionalidade é o `location_forward()`. Deste modo, o interceptador fica a procura até encontrar um GSeq que possa receber sua requisição. Quando um GSeq é encontrado, o interceptador constrói a estrutura do `requestInfo` (figura 6.18) e, com isso, invoca o método `send_request`, enviando a requisição do cliente `requestInfo` (passo 1 da figura 6.19).

Para que o GSeq possa receber requisições, desviadas pelo interceptador, é necessário que possua uma interface CORBA para isso (figura 6.20). Esta interface declara as operações utilizadas nas três etapas da figura 6.19.

```
module CORBA {
  module FT {
    typedef sequence <Dynamic::Parameter> ParameterList;
    typedef long Timestamp;
    struct RequestInfo {
      string type_id;
      string operation;
      org.omg.Dynamic.Parameter[] parameters;
      any result;
    };
    interface Gseq {
      // atualiza a IOGR do grupo GSeq
      void update_global_gseq_iogr();
      void update_local_gseq_iogr();
      oneway void send_request(in RequestInfo request_info, in string id);
      oneway void reliable_multicast(in RequestInfo request_info, in string
id);
    };
  };
};
```

Figura 6.20. A interface IDL do GSeq.

Etapa 2: comunicação entre os GSeq

A comunicação entre os GSeq, nesta etapa, se dá através de uma ferramenta de comunicação proprietária – Isis [Birman91]. A implementação do GSeq segue as descrições apresentadas no item 3.3.3.2 do capítulo 3, usando interceptadores e adaptadores de grupo [Oliveira99a, Lau00a].

Alternativamente, temos um protocolo de difusão confiável, baseado em [Chandra96], implementado sem o uso de uma ferramenta de comunicação de grupo. Neste caso, o protocolo é implementado nos interceptadores das réplicas do GSeq. A lista de membros utilizada para a difusão confiável é fornecida pelo SGR do domínio global através da ativação do método `update_global_gseq_iogr`.

Etapa 3: comunicação entre GSeq e um subgrupo

Após estabelecer uma ordem para uma mensagem *m*, na etapa 2, as réplicas do GSeq devem difundir esta mensagem para os seus respectivos subgrupos (passo 3 da figura 6.19). Como mostrado na figura 6.8, após o GSeq obter a IOGR do seu subgrupo (passo 4), envia a requisição do cliente para este subgrupo. Como isto pode ser feito por várias réplicas do GSeq, os interceptadores das réplicas do subgrupo devem descartar requisições repetidas.

6.4.5 Aspectos de configuração de domínios de tolerância a faltas

Por limitações físicas de nossa rede, este item apresenta apenas alguns aspectos de configuração do GroupPac. O protótipo do *GroupPac* foi testado em uma aplicação de sistema bancário executando sobre duas redes locais Ethernet 10Mps (lcmi.ufsc.br e nurcad.ufsc.br) na UFSC. O sistema bancário é formado por um grupo em que suas réplicas formam dois subgrupos, uma na rede do LCMI e outra no NURCAD. Foram utilizadas no total seis máquinas *Pentium*, com *Linux* ou *Windows98* instalado, sendo três em cada rede local – o que está longe ser uma rede de larga escala. Na figura 6.21, é mostrada uma possível composição de serviços do *GroupPac* em cada máquina das duas redes locais. Os serviços que estão dentro do círculo cinza fazem parte do domínio de TF global.

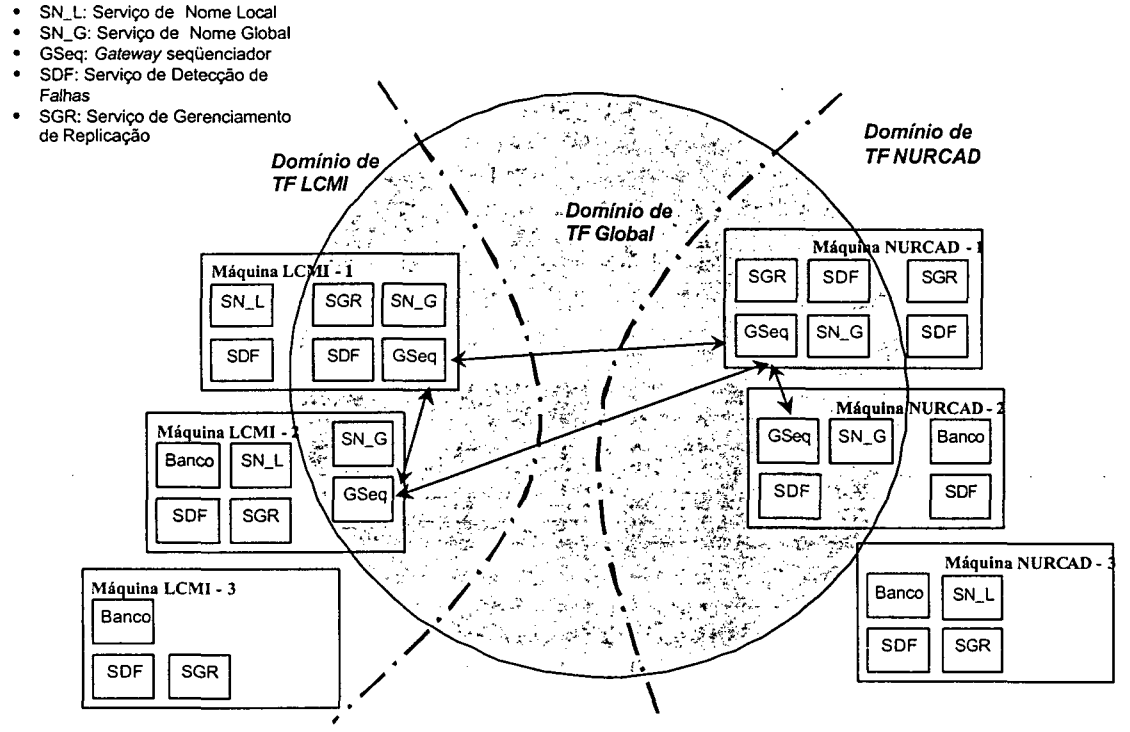


Figura 6.21. Exemplo de configuração para suporte a grupo interdomínio.

O grupo GSeq fornece suporte de comunicação de grupo através do ISIS. Em operações de escrita, o grupo GSeq se encarrega de difundir a atualização em cada membro do serviço bancário. No entanto, para operações de leitura o cliente pode localizar, através do SN_L, e acessar uma das réplicas do serviço bancário que esteja no seu domínio de TF, a mais próxima.

6.5 Considerações sobre o GroupPac

O modelo de hierarquia de domínios de tolerância a faltas, introduzido neste trabalho, implicou, conforme apresentado, em um conjunto de adaptações e extensões nas especificações FT-CORBA. Essas mudanças tiveram maior impacto na forma como o SGR e o SDF, das especificações FT-CORBA, fornecem suporte para gerenciamento de grupos. O SN_L e o SN_G refletem os resultados deste gerenciamento nas IOGRs registradas. A hierarquia proposta define dois níveis de domínios de TF. Hierarquias com mais níveis poderiam ser definidas ao custo de uma complexidade maior.

As soluções do *GroupPac* foram especificadas de modo que várias soluções algorítmicas podem ser usadas nos serviços de detecção de falhas e de comunicação de grupo. Por exemplo, para a detecção de falhas no grupo de detectores, o algoritmo envolvendo detectores não confiáveis de [Chandra96] poderia ser usado. Por outro lado, o modelo de detecção de falhas dos objetos da aplicação tem que ser assimétrico, devido à semântica definida para estes detectores nas especificações FT-CORBA. O modelo hierárquico de domínios adotado e os conceitos de detectores de falhas e de domínio de TF das especificações FT-CORBA colaboram para diminuir e confinar as mensagens envolvidas na detecção de falhas em sistemas de larga escala (propriedade de Localidade [Fritzke98]). Em termos de trabalhos relacionados, envolvendo o uso das especificações CORBA na detecção de falhas em sistemas distribuídos, a solução apresentada em [Chung98], embora não use os conceitos definidos pelo FT_CORBA, define um conjunto de detectores, chamados de *WatchDog*, para monitorar os objetos da aplicação de forma similar aos detectores introduzidos pelo FT-CORBA. Este trabalho se limita a redes locais e não trata do problema da falha dos detectores. No OGS [Felber98] a detecção de falhas é, também, similar à especificação, sendo que suas interfaces são diferentes do FT-CORBA. No entanto, o OGS apresenta um objeto de serviço de consenso que poderia ser adaptado para a detecção de falhas. Ambas propostas não consideram aspectos de escalabilidade.

O conjunto de extensões introduzidas no FT-CORBA, se deve, basicamente, ao suporte necessário para a comunicação de grupos interdomínios. Por exemplo, a necessidade de adicionar duas novas propriedades de TF (item 6.3.4.1) é devido à IOGR [OMG00a], conforme a especificação no FT-CORBA, sempre limitar um grupo em um

único domínio de TF. A proposta de definir um mesmo nome para cada IOGR de um subgrupo, exemplificada na tabela 6.1, permite resolver o problema tanto de gerenciamento como comunicação de grupo interdomínio.

Uma vez que os grupos GSeq e SN_G se apresentam como compostos de poucas réplicas, isto nos abstrai um pouco de um modelo de sistema de larga escala. Portanto, o modelo hierárquico proposto permite que diferentes protocolos de comunicação de grupo possam ser usados pelos grupos Gseq e SN_G. O suporte de comunicação presente nos domínios envolve objetos de serviços (SCG) que podem encapsular as funcionalidades de uma ferramenta de comunicação de grupo proprietária. Nos nossos desenvolvimentos foi usado o Isis [Birman91]. As nossas primeiras experiências em implementar estes objetos encapsuladores foram apresentadas em [Oliveira99b e Lau00a]. Comparações de abordagens e medidas de desempenho são apresentadas nos capítulo 3 e 4. Outras ferramentas, como o Horus [Renesse95], Newtop [Ezhilchel95], Totem [Melliar90], etc, também poderiam ser utilizadas. O sistema também pode conviver com objetos de serviço de diferentes domínios encapsulando diferentes ferramentas de comunicação de grupo. Mesmo o uso de um protocolo particular, implementado na forma de um objeto de serviço CORBA é perfeitamente possível no nosso modelo.

O modelo de comunicação de grupo dos *Gateways* Seqüenciadores (GSeq) é similar ao modelo apresentado em [Rodrigues96]. A diferença é que no protocolo de Rodrigues os processos podem mudar de função dinamicamente (seqüenciador ativo para processo passivo e vice-versa). O GSeq é baseado na abordagem servidor dedicado, o que o torna capaz de atender diversos grupos interdomínios ao mesmo tempo. Isto é possível porque o GSeq obtém a IOGR dos subgrupos em cada invocação (passo 4 da figura 6.8).

Em se tratando de interoperabilidade consideramos a potencialidade da comunicação entre objetos em ORBs distintos, implementados em diferentes linguagens de programação e sobre diferentes plataformas de hardware e software. Deste modo, concluímos que o GroupPac alcança esse grau de interoperabilidade, também, para a modalidade de comunicação de grupo interdomínios. Pois, a comunicação entre os GSeqs, apesar de ser, por enquanto, através de um suporte de comunicação proprietário, não influi na interoperabilidade entre clientes e grupos interdomínios. O GSeq serve apenas para a distribuição das mensagens, segundo um protocolo de comunicação, sobre os grupos

interdomínios. A comunicação dos clientes (objetos clientes e grupos interdomínio) desse serviço (GSeq) se dá através de IIOP. Portanto, podemos ter um sistema com clientes e subgrupos em diferentes ORBs, em um ambiente heterogêneo.

6.6 Conclusão

Fornecer suporte para o desenvolvimento de aplicações em redes de larga escala tem sido, atualmente, uma das principais preocupações da comunidade cientifica de sistemas distribuídos. Neste trabalho não objetivamos propor, especificamente, novos algoritmos distribuídos para sistemas de larga escala, uma vez que existem diversos na literatura, para uma larga faixa de aplicações. Nossas preocupações, sim, foram definir modelos que adequassem conceitos do FT_CORBA e soluções usuais de tolerância a faltas às necessidades de escalabilidade em sistemas de larga escala.

Neste capítulo, são explicados e analisados os meios como o modelo proposto pode ser implementado no CORBA sem causar maiores impactos nas especificações originais do padrão. Mostra um conjunto de extensões às especificações FT-CORBA que foram introduzidas para viabilizar este modelo. As soluções adotadas tiveram como princípio não modificar as interfaces do padrão. Os protocolos executados por trás dessas interfaces são transparentes na arquitetura como um todo. Este trabalho faz parte do projeto *GroupPac* que visa propor soluções para tolerância a faltas às aplicações CORBA em um contexto de sistema de larga escala.

CAPÍTULO 7
Conclusão

7.1 Revisão das motivações e objetivos

As especificações do FT-CORBA são recentes e, na sua versão 1.0, continuam a evoluir. Estas especificações, que ainda devem passar por varias revisões (e extensões), atendem, neste primeiro momento, apenas aos requisitos básicos para tolerância a faltas, que são úteis em praticamente todas aplicações que apresentam estes requisitos em sistemas distribuídos. A partir destas revisões deverão surgir serviços mais especializados. Ao que tudo indica, as direções apontam na definição de especificações para integrar suporte de comunicação de grupo na arquitetura CORBA [OMG00a], visando principalmente o suporte a implementação de replicações ativas para tolerância a faltas – não disponíveis nas especificações atuais do FT-CORBA. Talvez mais adiante, com a disponibilidade de especificações para comunicação de grupo se discuta também aspectos de escalabilidade, focando por exemplo, os grupos interdomínios um dos problemas tratados nesta tese.

Estas considerações têm sido alvo de intensas atividades de pesquisa e desenvolvimento em tolerância a faltas em sistemas distribuídos abertos. Diversos trabalhos foram produzidos no sentido de propor meios para introduzir suporte de grupo e tolerância a faltas no CORBA.

As nossas motivações, num primeiro instante, foram a inclusão de suportes para grupo e mecanismos de tolerância a falhas na arquitetura CORBA, sem que isto representasse a perda de propriedades da programação em sistemas abertos. Já diante das especificações, num segundo instante, as nossas pesquisas se concentraram na própria experimentação das especificações e na proposição das extensões que permitissem a utilização de seus conceitos em ambientes de larga escala.

7.2 Visão geral do trabalho

O tema tolerância a faltas em sistemas distribuídos, tratado neste trabalho, faz parte das linhas de pesquisas do Laboratório de Controle e Microinformática (LCMI) da Universidade Federal de Santa Catarina (UFSC). Esta tese faz parte de um interesse maior do nosso grupo de pesquisa em sistemas distribuídos. Na realidade, este trabalho encerra um ciclo de pesquisas envolvendo tolerância a faltas, segurança[1] [Merkle00] e tempo real[2] [Montez99] em *middleware* CORBA [Fraga01].

O projeto GroupPac[3] envolveu também alguns colaboradores, alunos de mestrado e de graduação (iniciação cientifica) durante todas as etapas do seu desenvolvimento, o que resultou em uma intensa atividade de trabalho em grupo e a participação destes nas publicações alcançadas com este projeto.

Esta tese iniciou com uma descrição de um estudo sobre requisitos e abordagens de suporte de grupo e tolerância a faltas em *middleware* CORBA. Em seguida, nossos esforços se concentraram na proposição de soluções para introduzir tolerância a faltas na arquitetura CORBA. O MetaFT é uma proposta baseada em protocolos meta-objetos para tolerância a faltas; e o GroupPac concebido dentro de uma perspectiva mais dentro das linhas da OMG, combina propriedades das abordagens de interceptação e de serviço. Um estudo comparativo, avaliando nossas propostas e outros trabalhos da literatura, é também apresentado. No sentido de validar nossas proposições, apresentamos um estudo sobre as especificações CORBA de serviço de nomes e a sua implementação agregando propriedades de tolerância a faltas através dos objetos de serviços do GroupPac.

Com o surgimento do padrão FT-CORBA, os nossos esforços se concentraram no entendimento dessas especificações e na adaptação do GroupPac às interfaces padronizadas pela OMG. Também, com este estudo, foram levantadas considerações sobre essas especificações tendo como enfoque principal aspectos de escalabilidade. Um estudo sobre suporte de grupo para redes de larga escala é realizado no sentido de compreender o problema da escalabilidade em processamento de grupo.

[1] JacoWeb – http://www.lcmi.ufsc.br/jacoweb
[2] http://www.lcmi.ufsc.br/~montez
[3] GroupPac – http://www.lcmi.ufsc.br/grouppac

7.3 Contribuições da tese

Dentro dos objetivos traçados para este trabalho e das atividades desenvolvidas durante este período, nós podemos enumerar os seguintes pontos de destaque desta tese:

- MetaFT – proposta de um modelo de programação baseado em reflexão computacional no sentido da gestão transparente da tolerância a faltas no CORBA [Lau96, Fraga97, Lau99c];
- GroupPac1.0 – proposta de uma abordagem híbrida, que combina propriedades das abordagens de serviço e de interceptação, para tolerância a faltas no CORBA [Oliveira99a, Oliveira99b, Lau00a, Lau00d];
- CosNamingFT – proposta de um serviço de nomes CORBA tolerante a faltas [Lau99a, Lau99b];
- Estudo e implementação das especificações FT-CORBA [Lau00c, Lau01b];
- GroupPac2.0 – proposta de um modelo de hierarquia de domínios para tolerância a faltas em redes de larga escala [Lau00a, Lau00b, Lau01a].

O MetaFT a nosso conhecimento foi a primeira proposta do uso de interceptação para inclusão dos controles de tolerância a faltas no CORBA. O GroupPac1.0 na sua combinação das abordagens de interceptação e de serviços, combina vantagens da transparência das ativações com a adequação à linha de objetos de serviço da OMG. Os objetos do GroupPac possibilitam a construção de *frameworks* próprios para os esquemas de tolerância a faltas desejados. Estes *frameworks* permitem um suporte mais adequado às necessidades da técnica de tolerância a falhas escolhida. Por fim, é proposto o GroupPac2.0 que é uma adequação da filosofia do GroupPac às especificações do FT-CORBA. Como uma contribuição indireta deste último trabalho, temos a identificação de inadequação de alguns mecanismos e conceitos introduzidos nestas especificações a necessidades de escalabilidade.

Fazendo o uso do conceito de Domínio de Tolerância a Faltas das especificações FT-CORBA, introduzimos a definição de um modelo de hierarquia de domínios para tratar com a escalabilidade no FT-CORBA. Esta hierarquia de domínios permite o

gerenciamento e comunicação de grupos em sistemas de larga escala, adequando-se às propriedades de minimidade e localidade identificadas em (Anexo B). Além disso, nesta proposta são identificados diferentes modelos de comunicação de grupo: comunicação de grupo intradomínios e interdomínios. A comunicação interdomínios nos levou a definição de *grupos interdomínios* onde réplicas de um mesmo grupo estão dispostas em diferentes domínios de TF (e de administração) e, portanto, sujeitas a protocolos e políticas diferentes. O modelo propõe soluções tanto na gestão como na comunicação destes grupos especiais.

7.4 Trabalhos futuros

Considerando o atual estágio deste projeto, algumas possibilidades para a sua continuidade são apresentadas a seguir.

A OMG lançou, recentemente, as especificações CORBA 3.0. As principais novidades dessa especificação foram a introdução o modelo de componente CORBA e um mecanismo de invocação de métodos assíncronos (*Assincronous Method Invocation* - AMI), além disso, considera as especificações FT-CORBA, CORBASec e RT-CORBA como os elementos chave desse novo padrão. Estas novidades consistem em um novo campo de pesquisa para a área de sistemas distribuídos abertos. A adequação do GroupPac, por exemplo, ao mecanismo de invocação assíncrono de mensagens e também a disponibilidade das especificações do *Unreliable Multicast Protocol* [OMG00b] abre novos desafios para a construção de serviços de membership e de difusão atômica.

O esquema de hierarquia de domínios de tolerância a faltas foi proposto de forma a "localizar" o uso de protocolos de gestão e comunicação de grupos no modelo CORBA. Uma possibilidade de continuidade deste trabalho seria a investigação de diferentes protocolos tanto de *membership* e como de difusão confiável, levando em conta aspectos de desempenho, e usados segundo a estrutura proposta. A aplicação deste modelo em redes de larga escala para testar as potencialidades do mesmo também se faz necessária.

Dentro da perspectiva de fornecer suporte para aplicações críticas em *middlewares* CORBA, o grupo de sistemas distribuídos do LCMI tem realizado pesquisas enfocando as especificações Fault-Tolerant CORBA (FT-CORBA [OMG00]), CORBA Security

(CORBASec [OMG00b]) e Real Time CORBA (RT-CORBA [OMG99]), estes estudos foram realizados de forma separada. As diferentes abstrações dessas especificações envolvendo *middlewares* CORBA certamente servem como base para implementações de controles de segurança, de tempo real e de tolerância a faltas em diferentes tipos de aplicações. A construção destas bases é sempre um desafio. As experiências existentes normalmente se concentram separadamente tratando uma destas qualidades de serviço. A medida que estas diferentes qualidades de serviço não demonstrem incompatibilidade conceitual, combinações podem ser tentadas. Estes processos de combinação devem seguir caminhos que venham a ser definidos pela OMG que, por exemplo, tem mostrado preocupação na obtenção da compatibilidade entre as tecnologias FT-CORBA e CORBAsec [OMG00].

Esta tese apresentou nossas pesquisas no sentido de introduzir tolerância a faltas na arquitetura CORBA que envolveram vários trabalhos. As principais preocupações eram como estas soluções poderiam ser adaptadas ao modelo de objeto CORBA, mantendo a conformidade com as especificações. Em geral, as especificações CORBA nem sempre se caracterizam como uma fonte objetiva e clara. Esperamos que os nossos trabalhos tenham contribuído para o melhor entendimento das mesmas e mostrado as possibilidades que se abrem para sistemas distribuídos dentro de uma filosofia de sistemas abertos.

ANEXO A
Tolerância a Faltas em Sistemas Distribuídos

A.1 Introdução

O objetivo deste anexo é apresentar a terminologia e conceitos básicos de tolerância a faltas a serem utilizados nesta tese. Além disso, é feito um estudo de técnicas de replicação de componentes de *software* em sistemas distribuídos. Técnicas de replicação (para dados ou processos) têm se mostrado um paradigma bastante útil para aumentar a disponibilidade e o desempenho dos serviços fornecidos e para permitir, através de mecanismos de tolerância a faltas, o fornecimento de serviços ininterruptos mesmo quando da ocorrência de falhas no sistema. Serão explorados dentro deste estudo os aspectos e as características de técnicas de replicação passiva, replicação ativa e replicação semi-ativa, visando, principalmente, à forma como cada abordagem busca garantir a consistência de estado. Na seqüência, é apresentada uma seleção de modelos clássicos de processamento replicado, implementados em plataformas de sistemas distribuídos conhecidos, divididos segundo essas três abordagens.

A.2 Classificação das faltas

Em um sistema distribuído, a ocorrência de falhas de componentes pode ser causada por erros provocados a partir de *hardware* 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, inundação etc.). Um serviço distribuído que não está preparado para agir de acordo com a ocorrência desses problemas, como conseqüência, pode causar sérios prejuízos aos usuários do serviço. Desta forma, um serviço distribuído pode ser estruturado, usando a redundância implícita de *hardware*, para fornecer serviço contínuo, mesmo na presença de componentes faltosos.

Um componente de um sistema 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. De acordo com a definição de [Laprie90] uma falta (*failure*) ocorre quando um serviço oferecido pelo sistema não está de acordo com as especificações. A terminologia em português na área de tolerância a faltas é ainda bastante controversa. Neste texto assumimos os seguintes termos e definições:

- A **falta** (***fault***) é definida como a causa fenomenológica de um erro no sistema;
- Um **erro** (*error*) é parte do estado do sistema que pode conduzir a falha do serviço especificado. Um erro é causado por uma falta;
- Uma **falha** (*failure*) ocorre, em um sistema, quando seu comportamento desvia daquele requerido pela sua especificação.

Em relação às faltas, podemos agrupa-las segundo a semântica de falha associada. As semânticas de falhas são definidas segundo dois domínios: tempo e valor; e classifica-las da seguinte forma:

- **Falhas por *crash* e de omissão**: um componente que não responde a uma dada requisição e, no entanto, responde a uma requisição subseqüente caracteriza uma falha por omissão. Um exemplo deste tipo de falha é a perda de uma mensagem na rede de comunicação. Para falhas de *crash*, o componente não responde a mais nenhuma requisição subseqüente. Há casos em que se atribui um grau de omissão f a um componente, quando esse limite é ultrapassado todas respostas subseqüentes deverão ser omitidas, caracterizando uma falha de *crash*.
- **Falhas por temporização**: ocorre quando um componente responde a uma dada requisição com um valor correto, mas fora do intervalo de tempo especificado, ex: um processador sobrecarregado pode produzir um valor correto, mas atrasado em relação a um intervalo especificado (falha por

atraso). Vale ressaltar que uma falha de temporização só se caracteriza se a especificação do sistema impor restrições temporais ao componente.

- **Falhas por valor**: ocorre quando uma resposta é devolvida dentro do intervalo de tempo estabelecido, mas com o valor fora do especificado, ex: mensagem corrompida (alterada) na rede de comunicação.

- **Falhas Arbitrárias**: engloba todas as classes de falhas citadas acima, o componente falha em ambos domínios (tempo e valor) de forma a impossibilitar classificá-la em uma das anteriores.

A falha por *crash* é a mais restritiva das classes de falha enquanto a falha arbitrária representa a menos restritiva, ambas formam os dois extremos do espectro de falhas. Falhas como perda ou corrupção de mensagem na rede de comunicação, *crash* do processador ou particionamento da rede são problemas que podem ocorrer em sistemas distribuídos e dificultam a manutenção da consistência da informação contida em cada componente do sistema. Contudo, é importante lembrar que o custo do projeto de um sistema é mais elevado quanto menos restritivas forem as classes de falhas e suas respectivas semânticas de falhas a serem toleradas.

A.3 Tolerância a faltas

A tolerância a faltas é uma propriedade do sistema de fornecer, por redundância, serviços de forma ininterrupta, de acordo com a especificação, mesmo na presença de faltas. A redundância pode ser de software, hardware ou ainda temporal. A tolerância a faltas envolve, normalmente, várias fases. Na literatura, usualmente, são identificadas as fases de detecção de erros, confinamento de erros, processamento de erros e tratamento de faltas.

A detecção de erros é concebida através de mecanismos de monitoramento no sistema, a fim de detectar possíveis falhas. Confinamento de erros significa delimitar a propagação de erros, a fim de evitar que se propague no resto do sistema. Em relação ao processamento de erros, as técnicas envolvidas se dividem em compensação de erros e recuperação de erros [Laprie92]:

- **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 (item A.4.2) aplicadas sistematicamente em um sistema;

- **Recuperação de erros**: esta técnica está baseada na detecção de erros. A recuperação de erro consiste em substituir um estado errôneo por um estado sem erro. Duas formas de recuperação de erro são identificadas na literatura:

    - **Recuperação em retrocesso do erro**: esta técnica exige que informações de estado do sistema sejam regularmente armazenadas em pontos específicos, denominados de pontos de recuperação (*checkpoint*). A partir da detecção de um erro, o estado do sistema é restaurado com os valores do último ponto de recuperação estabelecido;

    - **Recuperação em avanço do erro**: esta técnica consiste em transformar o estado falho em um estado livre de erro, a partir do qual o sistema retorna a sua operação normal, possivelmente em modo degradado.

Ao contrário das técnicas de compensação de erro, que independentemente da presença ou não de falha, apresentam sempre o mesmo *overhead* de processamento, as técnicas de recuperação de erros apresentam acréscimos de processamento somente quando um erro é detectado no sistema.

Por fim, o tratamento de faltas é responsável pelos procedimentos que visam impedir a reativação de uma falta. Desta forma, após o procedimento de erro, é necessário identificar os elementos faltosos, de modo a recuperá-los ou retirá-los do sistema. A primeira etapa do tratamento de faltas é então a diagnose de faltas que consiste em determinar a localização e a natureza das faltas. A segunda etapa, denominada de passivação de faltas, consiste em prevenir que as faltas sejam novamente ativadas. Após a passivação de falta, a reconfiguração do sistema somente é necessária quando os serviços fornecidos não atendem mais aos requisitos mínimos de tolerância a faltas. As diversas

fases da tolerância a faltas e seus mecanismos associados são examinadas em outras publicações [Anderson81, Laprie92].

A.4 Técnicas de replicação

É de consenso geral que a utilização de técnicas de replicação em sistemas distribuídos contribuem fortemente para uma melhora na confiabilidade, um aumento da disponibilidade de recursos e possivelmente, um melhor desempenho do sistema. O uso de replicação torna possível implementar serviços tolerantes a faltas para o usuário (exibe a propriedade de um componente único, mas que na realidade é formado por um conjunto de componentes replicados). A falha de um dos componentes replicados passa a ser considerada separadamente da falha do grupo como um todo. Com isso, a confiabilidade de um sistema está diretamente relacionada ao seu grau de replicação.

Fornecer suporte às técnicas de replicação implica no uso de protocolos de coordenação no sentido de assegurar a consistência de estado e a transparência do conjunto. Estes protocolos devem também ser responsáveis pelo controle da concorrência e pela recuperação em situação de falha parcial ou total das réplicas.

As técnicas de replicação podem ser divididas segundo três abordagens: *Réplicas Passivas*, *Réplicas Ativas* e *Réplicas Semi-ativas*. As abordagens de replicação passiva e semi-ativa são capazes de tratar as falhas que porventura ocorram em algum componente replicado do sistema. Já a abordagem de replicação ativa é capaz de mascarar falhas individuais de réplicas do conjunto. Além disso, a forma como estas abordagens são implementadas diferem fortemente. Na escolha de uma dessas abordagens devem ser considerados: o tipo de aplicação, a classe de falta que se deseja tolerar e as características do sistema distribuído.

A.4.1 A abordagem de replicação passiva

Nesta abordagem, somente um membro (primário) recebe, executa e responde as invocações dos clientes. As réplicas restantes do conjunto (*backups*) têm a função de substituir o membro primário caso este falhe. Usualmente, dependendo de como a técnica é implementada, o cliente faz as requisições sem saber qual é o primário da replicação. Para assegurar que os estados dos membros se mantenham mutuamente consistentes, o primário

envia periodicamente mensagens de *checkpoint* de seu estado para todos ou um número suficiente de *backups* (figura A.1). Em situação de falha do primário é ativado um protocolo de eleição que seleciona entre os *backups* o novo primário, que reassume a execução da operação a partir do *checkpoint* mais recente. É importante ressaltar que nenhuma invocação é processada até que o novo primário seja eleito.

A grande vantagem desta abordagem é exatamente a não necessidade do determinismo de réplicas[1], uma vez que o primário impõe seu estado sobre as réplicas *backups*. Devido ao fato do cliente comunicar-se somente com o primário, a interação cliente/servidor é simplificada. Uma situação que deve ser considerada é quanto à atualização dos *backups*. Se um *backup* não recebeu uma mensagem de *checkpoint* por um motivo qualquer, ele deve ser manipulado de forma que não possa ser eleito como o novo primário, até que seu estado seja atualizado. Isto pode ser feito através de um protocolo que reconheça um *backup* desatualizado e realize uma ação de atualização de seu estado para que esse possa ser potencialmente elegível.

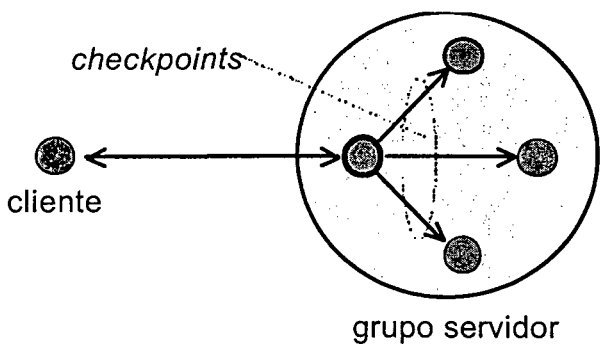


Figura A.1. Grupo réplicas passivas.

A detecção de falha do primário é um outro fator que deve ser considerado. Nesta abordagem é possível detectar falhas de *crash* ou de omissão do primário. Para esses casos, podem ser utilizados mecanismos de *timeout e keepalive.* A detecção pode ser realizada pelos *backups*, por um mecanismo independente ou pelo próprio usuário do serviço replicado. A freqüência com que é ativado o mecanismo de *keepalive* determina a rapidez da recuperação na falha do primário, mas se esta freqüência for muito elevado pode influenciar no desempenho do servidor. Uma situação que pode ocorrer é o primário falhar

[1]O determinismo de réplicas introduzido por [Schneider90] implica que réplicas corretas, partindo do mesmo estado inicial e processando o mesmo conjunto de entradas, na mesma ordem relativa, devem produzir as mesmas saídas. O determinismo de réplicas é uma condição para a consistência de estados entre réplicas ativas.

logo após completar a execução de uma requisição e enviar o *checkpoint* aos *backups* sem, no entanto, responder ao cliente. Neste caso o cliente pode reenviar a mesma requisição. Mas para evitar que o novo primário processe-a novamente é necessário, através de um mecanismo de retenção de resultado ou um controle de seqüência nas requisições submetidas, que ele reconheça a repetição da requisição e simplesmente retransmita o resultado retido [Birman85].

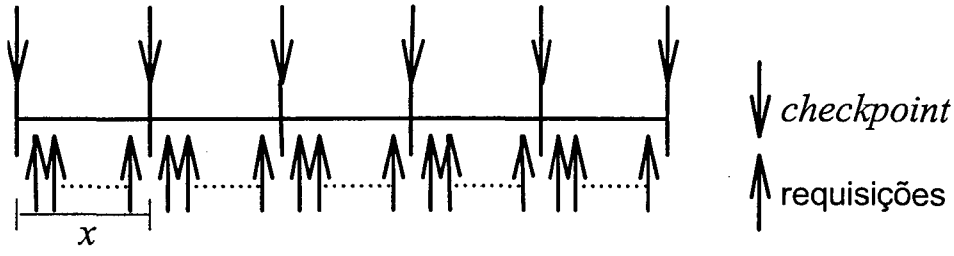


Figura A.2 Mecanismo de *checkpoint*.

O número de *backups* que recebem as mensagens de *checkpoint* do primário depende da aplicação e do sistema distribuído. Se o sistema é especificado para tolerar f nós faltosos, então é necessário que o *checkpoint* chegue em pelo menos f réplicas. As falhas dos *backups* não têm nenhuma influência sobre o sistema até atingir f réplicas, uma vez que até este valor existem reservas suficientes para tolerar as f falhas de componentes. A freqüência das operações de *checkpoint* pode diminuir o desempenho do serviço replicado. No entanto, é possível estabelecer uma taxa de envio de *checkpoint*. Como por exemplo, na figura A.2, a cada x requisições a ocorrência de um *checkpoint*. Esta solução é vantajosa em casos onde a ocorrência de falha do primário não for freqüente. Quando o primário falha, o cliente retransmite as requisições a partir do último *checkpoint*, e portanto poderá ter que reavaliar as respostas do novo primário para que não ocorram repetições.

Para evitar que o cliente tenha que retransmitir requisições em situação de falha do primário pode ser utilizado um mecanismo de *log* que guarde em disco mensagens de requisições entre intervalos de *checkpoint* (figura A.3).

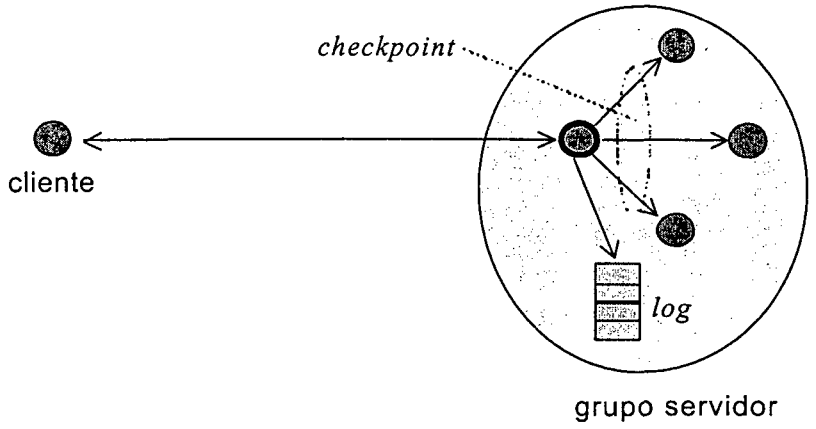


Figura A.3. Grupo réplicas passivas com mecanismo de *log*.

A.4.2 A abordagem de replicação ativa

A técnica de replicação ativa é baseada na abordagem de Máquina de Estado [Schneider90]. Nesta abordagem de replicação todas as réplicas não faltosas do grupo são ativas, isto é, recebem, executam e respondem a todas requisições dos clientes (figura A.4). Em relação à abordagem réplicas passivas o custo é maior, pois aloca mais recursos do sistema (memória, processador etc.) e o fluxo de mensagens no meio de comunicação é mais elevado. Por ter todas as réplicas ativas executando o mesmo conjunto de requisições é necessário garantir o determinismo de réplica, caso contrário podem surgir possíveis inconsistências de estado entre as réplicas. O retorno ao cliente da resposta do processamento do grupo servidor envolve algumas alternativas possíveis: o primeiro resultado a chegar é passado ao cliente; os resultados podem ser concatenados em seqüência e enviados ao cliente; ou ainda, os resultados passam por um votador ou ajustador que seleciona o resultado mais freqüente (maioria) para ser passado ao cliente.

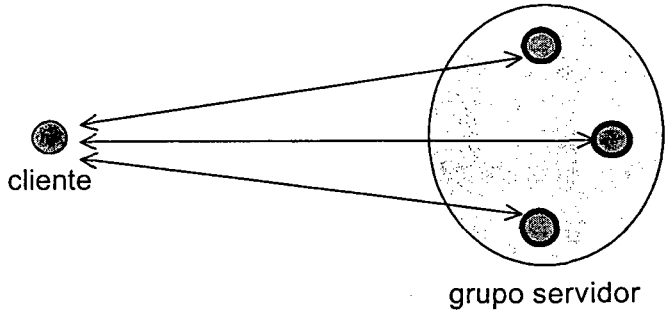


Figura A.4. Grupo de réplicas ativas.

Esta abordagem é mais apropriada em aplicações que requerem serviços ininterruptos com sobrecarga mínima em situações de falha (por exemplo, em aplicações tempo real), pois as falhas parciais são mascaradas.

O determinismo de réplicas nos modelos de réplicas ativas é conseguido assegurando as seguintes regras:

* **Acordo**: garante que uma mensagem (requisição) ou é recebida por todas réplicas corretas (não faltosas) ou é desconsiderada por todas estas réplicas corretas;

* **Ordenação**: as réplicas não faltosas recebem as requisições na mesma ordem relativa;

Alguns protocolos têm sido propostos com o propósito de relaxar a propriedade de acordo. Algumas semânticas de comunicação de grupo menos restritivas, por exemplo, as apresentadas em [Powell91], enfraquecem as necessidades de acordo para o engajamento de uma mensagem:

* ***At least K***: qualquer mensagem entregue a um participante deve ser entregue a no mínimo *K* participante corretos;

* ***Best effort K*** na ausência de falhas, qualquer mensagem entregue a um participante, deve ser entregue a *K* participante;

A abordagem réplicas ativas necessita, usualmente, da garantia de ordenação total. Neste caso, pode ser capaz de tolerar todo espectro de falhas (*crash,* omissão, temporização, valor ou arbitrária), desde que com algum ajuste no algoritmo de replicação para implementar um mecanismo de votação. Assumindo K como sendo o número de falhas de réplicas que o sistema pode tolerar, temos as seguintes especificações para tolerar as seguintes hipóteses de falhas:

- Falhas de *crash* e omissão: para tolerar K falhas de réplicas é necessário ter no mínimo K+1 réplicas no servidor replicado. Como não é preciso nenhum mecanismo de comparação de resultados, o protocolo pode ser simplificado, o primeiro resultado a chegar no cliente, é coletado e os restantes descartados. No caso de falhas de omissão é necessária a recuperação do

estado das réplicas faltosas. Esta recuperação de estado pode envolver a recuperação da fila de entrada (requisições).

- Falhas arbitrárias e de valor: para estas duas classes de falha é necessário que o serviço replicado tenha um total de 2K+1 réplicas. Ambos os tipos de falhas podem ser mascaradas envolvendo o voto majoritário. Para tanto, o cliente deve receber os resultados possíveis para que haja a maioria (K+1 resultados corretos).

- Falhas de temporização por atraso tem um tratamento idêntico ao do primeiro grupo. As falhas de temporização por antecipação correspondem ao segundo grupo e necessitam de mecanismos de voto majoritário [Powell91].

Em termos de requisitos de comunicação, é importante que mecanismos de disseminação confiável com garantia de ordem total sejam utilizados nesta abordagem, pois simplificam o atendimento dos requisitos de acordo e ordenação.

A.4.3 A abordagem de replicação semi-ativa

Alguns autores identificam uma terceira abordagem de replicação, chamada de semi-ativa (ou semi-passiva) [Powell91]. Esta abordagem apresenta uma combinação de algumas características das duas abordagens anteriores: as réplicas são todas ativas, sendo uma réplica privilegiada (figura A.5).

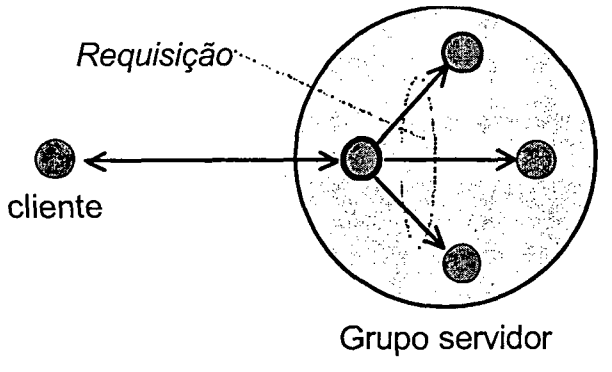


Figura A.5.Grupo réplicas passivas.

Nesta abordagem todas as réplicas executam os pedidos de serviço, no entanto, apenas a réplica privilegiada (líder) é responsável pela definição da ordem de execução dos

pedidos e também pela resposta aos clientes. Na falha da réplica privilegiada, uma das réplicas do grupo assume o seu papel, baseado em algum algoritmo de eleição. Nesta abordagem, como todas as réplicas são ativas, não existe a necessidade de recuperação de estado baseada em retrocesso (*rollback*).

A.5 Exemplos clássicos de técnicas de replicação

Apresentaremos nos próximos itens, de forma sucinta, alguns modelos de replicação de componente de *software*, segundo as três abordagens (passiva, ativa e semi-ativa), amplamente difundidas na literatura e implementadas em sistemas distribuídos.

A.5.1 Modelo de replicação *coordinator-cohort*

Este modelo de replicação passiva é implementado usando o sistema ISIS [Birman85] como suporte. As requisições emitidas por diferentes clientes são recebidas pelas réplicas do modelo. Uma réplica para cada replicação é nomeada *coordinator* (réplica ativa) enquanto que as outras operam como *cohort* (*backups*) prontas para substituir o *coordinator* em caso de falha. Cada *coordinator* pode atender a apenas um cliente. Devido ao modelo ser bastante complexo, mecanismos de ordenação total e causal são necessários.

O gerenciamento do número de membros participantes (*membership*) é feito de forma que cada membro saiba quais outros membros funcionais participam do processamento de uma requisição (quem são os *coordinators* e os *cohorts*). A lista de membros (view) é atualizada por um mecanismo de *membership*, que detecta a entrada ou a saída (normal ou por falha) de réplicas, e difunde o novo *view* para todos os membros por disseminação (*multicast*) confiável com ordem total (GBCAST). Requisições de diferentes clientes são enviadas ao grupo através de disseminação confiável com ordem total (ABCAST) assegurando que todas as réplicas funcionais recebam na mesma ordem relativa as requisições dos clientes.

Na figura A.6, um *coordinator* ao completar o processamento de uma requisição envia um *checkpoint* aos *cohorts*, para atualizarem seus estados, e depois responde ao cliente. O envio do *checkpoint* e da resposta para o cliente é feito através de disseminação

confiável com ordem causal (CBCAST), garantindo que todos os *cohorts* atualizem seus estados antes que qualquer requisição futura seja executada.

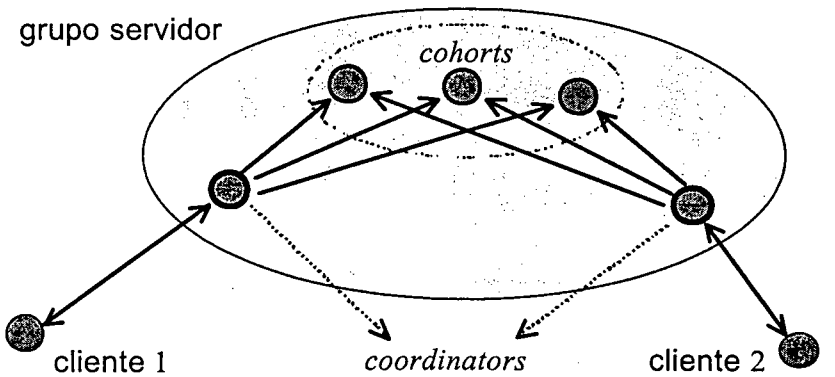


Figura A.6. Modelo de replicação *Coordinator-Cohort*.

Na situação de falha de um *coordinator* é feita a eleição de um novo dentre os *cohorts*. Devido ao fato de todos os membros do grupo terem uma mesma lista consistente e sempre atualizada de quais os membros operacionais, a escolha do novo primário não requer qualquer troca de mensagem entre os membros do grupo. Fazendo uso de uma mesma lista e de uma regra, todos os membros chegam à mesma decisão de qual réplica será o novo *coordinator* para a requisição em questão. A regra usada é baseada nos critérios: localização física do cliente (ex: uma réplica que está no mesmo nó que o cliente tem maior prioridade) e informação da taxa de balanceamento de carga (ex: número de requisições por unidade de tempo).

A.5.2 Modelo de replicação v*iewstamp*

Este esquema de replicação passiva está implementado no sistema Argus [Liskov87, Liskov88]. No Argus, objetos são denominados *guardians* e a unidade lógica de replicação é o *guardian*. A replicação envolve a criação de um grupo de *guardians*, que consiste de várias réplicas chamadas *cohort*. Este grupo se comporta como uma entidade lógica única. O conjunto de *cohorts* forma um grupo de configuração. Dentre os *cohorts* um é designado primário (réplica ativa) e o restante de *backups*. Se o primário falha um novo entre os *backups* é eleito. A falha de qualquer membro do grupo (primário ou *backups*) é registrada na lista de membros (*view*) que contém a informação de quem é o primário e quem são os *backups*.

Este esquema trata com falhas por particionamento da rede. Cada membro envia mensagens de "*are you alive*" para todos a fim de detectar possíveis falhas, portanto mudanças de *membership* são detectadas através de trocas de mensagens entre as réplicas. Na figura A.7 temos uma falha de comunicação entre o primário P1 (réplica *a*) e seus *backups* (*b, c, d, e*); o *view* do grupo antes da falha é *v1* = {*a: b, c, d, e, f*}. Como os *backups*, com exceção da réplica *f*, não conseguem comunicar-se com a réplica primária (ainda operacional), chegam ao consenso de sua falha e elegem um novo primário P2 (a réplica *b* da figura) com o *view* *v2* = {*b: c, d, e*}. Por parte do primário P1, este executa o algoritmo de mudança de *view*, como não consegue se comunicar com {*b, c, d,* e}, não obtém o consenso da maioria (só o voto de *f*), fica inativo para executar qualquer requisição que porventura receba. Quando uma mudança de *view* tem sucesso todos os membros pertencentes a este novo *view* iniciam num mesmo estado (no mais recente *checkpoint*). O *checkpoint* é emitido aos *backups* com garantia que pelo menos a maioria receba, como na falha do primário é o consenso da maioria que vale, é garantido que na transição de *v1* para *v2* pelo menos um membro possua o mais recente *checkpoint* do primário antigo (P1). Esse será o novo primário.

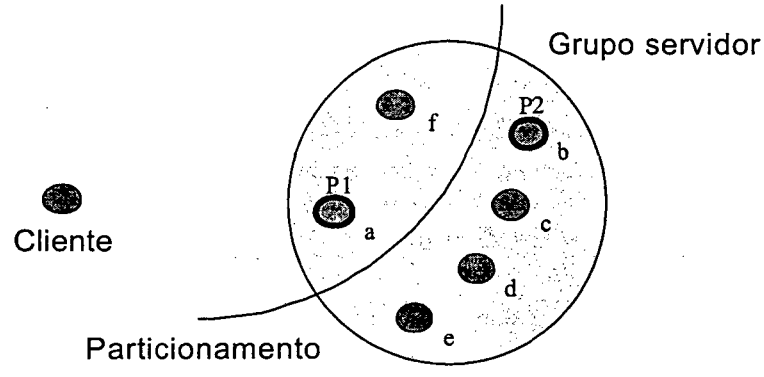


Figura A.7. Modelo de replicação *Viewstamp*.

Para implementar o algoritmo apresentado acima se utiliza de mecanismos do tipo *timestamp* e *view*. A associação destes dois mecanismos é chamada de *viewstamp*, onde o *view* dá o número de membros do grupo e indica quem é o primário e o *timestamp* estabelece a ordem lógica dos eventos.

A.5.3 Modelo de replicação ativa competitiva

No modelo de replicação competitiva todas as réplicas são ativas, mas apenas uma responde a uma dada requisição de entrada. A principal característica deste modelo é a competição entre as réplicas: somente a mais rápida responde a requisição. A coordenação da técnica é distribuída. Cada réplica possui um controlador associado, responsável pela recepção, disseminação e comparação das mensagens, ficando a réplica correspondente dedicada ao processamento das requisições. Para garantir a consistência entre as réplicas, todas as mensagens entre réplicas são transmitidas via disseminações confiáveis com ordem total.

O modelo de replicação competitiva pode ser configurado para tolerar dois conjuntos de falhas [Powell91]: *falhas por temporização*, envolvendo as semânticas de falha por *crash*, omissão e de temporização por atraso; e *falhas não controladas* (arbitrárias) que compreendem todo o espectro de falhas.

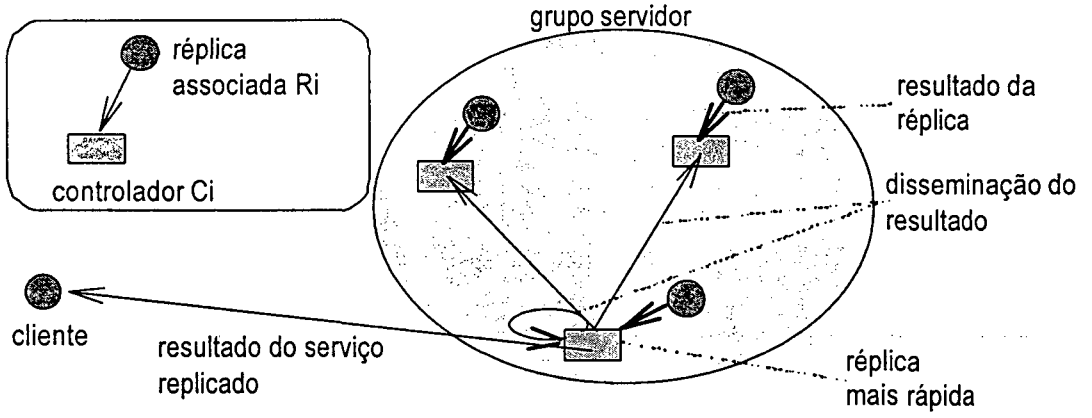


Figura A.8. Modelo de replicação competitiva para faltas de temporização.

A figura A.8 ilustra de maneira simplificada a replicação competitiva sob premissa de falhas de temporização. Neste caso, considerando o modelo Cliente/Servidor com o servidor replicado, uma requisição do cliente difundida no grupo servidor, é recebida pelos controladores `Ci`, que as repassam às replicas `Ri` associadas. Ao receber o resultado do processamento de sua réplica, cada controlador verifica se já possui mensagem de outro controlador do grupo com o resultado do mesmo processamento. Na ausência de mensagem, o controlador concatena um identificador aos resultados, e difunde a mensagem resultante no grupo de controladores. Se o controlador receber primeiro sua

própria mensagem depois da difusão, descobre que sua réplica foi a mais rápida e assim é o responsável pelo envio da resposta ao cliente; caso contrário, a sua mensagem é descartada. Esse algoritmo garante que apenas uma réplica responde ao cliente, pois todas as mensagens difundidas no grupo são observadas por cada membro na mesma ordem relativa (disseminação confiável com ordem total).

Por fim, a disseminação de uma mensagem de *fim_de_processamento* após o envio dos resultados ao cliente, pelo controlador da replica mais rápida encerra o ciclo de processamento referente à requisição do cliente. Esta mensagem dá possibilidade que se monte estratégias para a detecção de falha do controlador da réplica mais rápida e a sua substituição por outro controlador da replicação no envio dos resultados ao cliente [Brito95].

O protocolo citado acima mascara erros de temporização por atraso, de omissão e por *crash*. No que concerne o tratamento dos elementos falhos, dois procedimentos de detecção são previstos na literatura original [Powell91]:

* É admitido um fraco acoplamento entre o controlador e a réplica. Neste caso, no controlador são mantidos mecanismos de *timeout* para detectar a falha da réplica associada;

* A replicação competitiva privilegia a réplica mais rápida e, por conseqüência, pode levar a um assincronismo muito grande no conjunto de réplicas. Esta dessincronização é tratada realizando periodicamente um *rendez-vous,* onde todos os controladores difundem os resultados de suas réplicas entre si e o último a difundir envia o resultado ao cliente. Este *rendez-vous* é limitado no tempo de modo que possibilita também a detecção de controladores falhos.

Com premissas de faltas arbitrárias, o protocolo anterior deve ser acrescido de um mecanismo de comparação. Para tolerar f réplicas com falhas, o grupo deve conter ao menos $2f+1$ réplicas, isto porque é necessário comparar as saídas das várias réplicas e obter-se êxito na comparação de pelo menos de $f+1$ respostas concordantes. No protocolo, ao receber a resposta de sua réplica, cada controlador anexa sua assinatura e difunde a

mensagem resultante a todos os controladores do grupo. Devido à disseminação confiável com ordem total, as filas de mensagens dos controladores são idênticas e totalmente ordenadas. O consenso quanto à resposta é obtido quando um controlador do grupo receber e comparar com sucesso *f+1* respostas, incluindo a sua. O controlador que alcançou o limite *f+1* assinaturas comparadas e sua mensagem foi a *(f+1)*-ésima mensagem na comparação, envia a resposta ao cliente. Neste caso, as hipóteses de falhas incluem falhas de temporização por antecipação, que não eram consideradas no caso do protocolo anterior.

A.5.4 Modelo de replicação ativa cíclica

A principal característica deste modelo de replicação ativa é o uso de um mecanismo de passagem de bastão (*token*) entre os membros de um grupo replicado, onde a ordem de posse do bastão é definida segundo a seqüência de um anel lógico (*token ring*). Como no modelo de replicação competitivo, cada réplica tem uma ent_cont[2] que cuida de todo o processo de tolerância a falta do sistema. Este modelo, conhecido também de *round robin*, também pode ser configurado para tolerar falhas de temporização e falhas não-controladas. Disseminação confiável com ordem total é necessária para garantir a consistência entre as réplicas.

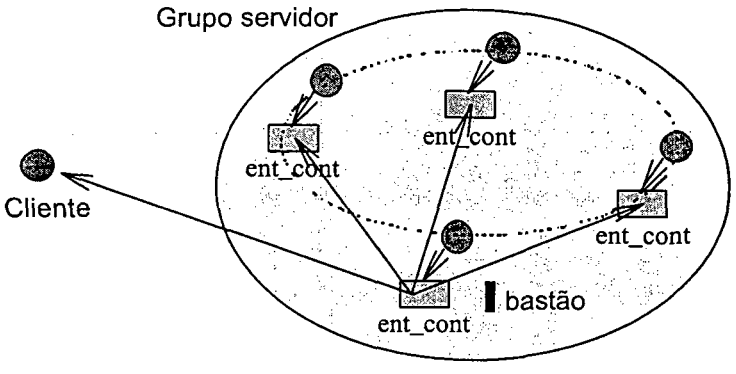


Figura A.9. Modelo de replicação cíclica para faltas de temporização.

A configuração para tolerar faltas restritas é basicamente a mesma que no modelo competitivo. A única diferença é que a ent_cont (entidade controladora) possuidora do bastão é o responsável pelo envio da mensagem para o cliente e pela disseminação para os membros restantes do grupo, quando feita a disseminação o bastão é passado para a outra

[2]entidade controladora

ent_cont da seqüência do anel lógico (figura A.9). As ent_cont não possuidores do bastão devem armazenar as mensagens recebidas até que chegue a sua vez de ser o privilegiado. Assim sendo, na recepção do bastão a ent_cont pode descartar as mensagens armazenadas até a especificada no bastão (*last_message*). Outra característica marcante deste modelo é que na falha da ent_cont privilegiada o modelo é reconfigurado para o modelo competitivo.

A tolerância a faltas não-controladas neste modelo também é basicamente idêntica ao modelo Competitivo. Quando a primeira ent_cont possuidora do bastão recebe uma mensagem da sua réplica controlada, ela anexa sua assinatura, emite por disseminação confiável a mensagem *claim* (mensagem original mais a assinatura) a todos os outros ent_cont do grupo ($2t$+1) e passa o bastão a próxima ent_cont definido no anel lógico. As subsequentes ent_cont's privilegiadas devem comparar a sua mensagem com as mensagens *claims* que receberam referentes ao mesmo processamento, e verificar se um total de t+1 mensagens são concordantes (mensagens iguais). No caso de êxito na comparação o possuidor do bastão envia a mensagem ao seu destino (ao cliente) e uma mensagem de *ack* (reconhecimento) as outras ent_cont determinando o fim da transação. No caso da comparação não atingir o limite de t+1 mensagens idênticas a ent_cont com o bastão simplesmente difunde sua mensagem *claim* e passa o bastão. Se até completar um ciclo do anel lógico não for alcançado o consenso da maioria (t+1 réplicas) é porque mais de t falhas de réplicas ocorreram no sistema, caracterizando uma falha no sistema como um todo. Se porventura ocorrer a falha da réplica privilegiada (perda do bastão), detectada através de um temporizador que marca o intervalo de tempo entre a recepção de uma mensagem original da réplica controlada e a recepção do bastão, o sistema é reconfigurado para o modelo competitivo.

A.5.5 Modelo de replicação líder/seguidores

Este esquema de replicação semi-ativa aproveita alguns mecanismos próprios das abordagens de réplicas passivas e ativas. A sua implementação se encontra no sistema Delta-4 [Barret90]. O modelo pode ser configurado para tolerar diferentes classes de falhas. Como foi mostrado anteriormente, protocolos de replicação semi-ativa requerem alguma forma de disseminação atômica que assegure que todos membros funcionais recebam um mesmo conjunto de mensagens e na mesma ordem relativa (os requisitos de acordo e ordenação para determinismo de réplica).

Neste modelo todas as réplicas são ativas, mas só a réplica líder responde as requisições do cliente e é responsável pelas decisões que afetam o determinismo das réplicas do conjunto. As decisões são propagadas do líder para seus seguidores via mensagem de sincronização (figura A.10). Qualquer mensagem (de requisição ou de resposta) é enviada pelo líder aos seus seguidores logo que são geradas, e quando os seguidores geram as mesmas respostas que o líder o próprio sistema de comunicação se encarrega de descartá-las, o modelo é especificado para nós *fail-silent*. Se o sistema é especificado para tolerar falhas de valor, o protocolo de replicação ativa usa um mecanismo de votação na réplica *líder* que usa as respostas das várias réplicas para detectar possíveis falhas. A fim de simplificar a apresentação deste modelo trataremos apenas a tolerância a falha por *crash*, pois não necessita utilizar um mecanismo de votação.

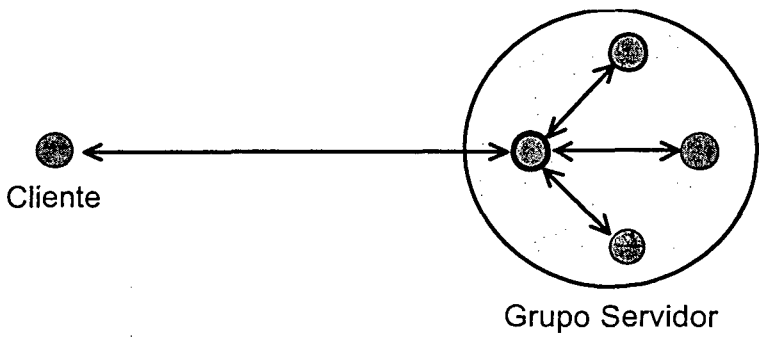


Figura A.10. Modelo de replicação líder/seguidores.

Na falha do *líder* um protocolo de eleição define o novo através de um *ranking* estático predefinido no momento da criação do grupo. A detecção de falha do *líder* e dos seus *seguidores* pode acontecer a partir:

* Do sistema de comunicação, quando o cliente tenta transmitir uma mensagem para o *líder*;

* Ou pelos *seguidores* que recebem periodicamente mensagens de "*I am alive*" do *líder*.

A grande vantagem deste modelo é que o líder pode iniciar o processamento de uma requisição assim que esta chega, ao contrário de outros modelos de replicação que precisam esperar por um acordo sobre a ordem das mensagens.

A.6 Conclusão

Apresentamos neste anexo uma introdução a alguns conceitos do processamento replicado, um estudo sobre técnicas de replicação de componentes de *software*, sobre as abordagens réplicas passivas, réplicas ativas e réplicas semi-ativas, enfatizando suas características na coordenação das replicas. A principal vantagem das técnicas de replicação de componentes de *software* sobre as chamadas técnicas de tolerância a faltas fortemente acoplada (em nível do suporte de *hardware*) é o menor custo de se adaptar aplicações tolerantes a faltas a diferentes arquiteturas de computadores ou atualizações (*updates*) destas.

	Replicação passiva	*Replicação ativa*	*Replicação semi-ativa*
Processamento de erros	Recuperação em retrocesso	Compensação de erro	Recuperação em avanço
Semântica de falhas	Crash	Sem restrições – faltas arbitrárias	Crash
Tempo de recuperação	Alto	Baixo	Baixo/médio
Acordo e ordem	Imposição da réplica privilegiada	Decisão distribuída	Imposição da réplica privilegiada
Réplicas deterministas	Em instantes precisos	Sim	Sim

Tabela A.1. Quadro comparativo das três abordagens.

Como apresentado, as técnicas de replicação podem seguir as abordagens réplicas ativas, passivas, ou uma variante de ambas, réplicas semi-ativas (ex.: modelo *líder-seguidores*). O quadro A.1 apresenta uma síntese comparativa destas três abordagens. A abordagem réplicas ativas tem um baixo *overhead* em situação de falha, pois todas réplicas são ativas e o mascaramento de falha de uma réplica é feito quase que instantaneamente. No sentido oposto, a abordagem réplicas passiva tem *overhead* maior, pois na falha do primário (ativo) um tempo é gasto para executar o protocolo de eleição do novo primário. A principal vantagem da abordagem réplicas passivas é a não necessidade do determinismo de réplicas (um custo menor em termos de suporte), pois apenas o primário é ativo e impõe seu estado sobre os *backups* através de *checkpoints,* porém esta abordagem tem a desvantagem de não tolerar classes de faltas menos restritivas.

ANEXO B

Redes de Larga Escala: Protocolos e Abordagens de Comunicação de Grupo

B.1 Introdução

O uso de comunicação de grupo para o desenvolvimento de aplicações confiáveis tem sido objeto de estudo em vários laboratórios de pesquisa. Os protocolos de difusão confiável, atômica ou ainda causal encontrados na literatura, entre eles [Amir93, Birman91, Chang84, Dolev93, Kaashoek91 e Melliar90], normalmente não são adequados para redes de larga escala. Os sistemas de larga escala são caracterizados pela grande distribuição espacial, com um caráter aberto, integrando quantidades significativas de recursos computacionais (tanto de *hardware* como de *software*) assinalados pela sua heterogeneidade. As dimensões e a complexidade destes sistemas torna proibitiva a verificação das propriedades embutidas nestes protocolos. Os custos de transmissão e a terminação dos protocolos usuais crescem com o aumento do tamanho dos sistemas [Hayden96].

Em [Guerraoui97] é lançada a noção de "implementação genuína" no sentido de favorecer a escalabilidade nestes protocolos. Uma implementação de *multicast* atômica é genuína se satisfaz as propriedades de Minimidade e de Localidade:

- **Minimidade**: os únicos processos envolvidos em uma implementação de *multicast* atômico são o emissor da mensagem e os processos receptores dos diferentes grupos;

- **Localidade**: limita as decisões de consenso em um simples grupo, ao invés de ter vários grupos envolvidos nisso. Esta abordagem favorece a

decomposição hierárquica do problema e uma implementação modular (permitindo que cada grupo tenha seu próprio protocolo de consenso).

Por outro lado, sistemas distribuídos de larga escala apresentam características de sistemas assíncronos. Os padrões de trafego e as velocidades de processamento em processadores no sistema não são conhecidos a priori.

Em [Fischer85], um trabalho clássico, era identificada a impossibilidade de um algoritmo determinista resolver um problema de consenso em sistemas assíncronos com uma falha. O consenso é um elemento de base se considerarmos a tolerância a faltas em sistemas distribuídos. Serviços de pertinência (*membership*), eleição de líder e acordo na entrega de mensagens ("*delivery*") são exemplos de uso deste elemento de base.

A impossibilidade de soluções deterministas implica no uso de abordagens probabilistas. Técnicas de randomização [Chor89], modelos de sincronismo parcial [Dolev87, Dwork88, Fetzer95] e detectores de falhas não confiáveis são exemplo destas abordagens encontradas na literatura.

O objetivo deste anexo é apresentar um estudo sobre comunicação de grupo em sistemas de larga escala. São explorados neste estudo sobre protocolos aspectos relacionados com a forma de garantir a atomicidade e a ordem.

B.2 Consenso em sistemas distribuídos

Protocolos de consenso buscam permitir que processos alcancem uma decisão comum mesmo em situação de falhas. Em [Chandra96] é apresentado um estudo bastante detalhado sobre a resolução deste problema em sistemas assíncronos através de propriedades atribuídas a detectores de falhas. Nesta abordagem processadores são associados com detectores de falhas que passam informações ao seu processador sobre o estado de funcionamento dos outros processadores do sistema. Os detectores de falhas são classificados segundo duas propriedades: completude (*completness*) e precisão (*accuracy*) [Chandra96].

- **Completude**: especifica que existe um tempo, após o qual, todos os processos corretos suspeitam permanentemente de todos os processos que estão falhos. Dois tipos de completude são identificados:

    - **Completude forte**: todo processo que falha terminará por ser permanentemente identificado como suspeito de falha por todos os processos corretos do sistema;

    - **Completude fraca**: todo processo que falha terminará por ser permanentemente identificado como suspeito de falha por algum (pelo menos um) processo correto do sistema;

- **Precisão**: existe um tempo, a após o qual, qualquer processo correto não suspeitará de processos corretos. Esta propriedade restringe os enganos (suspeitas errôneas) que um detector de falhas possa cometer. Vários tipos de precisão são identificados:

    - **Precisão forte**: nenhum processo correto pode ser suspeito de falha antes da ocorrência de sua falha. Esta é uma propriedade difícil de implementar (se não impossível) em sistemas assíncronos;

    - **Precisão fraca**: deve existir pelo menos um processo correto que nunca é tido como suspeito;

    - **Precisão forte não imediata**: existe um tempo, após o qual, processos corretos não são suspeitos por qualquer outro processo correto.

    - **Precisão fraca não imediata**: existe um tempo, após o qual, pelo menos um processo correto nunca é suspeito por qualquer outro processo correto.

Combinando as duas propriedades da completude com as quatro de precisão obtemos as oito classes de detectores de falha (tabela B.1). Completude forte com precisão forte formam a classe dos detectores perfeitos – impossíveis de se conseguir em sistemas assíncronos. A classificação dos detectores de falha é necessária porque, conforme

veremos nos próximos itens, os protocolos de comunicação de grupo para sistemas assíncronos apresentados estão baseados em uma dessas oito classes de detectores de falha definidas na tabela B.1

	Precisão forte	Precisão fraca	Precisão forte não imediata	Precisão fraca não imediata
Completude forte	*P perfeito*	*S forte*	*Não imediatamente perfeito ◊P*	*Não imediatamente forte ◊S*
Completude fraca	*Q*	*W fraca*	*◊Q*	*Não imediatamente fraca ◊W*

Tabela 6.1. Oito classes de detectores de falha definidos em termos da perfeição e precisão.

O algoritmo de consenso apresentado em [Chandra96] é baseado na classe *◊S* de detectores de falha. Neste algoritmo, os detectores de falha podem erroneamente adicionar e remover dinamicamente suspeitos de suas listas, sem afetar o funcionamento do protocolo como *◊S* (não imediatamente forte) é necessária uma maioria de processos corretos – o número de processos faltosos deve ser menos que a metade.

Neste protocolo, em cada rodada r um grupo de processos (com n membros) possui um coordenador *coord* definido dinamicamente de acordo ao anel virtual ($c = (r \bmod n) + 1$). Após n rodadas cada processo do grupo terá exercido a função de coordenador uma vez. Durante uma rodada, o coordenador atual é o centralizador de todas as mensagens do grupo. A função do coordenador é resolver o consenso, propondo um valor para apreciação aos outros processos do grupo. Se o coordenador corrente é correto e não suspeito por qualquer processo correto do grupo, então terá sucesso, e poderá disseminar o valor de consenso. Este valor de consenso deve ser um dos valores propostos pelos processos envolvidos no consenso.

Cada rodada possui quatro fases. Tomando como exemplo a figura B.1, apresentamos as quatro fases do algoritmo, de maneira sucinta, em uma condição livre de suspeita ou falha:

- Na primeira fase todos os processos enviam suas propostas contendo uma estampilha (número de seqüência e identidade do emissor) e o número da rodada ao coordenador.

- Na segunda fase, o coordenador atual espera pelas propostas até receber pelo menos da maioria do grupo de processos. Após isto, seleciona a maior das estampilhas e difunde o resultado v ao grupo.

- Na terceira fase, cada processo pode prosseguir de duas maneiras:

    - Ao receber o *propose*, enviar um *ack* ao *coord* indicando que aceita o resultado v ou;

    - Após acessar o módulo detector de falhas e detectar que *coord* é suspeito, enviar um *nack* ao *coord*. Em ambos casos, os processos prosseguem para a próxima rodada.

- Na quarta fase, *coord* espera pela maioria das respostas (*ack* ou *nack*). Se todas as respostas, desta maioria, forem *ack*, o *coord* assume que a maioria dos processos aceitou o resultado v e, portanto retém v. Então, o *coord* difunde uma mensagem *decide* requisitando a todos que decidam por v. Em qualquer instante, quando um processo q entrega a mensagem *decide* ele está assumindo a decisão do coordenador.

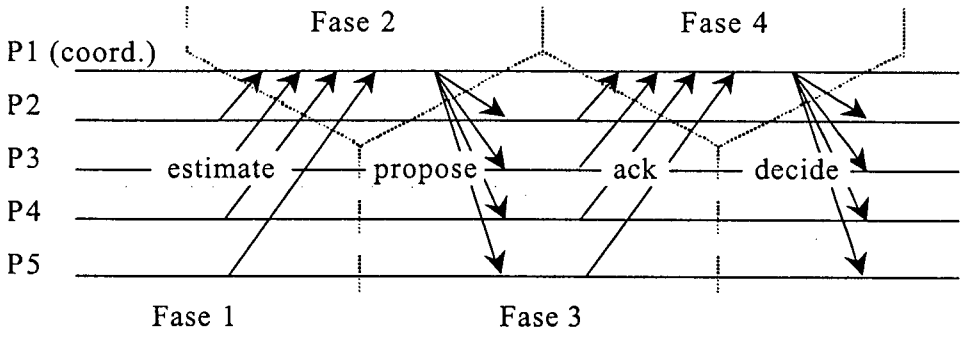


Figura B.1. Algoritmo de consenso em que nenhuma falha ou suspeita ocorre.

Em situações livres de suspeita ou falha do coordenador o número de mensagens trocadas será [3*(n-1)] mais a mensagem com o valor final do consenso, difundida pelo coordenador. Mais detalhes sobre este algoritmo, juntamente com provas, pode ser obtido em [Chandra96].

Outra contribuição em [Chandra96] é mostrar a equivalência entre o problema de consenso e a difusão atômica em sistemas assíncronos sujeitos a falhas de *crash*.

B.3 Comunicação de grupo em sistemas de larga escala

Diversos protocolos de comunicação de grupo em sistemas de larga escala têm sido propostos na literatura [Fritzke98, Hayden96, Rodrigues96, Rodrigues98, entre outros]. Esses protocolos podem ser classificados como simétricos, assimétricos ou híbridos. Nos algoritmos simétricos, as decisões de acordo e ordenação são estabelecidas por todos os processos de modo descentralizado. Os algoritmos baseados na abordagem assimétrica (também conhecidos como *sequencer-site*) apresentam um processo centralizador (ou mais de um) como responsável por definir a ordenação das mensagens aos outros processos membros do grupo. A abordagem híbrida consiste em combinar conceitos das duas abordagens anteriores. Define a formação de diferentes grupos, alguns atuando no modo simétrico e outros no modo assimétrico em um sistema distribuído.

B.3.1 Abordagem simétrica

Nesta seção são apresentados três protocolos: o Pbcast [Hayden96] que corresponde a um protocolo de difusão sustentado em técnicas probabilistas; o protocolo de [Fritzke98] que usa protocolos de consenso em várias fases para estabelecer estampilhas de mensagens difundidas; e o SCALATOM [Rodrigues98] que utiliza, da mesma maneira, um algoritmo de consenso para definir a ordem de entrega das mensagens.

B.3.1.1 Pbcast

Este protocolo se propõe a ser uma solução voltada para a difusão atômica em sistemas de larga escala. O Pbcast, apresentado em [Hayden96], é baseado num modelo de sistema probabilista. O algoritmo apresentado preenche as necessidades de escalabilidade tomando como base duas características: 1) os custos de transmissão da mensagem e a latência[1] crescem lentamente com o aumento do tamanho do sistema, 2) a probabilidade do protocolo falhar durante uma execução, aproxima exponencialmente de zero à medida que o número de processos do sistema aumenta. Quanto maior o número de processos no grupo maior a confiabilidade do protocolo. O algoritmo é baseado em um modelo "epidêmico" mostrando que uma doença contagiosa pode se alastrar em quase toda uma população ou quase ninguém pega esta doença [Bailey75].

[1] O tempo que leva para uma mensagem cruzar uma conexão de rede, do emissor para o receptor.

O modelo assume a existência de um grupo estático de processos, em que todos estão conectados com todos através de um canal de comunicação ponto-a-ponto confiável. A difusão Pbcast consiste na propagação de uma mensagem para todos os processos de um sistema através da retransmissão ocasional e insistente. Quando um processo recebe uma mensagem *m* pela primeira vez, seleciona, de forma aleatória, a quais processos do grupo deve retransmitir a mensagem, e só então pode entregar a mensagem para a aplicação. Mensagens repetidas são descartadas. Cada processo executa uma única vez a retransmissão aleatória.

Para ordenação total, o protocolo aproveita o mecanismo de relógios lógicos de [Lamport78], define uma estampilha para cada mensagem emitida. O Pbcast assume um número fixo de *R* rodadas, em que cada processo participa no máximo uma vez. Cada rodada é definida por *r*, sendo que no primeiro envio de uma mensagem *m* o valor de *r* é igual a *R*. Portanto, são agregados à mensagem *m* uma estampilha que contém o identificador do processo emissor (*Pi*), a estampilha (valor do relógio lógico do emissor (*c*)) e o número da rodada (*r*). Toda vez que um processo envia ou retransmite uma mensagem *m* deve decrementar em uma unidade o valor de *r*. Um processo que recebe uma mensagem *m* deve verificar se $r = 0$ (atingiu *R* rodadas), se sim, ele não retransmite a mensagem e entrega *m* para a camada da aplicação. Alcançar a ordenação total consiste em fazer os processos atrasarem a entrega de uma mensagem *m* até que qualquer outra mensagem mais recente (de rodadas posteriores), de acordo com a estampilha, já tenha sido recebida no processador, o que garante o *delivery* na ordem por estampilha.

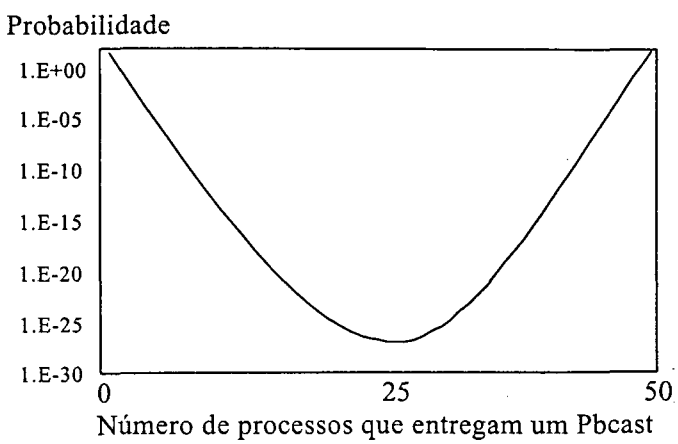


Figura B.2. Curva de distribuição do Pbcast.

O algoritmo garante uma difusão com "confiabilidade probabilista" e também ordem total no *delivery*. Isto é, garante numa distribuição de entrega bimodal que uma mensagem Pbcast é sempre entregue para a maioria ou para a minoria dos processos, e quase nunca para um número médio de processos. Na figura B.2, é apresentada a curva de distribuição de entrega, que indica a probabilidade de diferentes números de processos na entrega para a sua aplicação de uma mensagem Pbcast. Por exemplo, a probabilidade de que apenas 26 dos 50 processos entreguem um Pbcast é de 1.E-28. Embora os processos possam não entregar o mesmo conjunto de mensagens, todas as mensagens são entregues em uma ordenação total consistente, de acordo com a estampilha agregada na mensagem pelo emissor. O Pbcast pode ser utilizado em protocolos estilo votação onde, por exemplo, atualizações devem ser feitas na maioria dos processos para serem válidos.

B.3.1.2 Algoritmo de Fritzke et al.

O protocolo apresentado em [Fritzke98] trata da difusão atômica para múltiplos grupos em um contexto de sistema de larga escala em que os processos podem sofrer falha por paragem (*crash*). É assumido que os canais de comunicação são confiáveis. Especificamente, o protocolo trata de *multicast* atômico e segue a noção de "implementação genuína", introduzida em [Guerraoui97].

Neste protocolo, um sistema distribuído P é estaticamente particionado em vários subgrupos G_i não vazios e não sobrepostos. Esses subgrupos são definidos de acordo com a estrutura do sistema ou de acordo com as necessidades da aplicação.

Na fase 1 deste protocolo um processo para difundir a mensagem m para os subgrupos de P, usa um protocolo confiável para cada subgrupo G_i (G_1, G_2 e G_3, na figura B.3). Cada processo em um subgrupo G_i gera uma estampilha Ts (valor de seu relógio lógico e a identidade do subgrupo) para as mensagens que recebe. Na fase 2, cada processo receptor interage com os seus pares no seu subgrupo, executando um protocolo de consenso local (subgrupos podem executar protocolos de consenso diferentes) para a decisão sobre uma estampilha para a mensagem, a partir dos valores propostos de cada receptor no subgrupo considerado. Na terceira fase todos os processos do subgrupo difundem o valor de estampilha (Ts_i) obtido no consenso a todos os processos do grupo G.

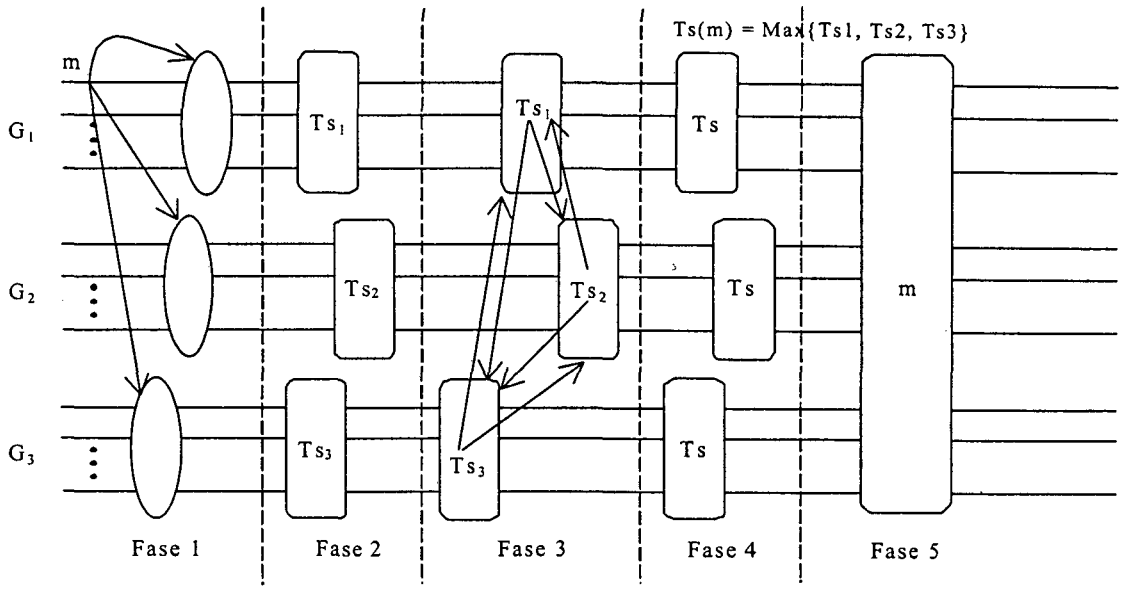


Figura B.3. As fases do protocolo de Fritzke et al.

Mensagens repetidas são tratadas pela camada de fiabilização. Como nem sempre os subgrupos decidem uma mesma estampilha para *m* é necessário que se alcance um consenso global sobre a estampilha definitiva para *m*. Uma vez que todos os processos de *G* recebam todas as estampilhas propostas (na figura B.3: Ts_1, Ts_2 e Ts_3), a estampilha global para *m* será definida pela maior de todas as estampilhas alcançadas pelos subgrupos G_i (Ts(m) = Max$\{Ts_i\}$ (fase 4 da figura B.3). Após isto, os processos dos subgrupos de G_i atualizam de forma consistente seus relógios lógicos de grupo de acordo com a estampilha definitiva da mensagem *m*. Por fim, a entrega da mensagem *m* é feita quando sua estampilha é a menor de entre todas as estampilhas das mensagens não entregues (fase 5 da figura B.3).

B.3.1.3 SCALATOM

O SCALATOM (*SCALable ATOmic Multicast*) apresentado em [Rodrigues98] também corresponde a um *multicast* atômico evolvendo múltiplos grupos (não sobrepostos) de um sistema distribuído de larga escala. O protocolo adota um modelo de grupo aberto (cliente não precisa pertencer ao grupo) que satisfaz a propriedade de minimidade.

Este protocolo também consiste em difundir um número de seqüência único (uma ordem total) para cada mensagem difundida no conjunto de subgrupos. As mensagens são

entregues para a aplicação (*delivery*) de acordo com a ordem estabelecida a partir destes números de seqüência.

O protocolo consiste em definir um número de seqüência único para cada mensagem *m* difundida. As mensagens são entregues para a aplicação (*delivery*) de acordo com a ordem do número de seqüência. Para definir um número de seqüência para uma mensagem *m* (SN(*m*)), o SCALATOM executa o protocolo de consenso baseado em detectores de falhas não confiáveis apresentado em [Chandra96] (item B.2). A classe dos detectores de falhas assumida é $\Diamond S$ (não imediatamente forte).

O protocolo define três filas para ordenar as mensagens: **Fila de Pendentes** (*Pending Buffer*), **Conjunto de Predecessores Potenciais** (PPS - *Potential Predecessor Set*) e a **Fila de Entrega** (*Delivery Queue*). A figura 3.4 apresenta as cinco fases desse protocolo. Na primeira fase, o processo *Pi* inicia o protocolo invocando a primitiva de difusão atômica (*TO-multicast* (m, Dst(m))) no conjunto destino Dst(m). A difusão atômica envolve um conjunto de passos. O primeiro passo é o *R-multicast* (difusão confiável) da mensagem *m* nos processos de destino Dst(m), ou seja, todos os processos do conjunto de subgrupos.

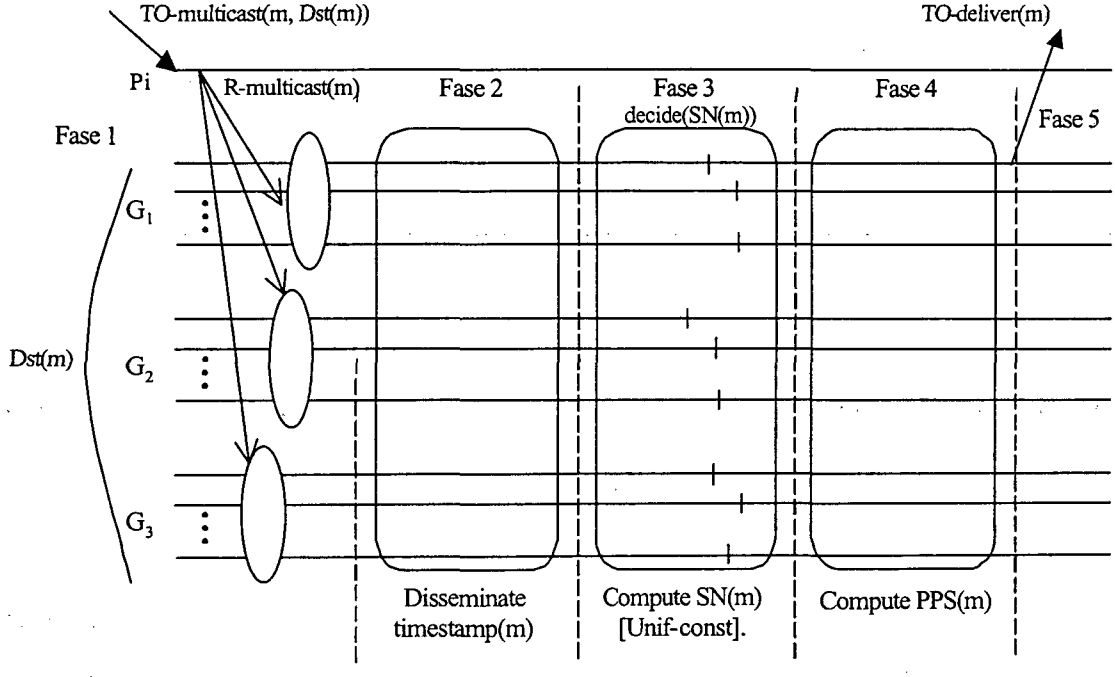


Figura B.4. As fases do protocolo SCALATOM.

Na segunda fase, o processo *P* que recebeu a mensagem pelo *R-multicast* atribui uma estampilha à mensagem *m,* atualiza o relógio lógico ([Lamport78)], e envia esta estampilha para todos os processos de destino Dst(m). Por fim, P armazena a mensagem na Fila de Pendentes.

Na fase três, o processo *P*, em função da fase dois, espera até coletar as estampilhas da maioria dos processos de Dst(m) para poder invocar a função de consenso e computar o número de seqüência (SN(m)) para a mensagem *m*. Quando a fase três termina um número de seqüência é definido para a mensagem *m*. Com esta definição todos os relógios lógicos são atualizados e a mensagem *m* é removida da Fila de Pendentes e inserida no Conjunto de Predecessores Potenciais (PPS).

Na fase 4, as informações da fila PSS de cada processador é disseminada no grupo usando um protocolo de *gossip*[2]. Esta disseminação é executada no sentido que os processos troquem históricos de mensagens (*history*(Pi, n)). Um histórico (*history*(Pi, n)) de um processo *Pi* corresponde ao conjunto de mensagens difundidas no grupo (difusão confiável) com estampilha menor que *n*. O PPS é calculado localmente, a partir das mensagens de todos os históricos recebidos *history*(Pj, SN(m)) da maioria dos membros do grupo. Isto assegura que PPS(*m*) contém todas as mensagens com número de seqüência menor ou igual a SN(*m*). Por fim, na fase cinco, a mensagem *m* é entregue à aplicação quando for cabeça de fila em PPS(*m*) [Rodrigues98].

B.3.2 Abordagem híbrida

Os protocolos que seguem a abordagem assimétrica requerem menos fases e, portanto são mais eficientes. No entanto, estão sujeitos a problema de contaminação[3] [Gopal92] e, na maioria dos casos, não são escaláveis. A principal característica desta abordagem é que a coordenação na ordenação e no *delivery* de mensagens é centralizada em um processo ou sítio (*sequencer-site*).

[2] Protocolo de *gossip* consiste de um mecanismo de disseminação aleatória de mensagens. Qualquer processo que recebe uma mensagem *m* pela primeira vez, seleciona um percentual dos processos do grupo que ainda não receberam uma mensagem e envia a mesma para estes. Todos processos executam este procedimento, para a mensagem *m*, uma única vez e mensagens duplicadas recebidas são descartadas.

[3] Propagação da inconsistência provocada por processos faltosos sobre os processos corretos.

Neste texto, retemos neste item protocolos que seguem a abordagem híbrida que é uma combinação de simétrica e assimétrica. Basicamente, esta abordagem consiste de grupos (ou ainda subgrupos) em um sistema distribuído em que seus membros executam protocolos simétricos e assimétricos, de acordo com as características dinâmicas do sistema. Nesta seção são apresentados dois protocolos: o Newtop [Ezhilchel95] que apresenta um conjunto de serviços (detecção de falhas, gerenciamento de membros, etc.) em camadas para prover difusão atômica (*multicast*) segundo uma abordagem híbrida; e o trabalho de [Rodrigues96] que também apresenta um algoritmo baseado em uma hierarquia de serviços para o fornecimento de difusão atômica.

B.3.2.1 Newtop

O Newtop (*NEWcastle Total Order Protocol*) [Ezhilchel95] é um sistema de suporte para processamento de grupo em sistemas assíncronos. Este suporte é composto por um conjunto de serviços estratificados em camadas, fornecendo nos níveis mais altos difusão atômica e gerenciamento de pertinência de membros (*membership*) no grupo. Neste suporte, a principal característica é a possibilidade que processos possam pertencer a diferentes grupos (grupos sobrepostos). Diferente dos sistemas apresentados anteriormente, o Newtop não objetiva simplesmente, a difusão de mensagens para todos os membros de todos os grupos ou subgrupos que formam o sistema de larga escala. E sim, garantir, através dos conceitos de sincronia virtual[4] [Birman91, Birman88], que um processo pertencendo a vários grupos (grupos sobrepostos) se mantenha consistente em relação aos *deliverys* nos diversos grupos a que pertence.

Sendo uma abordagem híbrida, os grupos no Newtop executam exclusivamente em modo simétrico ou assimétrico. Porém, processos podem executar em ambos modos desde que pertençam a diferentes grupos atuando em modos diferentes.

No Newtop, grupos simétricos executam um protocolo de difusão com ordem total fazendo uso de mecanismo de relógio lógico [Lamport78]. Já os grupos em modo assimétricos possuem um de seus membros como seqüenciador, concentrando as decisões de ordenação e *delivery* no grupo.

[4] Assegura que processos percebam falhas de processo e outras mudanças na configuração de grupo como ocorrendo no mesmo tempo lógico. Processos em uma mesma visão (da lista de membros) entregam (*delivery*) o mesmo conjunto de mensagens.

Este protocolo assume falhas por paragem (*crash*) e canais de comunicação ponto a ponto confiáveis – mensagens não corrompidas e ordem FIFO nestes canais. A detecção de falhas é baseada em *timeouts* cujos valores são definidos dinamicamente considerando as condições da rede. Estes detectores fornecem informações de processos suspeitos para o consenso executado na sentido de produzir, através do serviço de *membership*, a nova lista de membros (*view*) em cada processo correto.

B.3.2.2 Rodrigues et al.

O protocolo de difusão (*multicast*) com ordenação total para sistemas de larga escala apresentado em [Rodrigues96] é outro exemplo que segue a abordagem híbrida. Diferente do Newtop, este protocolo permite que processos em um mesmo grupo executem nos modos simétrico e assimétrico. Os processos de grupo podem mudar de um modo de execução para outro, com o objetivo de alcançar uma menor latência, adaptar-se a uma taxa de transmissão (*throughput*) ou devido a um atraso de rede. Segundo os autores, é ideal para aplicar em sistemas em que padrões de tráfego e topologia de rede não são conhecidos *a priori*.

A difusão atômica se utiliza de mecanismo de relógio lógico para ordenar mensagens segundo tempos lógicos. Neste protocolo, grupos são formados por processos ativos que operam em modos simétricos e assimétricos; e processos passivos que se executam somente segundo protocolos assimétricos. Os processos ativos são os responsáveis pela definição da ordem total no *delivery* das mensagens por estes e os processos passivos do mesmo grupo.

Na figura B.5, são apresentadas as três fases deste protocolo. Quando um processo (Pa_2) deseja enviar uma mensagem m para todos os grupos, ele, primeiro, difunde m ao grupo ao qual pertence. À mensagem m é concatenada uma estampilha (Pa_2, Ca_2) relativa ao grupo a, onde Pa_2 é o identificador do processo emissor e Ca_2 é o valor do contador de Pa_2. O seqüenciador do grupo a (processo Pa_1) procede no sentido de definir uma estampilha global para a mensagem m, definida pelos processos seqüenciadores de todos os grupos através de um algoritmo simétrico (fase 2). Finalmente, quando é alcançada a estampilha global da mensagem m, cada processo seqüenciador difunde para seus respectivos processos passivos a mensagem m com a estampilha global.

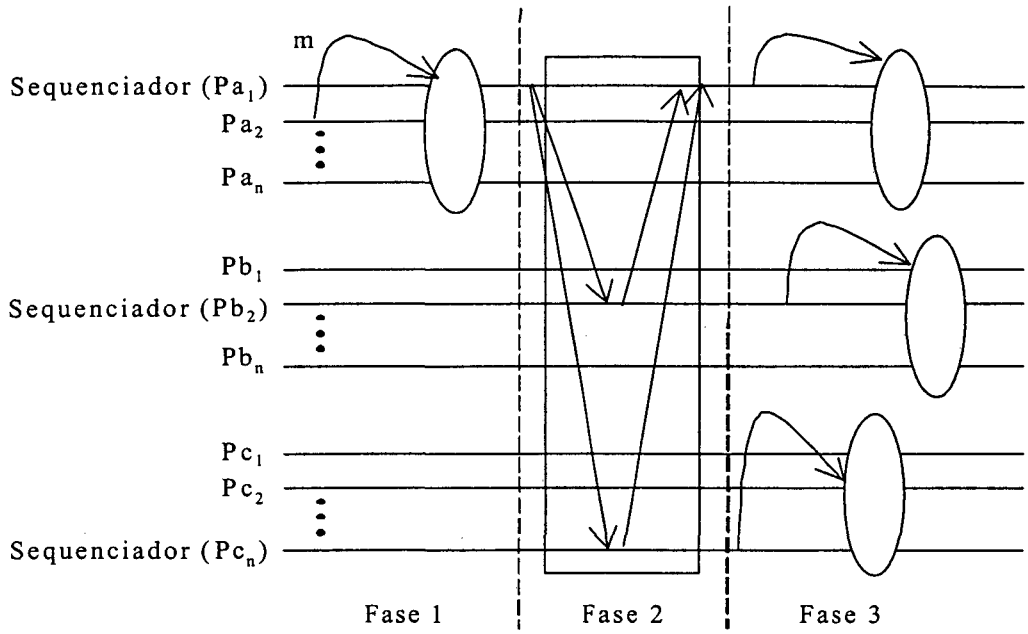


Figura B.5. Fases do protocolo.

Dependendo da dinâmica do sistema de larga escala cada processo pode mudar de função dentro do grupo, por exemplo, um processo passivo pode mudar de seqüenciador, um processo passivo pode mudar para processo ativo (seqüenciador) e, por último, um processo ativo pode mudar para processo passivo.

B.4 Conclusão do capítulo

Neste capitulo foi realizado um estudo de protocolos de comunicação de grupo para redes de larga escala. Foram introduzidos vários conceitos e propriedades, sobre as quais se fundamentam esses protocolos. Por fim, foram apresentados alguns exemplos significativos, presentes na literatura, divididos segundos três abordagens: simétrica, assimétrica e híbrida.

Nosso interesse nesse estudo teve o objetivo de identificar as principais características desses protocolos no sentido de tratar o aspecto da escalabilidade. O estudo apresentado neste anexo serviu de base na proposição de um modelo, baseado no CORBA, para comunicação de grupo em sistemas de larga escala, conforme apresentado no próximo capitulo 7.

Referências Bibliográficas

[Amir92] Y. Amir, D. Dolev, S. Kramer and D. Malki, **"Transis: A Communication Subsystem for High Availability"**, In 22nd International Symposium on Fault-Tolerant Computing, IEEE, julho de 1992.

[Amir93] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal and P. Ciarfella, **"Fast Message Ordering and Membership Using a Logical Token-Passing Ring"**. In Proceedings of the 13th International Conference on Distributed Computing Systems, pages 551-560, IEEE, 1993.

[Anderson81] T. Andeson and P. A. Lee, **"Fault Tolerance – Principles and Practices"**, Prentice-Hall, 1981.

[Bailey75] N. Bailey, **"The Mathematical Theory of Infectious Diseases"**, Charles Griffin and Company, London, 2 Edition, 1975.

[Barret90] P. A. Barret et al, **"The Delta-4 Extra Performance Architecture (XPA)"**, Proceedings of FTCS-20, Newcastle upon Tyne, June 1990.

[Birman85] K. Birman, et. al., **"Implementing Fault-Tolerant Distributed Object"**, IEEE Transactions on Software Engineering, June 1985.

[Birman88] K. Birman, T. Joseph and F. Schmuck, **"ISIS - A Distributed Programming User's Guide and Reference Manual"**, The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, March 1988.

[Birman91] K. P. Birman, **"The Process Group Approach to Reliable Distributed Computing"**, Technical Report Tr91-1216, Cornell University Computer Science Department, Ithaca, N.Y., July 1991.

[Brito95] O. F. G. Brito, O. G. Loques, **"Técnicas de Replicação de Módulos no Ambiente RIO"**, VI Simpósio de Computadores Tolerantes a Falhas, Canela - RS, 1995.

[Brose97] G. Brose, **"JacORB: Implementation and Design of a Java ORB"**, Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems, Cottbus, Germany, Chapman & Hall, Sep. 1997.

[Budhiraja93] N. Budhiraja, K. Marzulo, F. B. Schneider, S. Toueg, **"The Primary-Backup Approach"**, Distributed Systems (2nd Edition). Edited by S. Mullender, chapter 4. Adison-Wesley, 1993.

[Chandra96] T. Chandra and S. Toueg, **"Unreliable Failure Detectors for Reliable Distributed Systems"**, JACM, 43(1):225-267, March 1996.

[Chang84] J. Chang and N. Maxemchuck, **"Reliable Broadcast Protocols"**, ACM, Transactions on Computer Systems, 2(3), August 1984.

[Chen92] I. R. Chen and F. B. Bastini, F. B. **"Reliability of fully and partially replicated systems"**, IEEE Trans. Reliability, vol 41, nº 2, pp. 175-182, June 1992.

[Chiba93] S. Chiba, **"Open C++ Programmer's Guide"**, Technical Report 93-3, Department of Information Science, University of Tokio, 1993.

[Chor89] B. Chor AND C. Dwork, **"Randomization in Byzantine Agreement"**, Advances in Computer Research 5, 443-497.

[Chorus92] Chorus Systemes, **"Chorus Simulator v3 r4 - Programmer's Guide"**, 1992.

[Chung98] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, **"DOORS: Providing Fault Tolerance for CORBA Applications"**, in poster session of Middleware '98, Sept. 1998.

[Cristian90] F. Cristian, **"Asynchronous Atomic Broadcast"**, IBM Technical Disclosure Bulletin, 33(9):115-116, Feb 1991. Presented at the First IEEE Workshop on Management of Replicated Data, Houston, TX, Nov 1990.

[Cristian96] F. Cristian, **"Synchronous and Asynchronous Group Communication"**, Communications of the ACM, 39(4):88-97, Apr 1996.

[Cristian98] F. Cristian and C. Fetzer, **"The Timed Asynchronous Distributed System Model"**, 28th International Symposium on Fault-Tolerant Computing, IEEE Computer Society, 1998.

[Dolev87] D. Dolev, C. Work and L. Stockmeyer, **"On the Minimal Synchronism Needed for Distributed Consensus"**, Journal of the ACM 34, 1 (Jan), 77-97.

[Dolev93] D. Dolev, S. Kramer and D. Malki, **"Early Delivery Totally Ordered Multicast in Asynchronous Environments"**, In Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing, pages 544-553. IEEE, 1993.

[Dwork88] C. Work, N. Lynch and L. Stockmeyer, **"Consensus in the Presence of Partial Synchrony"**, Journal of the ACM 35, 2 (April), 288-323.

[Ezhilchel95] P. D. Ezhilchelvan, R. Macêdo and S. K. Shrivastava, **"Newtop: A Fault-Tolerant Group Communication Protocol"**, In Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE Computer Society. Pages 296-306, 1995.

[Fabre95] J. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud and Z. Wu, **"Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming"**, Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena (CA), June 1995.

[Felber98] P. Felber, **"The CORBA Object Group Service – A Service Approach to Object Groups in CORBA"**, PhD. Thesis, École Polytechnique Fédérale de Lausanne, Lausanne, 1998.

[Fetzer95] C. Fetzer and F. Cristian, **"On the Possibility of Consensus in Asynchronous Systems"**, In Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems, Newport Beach, CA, Dec 1995.

[Fischer85] M. J. Fischer, N. A. Lynch, M. S. Paterson, **"Impossibility of Distributed Consensus with One Faulty Process"**, Journal of the ACM, 32(2):374-382, Apr 1985.

[Fraga97] J. Fraga, C. Maziero, Lau L. and O. Loques, **"Implementing Replicated Services in Open Systems Using a Reflective Approach"**, Proceedings of the 3th IEEE International Symposium on Autonomous Descentralized Systems - ISADS 97, Berlin - Germany, April 1997.

[Fraga01] J. Fraga, Lau L., C. M. Westphall, C. B. Montez, **"Suporte para Aplicações Críticas nas Especificações CORBA: Tolerância a Faltas, Segurança, Tempo Real"**, Minicurso do XIX Simpósio Brasileiro de Redes de Computadores - SBRC'2001, Capítulo 2, SBC, Florianópolis – SC. Maio de 2001.

[Fritzke98] U. Fritzke Jr., P. Ingels, A. Mostefaoui, M. Raynal, **"Fault-Tolerant Total Order Multicast to Asynchronous Group"**, In Proceedings of the 17th International

Symposium on Reliable Distributed Systems - SRDS'98, IEEE Computer Society, 1998.

[Gopal92] A. S. Gopal, **"Fault-Tolerant Broadcast and Multicast: The Problem of Inconsistency and Contamination"**, Ph.D. Thesis, Cornell University, Ithaca, NY, March 1992.

[Guerraoui97] R. Guerrauoi and A. Schiper, **"Genuine Atomic Multicast"**, Proc. of the 11th Int. Workshop on Distributed Algorithms (WDAG), Saarbrucken, LNCS 1320, pp. 141-154, 1997.

[Hadzilacos93] V. Hadzilacos and S. Toueg, **"Fault-Tolerant Broadcast and Related Problems"**, In Distributed Systems, ACM Press (S. Mullender Ed.), New York, 1993, pp 97-145.

[Hagsand92] O. Hagsand, H. Herzog, K. P. Birman e R. Cooper, **"Object-Oriented Reliable Distributed Programming"**, IEEE, 2nd International Workshop on Object-Orientation in Operational Systems, I-WOOOS/1992.

[Hayden96] M. Hayden and K. Birman, **"Probabilistic Broadcast"**, Technical Report TR96-1606, http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html, September, 1996.

[Huang93] Y. Huang and C. Kintala, **"Software Implemented Fault Tolerance"**, in Proc. of 22nd Fault-tolerance Computing Symposium, pp. 2-10, 1993.

[Hurfin99] M. Hurfin, R. Macêdo, M. Raynal, F. Tronel, **"A General Framework to Solve Agreement Problems"**, 18th IEEE Symp. on Reliable Distributed Systems - SRDS'99, Lausanne, Suiça, October 1999.

[IONA95] Isis Distributed Systems Inc, IONA Technologies, Ltd. **"Orbix+Isis Programmer's Guide"**, Document D070-00, 1995.

[IONA97] IONA Techonologies, Ltd. **"OrbixWeb Programmer's Guide"**, 1997. www.iona.com.

[ISIS93] Isis Distributed Systems Inc., Ithaca, N.Y, **"Object Groups: A Response to the ORB 2.0 RFI"**, April 1993.

[ISIS95] Isis Distributed Systems Inc, IONA Technologies, Ltd. **"Orbix+Isis Programmer's Guide"**, Document D070-00, 1995.

[Joshi97] R. K. Joshi at al., **"Message Filters for Object-Oriented Systems"**, Software-Practice and Experience, vol. 27(6), 677-699, 1997.

[Kaashoek91] M. Kaashoek and A. Tanenbaum, **"Group Communication in the Amoeba Distributed Operating System"**, In Proceedings of the 11th International Conference on Distributed Computing System, pages 222-230. IEEE, 1991.

[Kiczales91] G. Kiczales, J. des Rivières, D. G. Bobrow, **"The Art of the Metaobject Protocol"**, MIT Press, 1991.

[Killijian98] M. Killijian, J. Fabre, J. Ruiz-Garcia, S. Chiba, **"A Metaobject Protocol for Fault-Tolerant CORBA Application"**, Proceedings of the 17th IEEE International Symposium on Reliable Distributed Systems (SRDS'98), West Lafayette (USA), October 1998.

[Lamport78] L. Lamport, **"Time, Clock, and Ordering of Events in a Distributed System"**, Communication of ACM, 21,7, pp 558-565, July 1978.

[Lamport82] L. Lamport, R. Shostak, and M. Pease, **"The Byzantine Generals Problem"**, ACM Transactions on Programming Languages and Systems, 4(3):382-401, July 1982.

[Laprie90] J. C. Laprie, C. Beounes, K. Kanoun, **"Definition and Analysis of Hardware-and-Software Fault-Tolerant Architecture"**, PDCS Esprit Basic Research Action, 23, May, 1990.

[Laprie92] J. C. Laprie (ed), **"Dependability: Basic Concepts and Terminology "**, Vol. 5 of Dependable Computing and Fault-Tolerant Systems, Springer Verlag. Wien, New York, 1992, pp 23-28.

[Lau95] Lau L., J. Fraga, C. A. Maziero, **"Suporte de Grupo de Objetos em uma Plataforma CORBA"**, Workshop ASAP - Ambiente para Suporte de Aplicações Distribuídas Baseado em Objetos, Florianópolis – SC. Outubro de 1995.

[Lau96a] Lau L., J. Fraga, C. A. Maziero, O. L. Filho **"Serviços Replicados em um Plataforma Aberta CORBA Usando uma Abordagem Reflexiva"**, II Workshop de Sistemas Distribuídos - Projeto ASAP do XIV Simpósio Brasileiro de Redes de Computadores - SBRC 96. Fortaleza-CE, Maio de 1996.

[Lau96b] Lau L., J. Fraga, C. A. Maziero **"Grupos de Objetos em uma Plataforma CORBA"**, VII Semana de Informática de Maringá - II Congresso de Tecnologia, Telecomunicações e Informática - CONTTEIN 96. Maringá – PR Setembro de 1996.

[Lau96c] Lau. C. L., **"Implementação de Técnicas de Replicação de Componentes de Software Sobre a Plataforma Aberta CORBA"**, Dissertação de Mestrado, DPGELL – Universidade Federal de Santa Catarina, Florianópolis, Maio de 1996.

[Lau97] Lau C. L., J. Fraga, C. Maziero,**"Uma Abordagem Reflexiva Usando Suporte de Grupo para Implementar Técnicas de Replicação em Ambientes Heterogêneos"**, Anais do VII Simpósio de Computadores Tolerantes a Falhas SCTF 97 - SBC - Campina Grande, PB, Julho 1997.

[Lau99a] Lau L., J. Fraga, J. Farines, **"CosNamingFT – Um Serviço de Nomes Tolerante a Falhas em Conformidade com o Padrão OMG"**, XVII Simpósio Brasileiro de Redes de Computadores - SBRC'99, SBC, Salvador - BA. Maio de 1999.

[Lau99b] Lau L., J. Fraga, J. Farines, M. Ogg, A. Ricciardi, **"CosNamingFT – A Fault-Tolerant CORBA Naming Service"**, Proceeding of the 18th IEEE Symp. on Reliable Distributed Systems - SRDS'99, Lausanne, Suiça, October 1999.

[Lau99c] Lau L., J. Fraga, C. Maziero, **"MetaFT - A Reflective Approach to Implement Replication Techniques in CORBA"**, Proceeding of the 19th International Conference of the Chilean Computer Science Society - SCCC'99, IEEE Computer Society. Talca, Chile - November 1999.

[Lau00a] Lau L., J. Fraga, J. Farines, J. R. Oliveira, **"Experiências com Comunicação de Grupo nas Especificações Fault Tolerant CORBA"**, XVIII Simpósio Brasileiro de Redes de Computadores - SBRC'2000, SBC, Belo Horizonte – MG. Maio de 2000.

[Lau00b] Lau L., J. Fraga, **"Detecção de Falha para Redes de Larga Escala no Fault-Tolerant CORBA"**, II Workshop de Testes e Tolerância a Falhas - WTF'2000, SBC, Curitiba – PR. Julho de 2000.

[Lau00c] Lau L., J. Fraga, **"Tolerância a Faltas no CORBA: Um Histórico das Pesquisas até a sua Padronização na OMG"**, Minicurso do II Workshop de Testes e Tolerância a Falhas - WTF'2000, SBC, Curitiba – PR. Julho de 2000.

[Lau00d] Lau L., J. Fraga, L. Souza, R. Padilha, **"*GroupPac*: Um *Framework* para Implementação de Aplicações Tolerantes a Falhas"**, XXVI Conferencia LatinoAmericana de Informatica - CLEI'2000. Estado de México, México – Setembro de 2000.

[Lau01a] Lau L., J. Fraga, R. Padilha, L. Souza, **"Adaptando as Especificações FT-CORBA para Redes de Larga Escala"**, XIX Simpósio Brasileiro de Redes de Computadores - SBRC'2001, SBC, Florianópolis – SC. Maio de 2001.

[Lau01b] Lau L., R. Padilha, L. Souza, J. Fraga, **"GroupPac: Uma Ferramenta para Desenvolvimento de Aplicações Tolerante a Faltas no CORBA"**, Salão de Ferramentas do XIX Simpósio Brasileiro de Redes de Computadores - SBRC'2001, SBC, Florianópolis – SC. Maio de 2001, (http://www.lcmi.ufsc.br/grouppac).

[LCMI00] Laboratório de Controle e Microinformática, **"SALE: Uma Plataforma Configurável de Comunicação Distribuída para Suporte a Aplicações de Empresas Virtuais em Sistemas Abertos de Larga Escala"**. LCMI/DAS/UFSC, Proposta de projeto, 2000.

[Lea94] D. Lea, **"Object in Groups"**, SUNY at Oswego/NY CASE Center, ECOOP 94.

[Liang90] L. Liang , S. T. Chanson e G. W. Neufeld, **"Process Groups and Group Communications: Classifications and Requirements"**, IEEE Computer, pp 56-65, February 1990.

[Liang96] D. Liang, S. C. Chou, S. M. Yuan, **"*Phoinix:* A Fault-Tolerant Object Service in OMA"**. 1996.

[Liang98] D. Liang, C-L Fang, S. M. Yuan, **"A Fault-Tolerant Object Service on CORBA"**. Journal of Systems and Software, 1998.

[Lisbôa96] M. L. B. Lisbôa, C. M. F. Rubira e L. E. Buzato, **"Arquitetura Reflexiva para o Desenvolvimento de *Software* Tolerante a Falhas"**, Anais do XXIII Seminário Integrado de Software e Hardware - SEMISH 96, XVI Congresso da SBC, Recife-PE, Agosto 1996.

[Liskov87] B. Liskov, **"Implementation of Argus"**, Proceedings of the 11th ACM Symposium on Operating Systems Principles, November 1987.

[Liskov88] B. Liskov, **"Distributed Programming in Argus"**, Communications of the CACM, Mach 1988.

[Little91] M. C. Little, **"Object Replication in a Distributed System"**, PhD. Thesis, University of Newcastle upon Tyne Computing Laboratory, september 1991.

[Maes87] P. Maes, **"Concepts and Experiments in Computational Reflection"** , OOPSLA 87 Proceedings, pp. 147-156, October 1987.

[Maffeis95a] S. Maffeis, **"Run-Time Support for Object-Oriented Distributed Programming"**, Ph.D. Thesis University of Zurich. Zurich, 1995.

[Maffeis95b] S. Maffeis, **"Adding Group Communication and Fault-Tolerance to CORBA"**, Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies. Monterey, CA: The USENIX Association, June 1995.

[Maffeis96] S. Maffeis, **"A Fault-Tolerant CORBA Name Server"**. 15th IEEE Symposium on Reliable Distributed Systems, October, 1996.

[Mishra93] S. Mishra, L. Peterson, R. Schlichting, **"Consul: A Communication Substrate for Fault-Tolerant Distributed Programs"**, Distributed Systems Engineering Journal 1,2, dezembro de 1993.

[Melliar90] P. Melliar-Smith, L. Moser and V. Agrawala, **"Broadcast Protocols for Distributed System"**, IEEE Transactions on Parallel and Distributed Systems, 1(1):17-25, January 1990.

[Montez99a] C. Montez, **"Um Modelo de Programação e uma Abordagem de Escalonamento Adaptativo Usando RT-CORBA"**, Tese de Doutorado, DPGELL – Universidade Federal de Santa Catarina, Florianópolis, Novembro de 1999.

[Moser98] L. E. Moser, P. M. P. Melliar-Smith, P. Narasimhan, **"Consistent Object Replication in the Eternal System"**, Theory and Practice of Object Systems, 4(2): 81-92, 1998.

[Naras97] P. Narasimhan, L. E. Moser, P. M. Melliar-Smith, **"Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance"**. Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, junho de 1997.

[Oliveira99a] J. S. Oliveira, **"Suporte a Grupo sobre a Plataforma CORBA: Um Estudo Comparativo entre as Abordagens"**, Dissertação de Mestrado, DPGELL – Universidade Federal de Santa Catarina, Florianópolis, Março de 1999.

[Oliveira99b] J. S. Oliveira, Lau C. L., J. S. Fraga, **"Uma Análise Comparativa das Estratégias de Suporte a Grupo sobre o CORBA"**, XXV Conf. Latinoamericana de Informática – CLEI'99, Asunción – Paraguay, Set. 1999.

[OMG96] Object Management Group, **"The Common Object Request Broker 2.0/IIOP Specification"**, Revision 2.0, OMG Document 96-08-04, 1996.

[OMG97] Object Management Group, **"CORBAservices: Common Object Services Specification"**, OMG Document. March 1997. www.omg.org.

[OMG98a] Object Management Group, **"Fault-Tolerant CORBA Using Entity Redundancy"**, OMG Document RFP Document orbos/98-04-01, October, 1998 www.omg.org.

[OMG98b] Object Management Group, **"Portable Interceptor"**, OMG Document RFP Document orbos/98-05-05, May, 1998 www.omg.org.

[OMG99] OMG, **'Real-time CORBA - Joint Revised Submission'**, Object Management Group (OMG), Document orbos/99-02-12, Mar. 1999.

[OMG00a] Object Management Group, **"Fault-Tolerant CORBA Specification V1.0"**, OMG document: ptc/2000-04-04, April, 2000. www.omg.org.

[OMG00b] Object Management Group, **"Unreliable Multicast Inter-ORB Protocol RFP"**, OMG document: orbos/99-11-14, Feb, 2000. www.omg.org.

[OMG00d] Object Management Group, **"Security Service:v1.5"**, OMG Document Number 00-06-25, June 2000 ftp://ftp.omg.org/pub/docs/formal/00-06-25.pdf.

[OMG00e] Object Management Group, **"Security Domain Membership Management Service – Revised Submission"**, OMG Document Number orbos/00-02-04, 14 February 2000.

[Powell91] D. Powell, **"Delta-4 Architecture Guide"**, Esprit II P2252, Delta-4 Phase 3, August 1991.

[Renesse93] Robbert V. Renesse, **"A MUTS Tutorial"** Dept. of Computer Science of the Cornell University, Mar 1993.

[Renesse95] R. V. Renesse and K. P. Birman, **"Protocol Composition in Horus"**, Dept. of Computer Science of the Cornell University, Mar 1995.

[Ricciardi91] A. Ricciardi and K. Birman, **"Using Process Groups to Implement Failure Detection in Asynchronous Systems"**. In Tenth ACM Symposium on the Principles of Distributed Computing . August, 1991.

[Rodrigues96] L. Rodrigues, H. Fonseca and P. Veríssimo, **"Totally Ordered Multicast in Large-Scale Systems"**, In Proceedings of the 16th International Conference on Distributed Computing System, IEEE, 1996.

[Rodrigues98] L. Rodrigues, R. Guerraoui, A. Schiper, **"Scalable Atomic Multicast"**, In Proceedings of the International Conference on Computer Communications and Networks, IEEE Computer, 1998.

[Schneider90] F. B. Schneider, **"Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial"**, ACM Computing Survey, 22(4):299-319, Dec 1990.

[Shapiro86] M. Shapiro, **"Structure and Encapsulation in Distributed Systems: The Proxy Principle"**, 198-204 IEEE, 1986.

[Shrivastava92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs and D. T. Seaton, **"Fail-Controlled Computer Architecture for Distributed Systems"**, Technical Report 333, University of Newcastle upon Tyne, August 1992.

[Wangham01] M. S. Wangham, Lau C. L., C. M. Westphall e J. S. Fraga. **"Integrating SSL to the JaCoWeb Security Framework: Project and Implementation"** IM'2001 - IFIP/IEEE International Symposium on Integrated Network Management, Seattle, Washington, USA, 14-18 May 2001.

[Westphall00] Carla Merkle Westphall, Joni da S. Fraga, Michelle S. Wangham e Lau C. L., **"JaCoWeb Security - A CORBA Security Discretionary Prototype"**, CLEI Electronic Journal, v3, n2, Dez. 2000. http://www.dcc.uchile.cl/~rbaeza/clei/index.html