**Uma Infra-Estrutura para o Gerenciamento
de Aplicações CORBA Tolerantes a Falhas**

Marcos Aurélio Mendes de Moura

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Markus Endler**

Durante o desenvolvimento deste trabalho, o autor recebeu apoio financeiro do CNPq.

— São Paulo, fevereiro de 2001 —

Uma Infra-Estrutura para o Gerenciamento de Aplicações CORBA Tolerantes a Falhas

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Marcos Aurélio Mendes de Moura
e aprovada pela comissão julgadora.

São Paulo, 22 de fevereiro de 2001

Banca examinadora:

- Prof. Dr. Markus Endler (orientador) (IME-USP)
- Prof. Dr. Francisco Carlos da Rocha Reverbel (IME-USP)
- Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)

Aos meus pais, irmãos e irmãs e
à minha namorada, Meire.

Agradecimentos

Ao meu orientador, Prof. Dr. Markus Endler, pelo acompanhamento e dedicação durante todo o desenvolvimento deste trabalho e por me abrir os caminhos do academicismo.

Ao Prof. Dr. Francisco Carlos da Rocha Reverbel, pelos comentários valiosos desde a concepção inicial do trabalho até a conclusão do mesmo.

Ao Prof. Dr. Luiz Eduardo Buzato, pelos comentários que ajudaram na escrita da versão final da dissertação.

À minha família, pelo apoio incondicional e pelo incentivo durante toda esta jornada. Além disso, pela compreensão, confiança e auxílio a qualquer instante.

À minha namorada, Meire, pelo amor, compreensão e carinho em todos os momentos. Por estar sempre ao meu lado, me dando muita força e trazendo muita tranquilidade, calma e serenidade mesmo nos momentos mais difíceis.

Aos amigos e amigas da minha cidade natal e espalhados mundo afora, que apesar da distância estiveram sempre comigo, trazendo muita segurança e apoio.

À turma do mestrado, Ricardo, Said, Uirá, Lucy, Lorena, Marcelo Brito, Maria do Carmo, Ariane e Francisco, pela amizade, companheirismo e pelos belos momentos de descontração. Em especial ao Ricardo, pela amizade, pelas discussões sobre o trabalho e ajuda na escrita da dissertação e ao Marcelo Brito, pela ajuda em algumas etapas do trabalho.

À turma do pensionato, Dannylo, Banik, Paulão, João, Cornélio, André, Vandrê, Fabrício, Adelmo, por terem me suportado durante toda a duração do mestrado e pelas grandes alegrias proporcionadas nos momentos de lazer. Em especial ao Dannylo, pela amizade, pelas inúmeras revisões da dissertação e pela ajuda contínua ao longo deste trabalho.

Ao Sr. João Claudino, pela confiança e pelo apoio irrestrito desde a decisão inicial de realizar este trabalho até a sua conclusão.

Marcos Aurélio Mendes de Moura

Resumo

Nesta dissertação é apresentado um serviço que fornece uma infra-estrutura robusta para o gerenciamento de aplicações CORBA tolerantes a falhas, denominado Juggler. Esse serviço disponibiliza interfaces que permitem o gerenciamento da configuração de aplicações CORBA tolerantes a falhas baseadas na redundância dos objetos que as compõem (grupos de objetos) e em mecanismos para a detecção e a recuperação de falhas desses objetos.

Através do Juggler, é possível definir o estilo de replicação e as propriedades de tolerância a falhas associados a cada grupo de objetos individualmente. Essas propriedades definem, por exemplo, a localização e a quantidade mínima de réplicas que compõem um grupo, e permitem escolher a forma mais adequada para a recuperação de falhas ocorridas nos membros de um grupo. Além disso, o Juggler permite a modificação dinâmica das propriedades de tolerância a falhas e do estilo de replicação associados a grupos de objetos.

O Juggler foi desenvolvido tomando-se como base o OGS, um serviço que disponibiliza os mecanismos básicos para a criação de grupos de objetos CORBA. O OGS foi escolhido como a infra-estrutura para o desenvolvimento do Juggler por definir uma arquitetura para Comunicação de Grupo que utiliza apenas estruturas padronizadas pela especificação CORBA, possibilitando a portabilidade e a interoperabilidade desse serviço em diferentes CORBA ORBs.

Abstract

In this thesis we describe a service, called Juggler, that provides a robust infrastructure for the management of fault-tolerant CORBA applications. This service provides interfaces for configuring fault-tolerant CORBA applications based on object replication (object groups) and mechanisms for the detection of and recovery from object faults.

Juggler allows to define the replication style and other fault tolerance properties for each object group. These properties determine, for instance, the locations and the minimum number of replicas that compose a group, and allow to choose the best approach to recover from faults of group members. Moreover, Juggler allows dynamic modification of the fault-tolerance properties and the replication style associated with object groups.

Juggler was developed using OGS, a service that provides the basic mechanisms to create CORBA object groups. OGS was chosen as the infrastructure for the development of Juggler mainly because its Group Communication architecture uses only CORBA standard constructs, which makes OGS portable and interoperable between different CORBA ORBs.

Sumário

1	Introdução	1
2	Tolerância a Falhas e CORBA	5
2.1	Comunicação de Grupo e Tolerância a Falhas	5
2.2	CORBA	6
2.3	Grupos de Objetos CORBA	8
2.3.1	Abordagem de Integração	9
2.3.2	Abordagem de Interceptação	10
2.3.3	Abordagem de Serviço	12
3	OGS – Serviço de Grupos de Objetos	14
3.1	Serviço de Grupo	14
3.1.1	Pertinência a Grupo	15
3.1.2	Multicast para Grupo	16
3.2	Serviço de Consenso	18
3.3	Serviço de Monitoramento	18
3.4	Serviço de Mensagem	19
3.5	Implementação e Uso	19
4	Trabalhos Relacionados	21
4.1	Piranha	21
4.1.1	Arquitetura	22
4.1.2	Critérios para Ativação de Grupos	23
4.1.3	Protocolos	23
4.2	Proteus	23

4.2.1	Ambiente de Execução de QuO	25
4.2.2	Gerenciador de Dependência	25
4.2.3	Fábrica de Objetos	27
4.2.4	Gateway	27
4.3	FT-CORBA	27
4.3.1	Domínios de Tolerância a Falhas	29
4.3.2	Endereçamento de Grupos de Objetos	30
4.3.3	Gerenciamento de Replicação	31
4.3.4	Gerenciamento de Falhas	31
4.3.5	Recuperação de Falhas e Logging	32
4.4	DOORS	33
4.4.1	Adequação ao Padrão FT-CORBA	33
5	Juggler	35
5.1	Arquitetura do Serviço	36
5.2	Modelo de Sistema	37
5.3	Especificação do Serviço	37
5.3.1	Estrutura Semantics	38
5.3.2	Estruturas GroupInfo, LocationInfo e FactoryInfo	39
5.3.3	Interface GroupManagementOperations	40
5.3.4	Interface GenericFactory	43
5.3.5	Interfaces ReplicaMonitoring e FaultNotifier	44
5.3.6	Interface ReplicationManager	46
6	Implementação	48
6.1	Replicação das Informações Gerenciadas pelo Juggler	48
6.2	Inicialização do Juggler	49
6.3	Interfaces Internas do Serviço	50
6.4	Criação de Grupos de Objetos	50
6.5	Recuperação de Falhas em Réplicas de Objetos	52
6.6	Modificação da Semântica de Grupos	55
6.7	Aspectos de Tolerância a Falhas no Juggler	55

7	Extensão do OGS	57
7.1	Motivação para a Extensão do Serviço	57
7.2	Modificação das Interfaces do OGS	58
7.2.1	Módulo mGroupAdmin	59
7.2.2	Módulo mGroupAccess	59
7.2.3	Módulo mGroupProtocol	62
7.3	Novos Estilos de Replicação	64
7.3.1	Replicação STATELESS	64
7.3.2	Replicação COLD_PASSIVE e WARM_PASSIVE	65
7.3.3	Replicação VIRTUAL_PRIMARY_COPY	66
7.4	Modificação Dinâmica do Estilo de Replicação	66
8	Testes	69
8.1	Criação de Grupos de Objetos	70
8.2	Adição e Remoção de Membros de um Grupo	72
8.3	Recuperação Automática de Falhas em Réplicas de Objetos	74
8.4	Multicasts para Grupos de Objetos	75
8.5	Modificação do Estilo de Replicação	76
8.6	Conclusão sobre os Testes	77
9	Conclusão	79
	Referências Bibliográficas	82

Lista de Figuras

2.1	Arquitetura de um CORBA ORB	7
2.2	Abordagem de integração	10
2.3	Abordagem de interceptação	11
2.4	Abordagem de serviço	12
3.1	Arquitetura do OGS	15
3.2	Serviço de Grupo do OGS	17
4.1	O Piranha na arquitetura do Electra	22
4.2	Arquitetura do Proteus	25
4.3	Arquitetura do FT-CORBA	28
4.4	Estrutura de uma Referência Interoperável para Grupo de Objetos (IOGR).	30
5.1	Arquitetura do Juggler	36
5.2	Arquitetura lógica do Juggler	36
5.3	Estrutura Semantics	38
5.4	Estruturas GroupInfo, LocationInfo e FactoryInfo	40
5.5	Interface GroupManagementOperations (parte 1)	41
5.6	Interface GroupManagementOperations (parte 2)	42
5.7	Interface GenericFactory	43
5.8	Interfaces ReplicaMonitoring e FaultNotifier	45
5.9	Propagação de eventos através do FaultNotifier	46
5.10	Interfaces ReplicationManager, LocationOperations e FactoryOperations	47
6.1	Interfaces GroupManager, GroupActor e GroupObserver	51
6.2	Protocolo para criação de um grupo de objetos	53

6.3	Protocolo para recuperação de falhas em réplicas de objetos	54
7.1	Extensão do módulo mGroupAdmin	60
7.2	Extensão do módulo mGroupAccess	61
7.3	Extensão do módulo mGroupProtocol	63
7.4	Estilo de replicação WARM_PASSIVE	65
7.5	Modificação dinâmica do estilo de replicação de um grupo de objetos	67
8.1	Interface BankAccount	70
8.2	Tempo gasto para a criação de grupos de objetos com o Juggler	71
8.3	Custo associado a cada etapa da criação de grupos de objetos	72
8.4	Tempo gasto para a adição e a remoção de membros de grupos de objetos . .	73
8.5	Tempo gasto para a recuperação automática da falha em uma réplica de objeto	74
8.6	Tempo gasto para a invocação de um multicast atômico	76

Capítulo 1

Introdução

Em muitas áreas, os computadores são utilizados para a execução de aplicações críticas, como por exemplo, o controle de tráfego aéreo, direcionamento de mísseis, monitoramento dos sinais vitais de um paciente e controle de usinas nucleares. Em sistemas onde são executadas aplicações dessa natureza, a ocorrência de alguns tipos de falhas ou a indisponibilidade temporária do sistema podem representar uma grande ameaça para a vida humana ou até mesmo causar uma catástrofe.

Nesse contexto, torna-se imprescindível a utilização de ambientes ou sistemas que garantam diversos tipos e graus de qualidade de serviço (QoS) para as aplicações. Alguns dos tipos mais comuns de QoS incluem confiabilidade e disponibilidade, os quais são particularmente importantes para tolerância a falhas [24], principal tema dessa dissertação.

O desenvolvimento de sistemas computacionais capazes de lidar com tarefas críticas tornou-se mais comum em meados da década de 80, com o surgimento de ambientes específicos para programação distribuída. Entre esses ambientes, há várias diferenças nos níveis de abstração e nos mecanismos utilizados para a programação de aplicações distribuídas, impossibilitando a integração e a cooperação entre aplicações.

Além disso, nesses ambientes há normalmente uma dependência da plataforma e da linguagem de programação utilizadas para o desenvolvimento de aplicações distribuídas. Isso dificulta a reutilização de código entre diferentes plataformas e exige que o programador de aplicação utilize uma linguagem de programação especialmente projetada para a criação de aplicações distribuídas. Dois dos ambientes mais conhecidos para programação distribuída desenvolvidos na década de 80 são o Isis [3] e o DCE (*Distributed Computing Environment*) [15].

No início dos anos 90, no entanto, começaram a surgir as primeiras arquiteturas de sistemas distribuídos que permitem o desenvolvimento de sistemas interoperáveis, como algumas arquiteturas de software (*middleware*) que oferecem suporte para programação distribuída orientada a objeto em ambientes abertos. Dentre as arquiteturas propostas, CORBA (*Common Object Request Broker Architecture*) [18] é provavelmente a mais conhecida e largamente utilizada.

A arquitetura CORBA, especificada pelo OMG¹ (*Object Management Group*), fornece uma infra-estrutura aberta, distribuída e orientada a objeto que possibilita a comunicação entre objetos, mesmo que eles sejam implementados em diferentes linguagens de programação e plataformas. CORBA também enfatiza outros aspectos relevantes para o desenvolvimento de aplicações distribuídas, como a integração de sistemas e a dissociação entre a funcionalidade provida por um serviço e a forma como ele é implementado. No entanto, até muito recentemente, não havia nenhuma especificação em CORBA que padronizasse mecanismos e interfaces para o desenvolvimento e o gerenciamento de aplicações distribuídas tolerantes a falhas.

Ao longo das últimas duas décadas, vários projetos de pesquisa na área de Sistemas Distribuídos foram desenvolvidos com o objetivo de permitir a criação de aplicações distribuídas que exigem um alto grau de confiabilidade e disponibilidade [2, 3, 22, 32, 40]. A maioria dos modelos descritos nestes trabalhos é baseada na replicação de processos e implementa primitivas para *Comunicação de Grupo*. Em geral, os modelos propostos possuem mecanismos que permitem a ordenação consistente de eventos distribuídos, garantindo assim o comportamento previsível das aplicações distribuídas. Além disso, são fornecidos mecanismos para detecção e recuperação de falhas parciais e temporárias no sistema. Assim, Comunicação de Grupo constitui uma técnica poderosa para o desenvolvimento de aplicações distribuídas tolerantes a falhas [4, 6].

Recentemente, vários esforços têm sido feitos para possibilitar a incorporação dos mecanismos de Comunicação de Grupo em CORBA [14, 25, 33]. Nesse contexto, as entidades replicadas são objetos CORBA, os quais constituem *grupos de objetos*. Os sistemas que incorporam Comunicação de Grupo em CORBA fornecem, normalmente, apenas as funcionalidades básicas para o desenvolvimento de aplicações tolerantes a falhas, tais como primitivas para Comunicação de Grupo confiável e ordenada, mecanismos para detecção de falhas em réplicas de objetos e protocolos para mudança de visão e transferência de estado entre os membros de um grupo de objetos.

Os sistemas que incorporam Comunicação de Grupo em CORBA fornecem uma interface de baixo nível para o gerenciamento de aplicações distribuídas, forçando o *administrador de aplicações* a lidar diretamente com as funcionalidades providas pelo sistema de Comunicação de Grupo. Nesses sistemas, o administrador de aplicações fica encarregado, por exemplo, do monitoramento de réplicas de objetos e da recuperação de falhas nas mesmas. Além disso, ele também fica responsável por adequar as aplicações às mudanças que ocorrem no ambiente computacional. Essas mudanças incluem a adição e a remoção de máquinas (*hosts*) no sistema e o gerenciamento do uso de recursos, como por exemplo, a memória dos *hosts* e a largura de banda disponível.

A fim de padronizar funcionalidades para o suporte a aplicações CORBA tolerantes a falhas e facilitar o desenvolvimento e gerenciamento dessas aplicações, o OMG elaborou, em abril de 1998, a RFP (*Request For Proposal*) intitulada “*Fault Tolerant CORBA Using*

¹ Consórcio formado por mais de 1000 empresas cuja missão é possibilitar a criação de software baseado em componentes através da introdução do conceito de software padronizado. O site do OMG na internet pode ser acessado através da URL <http://www.omg.org/>

Entity Redundancy” [17]. Atualmente, o processo de adoção de tecnologia para essa RFP está em fase de finalização e o OMG já disponibilizou a primeira versão da especificação do padrão para tolerância a falhas em CORBA, denominado FT-CORBA [21].

O padrão para tolerância a falhas em CORBA define um conjunto de interfaces, políticas e serviços que fornecem suporte para aplicações que exigem alta confiabilidade, principalmente aquelas nas quais não pode haver um único ponto de falha. O padrão é baseado no uso de objetos replicados (grupos de objetos) e descreve uma série de estratégias de tolerância a falhas e mecanismos para detecção, notificação e análise de falhas nas réplicas de objetos que compõem os grupos. Além disso, o padrão proposto pelo OMG também fornece suporte para aplicações que exigem recuperação automática de falhas e mecanismos que permitem à infra-estrutura de tolerância a falhas controlar a consistência dos dados armazenados nas réplicas de objetos.

Os principais objetivos dessa dissertação são a especificação e a implementação de uma infra-estrutura robusta para o gerenciamento de aplicações CORBA tolerantes a falhas semelhante àquela proposta pelo OMG. O serviço desenvolvido no contexto dessa dissertação, denominado Juggler, disponibiliza interfaces que permitem uma grande flexibilidade no gerenciamento de configuração de aplicações tolerantes a falhas. Esse serviço permite a criação e o gerenciamento de aplicações CORBA tolerantes a falhas através de redundância dos objetos que as compõem e de mecanismos para detecção e recuperação de falhas destes objetos.

O serviço Juggler possibilita a utilização de várias estratégias de tolerância a falhas que definem o comportamento de um grupo de objetos, como por exemplo, diferentes estilos de replicação e a especificação de propriedades que determinam, entre outras coisas, a localização e a quantidade de réplicas de objeto que compõem um grupo. Através do Juggler, também é possível modificar dinamicamente o estilo de replicação e outras propriedades associadas a grupos de objetos. Isso permite que o administrador de aplicações tolerantes a falhas, ou até mesmo o próprio Juggler, modifique a configuração de grupos de objetos de aplicação a fim de melhor adaptá-los às condições do ambiente computacional.

Da mesma forma como proposto na especificação do padrão FT-CORBA, o serviço Juggler disponibiliza uma interface de alto nível para o gerenciamento de aplicações tolerantes a falhas, livrando o administrador dessas aplicações de tarefas como o monitoramento e a recuperação de réplicas de objetos. Além disso, as interfaces de gerenciamento fornecidas pelo Juggler evitam que o administrador de aplicações tolerantes a falhas tenha que lidar com os detalhes inerentes ao sistema de comunicação utilizado para a manutenção de grupos de objetos.

Esta dissertação está organizada da seguinte forma. O capítulo 2 define a terminologia adotada ao longo dessa dissertação e descreve as principais abordagens para a incorporação de mecanismos de Comunicação de Grupo em CORBA. O capítulo 3 descreve o OGS, um serviço que permite a criação de grupos de objetos CORBA e que foi utilizado para a implementação da infra-estrutura de tolerância a falhas proposta nessa dissertação. No capítulo 4, são descritos alguns dos trabalhos relacionados diretamente com o tema dessa dissertação.

Os capítulos 5 e 6 descrevem, respectivamente, as principais interfaces e a implementação do serviço Juggler. Nos capítulos 7 e 8 são descritas algumas extensões feitas no OGS e os resultados dos testes realizados com o primeiro protótipo do Juggler. Por fim, no capítulo 9 são apresentadas algumas considerações finais e possíveis trabalhos futuros.

Capítulo 2

Tolerância a Falhas e CORBA

Nos últimos quinze anos, a área de Tolerância a Falhas em computação distribuída tem crescido bastante, principalmente devido à necessidade de sistemas confiáveis para a execução de aplicações críticas, tais como o controle de tráfego aéreo e o controle de usinas nucleares. Uma abordagem muito explorada para o desenvolvimento de aplicações tolerantes a falhas é a utilização de mecanismos de Comunicação de Grupo [2, 3, 22, 32, 40].

Apesar da crescente necessidade por sistemas confiáveis, até o início deste ano, não havia um padrão que definisse mecanismos e interfaces específicos para a criação de aplicações tolerantes a falhas em CORBA. No entanto, desde 1994, alguns projetos de pesquisa têm focado esforços na incorporação de mecanismos de Comunicação de Grupo em CORBA, permitindo assim a criação de aplicações tolerantes a falhas nessa arquitetura. Nesse capítulo, são descritas as diferentes abordagens adotadas nesses trabalhos. Além disso, são definidos alguns termos e conceitos utilizados na área de Tolerância a Falhas e nos sistemas que possibilitam Comunicação de Grupo e é apresentada uma visão geral da arquitetura CORBA.

2.1 Comunicação de Grupo e Tolerância a Falhas

A idéia de estruturar um sistema distribuído como um conjunto de processos cooperantes (grupos de processos) foi originalmente concebida no sistema V [9]. Desde então, vários sistemas que fornecem primitivas para comunicação com grupos de processos foram desenvolvidos, como o Isis [3], Transis [2], Totem [32], Horus [40] e Ensemble [22].

Esses sistemas, também denominados Sistemas de Comunicação de Grupo, fornecem protocolos de *multicast*, os quais permitem que um processo envie uma mensagem para um sub-conjunto dos processos existentes no sistema. Uma mensagem enviada através de protocolos de *multicast* é comumente denominada *mensagem multicast*, ou simplesmente *multicast*. Os sistemas que fornecem suporte para Comunicação de Grupo normalmente implementam protocolos que garantem variados graus de ordenação e confiabilidade para a entrega de mensagens multicast.

Os Sistemas de Comunicação de Grupo usualmente garantem a entrega de mensagens multicast com ordenação FIFO, causal ou total [5]. Além disso, alguns desses sistemas garantem a entrega consistente de multicasts mesmo quando processos entram ou saem dos grupos dinamicamente. Para garantir este grau de confiabilidade e dinamismo, os Sistemas de Comunicação de Grupo implementam mecanismos que permitem a detecção e a recuperação de falhas parciais e temporárias no sistema, como por exemplo a falha de um processo que pertence a um grupo. Devido a todas essas garantias de confiabilidade e à sua flexibilidade, os mecanismos de Comunicação de Grupo são considerados uma técnica poderosa para o desenvolvimento de aplicações distribuídas tolerantes a falhas [4, 6].

Um *sistema de computação distribuída*, freqüentemente denominado na literatura simplesmente como *sistema*, é um conjunto de programas que são executados em um ou mais computadores e que coordenam suas ações através de troca de mensagens [5]. No âmbito desta dissertação há um interesse particular por sistemas confiáveis, principalmente aqueles com requisitos de tolerância a falhas e alta disponibilidade. Antes de uma definição mais precisa desses termos, no entanto, é necessária a compreensão de alguns outros conceitos como *falta*, *erro* e *falha*. A terminologia adotada neste trabalho segue as definições descritas em [26], [10] e [24].

A *falta* (*failure*) de um sistema ocorre quando o seu comportamento desvia da sua especificação, ou seja, quando o sistema não fornece o serviço para o qual foi projetado. Um *erro* é uma parte do estado do sistema, normalmente inconsistente e indesejável, que pode levar à subsequente falta do sistema. A causa de um erro, e indiretamente da falta de um sistema, é uma *falha*. Em geral, uma falha é associada à noção de defeito e pode ser definida como um defeito que tem o potencial de gerar erros.

Um sistema é dito tolerante a falhas se ele for capaz de mascarar a presença de falhas nos componentes que o constituem. O objetivo de tolerância a falhas é, portanto, evitar a falta de um sistema mesmo quando ocorrem falhas em alguns dos seus componentes. O aspecto-chave para o desenvolvimento de sistemas tolerantes a falhas é a redundância dos componentes que o constituem. As propriedades mais importantes para tolerância a falhas são a confiabilidade e a disponibilidade, as quais podem ser compreendidas, respectivamente, como a garantia de que um sistema é capaz de prover sua funcionalidade e como a prontidão para uso do sistema [24].

2.2 CORBA

Nos últimos dez anos, o OMG vem promovendo a adoção de diversos padrões para o desenvolvimento de aplicações baseadas em objetos distribuídos. Um dos principais resultados alcançados é a especificação CORBA (*Common Object Request Broker Architecture*) [18], uma arquitetura aberta e orientada a objeto que possibilita a comunicação entre objetos distribuídos.

No contexto de CORBA, um objeto é uma entidade identificável (através de uma referência para objeto) que fornece um ou mais serviços para um cliente. Um cliente, por sua

vez, é uma entidade que requisita um serviço disponibilizado por um objeto. Um objeto CORBA que disponibiliza seus serviços para um cliente também é denominado implementação de objeto.

Objetos CORBA são especificados através de uma linguagem para definição de interfaces, denominada OMG IDL. Esta linguagem é puramente declarativa, permitindo uma separação nítida entre a interface que define um serviço e a forma como ele é implementado. A partir da definição de um objeto em OMG IDL, o compilador de IDL gera *stubs* de comunicação que permitem uma aplicação enviar requisições para este objeto, independente de ele ser um objeto local ou remoto.

CORBA também define como as construções especificadas em OMG IDL são mapeadas para diversas linguagens de programação, tais como C, C++, Java e Smalltalk. Através da OMG IDL, CORBA permite que uma aplicação cliente acesse objetos de maneira uniforme, independente das linguagens de programação utilizadas para a implementação dos mesmos. Os principais componentes da arquitetura CORBA são ilustrados na figura 2.1 e descritos a seguir.

Figura 2.1: Arquitetura de um CORBA ORB

O Núcleo do ORB (*ORB Core*) permite a invocação transparente e confiável de objetos remotos em um ambiente distribuído. O ORB pode ser visto como um barramento de objetos, através do qual objetos heterogêneos podem interoperar. O ORB é responsável por encontrar a implementação de objeto a quem se destina uma requisição, transmitir para ela a requisição enviada por um cliente e, possivelmente, retornar para o cliente uma resposta relativa à requisição.

Os *Stubs* IDL, também conhecidos como Interface de Invocação Estática (SII), fornecem uma Interface para o Programador de Aplicação (API) de comunicação específica para cada tipo de objeto que um cliente pode acessar. Sob a perspectiva do cliente, um stub é um *proxy* local para um objeto remoto. Os Esqueletos IDL Estáticos fornecem interfaces estáticas para implementações de objeto. Eles armazenam o código necessário para a invocação de métodos nas implementações de objeto. Tanto os *Stubs* IDL quanto os Esqueletos IDL Estáticos são gerados pelo compilador de IDL a partir da interface de um objeto definida em OMG IDL.

A Interface de Invocação Dinâmica (DII) e a Interface de Esqueleto Dinâmico (DSI) permitem a invocação dinâmica de métodos em objetos remotos, ou seja, permitem que um cliente invoque operações definidas em interfaces desconhecidas em tempo de compilação. A Interface ORB disponibiliza uma API para serviços do ORB que podem ser acessados localmente pelos clientes e pelas implementações de objetos. O Adaptador de Objetos fornece operações que permitem a criação de referência para objetos, bem como o registro e a ativação de implementações de objetos.

CORBA também define um Repositório de Interfaces e um Repositório de Implementações, os quais não estão ilustrados na figura 2.1. O Repositório de Interfaces armazena definições de interfaces e permite a um cliente obter e modificar as descrições dessas interfaces. As informações contidas no Repositório de Interfaces são utilizadas, pela DII e pela DSI, para a criação e a invocação dinâmica de requisições. O Repositório de Implementações, por sua vez, armazena informações que permitem ao ORB localizar e ativar implementações de objetos.

2.3 Grupos de Objetos CORBA

A incorporação de mecanismos de Comunicação de Grupo em CORBA é uma maneira natural de fornecer tolerância a falhas para as aplicações desenvolvidas nessa arquitetura [13]. Os sistemas que oferecem esse tipo de suporte podem ser classificados pela maneira como os mecanismos de Comunicação de Grupo são incorporados em CORBA [14]. Assim, é possível distinguir três abordagens principais: integração, interceptação e serviço. No restante dessa seção, é apresentada uma descrição conceitual de cada uma dessas abordagens e são mencionadas algumas das principais vantagens e desvantagens decorrentes da utilização das mesmas.

A terminologia adotada nesta dissertação, no contexto de grupos de objetos, é baseada nas definições descritas em [13]. Assim, um *grupo de objetos* funciona como uma entidade lógica endereçável que é constituída por um conjunto de objetos logicamente relacionados. Os objetos que constituem um grupo não precisam, necessariamente, ser cópias de um mesmo objeto replicado.

Um *objeto replicado* O , por sua vez, é um conjunto de objetos O_i que agem como cópias do mesmo objeto O . No entanto, como é comum na literatura, os objetos que constituem um grupo serão chamados de *réplicas de objeto*, ou simplesmente *réplicas*, mesmo que eles não sejam cópias de um mesmo objeto. Além disso, para uma entidade (cliente) que requisita

um serviço de um grupo, este se comporta como um único objeto, ou seja, há transparência quanto à localização e à quantidade de objetos que constituem o grupo.

2.3.1 Abordagem de Integração

A principal idéia da abordagem de integração [25] é a modificação do ORB para que ele possa tratar diferenciadamente referências para objetos simples (*singleton objects*) e referências para grupos de objetos. Além disso, todas as operações usadas para gerenciar grupos de objetos ficam embutidas no próprio ORB, permitindo que qualquer objeto no sistema possa se tornar membro de um grupo de objetos.

Nos sistemas que implementam essa abordagem, tais como o Electra [30] e o Orbix+Isis [23], toda requisição CORBA destinada a um grupo de objetos é repassada para o Sistema de Comunicação de Grupo em uso. Este se encarrega, então, de difundir a requisição, através de *multicast*, para as réplicas de objeto (implementações de objeto replicadas).

Durante a execução de uma requisição destinada a um grupo de objetos segundo a abordagem de integração, como ilustrado na figura 2.2, a seguinte seqüência de eventos pode ser identificada:

1. A requisição é enviada pelo cliente até o ORB, o qual reconhece que ela é endereçada a um grupo de objetos;
2. O ORB repassa a requisição para o Sistema de Comunicação de Grupo;
3. O Sistema de Comunicação de Grupo difunde a requisição, através de *multicast* confiável e ordenado, para as implementações de objeto destinatárias;
4. O ORB invoca a operação requisitada em cada uma das implementações de objeto;
5. As respostas produzidas pelas implementações de objeto são retornadas, através do ORB, para o Sistema de Comunicação de Grupo;
6. O Sistema de Comunicação de Grupo coleta as respostas e as envia, através do ORB, para o cliente.

Normalmente, uma única resposta é retornada para o cliente, tornando a existência do grupo transparente para o mesmo. Caso seja necessário, também é possível que o cliente receba todas as respostas produzidas pelas implementações de objeto. Para tanto, o compilador de IDL é estendido de forma a gerar dois tipos de operações: *padrão* e *especial*. Uma *operação padrão* é aquela que obedece as regras especificadas em CORBA para o mapeamento das linguagens de programação, tornando transparente o uso de grupos de objetos pelo cliente. Assim, o cliente recebe apenas uma das respostas retornadas pelas implementações de objeto. Por outro lado, uma *operação especial* permite que um cliente faça requisições não transparentes para grupos de objetos, possibilitando o retorno de mais de uma resposta para o cliente.

Figura 2.2: Abordagem de integração

As principais vantagens dessa abordagem são a transparência de replicação para o cliente e o alto desempenho nas comunicações de grupo [30]. No entanto, devido à necessidade de modificação do ORB, os sistemas que implementam essa abordagem apresentam problemas de interoperabilidade com outros CORBA ORBs. Por isso, as aplicações que utilizam o suporte para tolerância a falhas fornecido por esses sistemas são inerentemente não portáteis entre diferentes ORBs.

2.3.2 Abordagem de Interceptação

Na abordagem de interceptação [33], todos os mecanismos de comunicação de grupo ficam transparentes para o ORB. A principal idéia dessa abordagem é interceptar as requisições IIOP (*Internet Inter-ORB Protocol*) geradas pelo ORB e difundí-las para as implementações de objeto replicadas, através de uma primitiva para envio de *multicast* fornecida pelo Sistema de Comunicação de Grupo em uso.

Como ilustrado na figura 2.3, em uma requisição destinada a um grupo de objetos, segundo a abordagem de interceptação, a seguinte sequência de eventos pode ser identificada:

1. A requisição é enviada pelo cliente até o ORB;
2. O interceptor, no lado do cliente, mapeia a requisição IIOP para o formato de mensagem utilizado pelo Sistema de Comunicação de Grupo. Em seguida, a requisição é repassada para o Sistema de Comunicação de Grupo;
3. O Sistema de Comunicação de Grupo difunde a requisição, através de *multicast* confiável e ordenado, para as implementações de objeto destinatárias;
4. Em cada uma das localidades onde se encontram as implementações de objeto, um interceptor mapeia a requisição para o formato utilizado pelo protocolo IIOP. Em seguida, o ORB invoca a operação requisitada em cada uma das implementações de objeto;

5. As respostas produzidas pelas implementações de objeto são retornadas, através do ORB, para o Sistema de Comunicação de Grupo. No caminho entre o ORB e o Sistema de Comunicação de Grupo, a mensagem é interceptada e mapeada para o formato de mensagem utilizado pelo Sistema de Comunicação de Grupo;
6. O Sistema de Comunicação de Grupo coleta as respostas e as envia, através do ORB, para o cliente. No caminho entre o Sistema de Comunicação de Grupo e o ORB, a mensagem é novamente interceptada e mapeada para o formato de mensagem utilizado pelo protocolo IIOP.

Figura 2.3: Abordagem de interceptação

A interceptação de mensagens pode ser realizada através da utilização de protocolos de meta-objeto, implementados com o auxílio de reflexão computacional [29], interfaces específicas do sistema operacional [34] ou interceptores próprios da arquitetura CORBA (*Interceptors*) [18]. O Eternal [34] e a arquitetura AQuA [11] são exemplos de sistemas que implementam a abordagem de interceptação.

Essa abordagem tem a vantagem de fornecer suporte para replicação de uma forma transparente para as implementações de objetos, permitindo que aplicações CORBA já existentes se beneficiem do suporte para tolerância a falha fornecido. Outra vantagem é a interoperabilidade, pois um sistema que adota essa abordagem pode interceptar apenas as requisições destinadas a grupos de objetos, garantindo que as requisições para objetos simples (*singleton objects*) sejam tratadas pelo ORB da forma usual.

Uma desvantagem da abordagem de interceptação é a possível perda de portabilidade, pois, em alguns casos, mecanismos específicos do sistema operacional são explorados. Além disso, de forma similar ao que ocorre na abordagem de integração, os sistemas que adotam essa abordagem são altamente dependentes do Sistema de Comunicação de Grupo utilizado.

2.3.3 Abordagem de Serviço

Na abordagem de serviço [14], os mecanismos de Comunicação de Grupo são fornecidos de forma semelhante a um serviço CORBA [16], o qual executa sobre o ORB, mas não faz parte desse propriamente dito. Assim, o ORB não possui conhecimento sobre o uso de grupos de objetos, mas, para as aplicações cliente, o uso do serviço de grupo torna-se explícito. No entanto, um sistema que implementa essa abordagem pode tornar transparente a utilização de grupos de objetos para os clientes, fazendo uso da Interface de Invocação Dinâmica (DII) especificada em CORBA.

Durante a execução de uma requisição destinada a um grupo de objetos segundo a abordagem de serviço, como ilustrado na figura 2.4, a seguinte seqüência de eventos pode ser identificada:

1. O cliente contacta, através do ORB, o Serviço de Comunicação de Grupo, indicando o grupo com o qual deseja se comunicar e a requisição a ser enviada para o mesmo;
2. O Serviço de Comunicação de Grupo invoca, através do ORB, a operação requisitada em cada uma das implementações de objeto destinatárias. Essa requisição é difundida através de uma primitiva para o envio de *multicast* confiável e ordenado, implementada pelo Serviço de Comunicação de Grupo;
3. As respostas produzidas pelas implementações de objeto são retornadas, através do ORB, para o Serviço de Comunicação de Grupo;
4. O Serviço de Comunicação de Grupo coleta as respostas e as envia, através do ORB, para o cliente.

Figura 2.4: Abordagem de serviço

Normalmente, na abordagem de serviço, toda implementação de objeto deve implementar um conjunto específico de operações *callback* a fim de poder se tornar membro de um grupo. Essas operações são utilizadas para a recepção de notificações sobre mudanças na visão do grupo e para a transferência de estado entre os seus membros.

As principais vantagens do uso dessa abordagem são a portabilidade, interoperabilidade e compatibilidade com a especificação CORBA, pois nenhum Sistema de Comunicação de Grupo proprietário é necessário, como ocorre nas abordagens de integração e interceptação. Assim, a abordagem de serviço permite que o próprio Serviço de Comunicação de Grupo, bem como as réplicas de objeto que constituem os grupos, possam ser executados em diferentes ORBs.

A principal desvantagem da abordagem de serviço é um desempenho potencialmente pior nas comunicações com grupos de objetos, se comparado com o desempenho dos sistemas que implementam as abordagens de integração e interceptação. Isto se deve, em grande parte, ao fato do Serviço de Comunicação de Grupo utilizar as primitivas de comunicação do próprio ORB para implementar Comunicação de Grupo. Assim, a eficiência deste serviço passa a depender diretamente da implementação do ORB no qual ele é executado. Além disso, o Serviço de Comunicação de Grupo introduz duas indireções em cada requisição destinada a um grupo de objetos. A primeira indireção ocorre quando a requisição é enviada pelo cliente até o Serviço de Comunicação de Grupo (evento 1 na figura 2.4) e a segunda indireção ocorre quando as respostas são retornadas pelas implementações de objeto para o Serviço de Comunicação de Grupo (evento 3 na figura 2.4).

O *Object Group Service* (OGS) [13] e o *Object Fault-Tolerance Service* (OFS) [39] são exemplos de sistemas que implementam a abordagem de serviço. O OGS será descrito em detalhes no próximo capítulo, pois ele constitui a infra-estrutura básica para criação de grupos de objetos utilizada no desenvolvimento do Juggler, descrito no capítulo 5.

Ao longo da seção 2.3, foram mencionados alguns sistemas que implementam uma dentre as possíveis abordagens para a incorporação de mecanismos de Comunicação de Grupo em CORBA. No entanto, há sistemas que adotam simultaneamente mais de uma dessas abordagens, como o GroupPac [28], o qual implementa uma solução que combina as abordagens de interceptação e de serviço.

Capítulo 3

OGS – Serviço de Grupos de Objetos

O *Object Group Service* (OGS) [13] é um serviço que permite a criação de grupos de objetos em CORBA. Ele foi o primeiro sistema a implementar a abordagem de serviço (vide seção 2.3.3) para a incorporação de mecanismos de comunicação de grupo em CORBA.

Do ponto de vista do programador de aplicação, o OGS funciona de forma semelhante a um serviço CORBA [16], fornecendo funcionalidades que podem ser acessadas por qualquer objeto CORBA. Uma característica muito importante desse serviço é o fato dele não modificar o ORB a fim de fornecer mecanismos de comunicação de grupo, tornando o uso desse serviço totalmente transparente para o ORB. Assim, o OGS pode ser utilizado de forma interoperável em diferentes ORBs compatíveis com a especificação de CORBA. Além disso, na implementação do OGS foram utilizadas apenas estruturas padronizadas em CORBA, conferindo total compatibilidade do serviço com essa arquitetura.

A arquitetura do OGS, ilustrada na figura 3.1, é altamente modular e flexível, permitindo a extensão e a reutilização de seus componentes individualmente. Essa arquitetura é constituída por um conjunto de interfaces especificadas em OMG IDL, as quais definem os seguintes componentes: Serviço de Grupo, Serviço de Consenso, Serviço de Monitoramento e Serviço de Mensagem.

Ao longo deste capítulo, o termo “interface interna” será utilizado para designar uma interface implementada por objetos do próprio OGS. Por outro lado, uma “interface externa” é aquela que deve ser implementada por objetos criados pelo programador de aplicação, também denominados objetos de aplicação. A seguir, são apresentados os componentes (serviços) definidos na arquitetura do OGS. Por fim, na última seção do capítulo, são mencionados alguns aspectos relativos à implementação e às formas de uso do OGS.

3.1 Serviço de Grupo

O Serviço de Grupo é o principal componente do OGS. Ele fornece os mecanismos básicos para que o programador de aplicação possa utilizar grupos de objetos. Esse serviço

Figura 3.1: Arquitetura do OGS

implementa duas funcionalidades básicas: Pertinência a Grupo e Multicast para Grupo.

3.1.1 Pertinência a Grupo

Esse serviço é responsável por gerenciar a pertinência em grupos de objetos, ou seja, manter uma lista atualizada de todos os seus membros. Além disso, ele fornece mecanismos que possibilitam a entrada e a saída de membros de um grupo. É importante ressaltar que o OGS permite a existência de grupos dinâmicos, ou seja, grupos cujo conjunto de membros pode ser modificado após a sua criação. Para fornecer esse suporte, o OGS implementa dois tipos de protocolos:

- **Protocolo para Mudança de Visão** - garante que todos os membros de um grupo recebem notificações sobre qualquer mudança no conjunto de membros (mudança de visão). Assim, sempre que um novo objeto torna-se membro de um grupo ou quando um membro falha ou deixa o grupo voluntariamente, todos os demais membros recebem uma notificação. O OGS garante que as notificações sobre mudanças na visão de grupo são totalmente ordenadas entre si, assegurando que todos os membros do grupo recebem essas notificações na mesma ordem. Dessa forma, a qualquer momento, todos os membros de um grupo de objetos possuem a mesma visão do grupo.
- **Protocolo para Transferência de Estado** - garante que o estado de todos os membros de um grupo permanece consistente após as mudanças de visão. Esse protocolo é utilizado durante a entrada de um novo membro em um grupo de objetos. Basicamente, o protocolo de transferência de estado consiste em copiar o estado replicado pelos membros de um grupo para cada objeto que ingressa no mesmo grupo (novo membro).

O Serviço de Pertinência a Grupo define as interfaces `GroupAdministrator` e `Groupable`. A interface interna `GroupAdministrator` é implementada pelos objetos *group administrators*.

Essa interface define operações que permitem a entrada e a saída de membros de um grupo. Cada *group administrator* é associado a um único membro de um grupo de objetos, mas pode haver vários deles em um mesmo *host* ou processo.

A interface externa *Groupable* define operações que possibilitam ao OGS notificar os membros de um grupo sobre mudanças de visão e realizar transferência de estado entre os membros do grupo. Essa interface deve ser implementada por qualquer objeto de aplicação que eventualmente venha a se tornar membro de um grupo de objetos.

3.1.2 Multicast para Grupo

Esse serviço fornece primitivas que permitem o envio de mensagens para grupos de objetos (*multicast*). Segundo a classificação comumente adotada na literatura [5], a estrutura de grupos do OGS é aberta, ou seja, um grupo pode receber mensagens *multicast* tanto dos seus membros como de qualquer outro objeto do sistema. As primitivas de comunicação fornecidas por esse serviço possibilitam a escolha entre diversos graus de confiabilidade, ordenação e transparência para cada mensagem *multicast*. Além disso, um cliente de um grupo de objetos também pode escolher o número de respostas a serem recebidas para cada invocação *multicast*.

No que diz respeito à confiabilidade e ordenação de mensagens *multicast*, esse serviço permite, respectivamente, o uso de invocações confiáveis e não confiáveis e invocações sem nenhuma garantia de ordenação, com ordenação FIFO e com ordenação total. Quanto ao número de respostas, o cliente que faz uma invocação *multicast* em um grupo de objetos pode esperar as respostas de todos os seus membros, da maioria deles, de apenas um deles ou, até mesmo, não esperar nenhuma resposta. No último caso, o cliente pode esperar a conclusão da operação (invocação síncrona) ou apenas fazer a invocação e não ficar bloqueado esperando a conclusão da mesma (invocação assíncrona).

Além dessas diversas possibilidades para o envio de *multicasts*, o Serviço de Multicast para Grupo permite que um cliente envie invocações explícitas ou invocações transparentes para grupos de objetos. Quando o cliente realiza uma invocação explícita (não tipada), qualquer tipo de informação pode ser enviado para os membros de um grupo. Para isso, o conteúdo da mensagem é encapsulado em um atributo do tipo *CORBA::Any*. Além disso, invocações explícitas podem ser feitas em quaisquer grupos de objetos, independentemente das interfaces implementadas pelos seus membros.

No caso de invocações transparentes (tipadas), o OGS permite que um cliente invoque diretamente as operações definidas na interface dos membros de um grupo. Portanto, esse tipo de invocação só pode ser feita em grupos de objetos onde todos os membros implementam a mesma interface IDL. Uma funcionalidade adicional fornecida pelo OGS, no caso de invocações transparentes, é a especificação da semântica associada às operações disponibilizadas pelas réplicas de objeto que compõem o grupo. Assim, o programador de aplicação pode decidir qual deverá ser o grau de confiabilidade e a garantia de ordenação de cada uma das operações implementadas pelos membros de um grupo.

O Serviço de Multicast para Grupo define as interfaces `GroupAccessor` e `Invocable`. A interface interna `GroupAccessor` é implementada pelos objetos *group accessors*. Essa interface define operações que possibilitam às aplicações cliente enviar invocações *multicast* para grupos de objetos. Cada *group accessor* é associado a um único grupo no momento de sua criação, ou seja, um cliente deve obter referências para tantos *group accessors* quantos forem os grupos com os quais deseja se comunicar. A interface externa `Invocable` define apenas uma operação, a qual permite ao OGS entregar *multicasts* explícitos (não tipados) para os membros de um grupo de objetos.

O Serviço de Grupo como um todo é implementado através da estreita interação entre o Serviço de Pertinência a Grupo e o Serviço de Multicast para Grupo. As interfaces do Serviço de Grupo podem ser classificadas como: interface do cliente, interface dos membros e interface do serviço. A interface do cliente representa a visão que o OGS disponibiliza para as aplicações cliente e é definida pela interface interna `GroupAccessor`. A visão que os membros dos grupos de objetos têm do OGS é fornecida pela interface interna `GroupAdministrator`, também denominada interface dos membros. Por fim, a interface do serviço representa a visão que o OGS tem dos membros de um grupo de objetos, a qual é definida pela interface externa `Groupable`¹.

A figura 3.2 ilustra as principais interfaces definidas pelo Serviço de Grupo do OGS. Nessa figura, os objetos de aplicação são denotados por “OBJ. APL.” e os objetos implementados pelo Serviço de Grupo são denotados por “OBJ. SRV.”.

Figura 3.2: Serviço de Grupo do OGS

Como pode ser visto na figura, as interfaces `GroupAccessor` e `GroupAdministrator` representam a visão externa de um grupo de objetos para os clientes que o acessam e para os seus membros, respectivamente. Assim, os objetos do OGS que implementam essas interfaces, denominados *group accessors* e *group administrators*, podem ser vistos como referências para grupos de objetos no OGS. Portanto, as referências para grupos de objetos são implementadas como verdadeiros objetos CORBA e não como Referências Interoperáveis para

¹A interface `Groupable`, definida pelo Serviço de Pertinência, estende a interface `Invocable` definida pelo Serviço de Multicast.

Objetos (IORs), como definido na especificação CORBA.

3.2 Serviço de Consenso

O Serviço de Consenso fornece mecanismos genéricos que possibilitam a resolução de vários problemas relacionados com o consenso distribuído, tais como pertinência a grupo e ordenação total de mensagens. A principal função desse serviço é implementar um protocolo de consenso, o qual garante que os elementos participantes do protocolo alcançam uma decisão comum, de acordo com os valores iniciais (estimativas) propostos por cada participante.

O protocolo de consenso implementado no OGS é baseado no algoritmo descrito em [8], o qual resolve o problema de consenso distribuído em sistemas assíncronos com o auxílio de um mecanismo para detecção de falhas não confiável. No entanto, esse algoritmo só garante a resolução do problema de consenso distribuído se a maioria simples dos elementos participantes do consenso não falhar.

No Serviço de Consenso do OGS são definidas as interfaces *ConsensusManager* e *ConsensusParticipant*. A interface interna *ConsensusManager* é implementada pelos objetos *consensus managers*, os quais implementam o protocolo de consenso propriamente dito. Essa interface define operações que permitem a uma aplicação iniciar a execução do protocolo de consenso.

A interface externa *ConsensusParticipant* deve ser implementada por cada um dos participantes do protocolo de consenso. Essa interface define operações que possibilitam aos participantes do protocolo enviar suas estimativas e receber um valor que representa a decisão consensual dos objetos *consensus manager*. Na implementação do Serviço de Consenso no OGS, cada objeto *consensus manager* é sempre associado a um único objeto *consensus participant*.

3.3 Serviço de Monitoramento

O Serviço de Monitoramento fornece mecanismos para a detecção de falhas (monitoramento) de objetos CORBA. Ele funciona como o elemento detector de falhas necessário para o funcionamento correto do algoritmo de consenso distribuído proposto em [8], ou seja, esse serviço é essencial para a implementação do Serviço de Consenso do OGS, descrito na seção 3.2.

O Serviço de Monitoramento implementa um mecanismo “não confiável” para detecção de falhas, ou seja, o serviço pode assumir que um determinado objeto tenha falhado quando de fato ele ainda continua funcionando. Esse serviço define as interfaces *Monitor*, *Monitorable* e *Notifiable*. A interface interna *Monitor* é implementada pelos objetos *monitors* (detectores de falhas) e define operações para iniciar e encerrar o monitoramento de objetos.

A interface externa *Monitorable* deve ser implementada por cada objeto a ser monitorado pelo OGS. Ela define uma operação que permite ao Serviço de Monitoramento verificar periodicamente se um objeto monitorado está funcionando normalmente. A interface externa *Notifiable*, por outro lado, deve ser implementada pelos objetos interessados em receber notificações sobre as falhas dos objetos que estão sendo monitorados pelo OGS.

O OGS adota o modelo *pull* de monitoramento. Nesse modelo, um *monitor* envia mensagens periódicas para um objeto monitorado e espera a resposta do mesmo. Caso esse objeto não responda dentro de um intervalo de tempo pré-determinado, então o *monitor* suspeita que o objeto tenha falhado e informa os objetos interessados em receber notificações sobre falhas do objeto monitorado.

3.4 Serviço de Mensagem

O Serviço de Mensagem incorpora comunicação assíncrona confiável no OGS, fornecendo mecanismos para comunicação ponto-a-ponto e para comunicação um-para-muitos (*multicast*). Um dos objetivos desse serviço é permitir que as aplicações cliente não fiquem bloqueadas durante a invocação confiável de uma operação em um grupo de objetos. Esse serviço permite que os clientes especifiquem se uma invocação, seja ela ponto-a-ponto ou *multicast*, deve ser enviada de forma não confiável, confiável ou FIFO.

O algoritmo utilizado para implementar comunicação *multicast* confiável no OGS é baseado no algoritmo de *broadcast* confiável proposto em [8]. Esse algoritmo utiliza o reenvio de mensagens e garante que em algum momento todos os destinatários de uma requisição enviada através de *multicast* a receberão. Logicamente, a entrega da requisição só é garantida para os destinatários que não falharem durante a execução do protocolo de *multicast*.

3.5 Implementação e Uso

O OGS foi implementado inicialmente tanto na linguagem *C++* como na linguagem *Java*. Atualmente, no entanto, a versão publicamente acessível (0.8b1) só é disponibilizada em *Java*. Essa versão do OGS foi criada com o VisiBroker 3.2 CORBA ORB (desenvolvido pela *Inprise Corporation*), mas também foi testada com o Orbix (desenvolvido pela *IONA Technologies*).

Os grupos de objetos criados através do OGS podem ser compostos por objetos localizados em um mesmo *host* ou em diferentes *hosts* de uma rede. Um grupo pode conter uma quantidade arbitrária de membros, mas em testes realizados em [13] observou-se que o custo do envio de mensagens *multicast* para grupos de objetos cresce de forma exponencial com o aumento do número de membros do grupo. Assim, o OGS tende a ser inadequado para o uso de grupos de objetos compostos por centenas ou milhares de membros.

Durante a criação de um grupo, os objetos *group administrators* associados às réplicas de objetos são registrados no Serviço de Nomes de CORBA. Isso permite a obtenção das Referências Interoperáveis para Objetos (IORs) de cada um dos *group administrators* pertencentes ao grupo quando um novo objeto entra no grupo e quando objetos *group accessor* enviam *multicasts* para o grupo de objetos. Dessa forma, a fim de garantir que todos os membros do grupo tenham uma visão idêntica, as IORs dos *group administrators* precisam ser adicionadas ou removidas de forma sequencial do Serviço de Nomes.

Apesar de definir interfaces tanto para o estilo de replicação ativa [37] como para o estilo de replicação primária-secundária (*primary-backup*) [1], o OGS implementa apenas os protocolos utilizados para replicação ativa. A fim de contornar essa limitação e fornecer diferentes estilos de replicação no OGS, foram incorporadas algumas extensões nesse serviço no contexto desta dissertação, as quais são discutidas no capítulo 7.

O OGS pode ser utilizado como uma biblioteca de ligação dinâmica ou como um *daemon*. Quando a primeira alternativa é adotada, a biblioteca dinâmica do OGS é ligada ao código da aplicação, aumentando a performance das comunicações de grupo e dispensando o tratamento de falhas no próprio serviço.

Se o OGS for utilizado como um *daemon*, então uma instância do mesmo deve ser executada em cada um dos *hosts* onde se deseja criar réplicas de objetos. Nessa abordagem, o OGS fica claramente separado da aplicação, mas torna-se necessário considerar a possibilidade de falhas no próprio serviço de comunicação de grupo, em *links* de comunicação e nos *hosts* do sistema.

Capítulo 4

Trabalhos Relacionados

A incorporação de mecanismos de Comunicação de Grupo em CORBA é uma abordagem adequada para possibilitar o desenvolvimento de aplicações distribuídas confiáveis e tolerantes a falhas nessa arquitetura. A maioria dos sistemas que oferecem esse tipo de suporte disponibilizam apenas operações básicas para o gerenciamento de grupos de objetos, como aqueles descritos no capítulo 2.

No entanto, o gerenciamento de aplicações distribuídas tolerantes a falhas requer tipicamente um suporte adicional. Este inclui a recuperação automática de falhas, a adaptação às mudanças que ocorrem no ambiente computacional (por exemplo, a modificação no conjunto de *hosts* do sistema) e a modificação dinâmica dos requisitos das aplicações. Assim, torna-se necessária a disponibilização de interfaces e mecanismos de alto nível para o administrador de aplicações, permitindo a definição de políticas mais abrangentes e flexíveis para o gerenciamento de aplicações tolerantes a falhas.

Neste capítulo, serão descritos os trabalhos mais recentes na literatura que definem mecanismos e interfaces de alto nível para o gerenciamento de aplicações CORBA tolerantes a falhas.

4.1 Piranha

O Piranha [31] é um serviço composto por uma coleção de objetos que fornecem suporte para o gerenciamento de aplicações CORBA que exigem alta disponibilidade. Ele funciona como um monitor e notificador de falhas em objetos replicados. O Piranha também age como um gerenciador de aplicações, fornecendo suporte para reinicialização automática de objetos que tenham falhado e replicação dinâmica (em tempo de execução) de objetos que possuem estado.

Além disso, esse serviço possibilita a migração de objetos e a manutenção do grau de replicação para aplicações baseadas em objetos replicados na arquitetura CORBA. O Piranha foi concebido para ser executado no Electra [30], um CORBA ORB que fornece mecanismos

para o uso de grupos de objetos, tais como *multicast* confiável, transferência de estado entre os membros de um grupo, detecção de falhas de objetos e sincronia virtual [5].

4.1.1 Arquitetura

O Piranha é composto por processos *Monitor* e processos *Graphical User's Interface* (GUI). Os processos *Monitor* são responsáveis pelo monitoramento e reinicialização de objetos. Eles também fornecem informações sobre a carga e o desempenho dos *hosts* onde estão executando. O Piranha define um grupo que contém todos os processos *Monitor* executando no sistema. Uma das funções desse grupo é manter replicada as informações sobre os objetos executando em todos os *hosts* do sistema. Com isso, o Piranha torna-se capaz de reinicializar todos os objetos que estavam executando em um *host* que tenha falhado. Outra função do grupo de processos *Monitor* é controlar exatamente em quais *hosts* do sistema as réplicas de um objeto podem ser criadas.

Os processos *GUI* fornecem uma interface gráfica para o usuário. Esses processos permitem que os clientes do Piranha criem grupos de objetos e obtenham informação sobre quais objetos estão executando em quais *hosts* do sistema. Os processos *GUI* também fornecem notificações sobre falhas em membros dos grupos de objetos e permitem a reinicialização de objetos que tenham falhado. Além disso, esses processos permitem que o usuário mova réplicas de objetos entre os *hosts* do sistema. A figura 4.1 ilustra os componentes do Piranha na arquitetura do Electra.

Figura 4.1: O Piranha na arquitetura do Electra

4.1.2 Critérios para Ativação de Grupos

O Piranha permite a especificação de vários critérios de ativação de objetos no momento da criação de grupos de objetos. Esses critérios permitem que um cliente especifique os *hosts* onde as réplicas de objeto devem ser criadas, o número de réplicas a serem criadas e o tempo de vida das mesmas. Além disso, o cliente pode escolher a arquitetura, a carga e o desempenho dos *hosts* onde as réplicas de objeto devem ser criadas. Alguns critérios mais avançados permitem, por exemplo, que o cliente especifique se os objetos devem ser removidos do sistema após um certo período de inatividade (sem processar requisições) e se o Piranha deve realizar a recuperação automática de objetos que tenham falhado.

4.1.3 Protocolos

O Piranha implementa basicamente protocolos que permitem a criação de grupos de objetos, recuperação de falhas de objetos em um grupo e movimentação de objetos entre os *hosts* do sistema. Esses protocolos são resumidos a seguir:

- **Criação de Grupos** - quando um cliente deseja criar um grupo de objetos, ele invoca a operação *create_object* no grupo de processos *Monitor*. Como argumentos para essa invocação, são especificados o nome do grupo a ser criado e os critérios de ativação a ele associados. Em seguida, cada réplica do objeto requisitado é criada nos *hosts* especificados no critério de ativação do grupo. Por fim, uma referência para o novo grupo de objetos é criada e retornada para o cliente. É importante ressaltar que o Piranha só permite a criação de uma réplica do mesmo objeto em cada *host* do sistema.
- **Recuperação de Falhas** - quando ocorre a falha de uma réplica de objeto, esse protocolo simplesmente reinicia uma outra réplica do mesmo objeto no *host* onde se encontrava o objeto que falhou. Caso um *host* falhe, então o Piranha reinicia todas as réplicas de objetos que foram destruídas em outro *host* do sistema. No entanto, cada nova réplica só pode ser reiniciada em um *host*, caso não exista nenhuma outra réplica do mesmo objeto em execução neste *host*.
- **Movimentação de Objetos** - quando um cliente requisita a movimentação de um objeto, o Piranha verifica se já existe alguma réplica do mesmo executando no *host* de destino (para onde o objeto deve ser movido). Se esse for o caso, então o objeto não é movimentado. Caso contrário, uma nova réplica do objeto é criada no *host* de destino e o objeto original é destruído.

4.2 Proteus

O Proteus [36] é um serviço que fornece tolerância a falhas na arquitetura AQuA [11] através do gerenciamento dinâmico de objetos distribuídos replicados. Uma das suas funções

é decidir como fornecer tolerância a falhas para uma aplicação de acordo com os requisitos de disponibilidade especificados para a mesma. Esta decisão envolve a escolha do estilo de replicação, tipo de votação a ser utilizado, grau de replicação e tipos de falhas a serem toleradas por uma aplicação.

Assim, o Proteus pode ser visto como um serviço cuja função é garantir os requisitos de Qualidade de Serviço (QoS) relativos à confiabilidade e à disponibilidade das aplicações. A outra função do Proteus é permitir a modificação dinâmica da configuração de uma aplicação de acordo com o esquema de tolerância a falhas mais adequado para a mesma.

As aplicações a serem gerenciadas pelo Proteus devem satisfazer algumas exigências, que, por sua vez, refletem restrições de projeto ou de implementação da arquitetura AQuA e do próprio Proteus. As principais exigências impostas às aplicações desenvolvidas para a arquitetura AQuA são:

- A aplicação não pode utilizar o Serviço de Nomes nem o Serviço de Ciclo de Vida especificados pela arquitetura CORBA. Isso se deve ao fato do Proteus lidar diretamente com o estabelecimento de conexões entre objetos (*binding*) e com a criação e a destruição de objetos.
- A aplicação não pode utilizar comunicação assíncrona de CORBA. Apenas comunicação síncrona e síncrona postergada (*deferred synchronous*) podem ser utilizadas.
- O comportamento de todo objeto replicado deve ser determinístico, ou seja, objetos não podem utilizar mecanismos não determinísticos em sua implementação, tais como *threads* ou geradores de números aleatórios.
- Qualquer objeto cuja implementação utiliza ORBs multitarefa com escalonamento não determinístico, comunicação síncrona postergada ou comunicação com múltiplos objetos não pode ser replicado.
- Toda comunicação entre objetos deve ser feita através da arquitetura AQuA.
- A aplicação deve utilizar o sistema operacional Linux.

A arquitetura AQuA faz pouco uso da infra-estrutura fornecida por CORBA, dado que alguns serviços CORBA [16], como o Serviço de Nomes e Serviço de Ciclo de Vida, não podem ser utilizados. Além disso, todas as comunicações entre objetos são feitas através do Sistema de Comunicação de Grupo Ensemble [22] e não através do ORB.

A arquitetura do Proteus, ilustrada na figura 4.2, é composta dos seguintes elementos: Ambiente de Execução de QuO, Gerenciador de Dependência¹, Fábrica de Objetos e Gateway. Esses elementos são descritos em detalhes nas próximas seções.

¹O termo “dependência” está empregado como uma tradução do termo original em inglês “dependability”.

Figura 4.2: Arquitetura do Proteus

4.2.1 Ambiente de Execução de QuO

O Ambiente de Execução de QuO permite o uso de Objetos de Qualidade (QuO) [41] na arquitetura AQuA. O principal objetivo de QuO é fornecer uma interface orientada a objeto que permite a especificação dinâmica de requisitos de Qualidade de Serviço (QoS) para aplicações distribuídas.

Através de objetos QuO, o programador de aplicação especifica os requisitos de QoS usando uma linguagem para a definição de contratos denominada *Contract Definition Language* (CDL). Baseado no contrato especificado, o ambiente de execução de QuO interage com o Gerenciador de Dependência, a fim de configurar as aplicações de acordo com os requisitos de QoS especificados para as mesmas. Esses requisitos de QoS podem definir, por exemplo, as ações a serem tomadas pelo Ambiente de Execução de QuO para garantir o grau de replicação de uma aplicação, bem como a confiabilidade e a ordenação das mensagens enviadas para as aplicações desenvolvidas na arquitetura AQuA.

4.2.2 Gerenciador de Dependência

O Gerenciador de Dependência é o componente do Proteus que traduz os requisitos de QoS das aplicações, especificados através do Ambiente de Execução de QuO, para uma configuração correspondente do sistema. Esse componente também é responsável pelo tratamento de falhas ocorridas nas aplicações executando na arquitetura AQuA (aplicações AQuA). Além disso, o Gerenciador de Dependência faz chamadas ao Ambiente de Execução de QuO quando os requisitos de QoS de uma aplicação são alcançados com sucesso ou quando esses requisitos não podem mais ser mantidos.

Na implementação atual do Proteus, o Gerenciador de Dependência não é replicado. Assim, quando o mesmo falha, as aplicações AQuA perdem todo o suporte para o gerenciamento de aplicações fornecido pelo Proteus. O Gerenciador de Dependência é composto por um Conselheiro e um Coordenador de Protocolo, descritos a seguir.

Coordenador de Protocolo

O Coordenador de Protocolo é um objeto CORBA que permite a comunicação do Gerenciador de Dependência com o Ambiente de Execução de QuO, com as Fábricas de Objetos e com os Gateways. Ele também gerencia as estruturas do Gerenciador de Dependência que mantêm as informações sobre a configuração das aplicações que estão executando na arquitetura AQuA.

O Coordenador de Protocolo interage diretamente com o Conselheiro, informando-o sobre a ocorrência de eventos inesperados na configuração do sistema, tais como falhas nas aplicações. Ele também define operações para recepção de mudanças na visão de grupo, registro de Fábricas de Objetos, inicialização e destruição de aplicações, obtenção da carga dos *hosts* do sistema e criação de Regiões de Dependência. Nas Regiões de Dependência são definidos predicados sobre o estado dos objetos que avaliam as condições do sistema (*system condition objects*), os quais fornecem informações para o Ambiente de Execução de QuO sobre a manutenção dos requisitos de QoS especificados para as aplicações.

As operações para criação e destruição de objetos de aplicação são requisitadas pelo Conselheiro, a fim de garantir que os requisitos de QoS das aplicações sejam mantidos. Por outro lado, a operação para criação de Regiões de Dependência faz parte da interface entre o Gerenciador de Dependência e o Ambiente de Execução de QuO.

Conselheiro

Esse componente do Gerenciador de Dependência é responsável pelo tratamento de falhas e pelo processamento de requisições para a modificação da configuração das aplicações. Ele define operações que permitem a recepção de notificações sobre falhas nos objetos de aplicação, enviadas pelo Coordenador de Protocolo. Além disso, são definidas operações que permitem a criação e a destruição de Regiões de Dependência.

No caso de receber uma notificação de falha em uma réplica de objeto de uma aplicação, o Conselheiro informa o Coordenador de Protocolo que uma nova réplica deve ser criada em um *host* que não esteja executando outra réplica do mesmo objeto de aplicação. Por outro lado, se for requisitada a criação de uma Região de Dependência, então o programador de aplicação deve especificar a quantidade de falhas (n) do tipo queda (*crash*) a serem toleradas. Se nenhuma réplica do objeto requisitado estiver executando, então o Conselheiro solicita que o Coordenador de Protocolo inicialize $n+1$ réplicas. Caso já existam réplicas do objeto executando, então o Conselheiro solicita que o Coordenador de Protocolo inicialize ou destrua réplicas do objeto de forma que $n+1$ réplicas permaneçam executando.

Se o Conselheiro não obtiver êxito durante a recuperação de uma falha em uma réplica de objeto ou na criação de uma Região de Dependência, então ele informa o Ambiente de Execução de QuO sobre a impossibilidade de garantir os requisitos de QoS especificados para a aplicação correspondente.

4.2.3 Fábrica de Objetos

O principal objetivo de uma Fábrica de Objetos é inicializar e destruir réplicas de objetos. Na arquitetura AQuA, existe uma Fábrica de Objetos em cada *host* do sistema onde é executado o Gerenciador de Dependência do Proteus. Outra função de uma Fábrica de Objetos é fornecer informações periódicas sobre o *host* no qual ela reside. Estas informações são enviadas para o Gerenciador de Dependência, permitindo que ele escolha o melhor local para a criação de réplicas de objeto.

A primeira tarefa executada durante a inicialização de uma Fábrica de Objetos é a leitura de um arquivo (específico do *host* onde ela se localiza) contendo o nome, o caminho para o código executável e os parâmetros de inicialização de cada uma das aplicações que podem ser criadas. Em seguida, a Fábrica de Objetos se registra no Gerenciador de Dependência, informando-o que a mesma já está pronta para criar novas réplicas de objetos.

4.2.4 Gateway

Os Gateways são responsáveis por toda comunicação entre os elementos da arquitetura AQuA. Um Gateway é acoplado a cada um dos objetos das aplicações AQuA, ao Gerenciador de Dependência, aos objetos de QuO e às Fábricas de Objetos. É nos Gateways que são implementados os estilos de replicação e de votação fornecidos pela arquitetura AQuA, bem como os mecanismos para a detecção e a notificação de falhas para o Gerenciador de Dependência. Além disso, os Gateways são responsáveis pela tradução de mensagens IIOP para mensagens a serem enviadas pelo Sistema de Comunicação de Grupo Ensemble [22], e vice-versa.

A estrutura dos Gateways permite a implementação de vários estilos de replicação e de votação. No entanto, a implementação atual desses componentes só fornece suporte para replicação ativa e para a técnica de votação *leader pass first*, descritos detalhadamente em [36]. Além disso, na implementação atual do Proteus, os Gateways só fornecem mecanismos para detecção e recuperação de falhas do tipo queda de hosts (*crash*).

4.3 FT-CORBA

A especificação do padrão para tolerância a falhas em CORBA (FT-CORBA) [21], proposta pelo OMG, define um conjunto de interfaces e serviços que oferecem suporte para aplicações CORBA que exigem um alto grau de confiabilidade. A infra-estrutura proposta

no FT-CORBA é baseada na redundância de objetos e em mecanismos para detecção e recuperação de falhas nos mesmos, de maneira similar ao que é proposto em trabalhos como o Piranha [31] e o Proteus [36].

O padrão FT-CORBA contempla uma grande flexibilidade no gerenciamento de grupos de objetos. Ele fornece suporte para diversas estratégias de tolerância a falhas, como replicação ativa e passiva (*primary-backup*), e ainda, quando replicação passiva é utilizada, redirecionamento de requisições e reinvocação transparente de métodos. O FT-CORBA define mecanismos que permitem a detecção, a notificação e a análise de falhas em réplicas de objetos. Além disso, o padrão fornece suporte para aplicações que exigem recuperação automática de falhas e mecanismos que permitem à infra-estrutura de tolerância a falhas controlar automaticamente a consistência entre as réplicas de objeto em um grupo (*checkpoint* e *logging*).

A figura 4.3 ilustra os principais componentes definidos no padrão FT-CORBA. Os componentes ReplicationManager, FaultNotifier, Factory e FaultDetector são implementados como objetos CORBA. Por outro lado, os mecanismos de recuperação de falhas, de *logging* e de tratamento de mensagens não são objetos CORBA, mas podem ser implementados como uma parte do ORB ou como um serviço que funciona entre o ORB e o sistema operacional.

Figura 4.3: Arquitetura do FT-CORBA

O componente `ReplicationManager` permite uma aplicação controlar a criação e a destruição de grupos de objetos, bem como a modificação das propriedades a eles associadas. O `FaultDetector` é responsável pela detecção de falhas nas réplicas de objetos e o `FaultNotifier` pela notificação dessas falhas. Os componentes `Factory` possibilitam a criação e a destruição tanto de objetos replicados (grupos de objetos) como de objetos não replicados (*singleton objects*). Por fim, os mecanismos de recuperação de falhas e de *checkpoint* e *logging* permitem a recuperação automática de falhas.

Logicamente, há uma única instância do `ReplicationManager` e do `FaultNotifier`, mas fisicamente eles são replicados para permitir proteção contra falhas. Os objetos do tipo `Factory` e `FaultDetector` podem estar presentes em cada *host* e são específicos do *host* no qual se encontram. Neste caso, esses objetos não são replicados. No entanto, pode haver objetos `FaultDetector` replicados, os quais não são específicos de um dado *host* do sistema. Por outro lado, os mecanismos de tratamento de mensagens, de recuperação de falhas e de *checkpoint* e *logging* devem estar presentes em cada *host* do sistema onde réplicas de objetos podem ser criadas.

Apesar de padronizar um conjunto de interfaces e serviços para o gerenciamento de aplicações CORBA tolerantes a falhas, o padrão FT-CORBA apresenta algumas limitações quanto ao uso de aplicações cliente que utilizam CORBA ORBs já existentes e quanto à interoperabilidade entre as infra-estruturas de tolerância a falhas fornecidas por diferentes fabricantes.

O FT-CORBA permite que um cliente desenvolvido em um ORB que não implementa este padrão possa invocar operações em objetos replicados. No entanto, tal cliente não se beneficia de todas as propriedades de tolerância a falhas fornecidas pelo uso de grupos de objetos. Além disso, o padrão FT-CORBA exige que em cada Domínio de Tolerância a Falhas (vide seção 4.3.1) todos os ORBs utilizados sejam de um mesmo fabricante, a fim de garantir a interoperabilidade e o uso de todas as funcionalidades de tolerância a falhas dentro de um domínio. Assim, os membros de um grupo de objetos devem utilizar ORBs e infra-estruturas de tolerância a falhas de um mesmo fabricante. Nas próximas seções é feita uma descrição mais detalhada dos principais conceitos, componentes e mecanismos especificados no padrão FT-CORBA.

4.3.1 Domínios de Tolerância a Falhas

Prevendo o gerenciamento de aplicações tolerantes a falhas arbitrariamente grandes e complexas, o padrão FT-CORBA define domínios de tolerância a falhas. Um domínio tipicamente contém vários *hosts* e diversos grupos de objetos. Um *host* pode fazer parte de mais de um domínio de tolerância a falhas e todos os grupos de objetos em um domínio são criados e gerenciados por um único componente lógico `ReplicationManager`. Além disso, FT-CORBA permite que um grupo de objetos definido em um domínio invoque e seja invocado por objetos definidos em outros domínios.

4.3.2 Endereçamento de Grupos de Objetos

CORBA define um mecanismo flexível de endereçamento que identifica cada objeto unicamente, denominado Referência Interoperável para Objeto (IOR) [18]. FT-CORBA padroniza o formato da IOR utilizada para endereçar as réplicas de objeto que compõem um grupo. Além disso, o padrão define uma IOR específica para grupos de objetos, denominada Referência Interoperável para Grupo de Objetos (IOGR), ilustrada na figura 4.4.

Uma IOGR contém múltiplos perfis TAG_INTERNET_IOP, os quais são utilizados para acessar as réplicas de objeto que compõem um grupo. Cada perfil TAG_INTERNET_IOP deve conter um componente TAG_FT_GROUP e pode conter outros componentes, como TAG_IOP_ALTERNATE_ADDRESS. Além disso, um perfil TAG_INTERNET_IOP pode conter um componente TAG_FT_PRIMARY.

Figura 4.4: Estrutura de uma Referência Interoperável para Grupo de Objetos (IOGR).

O componente TAG_FT_GROUP armazena informações que permitem acessar um grupo de objetos. Esse componente armazena um identificador para o grupo de objetos e um identificador para o domínio de tolerância a falhas no qual o grupo é definido. O componente TAG_FT_PRIMARY indica que um perfil pode ser utilizado preferencialmente em detrimento de outros perfis TAG_INTERNET_IOP. Em uma IOGR, no máximo um perfil TAG_INTERNET_IOP pode conter um componente TAG_FT_PRIMARY. Esse componente pode ser utilizado, por exemplo, para acessar o membro primário em um grupo de objetos que adota o estilo de replicação COLD_PASSIVE ou WARM_PASSIVE, descritos na próxima seção.

4.3.3 Gerenciamento de Replicação

O `ReplicationManager` é o componente do FT-CORBA que garante a criação e o gerenciamento de grupos de objetos. Esse componente pode ser replicado, para permitir tolerância a falhas, mas logicamente há um único `ReplicationManager` por domínio de tolerância a falhas. O `ReplicationManager` herda as seguintes interfaces: `PropertyManager`, `GenericFactory` e `ObjectGroupManager`.

A interface `PropertyManager` permite a escolha das propriedades associadas a um grupo de objetos. Algumas dessas propriedades podem ser modificadas dinamicamente, como a quantidade inicial e mínima de réplicas em um grupo de objetos. Outras propriedades, como os estilos de replicação, de pertinência e de consistência utilizados por um grupo de objetos não podem ser modificados após a criação de um grupo. O FT-CORBA permite que a pertinência a um grupo e a consistência entre os membros de um grupo sejam controladas pela infra-estrutura de tolerância a falhas ou pela própria aplicação. Além disso, o padrão define os estilos de replicação `STATELESS`, `COLD_PASSIVE`, `WARM_PASSIVE` e `ACTIVE`.

No estilo de replicação `STATELESS`, o comportamento de um grupo de objetos não é afetado pelo histórico de invocações feitas no grupo. Nos estilos `COLD_PASSIVE` e `WARM_PASSIVE` apenas um membro do grupo, denominado membro primário ou réplica primária, executa as operações requisitadas em um grupo de objetos. Os outros membros do grupo são utilizados como cópias (*backup*) do membro primário e só executam operações no caso de falha da réplica primária.

No estilo `COLD_PASSIVE`, ao ocorrer a falha no membro primário, o seu estado é extraído de um *log* e carregado em um membro de *backup*, o qual se torna a nova réplica primária do grupo. Por outro lado, no estilo `WARM_PASSIVE`, o estado de um ou mais membros de *backup* é periodicamente atualizado pelo membro primário, de forma que algum deles possa assumir o papel de primário no caso de falha deste. No estilo de replicação `ACTIVE` todos os membros de um grupo de objetos executam todas as operações requisitadas ao grupo. Nesse estilo de replicação, requisições e respostas duplicadas são detectadas e suprimidas pela infra-estrutura de tolerância a falhas.

A interface `GenericFactory` é usada por uma aplicação para criar e destruir grupos de objetos. Além disso, essa interface é utilizada pelo `ReplicationManager` para a criação e a destruição das réplicas de objeto que compõem um grupo. A interface `ObjectGroupManager`, por outro lado, possibilita uma aplicação criar e destruir membros de um grupo, bem como adicionar objetos existentes a um grupo de objetos. Essa interface também permite que uma aplicação defina o membro primário de um grupo de objetos e obtenha a localidade e a referência de cada membro que compõe o grupo.

4.3.4 Gerenciamento de Falhas

A infra-estrutura para gerenciamento de falhas definida pelo FT-CORBA é composta pelos componentes `FaultDetector`, `FaultNotifier` e `FaultAnalyzer`. O `FaultDetector` é responsável por detectar falhas nas réplicas de objeto através do modelo *pull* de monitora-

mento. Nesse modelo, o `FaultDetector` invoca, em intervalos freqüentes, um método nas réplicas de objetos que são monitoradas. Caso uma réplica não responda a essa invocação de método, então o `FaultDetector` considera que ela falhou. O `FaultDetector` só detecta falhas do tipo queda (*crash*) e não define mecanismos para o tratamento de falhas de particionamento de rede (*network partitioning faults*), falhas bizantinas (*byzantine faults*) e falhas correlatas (*correlated faults*).

O componente `FaultNotifier` define operações para registros de objetos interessados em receber notificações sobre eventos de falhas. Além disso, o `FaultNotifier` também define operações para a transmissão dessas notificações para os consumidores de eventos de falhas (objetos registrados). Sempre que é detectada a falha de uma réplica de objeto, o `FaultDetector` envia um relatório sobre a falha para o `FaultNotifier`, o qual fica responsável por difundir uma notificação para os consumidores de eventos de falhas.

O FT-CORBA define ainda um componente que permite a correlação de eventos de falhas e a geração de relatórios de falhas condensados, denominado `FaultAnalyzer`. Para isso, o `FaultAnalyzer` se registra com o `FaultNotifier` a fim de receber notificações sobre os eventos de falhas e utiliza o próprio `FaultNotifier` para disseminar os relatórios de falhas condensados. A análise de falhas é específica para cada tipo de aplicação, por isso o padrão FT-CORBA não define uma interface para o `FaultAnalyzer`.

4.3.5 Recuperação de Falhas e Logging

O FT-CORBA prevê que a consistência entre os membros de um grupo seja controlada pela própria aplicação ou pela infra-estrutura de tolerância a falhas. No primeiro caso, os mecanismos para manter consistente o estado armazenado nos membros de um grupo e para recuperação de falhas em réplicas de objetos devem ser implementados diretamente pelo programador de aplicação. No caso de consistência controlada pela infra-estrutura, esses mecanismos são fornecidos pela própria infra-estrutura de tolerância a falhas e se tornam transparentes para o programador de aplicação.

Durante o funcionamento normal de um grupo de objetos com replicação passiva, o mecanismo de *checkpoint* e *logging* armazena o estado e as operações processadas pelo membro primário de um grupo em um *log*. Caso o grupo adote o estilo de replicação `WARM_PASSIVE`, esse mecanismo é responsável por enviar periodicamente as atualizações do estado do membro primário para os membros de *backup*. Caso o grupo adote o estilo de replicação `COLD_PASSIVE`, o mecanismo de *checkpoint* e *logging* é utilizado para armazenar apenas o estado completo do membro primário de um grupo de objetos em um *log*.

Quando ocorre uma falha na réplica primária de um grupo com replicação passiva, a infra-estrutura promove um dos membros *backup* do grupo para réplica primária. Neste caso, o mecanismo de recuperação de falhas atualiza o estado da nova réplica primária de acordo com as informações armazenadas no *log*. O mecanismo de recuperação de falhas garante, portanto, o estado consistente de um antigo membro *backup* de um grupo com replicação passiva que foi promovido a réplica primária. Além disso, esse mecanismo é utilizado durante

a criação de novos membros em grupos que adotam o estilo de replicação ACTIVE, a fim de garantir que todos os membros do grupo armazenem o mesmo estado.

4.4 DOORS

O DOORS [35] é um serviço para tolerância a falhas em CORBA que implementa subconjunto das funcionalidades definidas pelo padrão FT-CORBA. O DOORS começou a ser desenvolvido antes da adoção do FT-CORBA e vários conceitos e componentes definidos nesse serviço foram integrados no padrão especificado pelo OMG. Assim, a arquitetura do DOORS é bastante semelhante àquela definida pelo FT-CORBA, a qual é ilustrada na figura 4.3.

O DOORS define os componentes `ReplicationManager`, `FaultDetector` e `SuperFaultDetector`. O `ReplicationManager` encapsula a maioria das operações relativas ao gerenciamento de propriedades e ao gerenciamento de grupos definidas no FT-CORBA. Os componentes `FaultDetector` e `SuperFaultDetector` fornecem suporte para detecção hierárquica de falhas. Assim, o `FaultDetector` é responsável pela detecção de falhas em réplicas de objetos, enquanto que o `SuperFaultDetector` realiza detecção de falhas de *hosts*.

Os componentes `FaultDetector` e `SuperFaultDetector` também funcionam como notificadores de falhas e propagam as notificações de eventos de falhas para o `ReplicationManager`. O DOORS suporta tanto o modelo *pull* de monitoramento (adotado no FT-CORBA) quanto o modelo *push*. Nesse último modelo, as réplicas de objetos enviam mensagens frequentes, denominadas *heartbeats*, para o `FaultDetector`. Caso uma réplica deixe de enviar *heartbeats*, o `FaultDetector` considera que ela falhou.

Todos os componentes da infra-estrutura de tolerância a falhas definidos pelo DOORS são monitorados pelo `SuperFaultDetector` através do modelo *push*. Além disso, os `SuperFaultDetector` são replicados e monitoram-se mutuamente. Assim, através de replicação e monitoramento dos componentes de sua infra-estrutura, o DOORS garante que não há um único ponto de falta no serviço.

4.4.1 Adequação ao Padrão FT-CORBA

Como mencionado anteriormente, o DOORS implementa um subconjunto das funcionalidades especificadas pelo padrão FT-CORBA. Nesta seção são descritas as funcionalidades do FT-CORBA que não foram implementadas pelo DOORS e algumas extensões feitas ao padrão.

Funcionalidades não Implementadas

Apesar do `ReplicationManager` do DOORS ter sido projetado para lidar com a criação e recuperação de falhas em grupos que adotam o estilo de replicação `ACTIVE`, a versão atual do serviço não implementa protocolos para Comunicação de Grupo. Assim, o DOORS só permite o uso de grupo de objetos que adotam os estilos de replicação `STATELESS`, `COLD-PASSIVE` e `WARM-PASSIVE`.

O DOORS não fornece mecanismos próprios para recuperação automática de falhas e *logging*. Dessa forma, o serviço exige que as aplicações tolerantes a falhas se responsabilizem pela implementação desses mecanismos. A infra-estrutura fornecida pelo DOORS também não disponibiliza um objeto do tipo `FaultNotifier`, como proposto no padrão FT-CORBA. Ao invés disso, as funcionalidades para detecção e notificação de falhas do DOORS são combinadas no componente `FaultDetector`.

Extensões ao Padrão

Cada Referência Interoperável para Grupo de Objetos (IOGR) criada pelo DOORS armazena um perfil para cada réplica de objeto que faz parte de um grupo, como definido pelo FT-CORBA. Além disso, cada IOGR armazena um perfil que contém uma referência para um objeto do tipo `ReplicationManager` e a identificação única do grupo. Assim, o DOORS permite que um cliente acesse um grupo de objetos mesmo quando todos os membros referenciados na IOGR já não mais fizeram parte do grupo.

Os componentes `FaultDetector` do DOORS utilizam interfaces proprietárias para enviar notificações sobre falhas em réplicas de objetos para o `ReplicationManager`. As interfaces para notificação de falhas implementadas pelo DOORS são mais eficientes do que aquelas especificadas pelo FT-CORBA. A menor eficiência na transmissão de notificações no FT-CORBA deve-se ao fato das interfaces definidas no padrão serem baseadas no Serviço de Notificação de CORBA [19], o qual transmite informações através da passagem de valores do tipo `CORBA::Any`.

O DOORS define algumas propriedades adicionais para identificar o tipo das réplicas de objeto que compõem um grupo. Possíveis tipos de réplicas são: `PRIMARY`, `BACKUP` e `IDLE`. Além disso, o DOORS também define uma propriedade que determina quantas vezes o `FaultDetector` deve tentar contactar uma réplica de objeto antes de assumir que ela tenha falhado. Essa propriedade é importante para evitar que a infra-estrutura de tolerância a falhas inicie desnecessariamente o mecanismo de recuperação de falhas, evitando o desperdício de recursos.

Capítulo 5

Juggler

O Juggler é um serviço que fornece uma infra-estrutura robusta para o gerenciamento de aplicações CORBA que exigem um alto grau de confiabilidade, principalmente aquelas onde não deve haver um único ponto de falta. Esse serviço disponibiliza interfaces que permitem o gerenciamento da configuração de aplicações CORBA tolerantes a falhas, as quais são baseadas na redundância dos objetos que as compõem (grupos de objetos) e de mecanismos para a detecção e a recuperação de falhas desses objetos.

Através do Juggler, é possível definir o estilo de replicação e as propriedades de tolerância a falhas associados a cada grupo de objetos individualmente. Essas propriedades definem, por exemplo, a localização e a quantidade mínima de réplicas que compõem um grupo, e permitem escolher a forma mais adequada para a recuperação de falhas ocorridas nos membros de um grupo. Além disso, o Juggler permite a modificação dinâmica das propriedades de tolerância a falhas e do estilo de replicação associados a grupos de objetos.

O Juggler foi desenvolvido tomando-se como base o OGS [13], um serviço que disponibiliza os mecanismos básicos para a criação de grupos de objetos CORBA (veja o capítulo 3). O OGS foi escolhido como a infra-estrutura de tolerância a falhas para o desenvolvimento do Juggler por definir uma arquitetura para Comunicação de Grupo que utiliza apenas estruturas padronizadas pela especificação CORBA, possibilitando a portabilidade e interoperabilidade desse serviço em diferentes CORBA ORBs.

Outra característica do OGS imprescindível para o desenvolvimento do Juggler é o suporte fornecido para o monitoramento de objetos, o que permite ao Juggler obter notificações sobre eventuais falhas nas réplicas de objetos que constituem um grupo. Além disso, o OGS também disponibiliza um conjunto de mecanismos e serviços que tornam possível a implementação de outros estilos de replicação diferentes de replicação ativa, como descrito no capítulo 7.

A figura 5.1 apresenta uma visão geral dos principais componentes do Juggler, denominados GroupManager, GroupActor e GroupObserver, e a interação desse serviço com outras camadas de software. O Juggler encontra-se na camada imediatamente superior à do OGS, disponibilizando para o administrador de aplicações uma interface de alto nível para o ge-

renciamento de grupos de objetos.

Dessa forma, os mecanismos e os serviços do OGS utilizados para a manutenção de grupos se tornam transparentes para o administrador de aplicações tolerantes a falhas. Assim, este não precisa mais lidar com a execução de algumas tarefas relativas ao gerenciamento de grupos de objetos, como o monitoramento e a recuperação de réplicas de objetos.

Figura 5.1: Arquitetura do Juggler

5.1 Arquitetura do Serviço

Pode-se considerar que os grupos de objetos criados através do Juggler são gerenciados por uma única entidade lógica, composta pelos componentes GroupManager, GroupObserver e GroupActor. A interação entre esses componentes é ilustrada na figura 5.2.

Figura 5.2: Arquitetura lógica do Juggler

O componente GroupManager lida basicamente com a criação e a destruição de grupos de objetos e com a modificação de propriedades de tolerância a falhas associadas a esses

grupos. Para tanto, o GroupManager invoca operações no GroupActor, o qual é responsável pela criação e destruição de réplicas de objetos. O componente GroupObserver, por sua vez, é responsável pelo monitoramento dos membros dos grupos de objetos e pela notificação de suspeitas de falhas desses objetos, as quais são propagadas para o GroupManager.

Assim que recebe uma dessas notificações, o GroupManager verifica as propriedades associadas ao grupo do qual o objeto que falhou é membro e, então, toma as ações necessárias para garantir que as propriedades associadas ao grupo sejam respeitadas. Essas ações podem envolver, por exemplo, a criação de uma nova réplica de objeto no grupo ou a remoção do objeto que falhou da lista de membros do respectivo grupo.

Os componentes GroupManager, GroupObserver e GroupActor são especificados como interfaces definidas em OMG IDL. Na seção 5.3 são descritas apenas as principais interfaces herdadas por esses componentes e algumas operações importantes para a compreensão da funcionalidade dos mesmos. As operações e interfaces mais relacionadas à implementação desses componentes serão descritas no capítulo 6.

5.2 Modelo de Sistema

O modelo de comunicação e de faltas adotado pelo Juggler segue o mesmo modelo suposto pelo OGS. Assim o Juggler adota um modelo de sistema assíncrono [38] baseado em um detector de falhas não confiável [8], o qual é implementado pelo Serviço de Monitoramento do OGS. Nesse modelo, nada pode ser dito sobre a velocidade relativa do sistema de comunicação, dos processadores e dos processos em uma rede. O Juggler também assume que as primitivas de comunicação fornecidas pelo OGS garantem a entrega confiável e ordenada de mensagens multicast.

No modelo de faltas adotado pelo Juggler, supõe-se que os elementos do sistema (processos, processadores e objetos) falham independentemente e segundo o modelo *fail-stop* [5]. Esse modelo garante que um elemento do sistema ou funciona corretamente ou simplesmente pára de funcionar (*crash*) sem tomar nenhuma ação incorreta devido a uma falha.

Portanto, a infra-estrutura para tolerância a falhas desenvolvida no contexto deste trabalho, de forma semelhante ao FT-CORBA, não fornece provisões para o tratamento de falhas de particionamento de rede, falhas bizantinas e falhas correlatas [21].

5.3 Especificação do Serviço

O Juggler é especificado através de uma coleção de interfaces definidas em OMG IDL. Ao longo desta seção, serão descritas em detalhes as operações definidas nas principais interfaces definidas pelo Juggler, bem como algumas estruturas essenciais para a compreensão das mesmas.

Algumas especificações utilizadas no Juggler são precedidas pelo prefixo `FT::`, indicando que o Juggler utiliza o elemento correspondente (interface, estrutura ou tipo) como definido no FT-CORBA [21]. Da mesma forma, as especificações precedidas pelos prefixos `mGroupAdmin::` e `mMonitoring::` indicam definições feitas no OGS (vide capítulo 7).

5.3.1 Estrutura Semantics

A cada grupo de objetos criado através do Juggler é associado um conjunto de propriedades de tolerância a falhas. Essas propriedades incluem o tipo e a localização dos objetos que compõem o grupo, as quantidades inicial e mínima de réplicas, o estilo de replicação adotado e a maneira como eventuais falhas nas réplicas de objeto devem ser tratadas. Todas as propriedades de tolerância a falhas associadas a um grupo de objetos são armazenadas na estrutura `Semantics`, a qual é ilustrada na figura 5.3.

```
// OMG IDL
struct ReplicationProperties {
    PrimaryReplicasLocationsValue primaries;
    mGroupAdmin::CheckpointIntervalValue checkpoint_interval;
    mGroupAdmin::UpdatesBetweenCheckpointValue updates_between_checkpoint;
};

struct ReplicationStyle {
    mGroupAdmin::ReplicationStyleValue style;
    ReplicationProperties properties;
};

struct Semantics {
    FT::TypeId type_id;
    LocationSeq the_locations;
    FT::InitialNumberReplicasValue initial_number_replicas;
    FT::MinimumNumberReplicasValue minimum_number_replicas;
    ReplicationStyle replication_style;
    AutomaticRecoveryValue automatic_recovery;
};
```

Figura 5.3: Estrutura Semantics

O atributo `type_id` armazena o tipo (identificador no Repositório de Interfaces CORBA) dos objetos que formam o grupo. O atributo `the_locations` contém uma lista das localidades onde as réplicas de objeto (membros do grupo) estão sendo executadas. Uma localidade é uma localização física (um *host*, um *cluster de hosts*, etc.) na qual réplicas de objetos podem ser criadas. Para um dado grupo de objetos e uma localidade, apenas um membro do grupo pode existir nessa localidade. Na implementação atual do Juggler, uma localidade é equivalente a um *host* e deve conter o nome do mesmo.

Os atributos `initial_number_replicas` e `minimum_number_replicas` armazenam, respectivamente, a quantidade inicial de réplicas instanciadas durante a criação do grupo e a quantidade mínima de réplicas que deve haver no grupo. O atributo `automatic_recovery` é utilizado para indicar se o Juggler deve realizar recuperação automática de falhas nas réplicas de objeto que formam o grupo.

Caso o atributo `automatic_recovery` indique que a recuperação de falhas não deve ser realizada pelo Juggler, o valor armazenado no atributo `minimum_number_replicas` não possui nenhum significado. Nesse caso, a quantidade de réplicas existentes no grupo deve ser garantida por algum elemento externo ao Juggler.

O estilo de replicação associado a um grupo de objetos é armazenado no atributo `replication_style`. O Juggler permite a escolha dos estilos de replicação `STATELESS`, `COLD_PASSIVE`, `WARM_PASSIVE`, `ACTIVE` e `VIRTUAL_PRIMARY_COPY`, os quais são descritos detalhadamente no capítulo 7.

No atributo `replication_style` também são armazenadas propriedades específicas do estilo de replicação em uso. A propriedade `primaries` indica a localização da(s) réplica(s) primária(s) em um grupo, enquanto que as propriedades `checkpoint_interval` e `updates_between_checkpoint` definem, respectivamente, o intervalo de tempo (em milissegundos) e o número operações executadas na(s) réplica(s) primária(s) entre transferências completas do estado desta(s) réplica(s) para as réplicas secundárias (*backup*).

5.3.2 Estruturas `GroupInfo`, `LocationInfo` e `FactoryInfo`

Uma aplicação pode obter diversas informações sobre os grupos de objetos criados através do Juggler, sobre as localizações onde o serviço está sendo executado e sobre as Fábricas de Objetos (`GenericFactory`) registradas no Juggler. Essas informações são armazenadas, respectivamente, nas estruturas `GroupInfo`, `LocationInfo` e `FactoryInfo`, as quais são ilustradas na figura 5.4.

A estrutura `GroupInfo` armazena todas as informações relativas a um grupo de objetos. Os atributos `object_group` e `the_semantics` armazenam, respectivamente, o nome e a semântica associados ao grupo. O último atributo da estrutura `GroupInfo`, denominado `the_members`, contém as informações sobre os membros do grupo. Essas informações incluem, para cada membro, a sua Referência Interoperável para Objeto (IOR), a localidade onde ele foi criado e um identificador utilizado para a posterior remoção do membro do grupo, denominado `factory_creation_id`.

As informações sobre as localidades onde o Juggler está sendo executado são armazenadas na estrutura `LocationInfo`. Estas informações incluem a identificação da localidade, bem como a arquitetura, o nome e a versão do sistema operacional que está sendo executado nela. Os dois últimos atributos definidos na estrutura `LocationInfo`, denominados `load` e `performance`, armazenam a carga e o desempenho atuais da localidade. Os valores desses atributos devem ser calculados e atualizados periodicamente por algum Serviço para Gerenciamento de Recursos (*ResourceManager*) e difundidos para o Juggler.

```
// OMG IDL
struct GroupMemberInfo {
    Object the_object;
    FT::Location the_location;
    FT::GenericFactory::FactoryCreationId factory_creation_id;
};
typedef sequence<GroupMemberInfo> GroupMemberInfoSeq;

struct GroupInfo {
    ObjectGroup object_group;
    Semantics the_semantics;
    GroupMemberInfoSeq the_members;
};

struct LocationInfo {
    FT::Location the_location;
    string os_arch;
    string os_name;
    string os_version;
    float load;
    float performance;
};

struct FactoryInfo {
    GenericFactory the_factory;
    Location the_location;
    Criteria the_criteria;
};
```

Figura 5.4: Estruturas GroupInfo, LocationInfo e FactoryInfo

A estrutura `FactoryInfo` armazena as informações sobre as Fábricas de Objetos (`GenericFactory`, vide seção 5.3.4) registradas no Juggler. Essas informações incluem a Referência Interoperável para Objeto (IOR) da `GenericFactory`, a localidade onde ela pode criar réplicas de objetos e os critérios utilizados para a criação de réplicas, os quais são específicos da `GenericFactory` e não são interpretadas pelo Juggler. A definição da estrutura `FactoryInfo` no Juggler é idêntica àquela feita no FT-CORBA [21].

5.3.3 Interface GroupManagementOperations

A interface `GroupManagementOperations` é a principal interface herdada pelo componente `GroupManager`. Ela fornece operações que tornam possível uma aplicação controlar a criação, a destruição e a modificação de grupos de objetos. Essas operações são ilustradas na figura 5.5. Além disso, essa interface também disponibiliza operações que fornecem informações sobre os grupos de objetos gerenciados pelo Juggler e sobre os membros que os

constituem, as quais são ilustradas na figura 5.6.

```
// OMG IDL
interface GroupManagementOperations {
    GroupInfo create_group(in ObjectGroup object_group,
                          in Semantics the_semantics,
                          in FT::Criteria the_criteria)
        raises(GroupAlreadyExists,
              InvalidSemantics,
              CannotMeetSemantics);
    GroupInfo add_members(in ObjectGroup object_group,
                        in LocationSeq the_locations,
                        in FT::Criteria the_criteria)
        raises(FT::ObjectGroupNotFound,
              FT::MemberAlreadyPresent,
              CannotMeetSemantics);
    GroupInfo remove_members(in ObjectGroup object_group,
                           in LocationSeq the_locations)
        raises(FT::ObjectGroupNotFound,
              FT::MemberNotFound,
              CannotMeetSemantics);
    GroupInfo set_replication_style(in ObjectGroup object_group,
                                   in ReplicationStyle the_replication_style)
        raises(FT::ObjectGroupNotFound,
              mGroupAdmin::BadReplicationStyle);
    GroupInfo set_minimum_number_replicas(
        in ObjectGroup object_group,
        in FT::MinimumNumberReplicasValue minimum_number_replicas)
        raises(FT::ObjectGroupNotFound,
              CannotMeetSemantics);
    GroupInfo set_recovery_style(in ObjectGroup object_group,
                                in AutomaticRecoveryValue automatic_recovery)
        raises(FT::ObjectGroupNotFound);
    void delete_group(in ObjectGroup object_group)
        raises(FT::ObjectGroupNotFound,
              CannotMeetSemantics);
};
```

Figura 5.5: Interface GroupManagementOperations (parte 1)

A operação **create_group** é utilizada para criar um novo grupo de objetos. O parâmetro `object_group` identifica unicamente o grupo de objetos a ser criado e o parâmetro `the_semantics` especifica a semântica que deve ser associada ao grupo de objetos (vide seção 5.3.1). O último parâmetro, `the_criteria`, possibilita a passagem de informações diretamente para as Fábricas de Objetos (*GenericFactory*, vide seção 5.3.4) responsáveis pela criação das réplicas de objeto que constituem o grupo.

Caso já exista um grupo com essa mesma identificação (`object_group`), a exceção *GroupAlreadyExists* é lançada. As outras duas exceções, *InvalidSemantics* e *CannotMeetSemantics*, serão lançadas, respectivamente, se a semântica não estiver especificada corretamente ou se pelo menos um dos critérios contidos na semântica não puder ser garantido.

Isso pode ocorrer, por exemplo, se um cliente do Juggler requisitar a criação de um membro de um grupo em uma localidade inexistente (onde o Juggler não está sendo executado).

As operações **add_members** e **remove_members** são utilizadas, respectivamente, para adicionar e remover membros de um grupo de objetos. Na operação **add_members** são passados como parâmetros a identificação do grupo, as localidades onde os novos membros devem ser criados e os critérios específicos para a inicialização dos mesmos. De forma similar, na operação **remove_members**, um cliente do Juggler deve informar a identificação do grupo e as localidades dos membros a serem removidos do grupo de objetos.

Além de possibilitar a adição e a remoção de membros de grupos de objetos, o Juggler também disponibiliza operações para a modificação dinâmica de outras propriedades de tolerância a falhas associadas a um grupo de objetos. A operação **set_replication_style** permite a modificação do estilo de replicação associado a um grupo que está em funcionamento. O protocolo que descreve a modificação dinâmica do estilo de replicação é explicado em detalhes na seção 7.4.

A operação **set_minimum_number_replicas** possibilita a especificação do número mínimo de réplicas que deve haver em um grupo, enquanto que a forma de recuperação de falhas em réplicas de objetos que compõem um grupo pode ser alterada através da operação **set_recovery_style**.

A operação **delete_group** é utilizada para destruir um grupo de objetos, ou seja, para remover todas as réplicas de objeto que o constituem. Após a destruição de um grupo, qualquer nova requisição destinada a ele será respondida com uma exceção, informando que o grupo requisitado não existe.

```
// OMG IDL
interface GroupManagementOperations {
    LocationSeq locations_of_members(in ObjectGroup object_group)
        raises(FT::ObjectGroupNotFound);
    Object get_member_ref(in ObjectGroup object_group,
        in FT::Location the_location)
        raises(FT::ObjectGroupNotFound,
            FT::MemberNotFound);
    ObjectSeq get_all_members_ref(in ObjectGroup object_group)
        raises(FT::ObjectGroupNotFound);
    GroupInfo get_group_info(in ObjectGroup object_group)
        raises(FT::ObjectGroupNotFound);
    GroupInfoSeq get_all_groups_info();
};
```

Figura 5.6: Interface GroupManagementOperations (parte 2)

A operação **locations_of_members** retorna as localidades onde se encontram as réplicas que constituem um grupo de objetos, enquanto as operações **get_member_ref** e **get_all_members_ref** retornam referências para essas réplicas. Mais precisamente, a operação **get_member_ref** retorna a referência para uma réplica que se encontra em uma de-

terminada localidade e `get_all_members_ref` retorna as referências para todas as réplicas de objeto que constituem um grupo.

As informações sobre os grupos de objetos gerenciados pelo Juggler podem ser obtidas através das operações `get_group_info` e `get_all_groups_info`. Essas operações retornam, respectivamente, as informações sobre um grupo de objetos específico e sobre todos os grupos de objetos gerenciados pelo Juggler. As informações relativas a um grupo de objetos são armazenadas em uma estrutura do tipo `GroupInfo`, descrita na seção 5.3.2.

5.3.4 Interface GenericFactory

A interface `GenericFactory`, apresentada na figura 5.7, possibilita a criação e a destruição de objetos, independente de serem replicados ou não. Essa interface deve ser implementada pelas Fábricas de Objetos Locais (`Factories`), que podem criar e destruir tanto objetos simples (*singleton*) como réplicas de objetos.

A `GenericFactory` é a principal interface herdada pelo `GroupActor`, o qual permite ao Juggler exercer o controle sobre a criação e a destruição de réplicas de objetos. Além disso, a definição da interface `GenericFactory` no Juggler é idêntica àquela feita no FT-CORBA [21].

```
// OMG IDL
interface GenericFactory {
    typedef any FactoryCreationId;
    Object create_object(in TypeId type_id,
                       in Criteria the_criteria,
                       out FactoryCreationId factory_creation_id)
        raises(NoFactory,
              ObjectNotCreated,
              InvalidCriteria,
              InvalidProperty,
              CannotMeetCriteria);
    void delete_object(in FactoryCreationId factory_creation_id)
        raises(ObjectNotFound);
};
```

Figura 5.7: Interface `GenericFactory`

As `Factories` devem ser desenvolvidas pelo programador de aplicação e devem possibilitar a criação de objetos de um ou mais tipos, em uma ou mais localidades. Em contrapartida, a interface `GenericFactory`, como implementada pelo `GroupActor`, só permite a criação de réplicas de objetos e nas localidades onde o Juggler estiver sendo executado. O `GroupActor` de fato não cria nem destrói objetos, ele apenas delega essas operações para as `Factories` registradas no Juggler (vide seção 5.3.6).

A operação `create_object` possui dois parâmetros de entrada (do tipo *in*), `type_id` e `the_criteria`. O parâmetro `type_id` é utilizado para informar o tipo do objeto a ser criado, ou seja, a interface IDL implementada pelo mesmo. O parâmetro `the_criteria`

torna possível a especificação de critérios adicionais para a criação de um objeto, como por exemplo, valores de inicialização para a implementação de objeto e a localidade onde o objeto deve ser criado. A operação `create_object` também possui um parâmetro de saída (do tipo *out*), denominado `factory_creation_id`. O valor retornado nesse parâmetro deve ser retido pela entidade que invocou a operação `create_object`, permitindo assim a posterior remoção do objeto criado através da operação `delete_object`.

Como o parâmetro `factory_creation_id` é imprescindível para a remoção de um objeto, é necessário que as `Factories` armazenem nesse parâmetro informações que permitam a remoção de um objeto, mesmo que seja através de uma Fábrica de Objetos diferente daquela que o criou. Por isso, os `GroupActors` retêm em seu estado um mapeamento entre as referências para os objetos criados e os seus respectivos valores de `factory_creation_id`.

Além disso, o valor retornado pelo `GroupActor` no parâmetro `factory_creation_id` contém informações que possibilitam o monitoramento das réplicas de objetos pelo `GroupObserver`, como descrito na interface `ReplicaMonitoring` (vide seção 5.3.5). Essas informações incluem uma referência para o objeto criado, o seu tipo, a localidade onde ele foi criado e a identificação do grupo de objetos do qual ele é membro.

5.3.5 Interfaces `ReplicaMonitoring` e `FaultNotifier`

No Juggler, as operações relativas ao monitoramento dos membros dos grupos de objetos e à notificação de suspeitas de falhas desses objetos são definidas, respectivamente, nas interfaces `ReplicaMonitoring` e `FaultNotifier`. Essas interfaces são herdadas pelo `GroupObserver` e são ilustradas na figura 5.8.

A interface `FaultNotifier` fornece um subconjunto das funcionalidades oferecidas pelo Serviço de Notificação especificado em CORBA, comumente denominado `CosNotification` [19]. Além disso, a definição dessa interface no Juggler é idêntica àquela feita no FT-CORBA [21], com as modificações propostas em [20].

Na interface `ReplicaMonitoring` são definidas apenas duas operações, denominadas `start_monitoring_replica` e `stop_monitoring_replica`. Essas operações possibilitam ao `GroupObserver` iniciar e encerrar o monitoramento das réplicas de objetos criadas pelo Juggler através do `GroupActor`. As duas operações possuem um único parâmetro, denominado `factory_creation_id`, o qual contém informações que permitem ao `GroupObserver` identificar unicamente uma réplica de objetos no contexto do Juggler.

Assim, quando o `GroupObserver` recebe uma suspeita de falha relativa a uma réplica de objeto, ele é capaz de identificar precisamente o grupo do qual a réplica é membro e enviar uma notificação sobre a falha para os objetos consumidores registrados no `FaultNotifier`.

O Serviço de Notificação fornecido pelo Juggler assume que o canal de notificação usado para a propagação de eventos é baseado no modelo *push* de comunicação, ilustrado na figura 5.9. Além disso, o canal de notificação deve permitir a propagação de eventos estruturados (`CosNotification::StructuredEvent`) e de lotes de eventos (`CosNotification::EventBatch`).

```

// OMG IDL
interface ReplicaMonitoring {
    void start_monitoring_replica(
        in FT::GenericFactory::FactoryCreationId factory_creation_id);
    void stop_monitoring_replica(
        in FT::GenericFactory::FactoryCreationId factory_creation_id);
};

interface FaultNotifier {
    typedef unsigned long long ConsumerId;
    void push_structured_fault(in CosNotification::StructuredEvent event);
    void push_sequence_fault(in CosNotification::EventBatch events);
    ConsumerId connect_structured_fault_consumer(
        in CosNotifyComm::StructuredPushConsumer push_consumer);
    ConsumerId connect_sequence_fault_consumer(
        in CosNotifyComm::SequencePushConsumer push_consumer);
    void disconnect_consumer(in ConsumerId connection)
        raises(CosEventComm::Disconnected);
    void replace_constraint(
        in ConsumerId connection,
        in CosNotification::EventTypeSeq event_types,
        in string constr_expr);
};

```

Figura 5.8: Interfaces ReplicaMonitoring e FaultNotifier

Os objetos consumidores das notificações enviadas pelo FaultNotifier devem implementar a interface CosNotifyComm::StructuredPushConsumer ou a interface CosNotifyComm::SequencePushConsumer, caso desejem receber, respectivamente, eventos estruturados ou lotes de eventos, como especificado em [19].

As operações `connect_structured_fault_consumer` e `connect_sequence_fault_consumer` permitem que objetos consumidores se registrem no FaultNotifier a fim de receberem, respectivamente, eventos estruturados e lotes de eventos. Essas operações retornam um valor do tipo *ConsumerId*, que é utilizado para identificar unicamente o objeto consumidor no contexto do FaultNotifier. Essa identificação deve ser retida pelo consumidor, pois será utilizada por este para a sua posterior desconexão do FaultNotifier, através da operação `disconnect_consumer`.

Enquanto um consumidor estiver conectado ao FaultNotifier, ele poderá invocar a operação `replace_constraint` a fim de substituir uma restrição para uma determinada sequência de tipos de eventos. Essa restrição possibilita ao consumidor escolher os tipos de eventos que deseja receber do FaultNotifier. Assim, quando o FaultNotifier recebe um evento em um dado formato, ele o propaga para todos os consumidores que reconhecem aquele formato e cujas restrições indicam que eles desejam receber o evento.

Os objetos que geram eventos a serem enviados através do `FaultNotifier` são denominados Fornecedores de Eventos. As operações `push_structured_fault` e `push_sequence_fault` tornam possível a um Fornecedor de Eventos enviar, respectivamente, eventos estruturados e eventos em lotes para os consumidores de eventos registrados no `FaultNotifier`.

Figura 5.9: Propagação de eventos através do `FaultNotifier`

Para que um Fornecedor de Eventos envie um evento, é necessário apenas que ele obtenha uma referência para o `FaultNotifier` e invoque uma das duas operações para o envio de eventos. Portanto, ao contrário dos objetos consumidores de eventos, os Fornecedores de Eventos não precisam se registrar no `FaultNotifier` a fim de propagar os eventos que eles geram.

5.3.6 Interface `ReplicationManager`

A interface `ReplicationManager` disponibiliza a visão externa do Juggler para uma aplicação cliente. Ela pode ser vista como uma Interface para o Programador de Aplicação (API) que deve ser utilizada para a criação de clientes do serviço. Estes podem fornecer uma interface “modo texto” ou até mesmo uma interface gráfica para o usuário do Juggler. A interface `ReplicationManager`, ilustrada na figura 5.10, herda as interfaces `GroupManagementOperations`, `LocationOperations` e `FactoryOperations`.

As operações herdadas da interface `GroupManagementOperations` possibilitam o gerenciamento de grupos de objetos propriamente dito, como descrito na seção 5.3.3. Na interface `LocationOperations` são especificadas operações que tornam possível a atualização e a obtenção das informações relativas às localidades onde o Juggler está sendo executado. Por outro lado, a interface `FactoryOperations` define operações que permitem a adição e a remoção do registro das Fábricas de Objetos Locais (Factories) no Juggler.

Na interface `ReplicationManager`, a operação `register_fault_notifier` permite o registro de um Notificador de Falhas (`FaultNotifier`, vide seção 5.3.5) no `ReplicationManager`.

O parâmetro `fault_notifier` contém uma referência para o `FaultNotifier` a ser registrado. A operação `get_fault_notifier` torna possível um cliente do Juggler obter uma referência para um `FaultNotifier` previamente registrado com o `ReplicationManager`. Após obter essa referência, o cliente pode se conectar com o `FaultNotifier` a fim de receber notificações sobre eventos gerados pelo Juggler.

```
// OMG IDL
interface ReplicationManager : GroupManagementOperations,
                               LocationOperations,
                               FactoryOperations {
    void register_fault_notifier(in FT::FaultNotifier fault_notifier);
    FT::FaultNotifier get_fault_notifier()
        raises(FT::InterfaceNotFound);
};

interface LocationOperations {
    void set_location_info(in FT::Location the_location,
                          in float the_load,
                          in float the_performance)
        raises(LocationNotFound);
    LocationInfo get_location_info(in FT::Location the_location)
        raises(LocationNotFound);
    LocationInfoSeq get_all_locations_info();
};

interface FactoryOperations {
    void register_factories(in FactoryInfoSeq the_factories);
    void unregister_factories(in GenericFactorySeq the_factories);
    FactoryInfoSeq get_factories_info(in FT::Location the_location)
        raises(LocationNotFound);
    FactoryInfoSeq get_all_factories_info();
};
```

Figura 5.10: Interfaces `ReplicationManager`, `LocationOperations` e `FactoryOperations`

Na interface `LocationOperations`, implementada pelo `GroupObserver`, a operação `set_location_info` possibilita a atualização da carga e a performance de uma localidade (*host*). As informações sobre as localidades onde o Juggler está sendo executado podem ser obtidas através das operações `get_location_info` e `get_all_locations_info`. Essas operações retornam as informações sobre as localidades em estruturas do tipo `LocationInfo`, descrita na seção 5.3.2.

A interface `FactoryOperations` permite a adição e a remoção do registro das Fábricas de Objetos Locais (Factories) no Juggler, através das operações `register_factories` e `unregister_factories`. Essa interface é implementada pelo `GroupActor` e também define as operações `get_factories_info` e `get_all_factories_info`, as quais retornam as informações sobre as `Factories` registradas no Juggler em estruturas do tipo `FactoryInfo` (vide seção 5.3.2).

Capítulo 6

Implementação

Este capítulo descreve a organização dos componentes do Juggler, bem como os protocolos e os aspectos de tolerância a falhas desse serviço. No desenvolvimento do Juggler, foram utilizados a linguagem Java (SDK 1.3 da *Sun Microsystems Inc.*) e o CORBA ORB VisiBroker 4.0, desenvolvido pela *Inprise Corporation*. A linguagem Java foi escolhida principalmente devido à sua portabilidade de código, o que permite o uso do Juggler em diferentes plataformas sem nenhuma modificação no código fonte. O VisiBroker foi utilizado devido ao amplo conjunto de serviços CORBA que ele implementa e também devido ao fato do OGS ter sido desenvolvido e bastante testado nesse CORBA ORB.

6.1 Replicação das Informações Gerenciadas pelo Juggler

A fim de garantir a tolerância a falhas dos componentes do Juggler, os objetos que implementam as interfaces `GroupManager`, `GroupActor` e `GroupObserver` são replicados, podendo ser instanciados em diferentes localidades (*hosts*) de uma rede.

Existem três grupos de objetos criados no OGS que em conjunto constituem o serviço Juggler. Esses grupos de objetos são denominados `MANAGERS`, `ACTORS` e `OBSERVERS` e são compostos, respectivamente, por réplicas de objetos do tipos `GroupManager`, `GroupActor` e `GroupObserver`. Esses grupos adotam o estilo de replicação `ACTIVE`, a fim de garantir que os seus membros mantenham os seus estados perfeitamente sincronizados e possibilitar um alto grau de disponibilidade do serviço Juggler. Além disso, os elementos que constituem os grupos `MANAGERS`, `ACTORS` e `OBSERVERS` se comunicam através de comunicação *não tipada* (descrita na seção 3.1.2), que possibilita um maior desempenho nas comunicações com grupos de objetos.

O grupo `MANAGERS` mantém replicadas as informações sobre todos os grupos de objetos criados pelo Juggler em uma estrutura denominada `Group Information Table (GIT)`. As informações sobre cada grupo são armazenadas em uma estrutura do tipo `GroupInfo`,

descrita na seção 5.3.2.

No grupo ACTORS são mantidas duas estruturas replicadas, a Factory Information Table (FIT) e a Object FactoryCreationId Map Table (OFMT). Na FIT são armazenadas as informações sobre as Fábricas de Objetos (GenericFactory) registradas no Juggler. Essas informações são mantidas em estruturas do tipo FactoryInfo, descrita na seção 5.3.2. A OFMT, por sua vez, contém um mapeamento entre as referências para os objetos criados pelas GenericFactory e os seus respectivos FactoryCreationIds. Com a replicação da OFMT, torna-se possível a remoção de uma réplica de objeto mesmo se a GenericFactory que o criou tiver falhado, como visto na seção 5.3.4.

No grupo OBSERVERS são replicadas as informações sobre as localidades (*hosts*) onde o Juggler está sendo executado (onde estejam criados o GroupActor e o GroupObserver) e sobre o monitoramento das réplicas de objetos criadas pelo GroupActor. As informações sobre as localidades são mantidas na Location Information Table (LIT), que armazena informações em estruturas do tipo LocationInfo (vide seção 5.3.2). A estrutura que mantém replicadas as informações sobre o monitoramento das réplicas de objetos é denominada Object Information Table (OIT). Nessa estrutura são armazenados o `factory_creation_id` associado a cada réplica e a localização do monitor que é responsável pelo monitoramento da mesma.

Nas seções 6.4 a 6.6 ficará claro como as informações replicadas nos grupos que constituem o Juggler são utilizadas nas operações para o gerenciamento de grupos de objetos.

6.2 Inicialização do Juggler

O Juggler é implementado como um aplicativo *Java* que deve ser inicializado através da linha de comando. Durante a inicialização do Juggler em um *host*, são criados um objeto do tipo GroupActor e outro do tipo GroupObserver. Esses objetos são adicionados, respectivamente, aos grupos ACTORS e OBSERVERS. Opcionalmente, um objeto do tipo GroupManager também pode ser criado. Neste caso, ele é adicionado ao grupo MANAGERS. Todos os objetos instanciados pelo Juggler em um *host* são criados em um único processo. Dessa forma, garante-se que uma falha desse processo acarretará a falha de todos os objetos do serviço instanciados naquele *host*.

A instanciação facultativa do GroupManager, isto é, a possibilidade de não instanciá-lo em todos os *hosts* nos quais o Juggler pode criar réplicas de objetos, melhora a escalabilidade do serviço, pois evita a replicação das informações contidas na GIT em todos os *hosts* nos quais o Juggler é executado. Conseqüentemente, há uma redução na quantidade de mensagens que precisam ser trocadas entre os GroupManager durante a adição, a modificação ou a remoção de informações da GIT. Assim, o Juggler pode ser configurado de acordo com os requisitos de desempenho e tolerância a falhas exigidos para o próprio serviço de gerenciamento de grupos de objetos.

Quando o `GroupManager` é instanciado em um *host*, um objeto do tipo `ReplicationManager` também é criado. Este objeto disponibiliza o acesso ao Juggler para as aplicações cliente. Todos os objetos do tipo `ReplicationManager` são registrados no Serviço de Nomes de CORBA no Contexto de Nomes (*Naming Context*) denominado “Juggler”. Para que uma aplicação cliente possa acessar o Juggler, ela deve contactar o Serviço de Nomes a fim de obter uma referência para um objeto do tipo `ReplicationManager`.

6.3 Interfaces Internas do Serviço

Nesta seção são descritas as operações que possibilitam o gerenciamento das estruturas replicadas pelos componentes do Juggler. Essas operações são especificadas nas interfaces ilustradas na figura 6.1. Na interface `GroupManager` são disponibilizadas operações que permitem a adição, a atualização e a remoção das informações sobre grupos de objetos da `Group Information Table (GIT)`.

A interface `GroupActor` especifica operações que tornam possível adicionar e remover, da `Object FactoryCreationId Map Table (OFMT)`, as informações relativas ao mapeamento entre os objetos criados pelas Fábricas de Objetos Locais (`Factories`) e os seus respectivos `FactoryCreationIds`.

Na interface `GroupObserver` são definidas operações que possibilitam o gerenciamento das informações armazenadas na `Location Information Table (LIT)` e na `Object Information Table (OIT)`. As operações `add_location_info` e `remove_location_info` permitem, respectivamente, a adição e a remoção das informações relativas às localidades onde o Juggler está sendo executado. Por fim, as operações `add_object_monitoring_info` e `remove_object_monitoring_info` tornam possível a um `GroupObserver` adicionar e remover informações sobre o monitoramento de réplicas de objetos.

As interfaces `GroupManager`, `GroupActor` e `GroupObserver` herdam a interface `mGroupAdmin::Groupable` (vide seção 3.1), a qual define as operações que possibilitam a um objeto CORBA entrar e sair de grupos de objetos criados no OGS.

A interface `GroupManager` também herda a interface `CosNotifyComm::StructuredPushConsumer`, que permite ao `GroupManager` receber notificações sobre falhas das réplicas de objetos que compõem um grupo. Essas notificações são enviadas pelo `GroupObserver`, pois ele herda a interface `mMonitoring::Notifiable`, a qual possibilita a recepção de suspeitas de falhas em réplicas de objetos. As suspeitas de falhas são fornecidas para o `GroupObserver` através do Serviço de Monitoramento do OGS.

6.4 Criação de Grupos de Objetos

Nesta seção é descrito o protocolo que possibilita ao Juggler criar grupos de objetos. A fim de facilitar a compreensão desse protocolo, a criação de um grupo de objetos denominado

```
// OMG IDL
interface GroupManager : GroupManagementOperations,
                        mGroupAdmin::Groupable,
                        CosNotifyComm::StructuredPushConsumer {
    void add_group_info(in GroupInfo group_info);
    void update_group_info(in GroupInfo group_info);
    void remove_group_info(in ObjectGroup object_group);
};

interface GroupActor : FT::GenericFactory,
                      FactoryOperations,
                      mGroupAdmin::Groupable {
    void add_object_factoryId(
        in Object the_object,
        in FT::GenericFactory::FactoryCreationId factory_creation_id);
    void remove_object_factoryId(in Object the_object);
};

interface GroupObserver : ReplicaMonitoring,
                         FT::FaultNotifier,
                         LocationOperations,
                         mGroupAdmin::Groupable,
                         mMonitoring::Notifiable {
    void add_location_info(in LocationInfo location_info);
    void remove_location_info(in FT::Location the_location);
    void add_object_monitoring_info(
        in ObjectInfo object_info,
        in MonitorLocation monitor_location);
    void remove_object_monitoring_info(in Object the_object);
};
```

Figura 6.1: Interfaces GroupManager, GroupActor e GroupObserver

“Teste” é ilustrada na figura 6.2. Esse grupo adota o estilo de replicação ACTIVE e possui dois membros, que devem ser criados nos *hosts* A e C. Além disso, eventuais falhas dos membros do grupo devem ser recuperadas automaticamente pelo Juggler.

A criação de grupos de objetos exige uma interação entre os componentes do Juggler através de requisições para objetos simples e para grupos de objetos. Algumas das requisições para grupos são enviadas de forma não-bloqueante, ou seja, o objeto que fez a requisição não espera as respostas dos objetos destinatários. Durante a execução do protocolo para a criação de grupos, as seguintes etapas podem ser identificadas:

1. Um cliente do Juggler invoca a operação **create_group** em um objeto do tipo **ReplicationManager**. Este simplesmente delega a execução da operação para um dos objetos **GroupManager**;

2. O `GroupManager` invoca a operação `create_object` no grupo `ACTORS` e espera pelas respostas de todos os membros desse grupo;
3. Cada membro do grupo `ACTORS` verifica se uma réplica de objeto deve ser criada localmente. Em caso afirmativo, um novo objeto é criado pelo `GroupActor`, através da Fábrica de Objetos Locais (`Factory`). Caso contrário, o `GroupActor` não cria o objeto e retorna a resposta da requisição `create_object` para o `GroupManager`;
4. Os `GroupActors` que criaram uma réplica do objeto invocam a operação `add_object_factoryId`, de forma não-bloqueante, no grupo `ACTORS`;
5. Cada um dos `GroupActors` invoca a operação `start_monitoring_object` no `GroupObserver` instanciado no mesmo *host* em que ele está sendo executado;
6. Cada um dos `GroupObservers` invoca a operação `add_object_monitoring_info` no grupo `OBSERVERS` de forma não-bloqueante;
7. O `GroupActor` espera a conclusão da operação `start_monitoring_object` e retorna uma referência para o objeto criado para o `GroupManager`;
8. Após a recepção das respostas de todos os `GroupActors`, o `GroupManager` armazena as informações sobre o grupo em uma estrutura do tipo `GroupInfo` (vide seção 5.3.2) e invoca a operação `add_group_info` no grupo `MANAGERS` de forma não-bloqueante;
9. O `GroupManager` retorna as informações sobre o novo grupo de objetos para o cliente do Juggler.

Durante a execução do protocolo para criação de grupos de objetos implementado no Juggler, a ocorrência de falha de um membro do grupo `ACTORS` pode impedir a criação de uma réplica de objeto. A fim de não ficar bloqueado eternamente esperando uma resposta de um `GroupActor` que falhou, o `GroupManager` sempre é informado sobre mudanças na visão do grupo `ACTORS`. Dessa forma, se o membro (ou membros) do grupo `ACTORS` que falhou não precisar criar uma réplica do objeto, então a criação do grupo é realizada com sucesso. Caso contrário, uma exceção do tipo *CannotMeetSemantics* é retornada para o cliente que invocou a operação de criação de grupo e os membros do grupo que foram criados são destruídos.

6.5 Recuperação de Falhas em Réplicas de Objetos

O Juggler implementa um protocolo que possibilita a recuperação automática de falhas em réplicas de objetos. Para facilitar a compreensão desse protocolo, um exemplo da execução do mesmo é ilustrada na figura 6.3. Nesse exemplo, supõe-se a existência de um grupo de objetos denominado “Teste”, cujos membros estão localizados nos *hosts* A e C, e que ocorre uma falha da réplica localizada no *host* C.

Da mesma forma que o protocolo para a criação de grupos de objetos, durante a execução desse protocolo algumas das requisições para grupos são enviadas de forma não-bloqueante,

Figura 6.2: Protocolo para criação de um grupo de objetos

ou seja, o objeto que fez a requisição não espera as respostas dos objetos destinatários. O protocolo para recuperação automática de falhas implementado no Juggler pode ser descrito nas seguintes etapas:

1. Quando ocorre a falha de uma réplica de objeto, o GroupObserver responsável pelo seu monitoramento recebe uma notificação sobre a falha, através do Serviço de Monitoramento do OGS;
2. O GroupObserver invoca a requisição `remove_object_monitoring_info` no grupo OBSERVERS e envia uma notificação sobre a falha da réplica de objeto para o grupo MANAGERS;
3. Um dos GroupManagers é escolhido de forma determinista para realizar o procedimento de recuperação automática de falhas (vide seção 6.7). Então, ele invoca a operação `create_object` no grupo ACTORS e espera pelas respostas de todos os membros desse grupo;
4. Cada membro do grupo ACTORS verifica se uma réplica de objeto deve ser criada localmente. Em caso afirmativo, um novo objeto é criado pelo GroupActor, através da

- Fábrica de Objetos Locais (Factory). Caso contrário, o GroupActor não cria o objeto e retorna a resposta da requisição `create_object` para o GroupManager;
5. O GroupActor que criou a réplica do objeto invoca a operação `add_object_factoryId` no grupo ACTORS de forma não-bloqueante;
 6. O GroupActor invoca a operação `start_monitoring_object` no GroupObserver instanciado no mesmo *host* em que ele está sendo executado;
 7. O GroupObserver invoca a operação `add_object_monitoring_info` no grupo OBSERVERS de forma não-bloqueante;
 8. O GroupActor espera a conclusão da operação `start_monitoring_object` e retorna uma referência para o objeto criado para o GroupManager;
 9. Após a recepção das respostas de todos os GroupActors, o GroupManager atualiza as informações sobre o grupo e invoca a operação `update_group_info` no grupo MANAGERS de forma não-bloqueante;

Figura 6.3: Protocolo para recuperação de falhas em réplicas de objetos

A recuperação de falhas simultâneas é realizada de forma seqüencial pelo Juggler, pois como descrito na etapa 3 do protocolo, o GroupManager responsável por recuperar a réplica que falhou é escolhido de forma determinista e só realiza a recuperação de uma réplica de objeto por vez. Isso garante que o estado armazenado na GIT permanece consistente mesmo quando ocorrem falhas simultâneas de réplicas de objetos.

6.6 Modificação da Semântica de Grupos

O Juggler possibilita a modificação dinâmica da semântica de grupos de objetos. Basicamente, são disponibilizadas operações que permitem alterar o número mínimo de réplicas em um grupo, o estilo de replicação a ele associado e a forma de recuperação de falhas dos membros do grupo.

As mensagens enviadas pelo Juggler para a modificação da semântica de um grupo são transmitidas com ordenação atômica, garantindo que as requisições serão processadas na mesma ordem em todos os membros do grupo cuja semântica será modificada. Assim, o Juggler assegura que mesmo quando diversos clientes requisitam, concorrentemente, operações para modificar a semântica associada a um grupo, apenas uma modificação da semântica do grupo é realizada por vez.

No entanto, o Juggler não garante que após o processamento de requisições concorrentes, as informações contidas na GIT refletem a última modificação da semântica do grupo. Isso ocorre porque, apesar das modificações da semântica do grupo serem feitas seqüencialmente, não há garantia de que as atualizações da GIT também serão efetuadas na mesma ordem que as modificações da semântica do grupo.

As mudanças do número mínimo de réplicas e da forma de recuperação de falhas dos membros de um grupo são bastante simples, pois se referem apenas à modificação de atributos definidos na semântica de um grupo, os quais são armazenados e gerenciados pelo próprio Juggler. Por outro lado, a modificação dinâmica do estilo de replicação associado a um grupo de objetos requer uma interação entre os componentes do Juggler e do OGS e é descrita em detalhes na seção 7.4.

6.7 Aspectos de Tolerância a Falhas no Juggler

Na implementação do Juggler, a replicação das informações armazenadas nos componentes GroupManager, GroupActor e GroupObserver, como descrito na seção 6.1, torna possível a recuperação de falhas desses componentes. Além disso, são implementados alguns mecanismos que possibilitam tolerância a falhas no que se refere ao monitoramento e à notificação de falhas de réplicas de objetos.

O Juggler implementa um mecanismo de “paternidade” de monitoramento, o qual garante que todas as réplicas de objetos criadas pelo Juggler estejam sempre sendo monitoradas

por algum objeto do tipo `GroupObserver`. Assim, sempre que um `GroupObserver` falha, um outro membro do grupo `OBSERVERS` assume a responsabilidade pelo monitoramento das réplicas de objetos que estavam sendo monitoradas pelo membro que falhou. O membro que assume a responsabilidade pelo monitoramento é escolhido de forma determinista pelo Juggler e é sempre o membro cuja identificação aparece na posição mediana na lista de membros do grupo.

Além disso, quando um novo `GroupObserver` é inicializado no mesmo *host* onde o membro que falhou estava localizado, ele reassume a “paternidade” de monitoramento das réplicas existentes naquele *host*. Dessa forma, o Juggler não só garante que todas as réplicas de objetos estejam sempre sendo monitoradas, mas também tenta evitar que um único `GroupObserver` fique responsável por monitorar uma grande quantidade de réplicas de objetos.

Os objetos do tipo `GroupObserver` também são responsáveis pelo envio de notificações de falhas de réplicas de objetos. Quando um `GroupObserver` recebe do Serviço de Monitoramento do OGS uma notificação sobre a falha de alguma réplica de objeto, ele envia uma mensagem de notificação de falha, através de *multicast* confiável, para o grupo `MANAGERS`. Dessa forma, mesmo que um dos membros desse grupo falhe, a notificação ainda pode ser processada por algum outro membro e a recuperação da réplica de objeto que falhou pode ser realizada. O membro do grupo `MANAGERS` que fica responsável pela recuperação automática de falhas é escolhido de forma determinista pelo Juggler e é sempre o membro cuja identificação aparece na posição mediana na lista de membros do grupo.

Capítulo 7

Extensão do OGS

O processo de extensão do OGS envolveu a conversão desse serviço do VisiBroker 3.2 for Java para o VisiBroker 4.0. Essa conversão foi necessária porque o OGS disponibilizado publicamente se baseava na já ultrapassada especificação CORBA 2.0, que define o Adaptador de Objetos denominado *Basic Object Adapter* (BOA), o qual apresenta alguns problemas de interoperabilidade entre diferentes ORBs.

Com a especificação da versão 2.3 de CORBA, foi definido o *Portable Object Adapter* (POA), um Adaptador de Objetos portátil entre diferentes ORBs. Essa versão de CORBA é implementada pelo VisiBroker 4.0 for Java e pela maioria dos CORBA ORBs disponíveis atualmente. Assim, como uma das vantagens da abordagem de serviço usada no OGS é o aspecto de interoperabilidade, realizou-se também a conversão do OGS para adequá-lo à especificação CORBA 2.3.

Apesar de fazer uso do POA, o OGS não explora de forma adequada algumas potencialidades desse Adaptador de Objetos, como a ativação automática de serventes. A criação e a manutenção de grupos de objetos no OGS exigem, além da criação de serventes, a invocação de operações específicas em processos servidores para controlar a adição e a remoção de objetos pertencentes a um grupo. Assim, o modelo adotado pelo OGS permite a criação de um único servente, para um dado tipo de objeto, por processo servidor.

Nas próximas seções são descritas as principais razões que motivaram a extensão do OGS, as extensões feitas nas interfaces desse serviço e os protocolos que permitem a utilização de diversos estilos de replicação. Por fim, na última seção deste capítulo, é apresentado o protocolo que possibilita a modificação dinâmica do estilo de replicação associado a um grupo de objetos.

7.1 Motivação para a Extensão do Serviço

Como mencionado no capítulo 3 (seção 3.5), o OGS original permite apenas a criação de grupos de objetos que adotam o estilo de replicação ativa, também conhecido como replicação

baseada em máquina de estados [37]. Nesse estilo de replicação, o protocolo para manutenção de consistência entre os membros de um grupo garante que todas as requisições destinadas a um grupo de objetos são processadas, na mesma ordem, por todos os membros que não tenham falhado. Além disso, esse protocolo garante que o estado dos membros de um grupo permanece sincronizado, isto é, idêntico, após o processamento de cada requisição e também após a falha de qualquer membro do grupo.

O protocolo para manutenção de consistência utilizado no estilo de replicação ativa pode gerar uma sobrecarga de comunicação e um atraso considerável nas respostas durante o processamento de requisições enviadas para grupos. Isso ocorre porque toda requisição tem que ser entregue e processada por cada um dos membros do grupo. Além disso, é necessário que estes executem um protocolo de consenso para garantir a ordenação total na entrega de requisições para o grupo, gerando assim um número adicional de mensagens na rede.

No entanto, para muitas aplicações é suficiente que apenas um dos membros de um grupo processe as requisições destinadas ao mesmo. Esse estilo de replicação é denominado, genericamente, replicação passiva ou replicação primária-secundária (*primary-backup*) [1]. Na prática, o estilo de replicação passiva é muito utilizado, pois ele evita o processamento redundante nos membros do grupo e a execução de um protocolo de consenso para garantir que as requisições sejam processadas na mesma ordem pelos membros do grupo.

Para atender aos requisitos dessas aplicações, no contexto desta dissertação, o OGS foi estendido com o intuito de possibilitar o uso de grupos de objetos com o estilo de replicação primária-secundária, bem como algumas variantes do mesmo. Além disso, foram desenvolvidos mecanismos que permitem a modificação dinâmica do estilo de replicação associado a um grupo de objetos, ou seja, a mudança do estilo de replicação com o grupo em funcionamento.

7.2 Modificação das Interfaces do OGS

A especificação das interfaces e estruturas definidas pelo OGS é dividida nos módulos `mGroupAccess`, `mGroupAdmin` e `mGroupProtocol`. No módulo `mGroupAccess` são agrupadas as especificações que permitem a um cliente do OGS enviar multicasts para grupos de objetos. O módulo `mGroupAdmin` agrupa as definições que tornam possível um objeto entrar e sair de um grupo. Esses dois módulos definem, portanto, a visão externa do OGS para os clientes desse serviço e para as réplicas de objetos, respectivamente.

O módulo `mGroupProtocol`, por sua vez, é utilizado para a comunicação interna entre os componentes do OGS e não é visível externamente. Nesse módulo são definidas estruturas que permitem aos componentes `GroupAdministrator` (vide seção 3.1) a manutenção de uma informação consistente sobre a visão atual de um grupo de objetos.

A fim de possibilitar o uso de replicação passiva no OGS, algumas das definições feitas nos módulos `mGroupAccess`, `mGroupAdmin` e `mGroupProtocol` foram estendidas e novas interfaces e estruturas foram especificadas, como descrito a seguir.

7.2.1 Módulo mGroupAdmin

A extensão do OGS para a disponibilização de diversos estilos de replicação exigiu algumas modificações no módulo mGroupAdmin, como ilustrado na figura 7.1. Foi especificada a estrutura `ReplicationStyle`, que permite a escolha entre os estilos de replicação `STATELESS`, `COLD_PASSIVE`, `WARM_PASSIVE`, `ACTIVE` e `VIRTUAL_PRIMARY_COPY`, descritos detalhadamente na seção 7.3 e em trabalhos como [24] e [27]. Apesar do estilo de replicação ativa com votação (`ACTIVE_WITH_VOTING`) já estar especificado no módulo mGroupAdmin, ainda não foram implementados os mecanismos de votação que permitem a utilização desse estilo de replicação.

Na estrutura `ReplicationStyle` também são armazenadas propriedades específicas do estilo de replicação em uso (`ReplicationProperties`). A propriedade `primaries` contém as Referências Interoperáveis para Objetos da(s) réplica(s) primária(s) de um grupo. As propriedades `checkpoint_interval` e `updates_between_checkpoint` definem, respectivamente, o intervalo de tempo (em milissegundos) e o número de operações executadas na(s) réplica(s) primária(s) entre duas transferências completas do estado da(s) réplica(s) primária(s) para as réplicas secundárias. Assim, dependendo dos requisitos de uma aplicação, uma forma ou outra de atualização do estado das réplicas secundárias pode ser usada.

Os valores armazenados nos atributos da estrutura `ReplicationProperties` só têm significado para grupos que adotam algum dos estilos de replicação passiva, pois para grupos com estilo de replicação `ACTIVE` ou `STATELESS` não há distinção entre réplicas primárias e secundárias. Portanto, torna-se desnecessária a atualização periódica do estado dos membros de um grupo.

No módulo mGroupAdmin também é definida a interface `ExtendedGroupAdministrator`, que herda a interface `GroupAdministrator` (vide seção 3.1) e disponibiliza duas novas operações: `set_replication_style` e `get_replication_style`. A operação `set_replication_style` permite a modificação dinâmica do estilo de replicação de um grupo, como será descrito na seção 7.4. A outra operação, denominada `get_replication_style`, retorna o estilo de replicação associado a um grupo de objetos.

A interface `GroupAdministratorFactory`, previamente especificada no OGS, foi modificada de forma a criar e destruir objetos que implementam a interface `ExtendedGroupAdministrator`, denominados *extended group administrators*. Assim, a operação `create` exige que o estilo de replicação a ser utilizado por um grupo de objetos seja especificado durante a criação de um objeto do tipo `ExtendedGroupAdministrator`.

7.2.2 Módulo mGroupAccess

No módulo mGroupAccess é definida a estrutura `Ordering`, na qual são especificados os possíveis tipos de ordenação para a entrega de multicasts para um grupo de objetos. Essa estrutura foi modificada para incorporar dois tipos de mensagens, denominados `NON_CRITICAL` e `REPLICATION_STYLE_CHANGE`, como ilustrado na figura 7.2. Na estrutura `Ordering` eram definidos, originalmente, apenas os tipos de ordenação de mensagens e não tipos de mensa-


```
// OMG IDL
module mGroupAdmin {

    typedef unsigned short ReplicationStyleValue;
    const ReplicationStyleValue STATELESS = 0;
    const ReplicationStyleValue COLD_PASSIVE = 1;
    const ReplicationStyleValue WARM_PASSIVE = 2;
    const ReplicationStyleValue ACTIVE = 3;
    const ReplicationStyleValue ACTIVE_WITH_VOTING = 4;
    const ReplicationStyleValue VIRTUAL_PRIMARY_COPY = 5;

    struct ReplicationProperties {
        sequence<mGroupAccess::Invocable> primaries_;
        TimeBase::TimeT checkpoint_interval_;
        unsigned short updates_between_checkpoint_;
    };

    struct ReplicationStyle {
        ReplicationStyleValue style_;
        ReplicationProperties properties_;
    };

    interface ExtendedGroupAdministrator : GroupAdministrator {
        void set_replication_style(in ReplicationStyle replication_style)
            raises(BadReplicationStyle);
        ReplicationStyle get_replication_style();
    };

    interface GroupAdministratorFactory {
        ExtendedGroupAdministrator create(in string group_name,
                                           in ReplicationStyle repl_style)
            raises(mGroupAccess::GroupError,
                  mGroupAccess::InvalidGroupName);
        void release(in ExtendedGroupAdministrator adm)
            raises(mGroupAccess::GroupError);
    };
};
```

Figura 7.1: Extensão do módulo mGroupAdmin

gens propriamente ditos, como proposto nesta extensão do OGS.

Uma requisição do tipo `REPLICATION_STYLE_CHANGE` é enviada para um grupo de objetos a fim de modificar o estilo de replicação associado ao mesmo. Quando um cliente deseja enviar uma requisição desse tipo, ele a difunde para todos os membros do grupo destinatário, independente do estilo de replicação associado ao grupo. Mensagens do tipo `REPLICATION_STYLE_CHANGE` são enviadas com ordenação atômica (`ATOMIC`), a qual garante que *multicasts* confiáveis são entregues, na mesma ordem, por todos os membros de um grupo.

Um cliente pode utilizar o tipo de mensagem `NON_CRITICAL` para invocar uma operação que não modifica o estado dos membros de um grupo. Esse tipo de mensagem é útil para invocar operações somente de leitura (*read-only*) em grupos de objetos que utilizam replicação passiva.

A semântica de invocação de uma requisição `NON_CRITICAL` depende do estilo de replicação utilizado pelo grupo de objetos destinatário. Para um grupo que adota replicação ativa, esse tipo de mensagem é enviado com ordenação confiável (`RELIABLE`), a qual garante que uma requisição é entregue através de multicast confiável para todos os membros de um grupo.

Para o estilo de replicação `STATELESS`, uma requisição do tipo `NON_CRITICAL` é enviada para um membro qualquer do grupo, caso o cliente tenha especificado um número de respostas igual a 0 (zero) ou 1 (um) (vide seção 3.1.2). Caso o cliente espere respostas da maioria ou de todos os membros de um grupo, então a requisição é enviada para todos os membros do grupo. Para grupos de objetos que utilizam os estilos de replicação `COLD_PASSIVE`, `WARM_PASSIVE` e `VIRTUAL_PRIMARY_COPY`, uma requisição `NON_CRITICAL` é enviada para qualquer uma das réplicas secundárias pertencentes ao grupo.

```
// OMG IDL
module mGroupAccess {

    enum Ordering {
        UNRELIABLE,
        PESSIMISTIC_RELIABLE,
        RELIABLE,
        ATOMIC,
        NON_CRITICAL,
        REPLICATION_STYLE_CHANGE
    };

    struct GroupView {
        sequence<Invocable> composition_;
        unsigned long version_;
        unsigned short replication_style_;
        sequence<Invocable> primaries_;
    };
};
```

Figura 7.2: Extensão do módulo `mGroupAccess`

Outra extensão feita no módulo `mGroupAccess` foi a modificação da estrutura `GroupView`, que armazena a visão que um cliente possui de um grupo de objetos. Foram adicionados nessa estrutura os atributos `replication_style_` e `primaries_`, os quais contêm, respectivamente, a informação sobre o estilo de replicação utilizado pelo grupo e as Referências Interoperáveis para Objetos (IORs) para a(s) réplica(s) primária(s).

O atributo `primaries_` é um subconjunto do atributo `composition_` definido originalmente no OGS, pois este armazena as IORs para todos os membros de um grupo, sejam eles réplicas primárias ou não. Assim, com as informações contidas na estrutura `GroupView`, um objeto do tipo `GroupAccessor` (vide seção 3.1) associado a um cliente de um grupo pode enviar requisições especificamente para a(s) réplica(s) primária(s) ou para qualquer uma das réplicas secundárias que compõem o grupo. O valor armazenado no atributo `primaries_` só tem significado para grupos que adotam os estilos de replicação `COLD_PASSIVE`, `WARM_PASSIVE` e `VIRTUAL_PRIMARY_COPY`, pois para grupos com estilo de replicação `ACTIVE` ou `STATELESS` não existe distinção entre réplicas primárias e secundárias.

Além dessas modificações no módulo `mGroupAccess`, os objetos do OGS que possibilitam o envio de *multicasts* para grupos de objetos (*group accessors*) foram modificados. A nova implementação desses objetos permite que um *group accessor* receba notificações sobre mudanças na visão de um grupo e no estilo de replicação associado ao grupo enquanto um *multicast* é enviado. Quando um *group accessor* recebe uma dessas notificações, ele reenvia o *multicast* para o grupo, tornando a retransmissão de requisições transparente para o cliente de um grupo de objetos. Além disso, os novos *group accessors* são implementados de forma que os clientes de um grupo não fiquem eternamente bloqueados após o envio de um *multicast* mesmo quando algum membro do grupo destinatário falha.

7.2.3 Módulo `mGroupProtocol`

No módulo `mGroupProtocol` são especificadas estruturas que tornam possível a comunicação entre os componentes `GroupAdministrator` do OGS, os quais armazenam uma visão precisa sobre o conjunto de réplicas que compõem um grupo de objetos.

Nesse módulo, foram modificadas a enumeração `Ordering` e a estrutura `ServiceView`, como ilustrado na figura 7.3. A enumeração `Ordering` define os tipos de ordenação utilizados para a comunicação entre os componentes `GroupAdministrator`. Essa enumeração foi estendida para incorporar três tipos de mensagens: `REPLICATION_STYLE_CHANGE`, `CHECKPOINT` e `STATE_UPDATE`.

Uma requisição do tipo `REPLICATION_STYLE_CHANGE` é enviada para um grupo de objetos a fim de modificar o estilo de replicação ou as propriedades de tolerância a falhas associadas ao grupo. Quando um membro de um grupo recebe uma requisição desse tipo, ele a coloca na fila de mensagens a serem processadas, independente do estilo de replicação associado ao grupo no momento em que a requisição foi recebida. Além disso, uma requisição desse tipo é processada de forma atômica pelos membros do grupo.

Os tipos de mensagens `CHECKPOINT` e `STATE_UPDATE` são enviados pela(s) réplica(s) primária(s) para as réplicas secundárias de um grupo com estilo de replicação passiva. Em uma mensagem do tipo `CHECKPOINT`, o estado completo da(s) réplica(s) primária(s) é enviado, enquanto que numa mensagem `STATE_UPDATE` apenas a última atualização de estado da réplica primária é transmitida.

```
// OMG IDL
module mGroupProtocol {

    enum Ordering {
        UNRELIABLE,
        PESSIMISTIC_RELIABLE,
        RELIABLE,
        ATOMIC,
        VIEW_CHANGE,
        REPLICATION_STYLE_CHANGE,
        CHECKPOINT,
        STATE_UPDATE
    };

    struct ServiceView {
        GroupManagerSeq gms_;
        mConsensus::ConsensusManagerSeq cms_;
        unsigned long version_;
        mGroupAdmin::ReplicationStyleValue replication_style_;
        GroupManagerSeq primaries_gms_;
        mConsensus::ConsensusManagerSeq primaries_cms_;
    };
};
```

Figura 7.3: Extensão do módulo mGroupProtocol

Outra extensão feita no módulo mGroupProtocol foi a modificação da estrutura ServiceView, que armazena a visão de um grupo sob a perspectiva dos membros que o constituem. Foram adicionados nessa estrutura os atributos replication_style_, primaries_gms_ e primaries_cms_.

O atributo replication_style_ contém a informação sobre o estilo de replicação utilizado pelo grupo. Nos atributos primaries_gms_ e primaries_cms_ são armazenadas, respectivamente, as IORs para os *group administrators* (vide seção 3.1) e para os *consensus managers* (vide seção 3.2) associados à(s) réplica(s) primária(s) do grupo de objetos.

Os atributos primaries_gms_ e primaries_cms_ são definidos para possibilitar a execução de consenso entre um subconjunto dos membros de um grupo, permitindo assim a especificação do estilo de replicação VIRTUAL_PRIMARY_COPY. Os valores armazenados nesses atributos só têm significado para grupos que adotam o estilo de replicação VIRTUAL_PRIMARY_COPY, pois nesse estilo de replicação pode haver mais de uma réplica primária em um grupo de objetos.

7.3 Novos Estilos de Replicação

Na definição original do estilo de replicação primária-secundária em [1], apenas uma réplica de um grupo, denominada cópia primária (ou réplica primária), processa as requisições destinadas ao grupo. Esse estilo de replicação também é conhecido na literatura como replicação passiva.

Após o processamento de cada requisição, a réplica primária envia uma mensagem para que todas as outras réplicas que compõem o grupo (cópias secundárias) atualizem os seus estados. Somente após receber uma confirmação sobre essa atualização de estado, a réplica primária envia uma resposta para o cliente. Além disso, caso ocorra a falha da cópia primária, um mecanismo de *failover* é executado a fim de promover uma das cópias secundárias para o papel de cópia primária. Na implementação do OGS, sempre a primeira réplica que aparece na lista de membros de um grupo será escolhida como a nova réplica primária.

A principal diferença entre os estilos de replicação ativa e passiva é a forma como as falhas são tratadas. Com replicação ativa, as falhas dos membros de um grupo são totalmente mascaradas através de um protocolo de votação entre os membros restantes no grupo. Assim, a ocorrência de falhas se torna transparente para um cliente do grupo. Por outro lado, no estilo de replicação passiva, uma requisição para um grupo pode ser perdida caso a réplica primária do grupo tenha falhado e o mecanismo de *failover* ainda não tenha sido concluído. Nesse caso, torna-se necessário um mecanismo que realize a retransmissão de requisições.

Na extensão do OGS foram implementados os estilos de replicação STATELESS e três variantes do estilo de replicação primária-secundária, denominados COLD_PASSIVE, WARM_PASSIVE e VIRTUAL_PRIMARY_COPY, os quais serão descritos a seguir. Os protocolos que implementam esses estilos de replicação supõem que os elementos do sistema (processos, processadores e objetos) falham independentemente e segundo o modelo *fail-stop* [5].

7.3.1 Replicação STATELESS

No estilo de replicação STATELESS o comportamento de um grupo de objetos não é afetado pelo histórico de invocações no grupo. Normalmente esse estilo de replicação é utilizado para a criação de grupos de objetos que disponibilizam serviços somente de leitura (*read-only*).

Em grupos que adotam esse estilo de replicação, o OGS não executa o Protocolo de Transferência de Estado (vide seção 3.1) durante a entrada de novos membros no grupo. Além disso, as requisições enviadas com ordenação ATOMIC são entregues com ordenação RELIABLE, pois a ordem em que as requisições são processadas pelos membros de um grupo que utiliza o estilo de replicação STATELESS não é relevante. Assim, evita-se a execução desnecessária de um protocolo de consenso distribuído para garantir a entrega atômica de *multicasts*.

7.3.2 Replicação COLD_PASSIVE e WARM_PASSIVE

O protocolo para replicação WARM_PASSIVE implementado no OGS é semelhante àquele proposto em [7]. Esse protocolo garante que durante o funcionamento normal de um grupo há exatamente uma réplica primária, que recebe e processa as requisições enviadas pelos clientes do grupo. No caso de falha da réplica primária, o grupo permanece sem uma réplica primária até que o mecanismo de *failover* tenha sido concluído.

No protocolo usado no OGS, o processamento de uma requisição enviada por um cliente de um grupo, pode ser descrito através das seguintes etapas, ilustradas na figura 7.4:

1. O cliente envia a requisição para a réplica primária do grupo;
2. A réplica primária processa a requisição e atualiza o seu estado. Eventualmente, a requisição é colocada em uma fila e só é, de fato, processada após todas as requisições previamente enfileiradas terem sido processadas;
3. A réplica primária envia uma mensagem sobre a atualização do seu estado para as réplicas secundárias. Esta mensagem é denominada *atualização de estado* (*state update*);
4. Sem esperar por uma confirmação sobre o recebimento da *atualização de estado* pelas réplicas secundárias, a réplica primária envia uma resposta para o cliente.

Figura 7.4: Estilo de replicação WARM_PASSIVE

A única diferença entre o protocolo para replicação WARM_PASSIVE descrito em [7] e o protocolo implementado no OGS é o comportamento de uma réplica secundária quando esta recebe uma requisição.

No protocolo proposto em [7], todas as requisições enviadas para uma réplica secundária são simplesmente descartadas e, portanto, não são processadas. Por outro lado, no protocolo implementado no OGS, uma réplica secundária pode receber e processar requisições, desde que estas não alterem o estado da réplica. Essas operações são denominadas “não-críticas” e podem ser enviadas pelo cliente de um grupo que adota replicação WARM_PASSIVE através de *multicasts* do tipo NON_CRITICAL, descrito na seção 7.2.2.

Na extensão do OGS, também foi desenvolvido um protocolo que possibilita a utilização do estilo de replicação `COLD_PASSIVE`. A principal diferença entre os estilos de replicação `COLD_PASSIVE` e `WARM_PASSIVE` é que neste último, o estado das cópias secundárias é atualizado periodicamente. Por outro lado, no estilo `COLD_PASSIVE` o estado da réplica primária é armazenado, por um mecanismo de *logging* e *checkpoint*, em um *log* distribuído e só é carregado em uma outra réplica do grupo no caso de recuperação de falha da réplica primária.

7.3.3 Replicação `VIRTUAL_PRIMARY_COPY`

No estilo de replicação `VIRTUAL_PRIMARY_COPY` [27], uma ou mais réplicas são designadas primárias e formam uma *ilha de consistência*, ou simplesmente *ilha*. Dentro da ilha de consistência é utilizado o estilo de replicação `ACTIVE`, assim o conjunto das réplicas primárias é visto externamente como uma única réplica primária virtual. As réplicas de objeto que não se encontram na ilha fazem o papel de cópias secundárias (*backup*) da réplica primária virtual.

A interação entre a cópia primária virtual e as cópias secundárias no estilo de replicação `VIRTUAL_PRIMARY_COPY` é idêntica àquela descrita no estilo de replicação `WARM_PASSIVE`. A principal vantagem do estilo de replicação `VIRTUAL_PRIMARY_COPY` em relação ao estilo `WARM_PASSIVE` é a redução do tempo necessário para a execução do mecanismo de *failover*. Isso ocorre porque, assim que uma réplica primária falha, as outras réplicas da *ilha de consistência* continuam recebendo e processando requisições normalmente.

O estilo de replicação `VIRTUAL_PRIMARY_COPY` combina, portanto, algumas das características dos estilos de replicação `ACTIVE` e `WARM_PASSIVE`. Ele possibilita a rápida recuperação de falhas de uma réplica primária, mas evita o consenso entre todos os membros de um grupo de objetos para cada requisição, o que normalmente acarreta um grande número de mensagens na rede e aumenta o tempo de resposta para o cliente.

7.4 Modificação Dinâmica do Estilo de Replicação

A modificação dinâmica do estilo de replicação de um grupo torna mais flexível a configuração de uma aplicação (serviço) baseada em grupos de objetos, permitindo que ela se adapte melhor à quantidade de recursos (número de *hosts*, largura de banda em uma rede, etc.) disponíveis e às variações na carga de uma rede.

O OGS estendido para prover essa funcionalidade possibilita ao gerente de aplicações um controle refinado sobre a replicação de serviços cuja frequência de utilização (quantidade de requisições por unidade de tempo) é bastante variável. Além disso, essa funcionalidade pode ser explorada para o gerenciamento de aplicações que permitem, por exemplo, o relaxamento temporário dos requisitos de tolerância a falhas para a obtenção de um melhor desempenho do serviço.

A figura 7.5 ilustra o funcionamento do protocolo para a mudança do estilo de replicação associado a um grupo de objetos. Nesse exemplo, é realizada a modificação do estilo de replicação **ACTIVE** para o estilo **WARM_PASSIVE**. Além disso, a figura também apresenta o que ocorre com as requisições enviadas por um cliente enquanto o estilo de replicação do grupo é modificado.

Figura 7.5: Modificação dinâmica do estilo de replicação de um grupo de objetos

O protocolo para a modificação dinâmica do estilo de replicação de um grupo de objetos, ilustrado na figura 7.5, é descrito nas seguintes etapas:

1. Um cliente do Juggler obtém uma referência para um objeto do tipo `ReplicationManager` e invoca a operação `set_replication_style`;
2. O Juggler envia um *multicast* do tipo `REPLICATION_STYLE_CHANGE` para os objetos *extended group administrators* associados aos membros do grupo de objetos;
3. Os *extended group administrators* bloqueiam a recepção de novas requisições (*multicasts*). A partir desse momento, qualquer requisição enviada por um cliente do grupo é respondida com uma exceção do tipo *ChangingReplicationStyle*. Ao receber uma

exceção desse tipo, o *group accessor*, no lado do cliente, entra em um loop para re-enviar a requisição para o grupo até que a exceção *ChangingReplicationStyle* não seja retornada;

4. Os objetos *extended group administrators* verificam se o novo estilo de replicação está especificado corretamente e atualizam o estado dos membros a eles associados, caso seja necessário. Então, os *extended group administrators* atualizam as suas estruturas internas que armazenam as informações sobre o estilo de replicação associado ao grupo e incrementam a versão da visão do grupo;
5. Os *extended group administrators* desbloqueiam a recepção de requisições (*multicasts*) e voltam a processar novos *multicasts* enviados pelo cliente do grupo;
6. O cliente reenvia a requisição para o grupo. Os *extended group administrators* percebem que a visão do grupo retida pelo cliente é antiga e retornam uma exceção do tipo *GroupViewOutOfDate*, na qual é armazenada a visão mais atual do grupo;
7. O cliente, ao receber uma exceção do tipo *GroupViewOutOfDate*, atualiza a visão que ele possui do grupo;
8. A requisição do cliente é reenviada e processada pelos membros do grupo de acordo com o novo estilo de replicação associado ao mesmo e as respostas (eventualmente, pode ser apenas uma resposta) são retornadas para o cliente.

Esse protocolo é capaz de tolerar falhas de membros de um grupo enquanto o estilo de replicação do grupo estiver sendo modificado. O pior caso pode ocorrer quando o grupo deve ser modificado para utilizar algum estilo de replicação passiva e ocorre a falha da réplica primária. Apesar do grupo ficar inicialmente sem uma réplica primária, o Serviço de Monitoramento do OGS (vide seção 3.3) garante que as réplicas secundárias receberão uma notificação sobre a falha da réplica primária. Então, uma das réplicas secundárias será promovida a réplica primária e o grupo poderá ser utilizado normalmente.

Capítulo 8

Testes

Os testes realizados com o pacote Juggler OGS-Estendido tiveram como principal objetivo medir o tempo gasto pelo Juggler para realizar as principais tarefas relativas ao gerenciamento de grupos de objetos. Essas tarefas incluem a criação de um grupo de objetos, a adição e a remoção de membros de um grupo e a recuperação automática de falhas em réplicas de objetos.

Além de medir o tempo para a realização dessas tarefas, também foi avaliado o tempo que um objeto cliente gasta para enviar *multicasts* atômicos para um grupo de objetos. Estas medidas foram feitas tanto para a situação em que o grupo mantém fixo o estilo de replicação, como também para a situação em que o estilo de replicação do grupo é alterado dinamicamente, ou seja, concomitantemente com as requisições de *multicasts* dos clientes.

Para realizar esses testes, foi desenvolvida uma aplicação que mantém informações sobre contas bancárias fictícias. Essas informações são armazenadas em objetos CORBA replicados e gerenciados através do Juggler para garantir alta disponibilidade do serviço. A figura 8.1 apresenta a definição da interface BankAccount, que representa uma conta bancária. Nesta interface são definidas as operações *deposit*, *withdraw* e *balance*, que permitem respectivamente, efetuar um depósito, efetuar um saque e obter o saldo atual da conta bancária. O estado replicado nos objetos que compõem o grupo é o saldo da conta bancária. Esse estado foi armazenado apenas em memória, a fim de evitar a possível influência dos tempos de acesso a servidores de disco.

Os testes foram realizados em uma rede local Ethernet de 10 Mbps interconectando 9 estações Sun SPARCstation. Todas as estações estavam executando o sistema operacional Solaris 7 e durante os testes essas máquinas também estavam sendo compartilhadas com outros usuários. Os testes foram realizados com a opção TCP_NODELAY do ORB ativada. Dessa forma, o ORB realiza o envio imediato de requisições para objetos, ao invés de colocá-las em um buffer e enviá-las em lotes.

Para os testes de criação de grupos, adição e remoção de membros de um grupo e recuperação automática de falhas em réplicas de objetos, foram realizadas 25 iterações de cada uma dessas tarefas. Os outros dois tipos de testes realizados (*multicasts* para grupos de


```
// OMG IDL
module BankAccount {

    exception OverDraft();

    interface Account : mGroupAdmin::Groupable,
                        mMonitoring::Monitorable {
        readonly attribute float balance;
        void deposit(in float amount);
        void withdraw(in float amount)
            raises(OverDraft);
    };
};
```

Figura 8.1: Interface BankAccount

objetos e modificação dinâmica do estilo de replicação) foram realizados uma quantidade de iterações que variou de acordo com o número de clientes acessando um grupo de objetos. Os valores ilustrados nos gráficos apresentados ao longo desta seção representam as médias dos valores obtidos nas várias iterações realizadas para cada um dos testes. Além disso, em alguns gráficos também é ilustrado o desvio padrão do tempo gasto para a realização dos testes propostos.

É importante ressaltar que os testes realizados não tiveram o objetivo de avaliar, de forma completa e exaustiva, o desempenho do serviço Juggler. Para isso, seria necessário executar esses testes em uma quantidade maior de *hosts* e realizar um número bem maior de iterações para cada tipo de teste descrito neste capítulo. Além disso, seria importante avaliar o desempenho do serviço no gerenciamento de aplicações com diferentes características, como por exemplo, aplicações em que o estado armazenado nos membros de um grupo é bem maior que um simples tipo primitivo, como o da aplicação que foi utilizada para testar o Juggler.

8.1 Criação de Grupos de Objetos

No primeiro teste realizado, foi medido o tempo total gasto pelo Juggler para criar grupos de objetos. A figura 8.2 ilustra os resultados para a criação de grupos com 1, 3, 6 e 9 membros. A fim de realizar esse teste, o serviço Juggler foi inicializado em uma quantidade de *hosts* igual ao número de réplicas de objeto em cada grupo criado. Para o caso da criação de um grupo com apenas 1 réplica, o *GroupManager* foi inicializado em apenas um *host*, mas para os grupos com 3, 6 e 9 membros, o *GroupManager* foi instanciado em 3 *hosts*.

A partir do gráfico na figura 8.2, pode-se observar que com o aumento do número de réplicas que compõem um grupo de objetos, o tempo necessário para criar o grupo aumenta de forma praticamente exponencial. Grupos com 3, 6 e 9 réplicas foram criados, em média, em aproximadamente 3,7, 15,4 e 35,1 segundos, respectivamente. Tomando-se por base grupos com 3 réplicas, o tempo gasto para criar grupos com o dobro do número de réplicas (6 réplicas) foi um pouco maior que 4 vezes. Assim, poderia-se esperar que a criação de

Figura 8.2: Tempo gasto para a criação de grupos de objetos com o Juggler

grupos com 9 membros demorasse cerca de 7 vezes mais, o que de fato não ocorreu. De fato, para criar grupos compostos por 9 réplicas, o Juggler demorou um tempo 9,5 vezes maior do que aquele gasto para a criação de grupos com 3 réplicas.

O fator que mais contribui para este aumento tão grande no tempo de criação de grupos é o acesso ao Serviço de Nomes de CORBA (*CORBA Naming Service*). Isto ocorre porque o acesso a esse serviço tem que ser feito através de um protocolo de exclusão mútua, de forma a garantir que todas as réplicas possuam uma visão consistente do grupo ao final da criação deste, ou seja, as réplicas que compõem um grupo devem ser instanciadas sequencialmente.

A linha mais inferior no gráfico da figura 8.2 representa o tempo que a última réplica de objeto teve que esperar até que ela pudesse se registrar no Serviço de Nomes. Assim, pode-se concluir que o registro sequencial no Serviço de Nomes (vide seção 3.5) é, de fato, o fator determinante para o grande aumento no tempo de criação de grupos de objetos. Além disso, também pode-se perceber que o *overhead* efetivamente causado pelo protocolo de criação de grupos implementado no Juggler é relativamente pequeno em relação ao tempo gasto para o registro das réplicas de objetos no Serviço de Nomes.

Além de se medir o tempo total para a criação de grupos de objetos, também foram avaliados os tempos gastos para a execução de cada uma das etapas do protocolo de criação de grupos. Neste gráfico, a etapa relativa ao acesso ao Serviço de Nomes não foi contabilizada, pois o que se queria avaliar era apenas a variação do tempo gasto pelos *multicasts* enviados pelos componentes do Juggler e do tempo gasto pelas Fábricas de Objetos para a criação de

réplicas de objetos durante a criação de grupos com diferentes quantidades de réplicas. O gráfico da figura 8.3 mostra os resultados desse teste para grupos com 1, 3, 6 e 9 membros.

Figura 8.3: Custo associado a cada etapa da criação de grupos de objetos

É importante salientar que os valores apresentados nesse gráfico são cumulativos. Por exemplo, os valores representados pela linha com rótulo “Multicast add_object_factory_id” incluem os tempos gastos para o envio do multicast *add_object_factory_id* e para criar a réplica de objeto propriamente dita (representado pela linha com rótulo “Tempo de criação do objeto”). Considerou-se o tempo de um *multicast* como a soma dos tempos para criar o conteúdo da mensagem *multicast* e para enviar o *multicast* para os destinatários da mensagem. Além disso, as linhas apresentadas no gráfico representam os tempos médios de realização de cada uma das etapas avaliadas.

A partir do gráfico, pode-se perceber que o custo dos vários multicasts trocados pelos componentes do Juggler são responsáveis por aproximadamente 50% do tempo (custo) total para a criação das réplicas que compõem um grupo de objetos, independente da quantidade de réplicas criadas no grupo. Além disso, o custo associado à criação de objetos e dos *multicasts* cresce de forma não-linear quando a quantidade de réplicas em um grupo aumenta.

8.2 Adição e Remoção de Membros de um Grupo

Após os testes relativos à criação de grupos de objetos, foram avaliados os tempos gastos pelo Juggler para adicionar e para remover membros de grupos de objetos. Para a realização

desses testes, foram criados grupos com 3, 6 e 9 réplicas e, em seqüência, um dos membros do grupo foi removido e novamente adicionado ao grupo. Assim, os testes para remoção de membros foram executados com grupos compostos por 3, 6 e 9 membros, enquanto que os testes relativos à adição de membros foram feitos para grupos com 2, 5 e 8 membros.

Os resultados desse conjunto de testes são ilustrados na figura 8.4. Em média, o Juggler precisou de aproximadamente 0,8 segundos para remover um membro de um grupo com 3 réplicas. Para grupos com 6 e 9 réplicas, foram gastos 2,0 e 3,7 segundos, respectivamente.

Figura 8.4: Tempo gasto para a adição e a remoção de membros de grupos de objetos

Quanto à adição de um novo membro a um grupo de objetos, o Juggler gastou 1,8 segundos para grupos com 2 membros e 2,8 segundos para grupos com 5 membros. Por fim, a adição de um membro a um grupo com 8 réplicas consumiu, em média, 5,3 segundos.

Estes dados sugerem que os tempos para adição e remoção de réplicas apresentam uma taxa de aumento não-linear com relação ao tamanho do grupo, o que se deve ao protocolo de consenso distribuído executado entre as réplicas dos componentes do Juggler. Assim, fica claro que o serviço Juggler é mais apropriado para grupos pequenos de réplicas.

8.3 Recuperação Automática de Falhas em Réplicas de Objetos

Este teste teve como objetivo avaliar o tempo gasto pelo Juggler para reagir a uma falha de uma réplica de objeto e reinstanciar um novo objeto no mesmo *host* onde o objeto que falhou estava localizado. Para isso, foram criados grupos com 1, 3, 6 e 9 réplicas e, em seguida, foi simulada a falha de um dos membros do grupo.

Na execução desses testes, o tempo necessário para a *detecção* de uma falha não foi computado, pois ele essencialmente depende de parâmetros de monitoramento definidos no OGS e pode ser ajustado para valores arbitrários em cada *host* onde o OGS é inicializado. Para manter uma uniformidade durante a execução desse teste, o intervalo entre sucessivas mensagens enviadas pelo Serviço de Monitoramento do OGS para cada réplica de objeto foi ajustado para 10 segundos.

O processo de recuperação automática de falhas no Juggler consiste basicamente de três etapas: a notificação sobre a falha para os GroupManagers que compõem o grupo MANAGERS, a escolha de qual membro deste grupo será responsável por efetuar a recuperação da falha e a criação do novo membro do grupo. Na figura 8.5 são ilustrados os resultados obtidos durante a execução de cada uma das etapas durante a recuperação de falhas em réplicas de objetos. Da mesma forma que na figura 8.3, as linhas do gráfico representam valores acumulados.

Figura 8.5: Tempo gasto para a recuperação automática da falha em uma réplica de objeto

Para grupos com 3 membros, o tempo necessário para o envio das notificações sobre falhas foi de 234 milissegundos, em média. Já os tempos para a escolha do GroupManager responsável pela recuperação da falha e para a criação de uma nova réplica de objeto foram de 115 milissegundos e 1289 milissegundos, respectivamente. Assim, o tempo total gasto pelo Juggler para recuperar uma réplica em grupos com 3 membros foi, em média, de 1640 milissegundos. Para a recuperação de uma réplica em grupos com 1 membro, esse tempo foi de 834 milissegundos e para grupos com 6 e 9 membros, este tempo aumenta para 3656 e 6042 milissegundos, respectivamente.

A partir destes resultados percebe-se que a criação do objeto é o fator que mais contribui para o tempo total de recuperação de falhas. Além disso, nota-se que, apesar da contribuição dos tempos gastos nas etapas de notificação aos GroupManagers e da escolha do GroupManager a tratar a falha, o tempo total para a recuperação de uma réplica parece aumentar de forma muito semelhante ao tempo necessário para a criação de uma réplica durante a criação de um grupo, como ilustrado no gráfico da figura 8.3.

8.4 Multicasts para Grupos de Objetos

Esse teste teve como principal objetivo avaliar o tempo gasto por um cliente de um grupo OGS para enviar um *multicast* atômico para um grupo de objetos. Esse teste foi realizado para um grupo composto por 3 membros sendo acessado simultaneamente por 1, 5, 10 e 15 clientes. Além disso, o teste foi realizado para grupos que adotam replicação ativa (*ACTIVE*) e para grupos com estilo de replicação passiva (*WARM-PASSIVE*). O “envio” de um *multicast* corresponde, na verdade, ao envio do *multicast*, o processamento da requisição pelo grupo e o retorno das respostas de todos os membros do grupo para o cliente. Os resultados desse teste são ilustrados na figura 8.6.

Para a obtenção do tempo médio de envio de um *multicast* para um grupo foram transmitidos cerca de 250 *multicasts* para o caso de 1 cliente acessando o grupo. Para 5 e 10 clientes, a quantidade total de *multicasts* enviada para o grupo foi de aproximadamente 1100. Por fim, para 15 clientes, um grupo recebeu cerca 800 *multicasts*. O tempo médio para o envio de um *multicast* atômico para um grupo com replicação ativa, foi de 173,08 e 722,53 milissegundos para 1 e 5 clientes, respectivamente. Para 10 e 15 clientes, estes valores aumentaram para 929,99 e 1170,67 milissegundos.

Para a obtenção do tempo médio de envio de um *multicast* para um grupo com replicação passiva, foram transmitidos cerca de 400 *multicasts* para o caso de 1 cliente acessando o grupo. Para 5, 10 e 15 clientes, foram enviados aproximadamente 2000 *multicasts* para o grupo. O tempo médio para o envio de um *multicast* atômico para um grupo com replicação passiva, foi de 44,46 e 478,87 milissegundos para 1 e 5 clientes, respectivamente. Para 10 e 15 clientes, estes valores aumentaram para 586,07 e 778 milissegundos.

Os resultados desse teste mostram que o tempo gasto para o envio de *multicasts* para um grupo com replicação ativa, na média, foi de 50 a 60% maior que o tempo para o envio de *multicasts* para um grupo com replicação passiva, para uma quantidade de clientes maior

Figura 8.6: Tempo gasto para a invocação de um multicast atômico

que 4. Esperava-se inicialmente que essa diferença fosse bem maior, devido à execução do protocolo de consenso distribuído entre os membros de um grupo com replicação ativa e também porque em um grupo com replicação passiva, apenas a réplica primária processa as requisições. No entanto, os valores obtidos sugerem que a execução do protocolo de consenso não acarreta uma latência tão grande no processamento das requisições ou que o *multicast* para a atualização de estado, enviado pela réplica primária para as réplicas secundárias, contribui de forma considerável para um aumento no tempo de processamento de requisições.

8.5 Modificação do Estilo de Replicação

Além das medidas dos tempos de envio de *multicasts* para grupos com replicação ativa e com replicação passiva, também foi avaliado o tempo de duração de um *multicast* enviado por um cliente quando o estilo de replicação associado a um grupo estava sendo modificado dinamicamente.

Nesse teste, foram medidos os tempos para a modificação do estilo de replicação ativa (ACTIVE) para o estilo de replicação passiva WARM-PASSIVE e vice-versa. Para a realização desse teste, foi criado um grupo com 3 membros e, enquanto o grupo recebia *multicasts* dos clientes, o estilo de replicação associado ao grupo era modificado alternadamente entre replicação ativa e replicação passiva até perfazer um total de 20 mudanças do estilo de replicação.

Para um grupo sendo acessado por 1 cliente, um *multicast* enviado durante a modificação do estilo de replicação ativa para o estilo de replicação passiva demorou em média 162,73 milissegundos. Durante a mudança de replicação passiva para ativa um *multicast* levou, em média, 267,87 milissegundos para ser processado. Para um grupo sendo acessado por 5 clientes, a mudança do estilo de replicação ativa para replicação passiva demorou 2021,46 milissegundos e de replicação passiva para replicação ativa houve uma demora de 3195,82 milissegundos.

Para grupos sendo acessados por 10 e 15 clientes, a mudança do estilo de replicação ativa para o estilo de replicação passiva demorou, respectivamente, 2161,25 e 6711,90 milissegundos. Durante a mudança do estilo de replicação passiva para o estilo de replicação ativa, no entanto, o grande número de requisições concorrentes de clientes fez com que as informações sobre o estilo de replicação do grupo (armazenadas nos objetos *extended group administrators* do OGS) se tornassem inconsistentes, ocasionando o bloqueio indefinido do grupo para a recepção de novas requisições.

Esse problema ocorre quando uma réplica primária recebe muitas requisições simultaneamente. Nesse caso, a entrega de uma requisição para modificação do estilo de replicação em uma réplica primária pode demorar mais tempo do que a entrega dessa mesma requisição nas réplicas secundárias. Dessa forma, as réplicas secundárias podem iniciar um protocolo de consenso, pois ele exige apenas as respostas da maioria simples dos membros de um grupo para ser executado, e alcançar uma decisão consensual sobre o novo estilo de replicação a ser adotado pelo grupo. Assim, alguns dos objetos *extended group administrators* associados às réplicas de objetos podem armazenar informações desatualizadas sobre o estilo de replicação utilizado pelo grupo, causando problemas durante a recepção e o processamento de novos *multicasts* enviados para o grupo.

8.6 Conclusão sobre os Testes

Os resultados dos testes discutidos ao longo deste capítulo possibilitaram identificar aspectos relevantes sobre os protocolos implementados no Juggler. Quanto à criação de grupos de objetos e adição e remoção de membros de grupos, pôde-se observar que o desempenho do serviço durante a execução dessas tarefas é influenciado diretamente pelas restrições da própria arquitetura de grupos de objetos implementada pelo OGS. Mais especificamente, o fator determinante para a demora na execução dessas tarefas é o acesso sequencial ao Serviço de Nomes para a adição (ou remoção) do registro das réplicas de objetos pertencentes a um grupo, conforme mencionado na seção 3.5.

Durante o teste para recuperação automática de falhas em réplicas de objetos, observou-se que o tempo gasto para a execução das operações de notificação da falha e escolha do objeto GroupManger responsável pela recuperação da falha representa apenas uma pequena fração do tempo total para recuperação de uma réplica. Além disso, tanto nesse teste como nos testes para criação de grupos de objetos e adição e remoção de membros de grupos foi possível observar que o tempo gasto para a execução dessas operações cresce de forma não

linear quando a quantidade de réplicas em um grupo aumenta. Isso se deve ao protocolo de envio de *multicast* confiável implementado pelo OGS, o qual gera uma quantidade de mensagens que cresce exponencialmente com o número de membros de um grupo.

Nos testes realizados para medir o tempo gasto durante o envio de *multicasts* para grupos de objetos, foi possível constatar que a latência observada por um cliente durante o envio de um *multicast* aumenta de forma considerável à medida que aumenta o número de clientes usando o grupo simultaneamente. Além disso, a latência observada por um cliente durante a modificação do estilo de replicação ativa para passiva foi até três vezes maior do que a latência durante o uso do grupo com replicação ativa.

Por outro lado, quando o estilo de replicação de um grupo foi modificado de replicação passiva para ativa, a latência observada pelos clientes de um grupo foi ainda maior do que no outro teste de modificação dinâmica do estilo de replicação. Isso ocorre porque na mudança de replicação passiva para replicação ativa, alguns fatores podem influenciar bastante o tempo total para a execução dessa operação, como por exemplo, a frequência com que é realizada a transferência do estado da réplica primária para as réplicas secundárias de um grupo.

Adicionalmente, os testes realizados permitiram observar que o protocolo para modificação dinâmica do estilo de replicação de um grupo não lida com a concorrência de requisições para um grupo enquanto o estilo de replicação a ele associado estiver sendo modificado. Nesse caso, as informações armazenadas nos *extended group administrators* podem se tornar inconsistentes, levando o grupo a um comportamento imprevisível, como por exemplo, o bloqueio indefinido para a recepção de novas requisições de clientes.

Devido principalmente aos problemas de escalabilidade apresentados pelos protocolos para o envio de mensagens *multicast* no OGS, pode-se concluir que esse serviço é mais adequado para o desenvolvimento de aplicações reais compostas por pequenos grupos de objetos. Assim, como a implementação do Juggler é baseada no uso de grupos de objetos mantidos pelo OGS, o próprio serviço para gerenciamento de grupos também apresenta as restrições de escalabilidade e desempenho impostas pelo OGS.

Capítulo 9

Conclusão

Muitas aplicações distribuídas possuem requisitos de tolerância a falhas. Por isso, vários ambientes e serviços têm sido implementados para possibilitar o desenvolvimento de aplicações que exigem um alto grau de confiabilidade e disponibilidade. Esses ambientes geralmente fornecem apenas um conjunto mínimo de mecanismos que permitem a criação e a manutenção de grupos de objetos.

No entanto, o gerenciamento de aplicações distribuídas tolerantes a falhas tipicamente requer alguns mecanismos adicionais, como a recuperação automática de falhas, a adaptação das aplicações às mudanças que ocorrem no ambiente computacional e a modificação dinâmica dos requisitos das aplicações. Assim, torna-se necessária a disponibilização de interfaces e mecanismos de alto nível para o administrador de aplicações que permitem a definição de políticas mais abrangentes e flexíveis para o gerenciamento de aplicações tolerantes a falhas.

As principais contribuições desta dissertação são a especificação e a implementação de uma infra-estrutura robusta para o gerenciamento de aplicações CORBA tolerantes a falhas, denominada Juggler, e a extensão de um serviço que permite a criação de grupos de objetos CORBA, denominado OGS.

O Juggler é um serviço para Tolerância a Falhas portátil e interoperável em diferentes CORBA ORBs que fornece uma interface de alto nível para o gerenciamento de grupos de objetos. Esse serviço possibilita o uso de diferentes estilos de replicação para grupos de objetos e a especificação de propriedades que definem, por exemplo, a localização e a quantidade de réplicas de objeto que constituem um grupo. Através do Juggler, também é possível modificar dinamicamente o estilo de replicação e outras propriedades de tolerância a falhas associadas a um grupo de objetos.

A outra contribuição dessa dissertação é a extensão das interfaces e a modificação dos componentes do OGS a fim de disponibilizar um serviço que torna possível a criação de grupos de objetos CORBA que adotam diferentes estilos de replicação. Essas extensões foram essenciais para permitir a implementação de algumas funcionalidades propostas no Juggler, como a especificação de vários estilos de replicação para grupos de objetos e a

modificação dinâmica entre eles.

No desenvolvimento da infra-estrutura para gerenciamento de aplicações CORBA tolerantes a falhas proposta nesta dissertação, foi inicialmente implementado um protótipo que disponibilizava apenas algumas das funcionalidades apresentadas neste trabalho. Esse protótipo [12] permitiu avaliar os protocolos para gerenciamento de grupos implementados pelo Juggler e identificar possíveis melhorias nos mesmos. Além disso, com a adoção do padrão FT-CORBA, tentou-se adequar algumas interfaces e mecanismos do serviço Juggler à especificação proposta pelo OMG. No entanto, a adequação total do Juggler ao padrão FT-CORBA não foi possível devido a detalhes da arquitetura para manutenção de grupos de objetos disponibilizada pelo OGS.

O pacote Juggler OGS-Estendido¹ foi implementado e testado e o seu desempenho foi avaliado sob diversos aspectos para grupos de objetos com uma pequena quantidade de membros. De maneira geral, o desempenho apresentado pelo pacote Juggler OGS-Estendido é adequado para diversos tipos de aplicações que exigem suporte para tolerância a falhas, principalmente aquelas que não necessitam de um alto desempenho nas comunicações com grupos de objetos e que não exijam processamento em tempo real.

Os resultados dos testes realizados com o Juggler foram bastante úteis para identificar os custos associados às principais tarefas relativas ao gerenciamento de grupos de objetos. Esses testes também permitiram identificar quais aspectos, durante a realização dessas tarefas, são determinantes do desempenho do serviço. Além disso, foi possível a identificação de algumas limitações e problemas do OGS, como por exemplo, o acesso sequencial ao Serviço de Nomes durante a criação de um grupo de objetos e o possível bloqueio de um grupo de objetos quando este recebe uma grande quantidade de requisições concorrentes durante a modificação dinâmica do estilo de replicação.

A experiência com o uso do OGS como a infra-estrutura básica para a criação de grupos de objetos foi considerada bastante positiva durante o desenvolvimento do Juggler. A estrutura modular do OGS permitiu identificar claramente quais aspectos desse serviço precisariam ser modificados a fim de disponibilizar o conjunto de novas funcionalidades necessárias para o desenvolvimento do Juggler. Além disso, a modificação e a extensão de alguns serviços e mecanismos disponibilizados pelos OGS possibilitaram uma melhor compreensão de como os mecanismos de Comunicação de Grupo podem ser implementados para possibilitar a criação de aplicações tolerantes a falhas.

Se comparado com outros serviços que possibilitam o gerenciamento de aplicações CORBA baseadas em grupos de objetos, pode-se considerar que a implementação atual do Juggler é satisfatória, pois esse serviço disponibiliza um conjunto de funcionalidades adequado para o gerenciamento de diversos tipos de aplicações CORBA tolerantes a falhas, inclusive aquelas aplicações que exigem a modificação dinâmica dos seus requisitos de tolerância a falhas.

¹O pacote Juggler OGS-Estendido juntamente com um manual para instalação e uso encontra-se publicamente disponível na URL <http://www.ime.usp.br/~sidam/tools/juggler.html>

Futuramente, algumas melhorias podem ser feitas no pacote Juggler OGS-Estendido, como a modificação de alguns protocolos e a implementação de alguns novos mecanismos. Os protocolos implementados pelo Juggler devem ser modificados a fim de lidar com diversas situações de falhas dos componentes do serviço durante a sua execução, evitando assim que o Juggler fique bloqueado e se torne inacessível. Esses protocolos também devem ser modificados, a fim de garantir que o serviço mantenha consistente as informações relativas aos grupos de objetos gerenciados, mesmo quando o serviço processa concorrentemente, modificações em um mesmo grupo de objetos. Além disso, é interessante a implementação de funcionalidades que permitam a filtragem dos eventos gerados e notificados através do Juggler e a criação de Domínios de Tolerância a Falhas, de forma a melhorar a escalabilidade do serviço.

Em relação ao OGS Estendido, o protocolo que possibilita a modificação dinâmica do estilo de replicação associado a um grupo deve ser modificado de forma a funcionar corretamente, mesmo quando um grupo de objetos estiver sendo acessado simultaneamente por uma grande quantidade de clientes. Isso pode envolver a modificação do protocolo de consenso implementado originalmente no OGS.

Referências Bibliográficas

- [1] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. *2nd International Conference on Software Engineering*, pages 562–570, 1976.
- [2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Subsystem for High Availability. In Dhiraj K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 76–84, Boston, MA, July 1992. IEEE Computer Society Press.
- [3] K. P. Birman. Replication and Fault-Tolerance in the Isis System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles, (10th SOSPP'85), Operating Systems Review*, pages 79–86, Orcas Island, Washington, December 1985. Published as *Proceedings of the Tenth ACM Symposium on Operating Systems Principles, (10th SOSPP'85), Operating Systems Review*, volume 19, number 5.
- [4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.
- [5] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996. ISBN 1-884777-29-5.
- [6] K. P. Birman and T. A. Joseph. Reliable Communications in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
- [7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Optimal Primary-Backup Protocols. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG '92*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378, Haifa, Israel, 2–4 November 1992. Springer.
- [8] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [9] D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.
- [10] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

- [11] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, Indiana, USA, October 1998. IEEE.
- [12] M. A. M. de Moura and M. Endler. Juggler - A Management Service for CORBA Object Groups. In *Proceedings of the 18th Brazillian Symposium on Computer Networks*, pages 227–242, Belo Horizonte, MG, May 2000. SBC.
- [13] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.
- [14] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, pages 150–161, Los Alamitos, CA, USA, October 1996. IEEE Computer Society Press.
- [15] Open Software Foundation. *Introduction to OSF/DCE*. Prentice Hall, New Jersey, 1992.
- [16] Object Management Group. CORBAServices: Common Object Services Specification. Technical Report formal/98-12-09, Object Management Group, December 1998.
- [17] Object Management Group. Fault Tolerant CORBA Using Entity Redundancy. Technical Report orbos/98-04-01, Object Management Group, April 1998. Request for Proposal.
- [18] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report formal/00-10-07, Object Management Group, October 2000. Version 2.4.
- [19] Object Management Group. CORBAServices: Notification Service Specification. Technical Report formal/00-06-20, Object Management Group, June 2000.
- [20] Object Management Group. Fault Tolerance FTF Report. Technical Report ptc/00-12-06, Object Management Group, December 2000.
- [21] Object Management Group. Fault Tolerant CORBA Specification, V1.0. Technical Report ptc/00-04-04, Object Management Group, April 2000.
- [22] M. G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.
- [23] Isis Distributed Systems Inc. and IONA Technologies Ltd. Orbix+Isis Programmer's Guide, 1995. Document D070-00.
- [24] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994. ISBN 0-13-301367-7.

- [25] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.
- [26] J. C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the 15th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-15)*, pages 2–11, Los Alamitos, CA, USA, June 1985. IEEE Computer Society Press.
- [27] R. Lenz. The “Virtual-Primary-Copy Approach” Compared to other Approaches with Weak Consistent Data Replication. In *II Workshop on Mobility and Replication, ECO-OP’96*, Linz, Austria, July 1996.
- [28] L. C. Lung, J. Fraga, and J.-M. Farines. CosNamingFT - Um Serviço de Nomes Tolerante a Falhas em Conformidade com o Padrão OMG. In *Proceedings of the 17th Brazilian Symposium on Computer Networks*, pages 549–564, Salvador, BA, May 1999. SBC.
- [29] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, Artificial Intelligence Lab., Brussels, Belgium, January 1987.
- [30] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, Switzerland, 1995.
- [31] S. Maffeis. Piranha: A CORBA Tool for High Availability. *IEEE Computer*, 30(4):59–66, April 1997.
- [32] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.
- [33] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR, USA, June 1997.
- [34] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica Consistency of CORBA Objects in Partitionable Distributed Systems. *Distributed Systems Engineering Journal*, 4(3):139–150, September 1997.
- [35] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Proceedings of the 2nd Distributed Applications and Objects (DOA) Conference*, Antwerp, Belgium, September 2000.
- [36] B. S. Sabnis. Proteus: A Software Infrastructure Providing Dependability for CORBA Applications. Master’s thesis, University of Illinois, USA, 1998.
- [37] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

REFERÊNCIAS BIBLIOGRÁFICAS

- [38] F. B. Schneider. What Good are Models and What Models are Good. In *Distributed systems (2nd edition)*, chapter 3, pages 17–26. Addison-Wesley & ACM Press, 1993. S. J. Mullender, Editor.
- [39] G. Sheu, Y. Chang, D. Liang, S. Yuan, and W. Lo. A Fault-Tolerant Object Service on CORBA. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, 1998.
- [40] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.
- [41] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.