LUELSON MARLOS NUNES

UMA INFRA-ESTRUTURA BASEADA EM FT-CORBA PARA O DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS CONFIÁVEIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas. Universidade Federal do Paraná.

Orientador: Prof. Elias Procópio Duarte Jr.

CURITIBA

2002

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno *Luelson Marlos Nunes*, avaliamos o trabalho intitulado, *"Uma Infra-Estrutura Baseada em FT-Corba para o Desenvolvimento de Aplicações Distribuídas Confiáveis"*, cuja defesa foi realizada no dia 05 de julho de 2002, às quatorze horas, no anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 05 de julho de 2002.

Prof. Dr. Elias Procópo Duarte Júnior
DINF/UFPR - Orientador

Profª. Dra. Keiko Verônica Ono Fonseca
CPGEI-CEFET/PR - Membro Externo

Prof. Dr. Marcos S. Sunye
DINF/UFPR

Sumário

Resumo

O padrão CORBA (*Common Object Request Broker Architecture*) possibilita a construção de sistemas distribuídos abertos em um modelo orientado a objetos. A especificação para tolerância a falhas do CORBA (*FT-CORBA - Fault Tolerant CORBA*) tem por objetivo fornecer suporte para aplicações que necessitam de confiabilidade. Neste trabalho apresentamos a implementação de uma infraestrutura baseada em FT-CORBA que permite a construção de aplicações distribuídas confiáveis baseadas em grupos de servidores replicados. Duas abordagens para monitoração das réplicas foram implementadas. Na primeira abordagem, apenas o servidor primário de cada grupo é periodicamente monitorado. Somente em caso de falha da réplica primária a monitoração das demais réplicas é efetuada sendo então um novo servidor primário eleito para o grupo de objetos. Na segunda abordagem, o processo de monitoração é feito periodicamente para todos os objetos de todos os grupos. Uma réplica falha é desconsiderada no momento da eleição de um novo membro primário para um grupo de objetos. Resultados experimentais mostram que a escolha do método de monitoração deve ser feita após uma avaliação do impacto de cada estratégia considerando o número total de réplicas monitoradas, bem como a banda disponível na rede.

Abstract

The CORBA Standard (*Common Object Request Broker Architecture*) allows the construction of open distributed applications based on the object oriented paradigm. The CORBA fault tolerance specification (*FT-CORBA – Fault Tolerant CORBA*) aims to support applications that need reliability. In this work we present the implementation of an infrastructure based on FT-CORBA that allows the construction of trustworthy distributed applications based on groups of replicated servers. Two monitoring approaches for the replicas were implemented. In the first approach only the primary server of each group is periodically monitored. Only in case of the primary replica being faulty, the other replicas are monitored, and then a new primary server is chosen for the group of objects. In the second approach the monitoring process is performed periodically for all objects of all groups. A faulty replica is dismissed when choosing a new primary member for a group of objects. Experimental results show that the choice of the monitoring method must be made after an evaluation of the impact of each strategy considering the total number of monitored replicas, as well as the network bandwidth.

Capítulo 1

Introdução

A computação distribuída oferece uma solução adequada para muitas aplicações que têm como requisito o compartilhamento de recursos através de uma rede de computadores, bem como a distribuição da carga de trabalho com objetivo de balanceamento ou o aumento da disponibilidade de recursos [5]. No entanto, falhas em tais sistemas não são ocorrências excepcionais, uma falha pode ser causada por problemas de hardware, *bugs* no software ou mesmo pela falta de energia, entre outros. Para que se possa minimizar os efeitos das falhas no sistema como um todo, são necessários procedimentos de tolerância à falhas, descritos no decorrer deste capítulo.

Além da computação distribuída, um outro paradigma bastante utilizado, para o desenvolvimento de aplicações, é o modelo de objetos. Este modelo combina em uma única entidade a estrutura (dados) e o seu comportamento, descrito em métodos que podem ser acessados via uma interface pública e bem definida. Uma grande vantagem do modelo de objetos é a possibilidade de separação entre a interface e a implementação dos componentes do sistema [21].

A integração entre a computação distribuída e o modelo de objetos permite a construção de sistemas de objetos distribuídos. Tal integração exige que haja interoperabilidade entre objetos distribuídos em ambientes heterogêneos. Esta interoperabilidade pode ser obtida através da utilização de padrões amplamente utilizados, entre estes padrões encontra-se o CORBA (*Common Object Request Broker Architecture*) [4]. O CORBA é independente de fornecedor, trata-se de uma especificação definida pela OMG (*Object Management Group*) entidade composta por grande número de organizações associadas que participam na elaboração das especificações.

A tolerância à falhas em sistemas distribuídos tem por objetivo permitir que o sistema como um todo continue funcionando na presença de falhas parciais, ainda que com desempenho degradado. A aplicação destas técnicas é natural devido à redundância de recursos intrínseca a estes sistemas. A especificação para tolerância à falhas do CORBA chama-se FT-CORBA (*Fault Tolerant CORBA)* [3].

O restante deste capítulo introduz conceitos de objetos distribuídos em CORBA, seguido de uma visão geral do padrão FT-CORBA.

1.1 Objetos Distribuídos Utilizando CORBA

O modelo CORBA oferece uma série de benefícios que facilitam a integração de aplicações em um ambiente distribuído. Este modelo permite que os projetistas de sistemas distribuídos tirem proveito de técnicas de orientação a objetos, como encapsulamento e herança [4]. Estas técnicas permitem que os objetos sejam capazes de executar determinadas tarefas, sem que o sistema precise conhecer o funcionamento interno destes objetos. Outro benefício é a definição clara das interfaces entre as partes do sistema, partes estas que podem ser alteradas sem afetar o sistema como um todo, obtendo desta forma uma melhor modularidade já que os módulos operam como maior independência [2].

A organização OMG (*Object Management Group*) [12] criou a arquitetura OMA (*Object Management Architecture*) com o objetivo de fornecer uma infra-estrutura conceitual a todas suas especificações. A arquitetura OMA define um módulo intermediário entre objetos clientes chamados consumidores e objetos servidores chamados fornecedores, este módulo chama-se ORB [4] (*Object Request Broker*) que é o componente chave da arquitetura CORBA.

Na arquitetura CORBA os clientes enviam requisições para o ORB solicitando algum serviço que pode ser executado por qualquer servidor que possa atender às suas requisições. No atendimento das requisições apenas o ORB precisa saber da localização tanto dos clientes quanto dos servidores na rede. A localização do servidor é transparente para o cliente e vice-versa. O ORB utiliza o componente CORBA chamado POA [4] (*Portable Object Adapter*)

para localizar, inicializar e invocar o método apropriado que deve responder à requisição do cliente.

Todos os serviços de um objeto CORBA são declarados utilizando a linguagem IDL (*Interface Definition Language*), que é uma linguagem declarativa que enfatiza a separação entre interface e implementação. A interface IDL é independente de linguagem de programação e dos protocolos de comunicação, ela é utilizada como um meio para descrever as operações, tipos de argumentos e tipos de retorno destas operações possibilitando assim a utilização dos serviços de um objeto servidor.

O uso dos serviços de um objeto servidor CORBA por um objeto cliente exige apenas que o objeto cliente conheça a interface IDL do servidor e não a linguagem de programação específica da implementação do objeto servidor. O fato de que a linguagem de programação utilizada na implementação dos métodos definidos na interface não influencia na comunicação entre os processos traz a vantagem que o código dos objetos cliente e servidor não precisam ser escritos na mesma linguagem.

Garantir a interoperabilidade é importante em um sistema distribuído. Buscando interoperabilidade entre diferentes implementações do padrão CORBA a comunicação entre estas implementações é feita via uma versão do protocolo geral entre ORBs, sendo este protocolo chamado GIOP (*General Inter-ORB Protocol*). O protocolo GIOP é um conjunto de especificações para mapear a troca de mensagens entre objetos. A versão do protocolo GIOP que roda sobre o protocolo de transporte TCP/IP (*Transmission Control Protocol/Internet Protocol*) é o protocolo IIOP (*Internet Inter-ORB protocol*).

Para que a interoperabilidade entre objetos CORBA esteja completa, todas as implementações dos objetos devem incluir suporte a um identificador único, independente de plataforma e fornecedor e que possa ser interpretado por qualquer outra implementação do padrão. O mecanismo que fornece esta característica chama-se IOR (*Interoperable Object Reference*) o qual deve ser compatível entre todas as implementações do protocolo IIOP. Quando se estiver utilizando um ambiente com múltiplos ORBs, o IOR passa a ser a única forma para se obter a referência inicial para um objeto.

Uma referência IOR contém a informação da versão IIOP implementada pelo ORB; o endereço IP e a porta do processo que contém o objeto que aguarda as requisições; uma chave única que identifica o objeto servidor; além de uma seqüência de informações adicionais aplicáveis a invocações de métodos dos objetos. Assim quando um objeto cliente obtém a referência IOR para o objeto servidor ele pode então utilizar o ORB para iniciar a comunicação com o servidor. A interoperabilidade é garantida mesmo que o cliente e o servidor estejam utilizando ORBs diferentes.

1.2 O Padrão FT-CORBA

A especificação FT-CORBA (*Fault Tolerant CORBA*) [4] foi publicada pela organização OMG em abril de 2000, onde foram definidos um conjunto de objetos que oferecem serviços genéricos para tolerância a falhas. Os serviços FT-CORBA estão divididos da seguinte forma:

Serviço de Gerenciamento de Replicação que subdivide-se em:

- Serviço de Gerenciamento de Grupos de Objetos;

- Serviço de Gerenciamento de Propriedades;
- Serviço de Fábrica de Objetos;

Serviço de Gerenciamento de Falhas que subdivide-se em:

- Serviço de Detecção de Falhas;
- Serviço de Notificação de Falhas;
- Serviço de Análise de Falhas;

Serviço de Gerenciamento de Logging e Recuperação que subdivide-se:

- Registro das Requisições;
- Atualização de Estado;

O Gerenciador de Replicação herda três interfaces de programação: a de gerenciamento de grupo, gerenciamento de propriedades e fábrica genérica de objetos. Os métodos herdados do Gerenciamento de Propriedades permitem a definição de propriedades associadas com os grupos criados pelo Gerenciador de Replicação. Os métodos herdados do Gerenciador de Grupos permitem à aplicação exercer controle sobre a adição, remoção e localização de membros de um grupo de objetos. Os métodos herdados da fábrica genérica de objetos permitem ao Gerenciador de Replicação criar e excluir grupos de objetos.

O Gerenciador de Falhas envolve as atividades de detecção, notificação e análise de falhas. A Detecção de Falhas é responsável por determinar se há falhas no sistema e gerar um relatório de falhas para o Notificador de Falhas. O notificador de falhas recebe os relatórios de falhas, filtra-os e propaga os relatórios filtrados como eventos de notificação para os consumidores que estiverem nele cadastrados.

O Analisador de Falhas avalia os relatórios de falhas que foram recebidos e produz um relatório agregado ou sumarizado das falhas o qual é enviado de volta para o Notificador de Falhas para o devido envio para outros consumidores do serviço. O Gerenciador de Replicação é um consumidor assim como o Analisador de Falhas. Uma aplicação também pode inscrever-se para receber as notificações de falha.

Existe um Notificador de Falhas por domínio de tolerância à falhas e normalmente este Notificador é replicado para tolerar falhas. Um sistema tolerante a falhas também pode ter um ou mais Analisadores de Falhas cada um deles recebendo relatórios de falhas e efetuando os diagnósticos a partir dos relatórios.

O Gerenciador de *logging* e recuperação oferece os mecanismos necessários a recuperação do estado e ações de um componente quando ocorre uma falha.

1.3 Uma Infraestrutura para o Desenvolvimento de Aplicações Distribuídas Confiáveis

A infraestrutura implementada permite a criação de aplicações distribuídas tolerantes a falhas que envolvem servidores replicados formando *grupos de objetos*. Cada *grupo de objetos contém réplicas de um mesmo* servidor sendo que uma das réplicas, a chamada réplica primária, é a responsável por responder às requisições dos clientes. As requisições são feitas pelos clientes ao grupo de objetos replicados que é o responsável por direcionar a requisição para a réplica primária daquele grupo.

A criação do grupo de objetos é feita por uma aplicação cliente através do gerenciador de replicações que utiliza o gerenciador de grupos de objetos para a criação das réplicas. A criação de novas réplicas de um grupo é feita dinamicamente com base na implementação de duas interfaces definidas pela infraestrutura e que são detalhadas no decorrer deste trabalho.

Após a sua criação, a réplica é incluída sob o contexto de grupo ao qual pertence. Este contexto será utilizado no processo de monitoração das réplicas bem como no processo de eleição de uma nova réplica primária válida e ativa, quando da ocorrência de uma falha. A réplica primária é eleita logo após a criação de todas as réplicas do grupo e a partir deste momento o grupo já se encontra em condições de responder requisições dos clientes.

O detector de falhas monitora continuamente cada grupo de objetos replicados de acordo com o intervalo de monitoração previamente definido. Caso durante o processo de monitoração verifique-se que a réplica primária de um grupo está falha uma nova réplica primária é eleita para o grupo.

Duas abordagens para monitoração das réplicas foram implementadas. Na primeira abordagem apenas o servidor primário de cada grupo é periodicamente monitorado. Somente em caso de falha da réplica primária a monitoração das demais réplicas é efetuada sendo então um novo servidor primário eleito para o grupo de objetos e configurado como servidor primário do grupo. Na segunda abordagem o processo de monitoração é feito periodicamente para todos os objetos de todos os grupos, checando se cada réplica disponível está falha. Uma réplica falha é desconsiderada no momento da eleição de um novo membro primário para um grupo de objetos.

Resultados experimentais mostram que a escolha do método de monitoração deve ser feita após uma avaliação do impacto de cada estratégia considerando o número total de réplicas monitoradas, bem como a banda disponível na rede.

1.4 Organização do Trabalho

O restante deste trabalho esta organizado da seguinte forma. O capítulo 2 apresenta uma visão geral do padrão CORBA. O Capítulo 3 apresenta o padrão FT-CORBA. O capítulo 4 descreve a implementação de uma infraestrutura tolerante a falhas baseada no FT-CORBA e os resultados experimentais obtidos. As conclusões seguem no capítulo 5.

Capítulo 2

Visão Geral do Padrão CORBA

Um sistema distribuído é uma coleção de processadores autônomos ligados em rede e executando múltiplos processos que cooperam para realizar tarefas específicas e permitem o compartilhamento de recursos. Basicamente existem quatro benefícios para utilização de sistemas distribuídos: compartilhamento de recursos, tolerância à falhas, trabalho cooperativo e performance [11].

Sistemas de objetos distribuídos são sistemas distribuídos no qual todas as entidades são modeladas como objetos. Nesta abordagem as aplicações são modeladas como um conjunto de objetos que cooperam entre si [2]. Um objeto distribuído é um componente de software que pode interoperar com outros objetos distribuídos através de linguagens, aplicações, redes e equipamentos diversos. O padrão *Common Object Request Broker Architecture* (CORBA) oferece entre outros benefícios uma forma de comunicação entre objetos distribuídos. Este capítulo descreve uma visão geral deste padrão.

2.1 A Organização OMG (*Object Management Group*)

Em 1989 foi fundada a organização *Object Management Group* (OMG) [12] que é um consórcio internacional da indústria de software. Seu propósito é criar padrões para sistemas baseados em objetos para ambientes distribuídos heterogêneos com características de reusabilidade, portabilidade e interoperabilidade.

A OMG não visa a produção de software, somente a produção de especificações. As especificações são criadas a partir de idéias e propostas discutidas pelas organizações membro. A vantagem destas especificações é que a maior parte das grandes empresas envolvidas com a computação distribuída estão entre as centenas de organizações internacionais filiadas à OMG.

A OMG publicou em 1991 a revisão 1.1 da especificação *Common Object Request Broker Architecture* (CORBA) que é uma descrição completa das interfaces e serviços que um ORB deve possuir. Desde então várias

implementações de produtos têm sido lançados no mercado baseados na especificação CORBA [6]. CORBA é a especificação de uma arquitetura e interface que permite que aplicações efetuem requisições a objetos de forma transparente e independente da linguagem de programação, sistema operacional ou mesmo da localização dos objetos.

2.2 O Modelo de Objetos CORBA

O modelo de objetos [12] fornece conceitos e a terminologia usada pela arquitetura. Um conceito básico é o de objeto CORBA. Um objeto é uma entidade que fornece serviços a clientes. Um cliente de um serviço é qualquer entidade capaz de requisitar serviços de eventos denominados requisições. Uma requisição possui informação associada que consiste basicamente da operação, do objeto destino, dos parâmetros e do contexto da requisição.

Um valor é uma instância de um tipo de dado definido no OMG IDL (que será visto adiante) que pode ser usado como parâmetro em uma requisição. Um valor que identifica um objeto é denominado nome de objeto, e uma referência de objeto é um nome de objeto que denota um objeto em particular.

Uma requisição pode ter parâmetros que podem ser de entrada, saída, ou de entrada e saída, e que são identificados pelas suas posições na requisição. Ela pode conter também um contexto que fornece informação adicional sobre a própria requisição. Uma requisição pode retornar um valor de resultado para os clientes, além de retornar os parâmetros de saída. Caso ocorra uma condição anormal, uma exceção é gerada.

Objetos CORBA podem ser criados e destruídos através de determinadas requisições. O resultado de uma criação de objeto é revelada ao cliente na forma de uma referência de objeto que denota o novo objeto.

Uma interface é uma descrição de um possível conjunto de operações que um cliente pode requisitar de um objeto. Diz-se que um objeto *satisfaz* uma interface se ele pode ser especificado como objeto destino em cada operação descrita pela interface. Uma herança de interface fornece o mecanismo de composição para permitir a um objeto suportar interfaces múltiplas.

Uma operação é uma entidade que denota um serviço que pode ser requisitado. Uma operação possui uma assinatura que, de um modo geral, descreve os valores válidos dos parâmetros, dos resultados retornados da requisição, a exceção definida pelo usuário que pode ser sinalizada para terminar uma requisição de operação, e a informação de contexto que será fornecida à implementação do objeto. Na implementação de objeto, um método é o código que é executado para fornecer o serviço e a execução de um método é denominada ativação do método.

2.3 A Arquitetura OMA (*Object Management Architecture*)

O modelo conceitual, sobre o qual são executados os trabalhos de padronização efetuados pelo OMG, é denominado *core object model* sobre uma arquitetura denominada *Object Management Architecture* (OMA). Esta arquitetura busca definir de uma forma abstrata a funcionalidade necessária à computação distribuída. No modelo OMA cada componente de software é

representado como um objeto. A OMA é composta por quatro elementos principais: Objetos de Aplicação (*Application Objects*), Serviços de objetos (*Object Services*), Ferramentas Comuns (*Common Facilities*) e o Agente de Requisição de Objetos (*Object Request Broker*), apresentados conforme a figura 1.

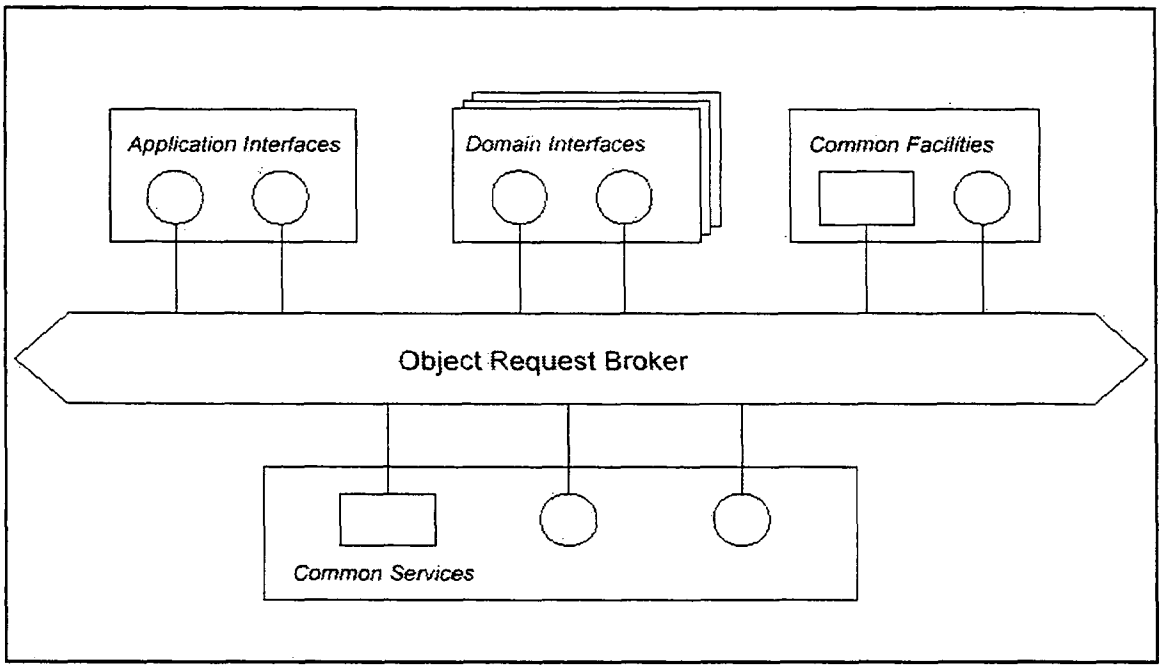


Figura 1: Modelo de Referência para a Arquitetura–OMA/OMG [14].

Os componentes da arquitetura OMA são descritos a seguir:

Object Request Broker : Quando um cliente invoca uma operação, o ORB é responsável por encontrar a implementação do objeto, ativá-lo de forma transparente, se isto for necessário, entregar a requisição do cliente para o objeto e retornar a resposta para o cliente.

Aplication Interfaces : Estas são interfaces desenvolvidas para uma aplicação específica, e devido ao fato de a OMG não desenvolver aplicações e

sim especificações, estas interfaces não são padronizadas. No entanto, com o decorrer do tempo, caso determinadas aplicações se mostrem como serviços amplamente utilizados para um determinado domínio de aplicações elas podem se tornar candidatas a futuros padrões OMG de interfaces de domínios.

Domain Interfaces: São as interfaces específicas dos Domínios, como: Finanças, Saúde, Manufatura, Telecom, Comércio Eletrônico, Transportes entre outros.

Common Facilities: uma coleção de serviços que muitas aplicações podem compartilhar, mas que não são fundamentais como os *common services*. Exemplo: Um serviço de correio eletrônico pode ser classificado como um *Common Facilities*.

Common Services : Uma coleção de serviços (interfaces e objetos) os quais suportam funções básicas para aproveitar e implementar objetos. Serviços são necessários para construir qualquer aplicação distribuída e são sempre independentes do domínio de aplicação. Exemplo : O serviço utilizado para encontrar objetos baseado no seu nome (*Naming Service*). Outros serviços também se enquadram na especificação de serviço de objetos como gerenciamento do ciclo de vida, segurança, transações, eventos entre outros [6].

2.4 A Arquitetura CORBA (*Common Object Request Broker Architecture*).

O padrão CORBA (*Common Object Request Broker Architecture*) foi criado pela OMG em 1991. Em 1995 a OMG definiu a versão 2.0 do padrão, que fez com que a interoperabilidade entre implementações de diferentes vendedores fosse mandatória [4]. Os componentes CORBA são ilustrados na figura 2 e descritos a seguir.

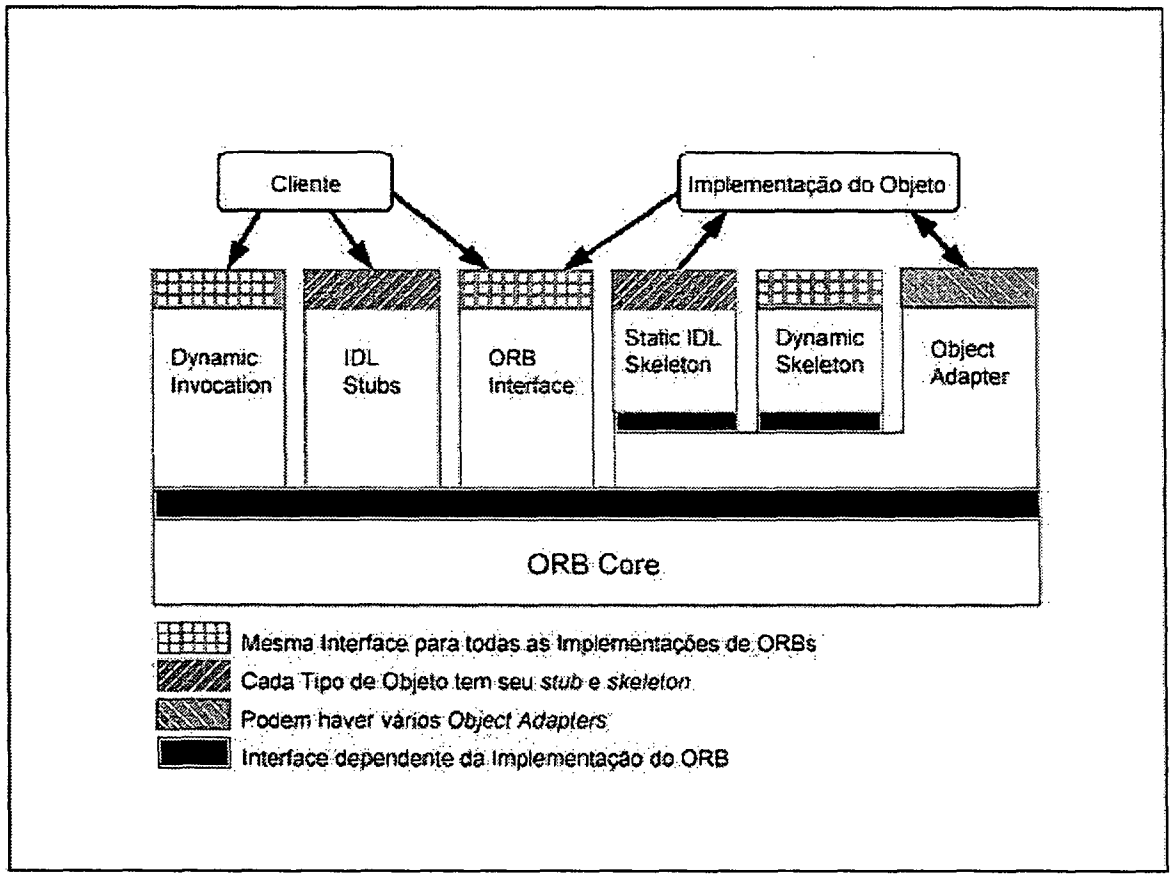


Figura 2: Os componentes da arquitetura CORBA.

Cliente: Esta é a entidade de software que invoca uma operação para uma implementação do objeto. O acesso à implementação do objeto remoto deve ser transparente para o cliente.

Implementação do Objeto: Este componente representa uma implementação servidora de um objeto que suporta as operações definidas pela IDL. Os servidores podem ser implementados em várias linguagens como C,C++, Java, Smalltalk entre outras.

Dynamic Invocation: permite que requisições sejam criadas em tempo de execução para os objetos. O cliente deve especificar qual o objeto a ser chamado, a operação desejada, o conjunto de parâmetros necessários e o modo de chamada. Estas informações estão geralmente disponíveis no repositório de interfaces. Um exemplo de aplicações que utilizam este componente são os *browsers* de objetos.

IDL *Stubs* e *Skeleton*: CORBA IDL *stubs* e *skeletons* servem respectivamente como objetos de ligação entre o cliente e aplicações servidoras, através do ORB. A transformação entre as definições de CORBA IDL e a linguagem de programação é feita automaticamente pelo compilador IDL.

ORB Interface: É a interface disponível para o cliente e a implementação acessarem o ORB *Core*. Deve ser a mesma para todas as implementações de ORBs (padrão). Como a maioria da funcionalidade do ORB é provida pelos *stubs*, *skeletons*, invocações dinâmicas ou adaptadores de objetos, apenas poucas operações comuns fazem parte da Interface do ORB.

Dynamic Skeleton: A DSI (*Dynamic Skeleton Interface*) permite a manipulação dinâmica das invocações aos objetos. *Skeletons* Dinâmicos podem

ser chamados via interfaces estáticas ou dinâmicas da mesma maneira e com os mesmos resultados. Este componente foi introduzido na especificação CORBA 2 com o principal objetivo de provar a interoperabilidade entre implementações ORBs de diferentes vendedores.

Object Adapter : Enquanto o *stub* e o *skeleton* representam o tipo formal dos objetos, o adaptador de objetos representa o estilo de implementação do objeto. Os métodos no *skeleton* devem interagir com o adaptador de objetos para iniciar a implementação no servidor e ativar o objeto se for necessário. O adaptador de objetos consiste em centralizar certas funcionalidades comuns a um conjunto de implementações. O CORBA especifica um adaptador genérico chamado *Portable Object Adapter* (POA). O *POA* tem uma interface IDL na qual estão definidas operações para criação e deleção de objetos; para obter a referência de dados associados a uma *referência de objeto*; possui também operações que assinalam quando uma implementação está pronta para utilização, ou não, e se o objeto está ativo, ou não. O *POA* está envolvido com várias partes do ciclo de vida do objeto: criação, destruição, ativação e desativação.

Object Request Broker: Objetos comunicam-se com outros objetos através do *Object Request Broker* (ORB), que é o elemento chave de comunicação. O ORB é o provedor dos mecanismos pelos quais os objetos são ativados, enviam requisições e recebem respostas. O ORB é um módulo intermediário entre clientes e servidores, apenas o ORB necessita conhecer a localização dos clientes e servidores na rede, com isso a localização do servidor é transparente para o cliente e vice-versa.

2.5 A Linguagem IDL

Todos os serviços são declarados utilizando uma linguagem declarativa chamada *Interface Definition Language* (IDL) que enfatiza a separação entre interface e implementação. A interface IDL é independente da linguagem de programação e dos protocolos de comunicação, sendo utilizada como um meio de descrever as interfaces dos objetos. A IDL pode ser utilizada como uma linguagem universal para definição de componentes de uma arquitetura de software [8]. A IDL não é, entretanto, uma linguagem de programação completa, não possuindo iteradores ou controle fluxo. A IDL permite a especificação de interfaces não fornecendo sua implementação.

Através de um compilador, que pode ser chamado de tradutor, pois o que ele faz é traduzir a IDL para alguma linguagem de programação de acordo com o mapeamento especificado na mesma, são criados os devidos códigos fonte para a linguagem utilizada. Após a execução do tradutor sobre a IDL são gerados os chamados *stubs* para o lado cliente e os chamados *skeletons* para a implementação do lado servidor.

O *stub* é o objeto local através do qual o cliente faz suas requisições. Não há nada a ser implementado no *stub*, pois o compilador IDL é responsável pelo devido mapeamento do *stub* para a linguagem adequada. O tradutor IDL também gera o código correto para a localização do *skeleton* e o código para o processo chamado de *marshaling* [6], que é a codificação dos dados para envio via rede, e codificação dos parâmetros se for necessário. O código do *stub* que

o compilador produz manipula todos os detalhes específicos da camada de transporte necessários à aplicação.

Devido ao fato dos *skeletons* serem os objetos que realmente inicializam e ativam a implementação do servidor, os *stubs* freqüentemente são chamados de objetos *proxy*. Este *proxy* é a referência para a implementação do objeto. O *skeleton* é a implementação equivalente ao *stub* do lado do servidor. O compilador IDL fornece o correto mapeamento do *skeleton* para a invocação do objeto preocupando-se com o processo de *unmarshaling* [6], que é a decodificação dos dados para utilização da informação enviada.

Os *skeletons* são chamados de *skeleton* porque eles são chamadas de métodos sem detalhes de implementação, deve-se preencher o objeto com os detalhes de implementação, adicionando código nas chamadas vazias dos métodos. Os *skeletons* também diferem dos *stubs*, pois são eles que coordenam a invocação do adaptador de objeto.

A independência de linguagem é, portanto, obtida através da construção das IDLs que permitem a implementação em diferentes linguagens de programação como "C", "C++", Ada, SmallTalk, Java e outras.

2.6 Java & CORBA

A associação da arquitetura CORBA com a linguagem de programação Java oferece uma alternativa para a implementação de sistemas baseados em objetos distribuídos. Aplicações java com a tecnologia CORBA proporcionam sistemas de objetos móveis portáveis entre qualquer plataforma de desenvolvimento. Java tem com principais características: portabilidade, suporte

multithread, sistema de *garbage collection* e gerenciamento de erros [13], o que facilita o desenvolvimento de aplicações robustas em redes de objetos.

Java complementa CORBA. CORBA oferece transparência entre redes e comunicação com sistemas diferenciados enquanto Java oferece transparência de implementação para plataformas diferenciadas. CORBA fornece a ligação entre as aplicações portáveis Java e o ambiente distribuído de objetos. Desta forma, Java é a linguagem de programação selecionada para a implementação associada a esta dissertação.

Capítulo 3

Tolerância a Falhas & CORBA

Aplicações de objetos distribuídos podem ser tolerantes a falhas através da replicação dos objetos que as compõe e da distribuição destes objetos em diferentes computadores na rede. A idéia da replicação de objetos é que a falha de uma réplica, ou do processador que hospeda a réplica, de objeto possa ser escondida do objeto cliente, de forma que outra réplica continue executando qualquer operação que o cliente precise do objeto servidor.

O padrão FT-CORBA fornece conceitos usados para o gerenciamento, controle e recuperação de falhas para o serviço de tolerância à falhas, estes

conceitos bem como os componentes da especificação de tolerância a falhas CORBA são descritos neste capítulo.

3.1 Conceitos

Esta seção descreve conceitos básicos de tolerância a falhas e replicação de objetos distribuídos.

3.1.1 Replicação e Grupos de Objetos

Para tornar um objeto tolerante a falhas, várias réplicas do objeto são criadas e gerenciadas por um *grupo* de objetos. Cada réplica de um objeto tem sua própria referência de objeto, o chamado *Interoperable Object Reference* (IOR). Para um grupo de objetos uma referência adicional é introduzida para o grupo de objetos como um todo, sendo esta referência chamada de *Interoperable Object Group Reference* (IOGR).

A referência IOGR é disponibilizada para uso dos objetos clientes. Assim o objeto cliente invoca métodos no grupo de objetos servidor. Os membros do grupo de objetos servidor executam os métodos requisitados e retornam as respostas aos clientes da mesma forma que um objeto convencional o faria.

Devido à abstração de grupo de objetos, os objetos clientes não precisam saber que os objetos servidores estão replicados, esta característica chama-se transparência de replicação. Os objetos clientes também não precisam saber se houveram falhas nos objetos servidores, ou se houve uma recuperação de falha, esta característica chama-se transparência de falhas.

3.1.2 Estilos de Replicação

Os estilos de replicação definem como os objetos replicados são atualizados. Os estilos de replicação estão divididos em : *replicação passiva fria, replicação passiva quente, replicação ativa e replicação sem estado* [3]. Para o estilo de replicação passiva fria (*cold passive replication style*) ou passiva quente (*warm passive replication style*), somente o membro primário, que é o membro que oferece os serviços, executa os métodos invocados no grupo de objetos replicados. O grupo de objetos contém membros *backup* adicionais que serão recuperados caso uma falha ocorra. Para o estilo de replicação ativa todos os membros do grupo de objetos executam os métodos invocados. Para o estilo de replicação sem estado, o comportamento do grupo de objetos não é afetado pela história das invocações, um exemplo típico é um servidor que fornece acesso de somente leitura a um conjunto de informações.

Para o estilo de replicação passiva fria, o estado do membro primário é extraído de um *log* e carregado para o membro *backup* quando há necessidade de recuperação. Para o estilo de replicação passiva quente, o estado do membro primário é carregado para um ou mais membros *backup* periodicamente durante operações normais. Para o estilo de replicação ativa todos os membros de objetos executam os métodos invocados, mas apenas um dos objetos replicados responde à requisição feita. No estilo de replicação sem estado, não é necessário o controle de estado das réplicas.

3.1.3 Domínios de Tolerância à Falhas

Aplicações que necessitam de tolerância à falhas podem ser grandes e complexas. Gerenciar uma aplicação tolerante a falhas como uma única entidade em um sistema de objetos distribuídos não é apropriado [3]. Assim vários domínios de tolerância à falhas são definidos, conforme exemplificado na figura 3.

Cada domínio de tolerância à falhas contém vários *hosts* e grupos de objetos. Na figura 3 os domínios de tolerância à falhas são mostrados envolvidos pelas linhas pontilhadas. Os *hosts* são mostrados envolvidos pelas linhas contínuas e cada membro de um grupo de tolerância à falhas é mostrado com o mesmo tom de cinza. Assim o grupo B é constituído pelos objetos B1, B2 e B3, o grupo C é constituído pelos objetos C1, C2 e C3 e o grupo D é constituído pelos objetos D1, D2, D3 e D4.

Todos os grupos de objetos dentro de um domínio de tolerância à falhas são criados e gerenciados por um único Gerenciador de Replicações. Os grupos de objetos podem ser invocados por objetos dentro de outro domínio de tolerância à falhas. O conceito de domínio de tolerância à falhas confere escalabilidade às aplicações, permitindo que um grande número de objetos seja dividido em pequenos números de objetos que são gerenciados através de cada Gerenciador de Replicação.

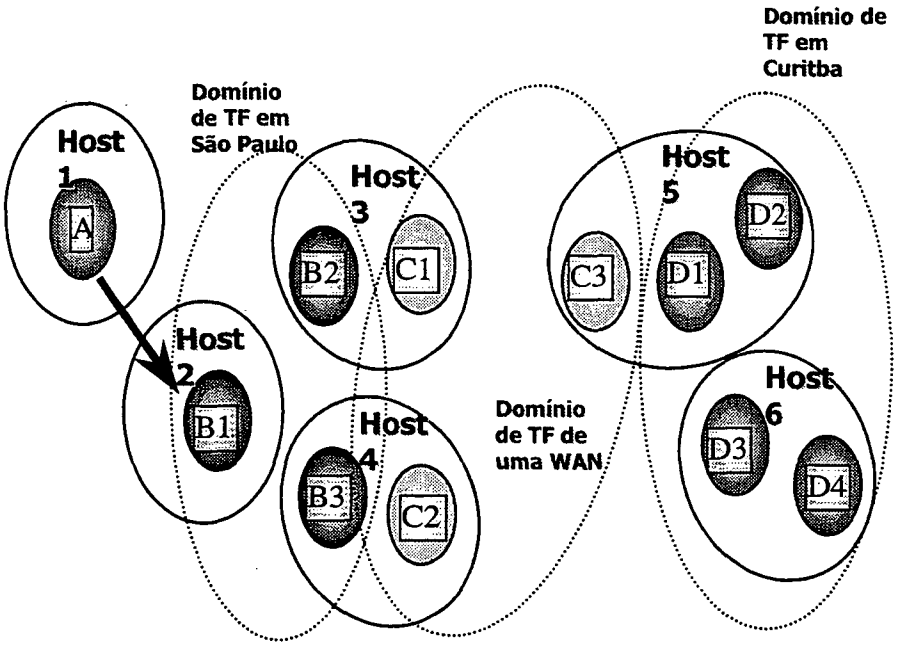


Figura 3 : Domínios de Tolerância a Falhas.

3.1.4 Propriedades de Tolerância a Falhas

Cada grupo de objetos possui um conjunto de propriedades que estão associadas a ele. Como por exemplo, propriedades que se referem ao estilo de replicação (passiva fria, passiva quente, ativa, sem estado etc), número inicial de réplicas, número mínimo de réplicas, etc.

É possível que sejam definidas propriedades para o serviço de tolerância à falhas que possam ser aplicadas para todos os grupos de objetos dentro de um domínio de tolerância à falhas ou para todos os objetos de um tipo específico. Também é possível configurar as propriedades de um grupo de

objetos quando o grupo é criado, e eventualmente mudar as propriedades dinamicamente depois que o grupo de objetos já foi criado.

3.1.5 Consistência de Réplicas

Consistência de réplicas tem como requisito que os estados de um grupo de objetos estejam consistentes quando os métodos são invocados no grupo de objetos e quando ocorrem falhas.

Para o estilo de replicação ativa, forte consistência de réplicas significa que no fim de cada invocação de métodos no grupo de objetos todos os membros têm o mesmo estado. Tanto para a replicação passiva fria como para a replicação passiva quente, no fim de cada transferência de estado todos os membros do grupo possuem o mesmo estado.

A consistência de réplicas é atingida através do uso de agrupamento forte de objetos bem como unicidade do objeto primário para o estilo de invocação passiva [10]. Agrupamento forte de objetos significa que para cada método invocado no grupo de objetos a infraestrutura de tolerância à falhas em todos os *hosts* têm a mesma visão dos membros do grupo de objetos. Unicidade do objeto primário significa que um e somente um membro do grupo de objetos executa os métodos invocados no grupo de objetos.

3.2 Visão Geral do Padrão *Fault-Tolerant* CORBA (FT-CORBA)

Redundância de entidades, detecção e recuperação de falhas são requisitos para um sistema tolerante a falhas [9]. A especificação para tolerância à falhas CORBA (FT-CORBA) tem por objetivo fornecer um suporte para aplicações que necessitam de um alto grau de confiabilidade e que necessitam de mais do que um único servidor *backup*.

A redundância de entidades através da qual a especificação oferece tolerância à falhas é a replicação de objetos através de grupos. Comparando-se a estratégia de réplicas de servidor com a estratégia de replicação de objetos através de grupos temos que a estratégia de replicação de objetos através de grupos tem como objetivo obter maior flexibilidade no gerenciamento de configurações do número de réplicas, no que se refere a diferentes *hosts*, o que não ocorre na estratégia de réplicas de servidor. É importante observar que objetos replicados podem invocar métodos de outros objetos replicados sem ter conhecimento da localização dos mesmos.

O padrão suporta um conjunto de estratégias de tolerância à falhas, incluindo reenvio de requisição, redirecionamento para um servidor alternativo, replicação passiva e replicação ativa, que permite recuperação mais rápida de eventuais falhas. O padrão permite que o usuário defina as propriedades de tolerância à falhas para grupos de objetos replicados, ou para todos os grupos de um tipo específico quando da criação do grupo de objetos, ou mudar as propriedades dinamicamente depois que o grupo é criado.

O padrão suporta aplicações que necessitam de uma infraestrutura para controlar a criação de réplicas de objetos da aplicação assim como aplicações que controlam diretamente a criação de réplicas de seus objetos. Ele suporta também aplicações que requerem infraestrutura de tolerância à falhas com forte consistência de réplicas sob condições normais e sob condições de falha.

O padrão oferece também suporte para detecção, notificação e análise de falhas de objetos. A figura 4 apresenta uma visão geral da arquitetura de um sistema tolerante a falhas, mostrando um exemplo de estratégia para implementação da especificação FT-CORBA [3]. O gerenciamento de tolerância à falha do padrão FT-CORBA divide-se em três grupos: Gerenciamento de Replicação, Gerenciamento de Falhas e Gerenciamento de *Log* e Recuperação [1].

Na parte de cima da figura 4 são apresentados vários componentes da infraestrutura de tolerância a falhas (Gerenciador de Replicação, Notificador de Falhas e Detector de Falhas), todos os quais são implementados como objetos CORBA. O Gerenciador de Replicação herda as interfaces do Gerenciador de Propriedades do Gerenciador de Grupo de Objetos e da Fabrica Genérica [3]. Existe uma única instância do Gerenciador de Replicação e Notificador de Falhas em cada domínio de tolerância à falhas, mas, fisicamente, eles podem estar replicados para proteger contra falhas, apenas como objetos de aplicação.

A parte de baixo da figura 4 mostra três *hosts* sendo um cliente no *host*1 o qual invoca um objeto servidor com duas réplicas: Servidor1 no *host*2 e Servidor2 no *host*3. São mostrados também os objetos Fábrica e DF (Detector de Falhas) que se apresentam em cada *host* sendo que estes componentes são

específicos para cada um dos *hosts*. Os objetos específicos do *host* não são replicados, diferentemente dos objetos mostrados no topo da figura 4 os quais são objetos replicados.

A parte de baixo da figura 4 também apresenta o mecanismo de *logging* e *Checkpoint* e o mecanismo de recuperação que não são objetos CORBA, mas devem fazer parte do ORB, ou devem estar localizados entre o ORB e o sistema operacional.

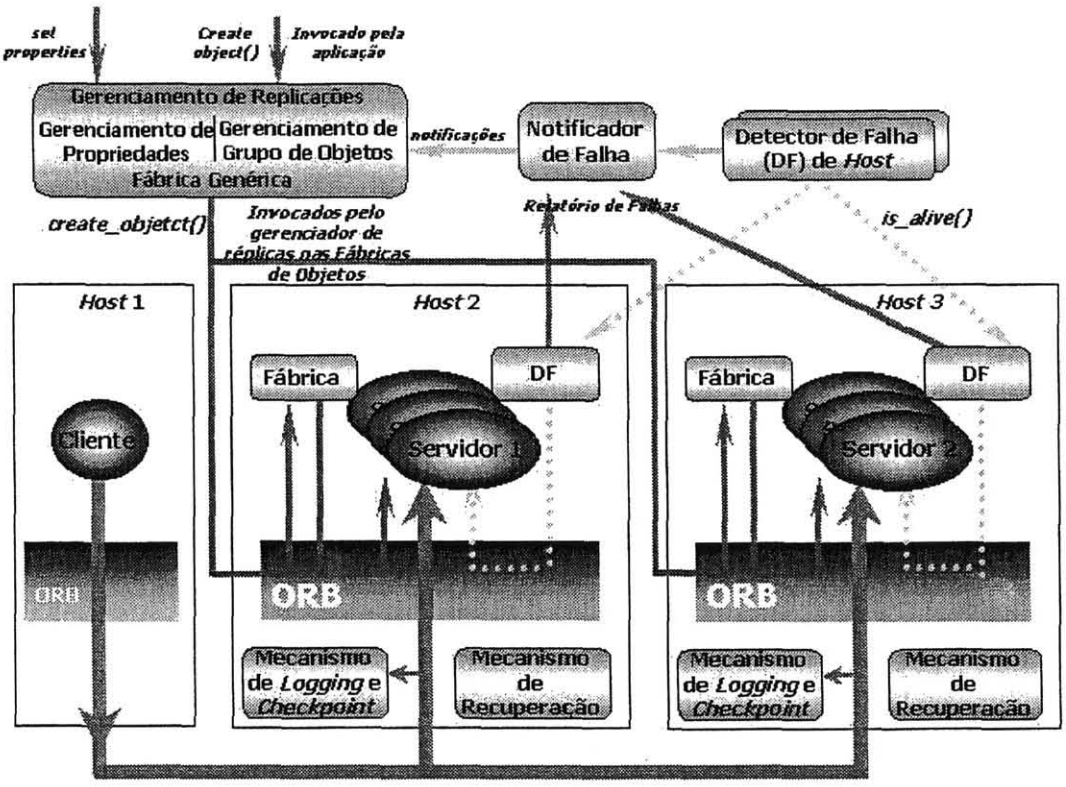


Figura 4 : Arquitetura FT-CORBA.

Como visto anteriormente o Gerenciador de Replicação comporta três gerenciamentos: o Gerenciamento de Propriedades, o Gerenciamento de Grupo de Objetos e a Fábrica Genérica. Temos também na arquitetura FT-CORBA o

Mecanismo de Recuperação e *Logging*. Estes componentes são descritos a seguir.

3.2.1 O Gerenciamento de Propriedades

O Gerenciamento de Propriedades permite ao usuário definir as propriedades de tolerância à falhas dos grupos de objetos. A especificação da interface do Gerenciamento de Propriedades foi projetada de forma a permitir o desenvolvimento de interfaces gráficas e para definir propriedades adicionais caso isto seja desejado.

Duas propriedades de grande relevância são a forma de gerenciamento de grupo e o estilo de consistência. A forma de gerenciamento de grupo define se um grupo de objetos será controlado pela infraestrutura ou pela aplicação. De forma similar o estilo de consistência define se a consistência dos estados dos membros de um grupo de objetos será controlada pela infraestrutura ou será controlada pela aplicação.

3.2.2 O Gerenciamento de Grupo de Objetos

O Gerenciamento de Grupo de Objetos proporciona a utilização de métodos de criação, adição, remoção e localização de membros de um grupo de objetos permitindo assim que a aplicação exerça controle sobre os membros de um grupo de objetos [4].

Enquanto cada réplica tem sua própria referência de objeto, o grupo de objetos como um todo tem que ter uma referência de objeto interoperável. Esta

referência é criada e recuperada pelo Gerenciador de Replicação através do Gerenciamento de Grupo de Objetos. A referência do grupo de objetos é retornada para a aplicação pelo Gerenciamento de Grupo de Objetos e é publicada pelo servidor de objetos.

O objeto cliente usa a referência do grupo de objetos para invocar métodos no servidor de objetos do grupo. Apenas o servidor de objetos do grupo irá usar a referência convencional do objeto para um objeto não replicado. Devido à abstração de grupo de objetos, os objetos clientes não sabem que os objetos servidores estão replicados (transparência de replicação) e também não sabem se uma falha ocorreu ou não nas réplicas do servidor ou se houve a recuperação de quando uma falha ocorreu (transparência de controle de falhas para o cliente).

3.2.3 A Fábrica Genérica

Usando o método de criação de objetos da interface de Fábrica Genérica a aplicação requisita a criação de um objeto replicado caso ele ainda não seja replicado. Este método é invocado pelo gerenciador de replicações que então invoca as fábricas, nos diferentes *hosts*, onde as réplicas são criadas usando o mesmo método de criação de objetos da interface de Fábrica Genérica.

3.2.4 O Mecanismo de Recuperação e *Logging*

O Mecanismo de Recuperação e *Logging* é utilizado na recuperação de um objeto replicado que foi restaurado ou eleito como membro primário em caso de falha do membro primário anterior.

O Mecanismo de Recuperação e *Logging* funciona da seguinte forma: todas as mensagens são passadas para o mecanismo de recuperação e *logging* de forma automática e invisível para a aplicação. O mecanismo de *logging* registra as mensagens no *log* para que o mecanismo de recuperação possa recuperar a mensagem. Periodicamente o mecanismo de *logging* invoca um método para obter o estado do objeto, então o estado pode ser registrado no *log*. Durante a recuperação o mecanismo de recuperação invoca o método para atualizar o estado do novo membro primário para o estado que foi gravado no *log*.

Os estilos de replicação utilizados pela especificação FT-CORBA são: replicação passiva fria, replicação passiva quente, replicação ativa e sem estado.

Para os estilos de replicação passiva fria e replicação passiva quente, em condições livre de falhas, somente um membro do grupo de objetos, o membro primário, executa a requisição e gera uma resposta. Se o detector de falhas suspeitar que o membro primário está falho, o gerenciador de replicações, reinicia o membro primário corrente ou promove um membro *backup* para se tornar o novo membro primário.

Para o estilo de consistência controlado pela aplicação o gerenciador de replicação não efetua o processo de recuperação e o novo membro primário é

responsável por recuperar seu próprio estado. Para o estilo de consistência controlado pela infraestrutura, o novo membro primário deve iniciar a operação com o estado apropriado, e deve executar a mesma seqüência de requisições que foram ou deveriam ter sido executadas pelo membro primário anterior tendo este falhado ou não.

3.2.5 O Detector e o Notificador de Falhas

O Detector de Falhas é responsável pela percepção da presença de falhas no sistema e geração de um relatório de falhas para o Notificador de Falhas. O Notificador de Falhas recebe os relatórios de falhas filtra-os e propaga os relatórios filtrados como eventos de notificação para os consumidores cadastros. O Analisador de Falhas avalia os relatórios de falhas que foram recebidos e produz um relatório agregado ou sumarizado das falhas. Os relatórios são enviados de volta para o Notificador de Falhas para envio a outros consumidores do serviço.

O Notificador de Falhas recebe relatórios do Detector de Falhas, e efetua uma seleção sobre eles visando eliminar relatórios desnecessários ou duplicados. Ele então envia um evento de notificação para os consumidores deste serviço. O Gerenciador de Replicação é como um consumidor assim como o Analisador de Falhas. Uma aplicação também pode inscrever-se para receber as notificações de falha. Existe um Notificador de Falhas por domínio de tolerância à falhas e normalmente este Notificador é replicado para tolerar falhas.

Um sistema tolerante a falhas também pode ter um ou mais analisadores de falhas cada um deles recebendo relatórios de falhas e efetuando a análise dos relatórios.

Capítulo 4

Descrição do Sistema Implementado e Resultados Experimentais

Neste capítulo a implementação do detector de falhas é descrita. Uma aplicação exemplo é utilizada para ilustrar o uso da infraestrutura proposta, um sistema cliente/servidor de hora. A descrição da implementação da infraestrutura tolerante a falhas apresenta seus componentes, o procedimento utilizado para detecção de falhas e os procedimentos necessários para integração de uma aplicação servidora com a infraestrutura.

São descritos também experimentos realizados para a avaliação das estratégias de monitoração propostas. A primeira abordagem efetua a monitoração periódica apenas das réplicas primárias de cada grupo de objetos e a segunda abordagem efetua a monitoração periódica de todas as réplicas de cada grupo de objetos. São medidos, para cada abordagem, o tempo de recuperação da réplica primária e o número de requisições necessárias para recuperação da mesma em caso de falha.

4.1 Sequência de Criação das Réplicas

Redundância é a base da tolerância a falhas. A infraestrutura proposta permite a criação de aplicações distribuídas tolerantes a falhas que envolvem servidores replicados formando grupos de objetos. Cada grupo de objetos contém réplicas de um mesmo servidor sendo que uma das réplicas, a chamada réplica primária, é a responsável por responder às requisições dos clientes. As requisições são feitas pelos clientes ao grupo de objetos replicados que é o responsável por direcionar a requisição para a réplica primária daquele grupo. Caso ocorra uma falha com a réplica primária do grupo de objetos uma nova réplica primária é eleita para o mesmo.

A figura 5 ilustra o procedimento utilizado pela infraestrutura para criação das réplicas de objetos. Uma aplicação que utiliza a infraestrutura, representada na figura 5 pelo componente *AplicaçãoQualquer*, solicita a referência do componente *ReplicationManager* ao componente *COS-Naming*. Após obter a referência, a aplicação (*AplicaçãoQualquer*) solicita a criação do grupo de objetos ao *ReplicationManager*.

O componente *ReplicationManager* solicita ao componente *ObjectGroup* a criação do grupo de objetos solicitado. O componente *ObjectGroup* por sua vez cria uma *thread*, representada na figura pelo componente *ThreadMember*, para a criação dos membros de cada grupo de objetos solicitado. O componente *ThreadMember* utiliza as informações recebidas de *ObjectGroup* para criação dinâmica de um nova instância de cada réplica e, a partir desta nova instância, o método responsável por sua disponibilização é invocado.

Após a sua criação, a réplica é incluída sob o contexto de grupo ao qual pertence. Este contexto será utilizado no processo de monitoração das réplicas bem como no processo de eleição de uma nova réplica primária válida e ativa, quando da ocorrência de uma falha. O processo de identificação de grupos em diferentes *hosts* também utiliza o componente *GenerateId* que tem como função fornecer um identificador único por *host*.

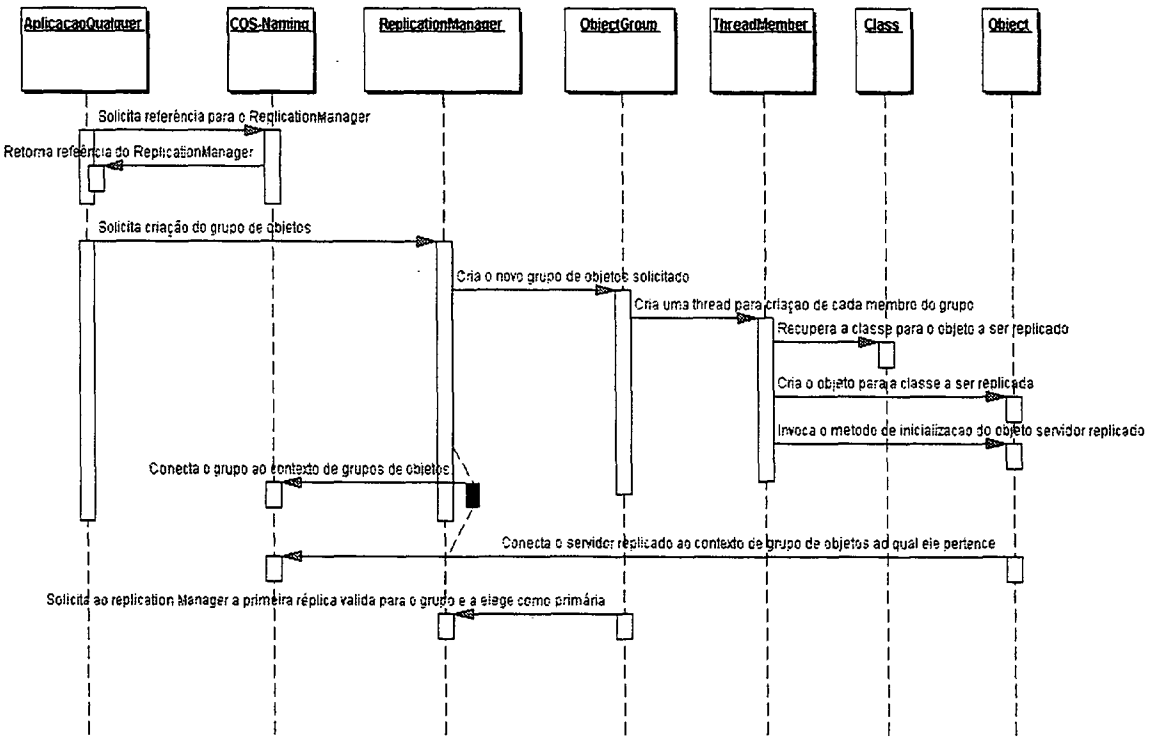


Figura 5 : Diagrama de seqüência da criação de réplicas.

4.2 Sequência da Interação Cliente com Servidor Tolerante a Falhas

A criação de um servidor de hora tolerante a falhas, utilizando a infraestrutura implementada, é utilizada como exemplo e é ilustrada na figura 6. Um servidor de hora é utilizado no exemplo. O componente *Client* representa um cliente que deseja utilizar o servidor. O componente *COS-Naming* representa o serviço de nomes do CORBA [3] onde os grupos e os objetos replicados são registrados. O componente *ReplicationManager*, neste exemplo, representa o objeto responsável por iniciar o processo de recuperação da referência da réplica do servidor primário do grupo de objetos *TimeServer* solicitado pelo cliente. O componente *FaultDetector* representa a classe responsável pela monitoração dos servidores replicados nos grupos de objetos.

Na figura 6 o componente *FaultDetector* deve ser considerado durante todo o período de execução da aplicação, pois o procedimento de detecção de falhas está ativo durante todo o período de execução da aplicação. A interação cliente servidor é descrita a seguir.

O componente *Client* efetua uma requisição ao *COS-Naming* para obtenção da referência ao *ReplicationManager*. A partir desta recuperação o *Client* solicita ao *ReplicationManager* a referência do servidor de hora pelo nome do grupo de servidores, no caso "*TimeServer*". O *ReplicationManager* por sua vez efetua uma consulta ao *COS-Naming* buscando a referência para o grupo de objetos desejado, que é representado na figura 6 pelo componente *ObjectGroup*. O *COS-Naming* obtém a referência para o grupo de objetos, este

retorna a referência do servidor primário do grupo desejado que é então retornado para o *ReplicationManager* e, por sua vez, retornado para o *Client* que pode então efetuar a solicitação desejada.

O componente *FaultDetector* monitora continuamente cada grupo de objetos de acordo com o intervalo de monitoração previamente definido. A interação representada na figura 6 entre o componente *FaultDetector* e o componente *ObjectGroup* representa a eleição de uma nova réplica primária para o grupo em questão, caso durante o processo de monitoração verifique-se que a réplica primária para o grupo está falha.

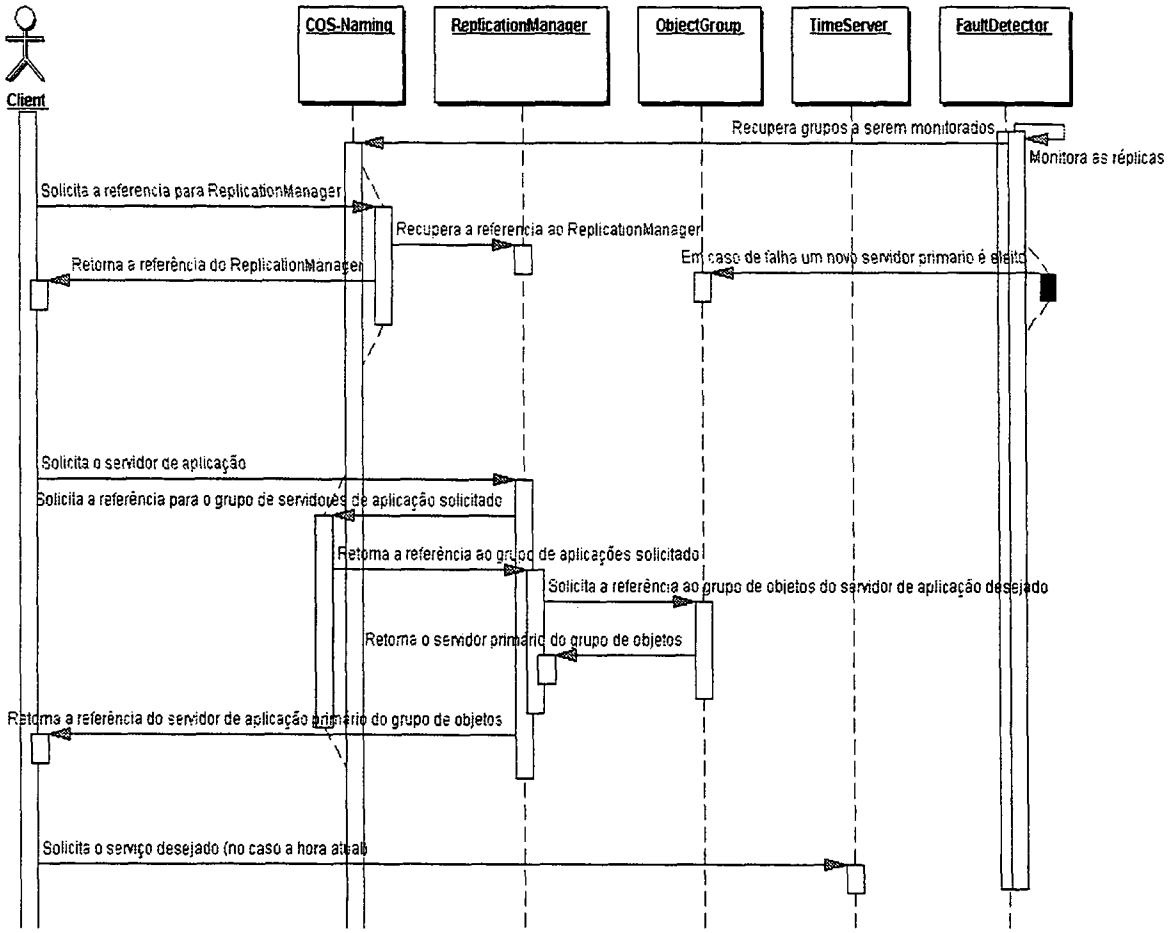


Figura 6: Diagrama de seqüência da interação cliente/servidor tolerante a falhas.

O processo de monitoração das réplicas utilizado por esta infraestrutura contém duas abordagens diferenciadas que são descritas a seguir.

4.2.1 Monitoração das Réplicas

Na primeira abordagem apenas o servidor primário de cada grupo é periodicamente monitorado, checando se o mesmo está falho ou sem falha. Somente em caso de réplica primária falha a monitoração das demais réplicas é efetuada buscando-se eleger um novo servidor primário para o grupo de objetos em questão e configurando esta nova réplica como servidor primário deste grupo.

Na segunda abordagem o processo de monitoração é feito periodicamente para todos os objetos de todos os grupos, checando se cada réplica disponível está falha. Uma réplica falha é desconsiderada no momento da eleição de um novo membro primário para um grupo de objetos.

A monitoração, na primeira abordagem, não é feita para todas as réplicas, o que oferece um ganho potencial de performance na monitoração das réplicas se considerarmos um grande número delas. Por outro lado, caso múltiplas réplicas falhem, o intervalo de tempo até que um novo servidor seja eleito pode ser consideravelmente elevado.

Na segunda abordagem, a monitoração é feita periodicamente para todos os objetos de todos os grupos, o que possibilita um ganho potencial no tempo de recuperação de uma réplica primária falha para o grupo em questão já que serão consideradas apenas réplicas ativas previamente monitoradas. No entanto, o custo de monitoração é maior do que o proposto na primeira abordagem.

Em ambos os casos o intervalo de monitoração, o número de tentativas de comunicação com o servidor e o tempo entre cada uma das tentativas antes de se considerar uma falha é considerado e é configurável. Os experimentos efetuados comparando-se as duas abordagens são apresentados adiante neste capítulo.

4.3 A Infraestrutura Tolerante a Falhas: Descrição da Implementação

Nesta seção a implementação da infraestrutura tolerante a falhas é descrita a partir do diagrama de classes representado pela figura 7 e considera um sistema isócrono utilizando o estilo de replicação *stateless* e o estilo de consistência controlado pela infraestrutura [3]. Para a demonstração do funcionamento da infraestrutura a aplicação exemplo que consiste de um sistema cliente-servidor de hora representado pela classe *TimeServerImpl*, foi considerada. É importante observar que todas as classes representadas na figura com fundo branco referem-se à infraestrutura para tolerância a falhas, apenas os componentes com fundo cinza representam os objetos de aplicação.

Para que o servidor de hora *TimeServerImpl*, representado na figura 7, possa ser monitorado é necessário que ele implemente duas interfaces e que, de forma facultativa, herde a classe *DefaultDetecImplInterfaces*. A primeira interface é chamada de *OrbInterface* e define o método *initOrb*. A segunda interface é chamada de *PullMonitorableOperations* e define a implementação do método *isAlive()*. O componente *DefaultDetecImplInterfaces*, é a classe

facultativa, que tem a função de reduzir o esforço de implementação das interfaces.

A interface *OrbInterface* viabiliza a instanciação dinâmica dos servidores replicados. Assim, qualquer servidor que venha a ser integrado com a infraestrutura tolerante a falhas deve implementar esta interface contemplando às seguintes atividades: inicialização do ORB, inicialização do POA, ativação do POA, criação do objeto CORBA do objeto servidor a ser replicado e inclusão da referência do objeto replicado no contexto de réplicas do seu grupo de objetos. Estas atividades são parametrizadas durante a instanciação dinâmica pelos argumentos definidos pela infraestrutura.

Devido ao fato das atividades necessárias para integração do servidor com a infraestrutura serem muito semelhantes para outras aplicações servidoras que venham a utilizar a infraestrutura tolerante a falhas com um dado ORB, esta implementação foi generalizada na classe *DefaultDetecImplInterfaces* que pode ser utilizada como classe pai do objeto servidor a ser integrado, diminuindo grande parte do esforço de desenvolvimento de um novo servidor de aplicação. Utilizando esta implementação padrão o novo servidor precisa apenas definir sua integração com a infraestrutura, esta integração pode ser vista como uma transformação do objeto específico da aplicação para o objeto genérico fornecido para a infraestrutura tolerante a falhas.

A necessidade da implementação da interface *OrbInterface* fica clara quando pensamos que a infraestrutura pode disponibilizar qualquer tipo de objeto que desempenhe papel de servidor em um ambiente distribuído. Esta interface permite a comunicação entre a infraestrutura tolerante a falhas e o

objeto servidor no momento da criação e disponibilização das réplicas servidoras.

A segunda interface a ser implementada pela aplicação servidora é chamada de *PullMonitorableOperations* que define a implementação do método *isAlive()*. Este método é utilizado para monitoração de falhas nos objetos replicados e é invocado periodicamente pelo detector de falhas para monitorar o(s) servidor(es) replicado(s) do grupo de objetos validando, desta forma, se o mesmo está sem falhas.

A implementação do método *isAlive*() pode considerar vários fatores para determinar se a réplica está ou não em condições de plena atividade, de acordo com a aplicação desenvolvida. Um exemplo seria uma aplicação que necessita verificar se a comunicação com o banco de dados está ativa, caso não esteja, o processo deve ser interrompido e uma falha deve ser lançada, ativando assim a infraestrutura tolerante a falhas que irá buscar outro servidor que possa atender tal requisição.

Caso o objeto servidor não esteja disponível por qualquer motivo como falta de energia ou mesmo falha no sistema ou *host*, a requisição feita pela infraestrutura para o método *isAlive*() não retorna pois o servidor simplesmente não existe mais. Este problema é abordado da seguinte forma: ao efetuar uma requisição ao objeto servidor e em não se obtendo resposta por um determinado período de tempo, novas requisições serão efetuadas em períodos de tempo pré-determinados antes de efetivamente considerar esta réplica falha. Após alcançar os limites definidos, a réplica é então considerada falha. O tempo que o cliente aguarda a resposta de uma requisição, o número de novas requisições e

o intervalo de tempo entre as novas requisições devem ser previamente configurados na infraestrutura.

Qualquer aplicação servidora monitorada pelo detector de falhas utiliza no momento de criação dos grupos de objetos a operação *create_object* que é definida na interface *GenericFactory* e implementada pela classe *ReplicationManager*. Esta operação é responsável pela criação dos grupos de objetos para cada tipo de objeto servidor.

A implementação da interface *GenericFactory* pelo *ReplicationManager* tem como principais funções a instanciação da classe *ObjectGroupImpl* e o registro desta classe no contexto de grupos de objetos. A classe *ObjectGroupImpl*, ao ser instanciada, imediatamente inicia o processo de criação das réplicas deste grupo considerando o número de réplicas configurado. Após a criação das réplicas a infraestrutura elege a réplica primária para aquele grupo. Para eleição da réplica primária são considerados a ordem de criação e o estado de atividade da mesma, elegendo sempre a primeira réplica criada que esteja sem falhas.

O processo de criação de réplicas é feito pelo *ObjectGroupImpl* através de *threads* que são representadas no sistema pela classe *ThreadMember*. Esta classe efetua dinamicamente a chamada do método *initOrb* que foi previamente implementado pela réplica servidora para atender a interface *OrbInterface*. Desta forma a disponibilização do objeto e o seu registro no contexto de réplicas estão definidos no próprio servidor replicado baseado nos parâmetros fornecidos pelo componente *ThreadMember*. É este processo que

permite a generalização do processo de criação de réplicas para qualquer tipo de servidor que implemente a interface OrbInterface.

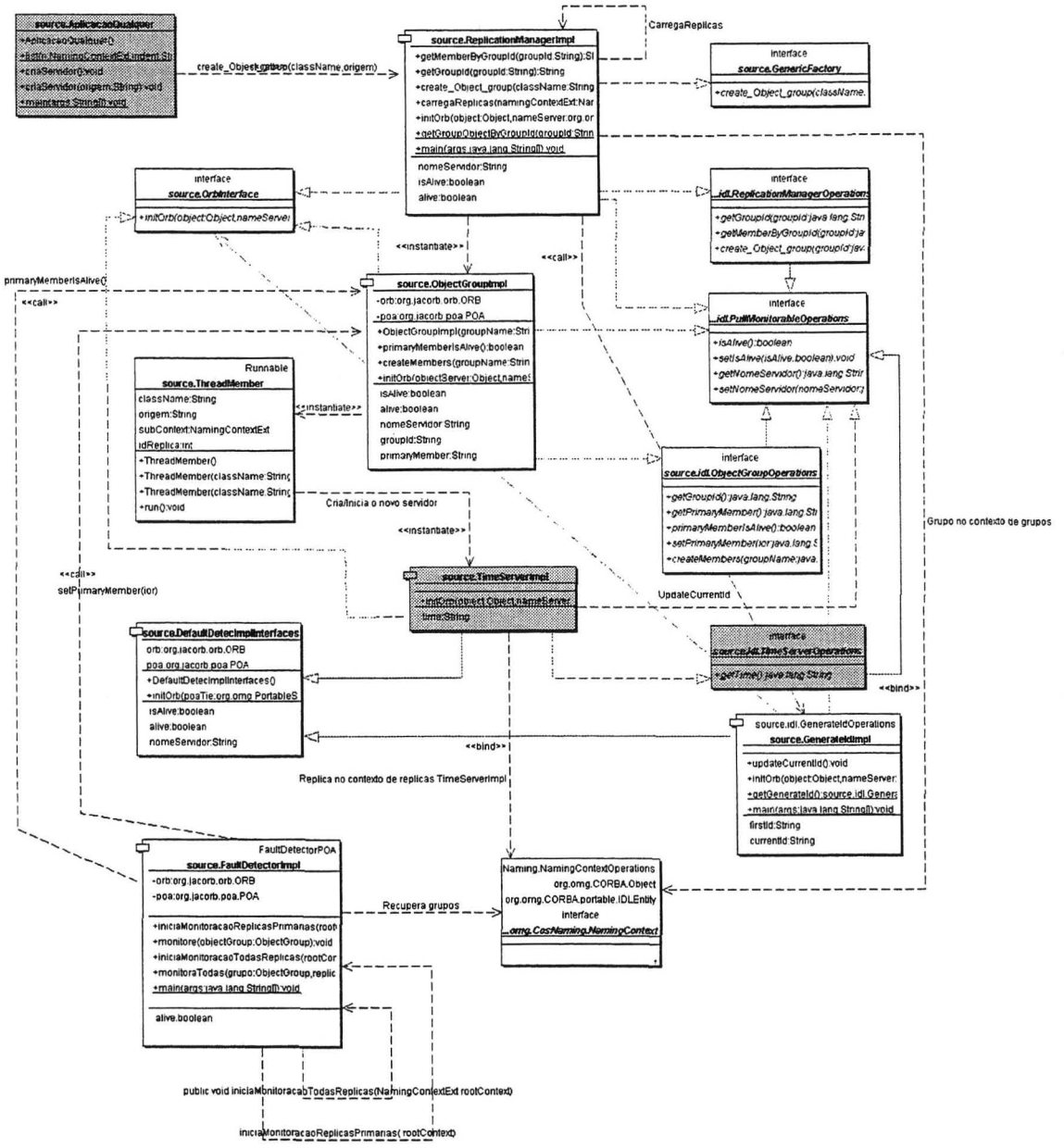


Figura 7: Integração de uma aplicação à infraestrutura.

A estrutura utilizada para controle de grupos e réplicas é baseada em contextos. Contextos são agrupamentos de informações afins. Os contextos aqui utilizados são de dois tipos: contexto de grupos e contexto de objetos replicados.

O contexto de grupos possui referências para objetos que contém informações sobre os grupos. A principal informação destes objetos é a referência para a réplica primária do grupo. O contexto de objetos replicados agrupa as réplicas dos objetos de determinado grupo.

4.4 Experimentos

Conforme descrito anteriormente duas abordagens para monitoração das réplicas foram implementadas. Na primeira abordagem apenas o servidor primário de cada grupo é periodicamente monitorado. Somente em caso de falha da réplica primária a monitoração das demais réplicas é efetuada e um novo servidor primário é eleito para o grupo de objetos e configurado como servidor primário do grupo.

Na segunda abordagem o processo de monitoração é feito periodicamente para todos os objetos de todos os grupos, checando se cada réplica disponível está falha. Uma réplica falha é desconsiderada no momento da eleição de um novo membro primário para um grupo de objetos.

Considerando as abordagens anteriormente descritas foram realizados experimentos que mostraram que a primeira abordagem possui um custo de monitoração pequeno. No entanto o tempo de recuperação da réplica primária, em caso de múltiplas falhas, é consideravelmente maior. Na segunda abordagem o tempo de recuperação, em caso de falha da réplica primária, é muito inferior. No entanto o custo de monitoração é maior devido ao potencialmente grande número de réplicas a serem monitoradas.

As medidas obtidas na avaliação das duas abordagens no que se refere ao tempo de recuperação dada uma falha da réplica primária de um grupo de objetos são apresentadas na tabela 1 e na figura 8. Os experimentos foram realizados em um computador rodando o sistema operacional conectiva linux, a implementação CORBA utilizada foi a Jacorb[17] e a versão java development kit 1.2.2. O computador utilizado tinha as seguintes características: processador Pentium III, clock de 933 Mhz e 512 MB de memória RAM. O tempo de monitoração entre cada bateria de testes foi de 2500 milisegundos. Uma réplica é considerada falha após 4 tentativas de conexão com intervalo de 500 milisegundos entre cada tentativa.

Considere a tabela 1, nela são apresentados resultados de 7 rodadas de testes denominados teste 1, teste 2, teste 3 e assim sucessivamente até teste 7. Cada grupo de testes contém 8 réplicas e a cada rodada de testes foram simuladas falhas simultâneas de forma que no teste 1 uma réplica sofre uma falha, no teste 2 duas réplicas sofrem falhas e assim sucessivamente até que 7 réplicas falhem simultaneamente no teste 7.

Os tempos de recuperação da réplica primária para o grupo de objetos replicados em cada rodada de teste foram medidos em milisegundos para ambas as abordagens propostas. De forma que os valores apresentados na tabela 1 considerando a primeira abordagem de monitoração são apresentados na coluna "Apenas as Réplicas Primárias" e têm os valores de 538 milisegundos para o teste 1, 3305 milisegundos para o teste 2, 7766 milisegundos para o teste 3 e assim sucessivamente até o valor de 13729 milisegundos para o teste 7. Já os valores apresentados na tabela 1 que consideram a segunda abordagem são

apresentados na coluna "Todas as Réplicas" e têm os valores de 538 milisegundos para o teste 1, 550 milisegundos para o teste 2, 562 milisegundos para o teste 3 e assim sucessivamente até o valor de 855 milisegundos para o teste 7.

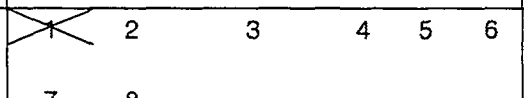
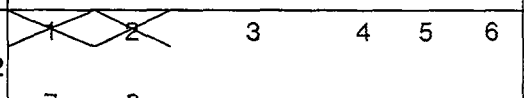
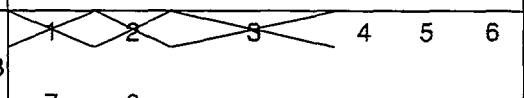
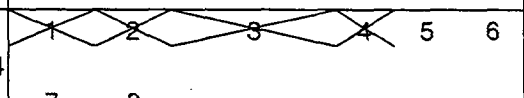
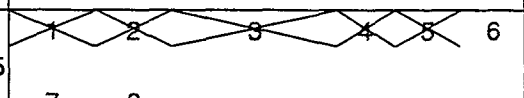
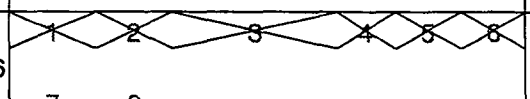
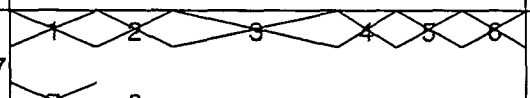
	Réplicas	Tempo de recuperação em milisegundos	
		Todas as Réplicas	Apenas as Réplicas primárias
Teste 1		538	538
Teste 2		550	3305
Teste 3		562	7766
Teste 4		573	8226
Teste 5		688	8326
Teste 6		854	10951
Teste 7		855	13729

Tabela 1 : Tempo de recuperação comparando-se diferentes abordagens.

Os dados apresentados na tabela 1 são mostrados graficamente na figura 8. Observe que o tempo de recuperação da réplica primária na segunda

abordagem permanece quase constante, ao contrário do que ocorre com a primeira abordagem. Isto acontece porque na segunda abordagem quando uma falha ocorre já se sabe qual a próxima réplica que está ativa e portanto será eleita como réplica primária do grupo.

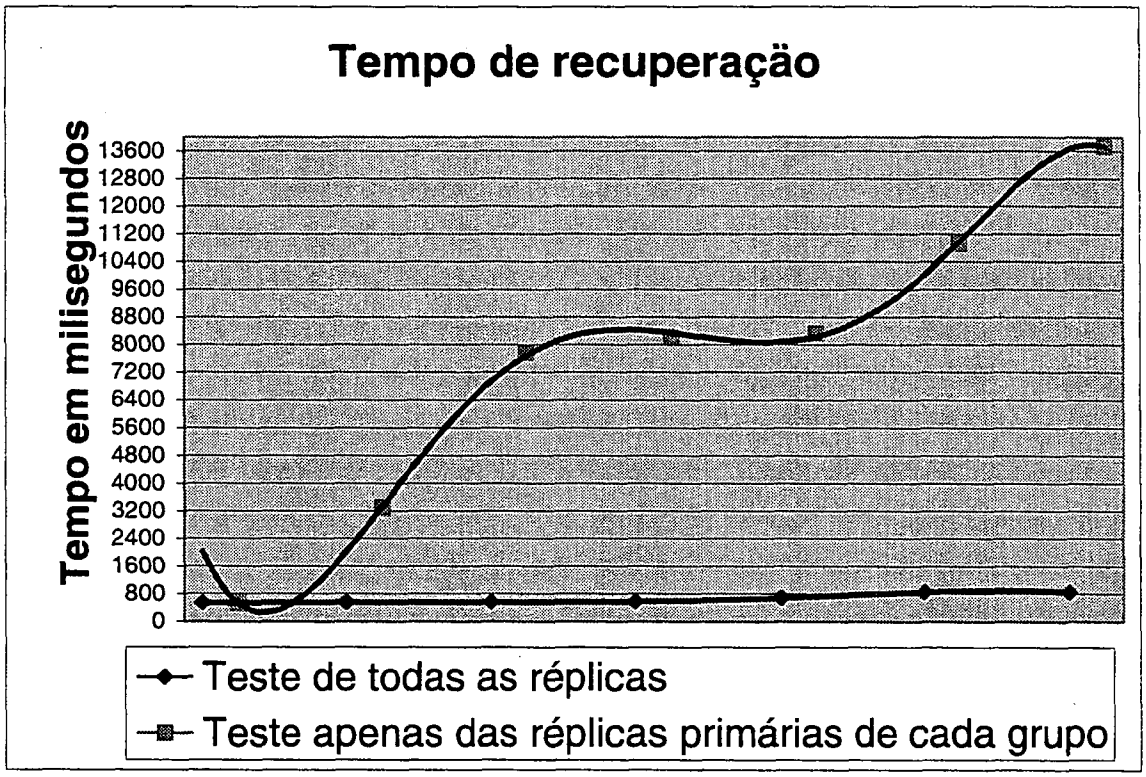


Figura 8 : Gráfico de linha do tempo de recuperação comparando-se diferentes abordagens.

No entanto, é importante observar que o custo da monitoração da primeira abordagem é muito inferior ao da segunda abordagem. Considerando N como o número de réplicas a serem monitoradas, Q o número de requisições, T o intervalo de tempo de monitoração entre cada bateria de testes e um tempo de avaliação A, temos que o número de requisições efetuadas para monitoração pode ser obtido a partir da seguinte expressão:

$$Q = N * (A/T)$$

Se compararmos as duas abordagens tomando como base o experimento efetuado considerando um tempo de avaliação (A) de 1 minuto (60000 milisegundos) o intervalo de monitoração entre as réplicas (T) de 2500 milisegundos e o número de réplicas monitoradas (N) pela primeira abordagem sendo igual a 1, pois apenas a réplica primária é monitorada, e na segunda abordagem igual a 8, pois todas as réplicas são monitoradas. Temos que o número de requisições (Q) feitas pela infraestrutura para monitoração das réplicas na primeira abordagem é de 24 requisições contra 192 requisições na segunda abordagem, a tabela 2 e a figura 9 mostram a relação de tempo de avaliação versus número de requisições considerando um intervalo de avaliação de 1 a 10 minutos.

Tempo Avaliação (A) (em minutos)	Numero de requisições (N) (Abordagem 1)	Numero de requisições (N) (Abordagem 2)
1	24	192
2	48	384
3	72	576
4	96	768
5	120	960
6	144	1152
7	168	1344
8	192	1536
9	216	1728
10	240	1920

Tabela 2: Número de requisições comparando-se diferentes abordagens.

Considerando um mesmo tempo de avaliação e um número diferenciado de réplicas observamos que o número de requisições aumenta proporcionalmente ao número de réplicas a serem monitoradas na segunda abordagem e se mantém bem inferior na primeira abordagem.

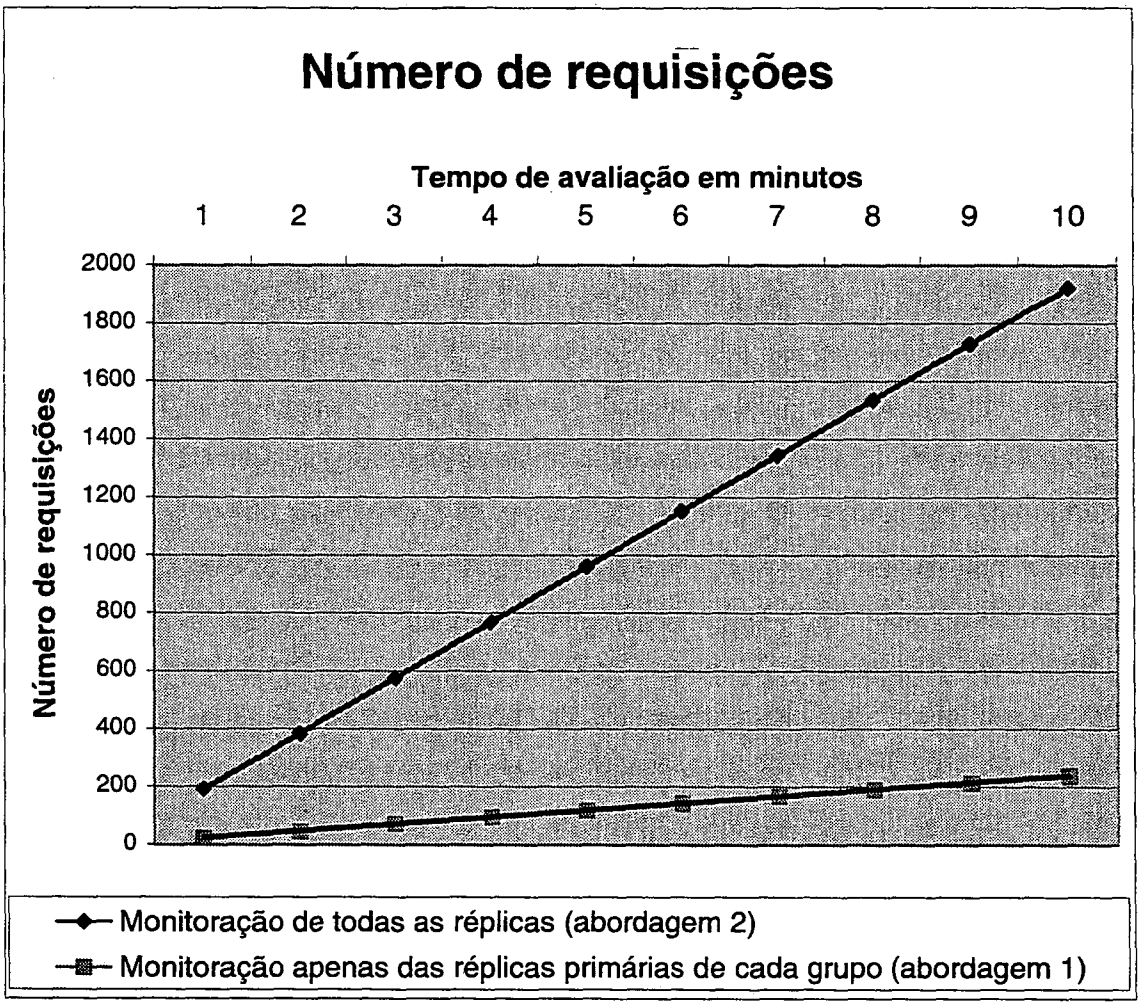


Figura 9 :Gráfico de linha comparando o número de requisições comparando-se diferentes abordagens.

Considerando uma rede de Ethernet de 100 Mbps e que cada requisição utilizada pelo processo de monitoração ocupa em média 200 Bytes da banda disponível, temos que a cada segundo uma requisição de monitoração ocupa em torno de 0,0016 % desta banda. Ou seja para monitorar 100 requisições simultâneas é utilizado 0,16 % da banda disponível a cada segundo e para monitorar mais de 600 requisições simultâneas 1 % da banda de rede seria utilizada. Se o número de requisições estiver dentro deste limite,

o custo da segunda abordagem pode ser considerado pequeno, justificando seu uso dada a latência de recuperação.

Capítulo 5

Conclusão

O padrão CORBA define um dos modelos de computação para objetos distribuídos mais amplamente utilizados. O CORBA é a especificação de uma arquitetura e interface que permite que aplicações efetuem requisições a objetos de forma transparente e independente da linguagem de programação, sistema operacional ou mesmo da localização dos objetos.

O padrão para tolerância a falhas FT-CORBA tem por objetivo fornecer um suporte para aplicações que necessitam de um alto grau de confiabilidade. Este trabalho contribui com a implementação de uma infraestrutura tolerante a falhas baseada no padrão FT-CORBA e demonstra o resultado de experimentos baseados nesta infraestrutura considerando duas diferentes abordagens para monitoração de réplicas. A comparação entre estas abordagens traz

informações que devem ser consideradas durante a implementação de sistemas distribuídos baseados em CORBA e tolerantes a falhas.

A primeira abordagem de monitoração demonstrada neste trabalho propõe a monitoração periódica de atividade apenas da réplica primária do servidor, iniciando o processo de monitoração para as demais réplicas somente em caso de falha da réplica primária. Neste caso o servidor primário é substituído por um novo servidor replicado sem falhas, desta forma, há uma eliminação de *overhead* de monitoração de múltiplas réplicas. Por outro lado, caso múltiplas réplicas falhem, o intervalo de tempo até que um novo servidor seja eleito pode ser consideravelmente elevado.

A segunda abordagem monitora periodicamente todas as réplicas sob os contextos de objetos replicados. Desta forma esta abordagem aumenta o *overhead* de monitoração mas diminui a latência de recuperação de uma réplica primária falha por uma nova réplica primária. Resultados experimentais mostram que a escolha do método de monitoração deve ser feita após uma avaliação do impacto de cada estratégia considerando o número total de réplicas monitoradas, bem como a banda disponível na rede.

Como trabalhos futuros, uma abordagem considerando intervalos de monitoração diferenciados, menores para as réplicas primárias dos grupos e maiores para as demais réplicas pode ser implementada. Além disso, diferentes abordagens para monitoração de falhas e coordenação de réplicas podem ser implementadas e integradas a esta implementação. Como exemplos de possíveis abordagens de monitoração podemos citar as baseadas em

diagnóstico distribuído [15,16] e como exemplo para abordagens de coordenação podemos citar as baseadas em protocolos de votação [18].

Um mecanismo que também pode ser integrado à implementação proposta são os interceptadores portáveis de requisições (*Portable Request Interceptors)* [20] que permitem modificar o comportamento do envio/recebimento de requisições sem necessidade de modificação no código do ORB ou da aplicação. Dentre as possíveis implementações que podem ser feitas utilizando este mecanismo está a identificação única das requisições de um cliente e o redirecionamento desta requisição entre diferentes réplicas de um servidor tolerante a falhas, viabilizando a transparência de falhas [19].

Referências Bibliográficas

[1] Gokhale, Aniruddha. "Fault Tolerant CORBA Extensions for JINI Pattern Language," *Lucent Technologies,* Bell Laboratories, 2000.

[2] J.R. Nicol, C.T. Wilkes, and F.A. Manola, "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Vol. 26 no.6, 1993.

[3] Object Management Group, "Fault Tolerant CORBA Specification V1.0," *OMG Document ptc/2000-04-04 ed.*, 2000.

[4] Object Management Group. "Common Object Request Broker: Architecture and Specification," *OMG Document*,1995.

[5] Olsen & Associates, and S. Maffeis, "Client/Server Term Definition," *Encyclopedia of Computer Science, International Thomson Computer Publishing*, 1997.

[6] A. Pope, "*The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*", Addison-Wesley, *1998.*

[7] R. M. Adler, "Group-Oriented Coordination Extensions to OMG's OMA/CORBA," *OMG presentation*, 1995.

[8] R.M. Soley,"Object Management Architecture Guide," *OMG Document*, 1992.

[9] S. Maffeis, "A Fault-Tolerant CORBA Name Server," In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, 1996

[10] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, USA,1995.

[11] S. Landis and S. Maffeis, "Building Reliable Distributed Systems with CORBA," *Theory and Practice of Object Systems*, John Wiley Publisher, 1997.

[12] *Object Management Group*, URL: http://www.omg.org/, acessado em julho de 2001.

[13] *Sun MicroSystems*, URL: http://www.javasoft.com/, acessado em outubro de 2001.

[14] D. Schmidt, "*Overview of CORBA*", URL: http://www.cs.wustl.edu/~schmidt/corba-overview.html, acessado em janeiro de 2001.

[15] E.P. Duarte Jr., L.C.P. Albini, and A. Brawerman "An Algorithm for Distributed Diagnosis of Dynamic Fault and Repair Events," In *Proceedings of the 7th IEEE International Conference on Parallel and Distributed Systems, IEEE/ICPADS'00*, Iwate, Japan, 2000.

[16] J.I. Siqueira, and E.P. Duarte Jr. "Uma Ferramenta para Diagnóstico Distribuído de Redes de Alta Velocidade" (*in portuguese)* In *Proceedings of the 2nd SBC Brazilian National Research Network (RNP) Workshop, SBC-WRNP'2*, Belo Horizonte, Brazil, 2000.

[17] Software Engineering and Systems Software Group at Freie Universitat Berlin and Xtradyne Technologies AG. URL: http://www.jacorb.org/, acessado em fevereiro de 2002.

[18] P. Jalote, "*Fault Tolerance in Distributed Systems*", Prentice Hall, 1994.

[19] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni, "An Interoperable Replication Logic for CORBA Systems," *In Proceedings of the 2nd Intenational Symposium on Distributed Objects and Applications*, Antwerpen, Belgium

[20] Object Management Group. "Portable Interceptor Specification", *OMG Document,* 1999.

[21] J. Rumbaugh, M. Blaha, W.Premerlani, F.Eddy, W.Lorensem, "*Modelagem e Projetos Baseados em Objetos*", Editora Campus, 1994.

Apêndice A – Documentação da Infraestrutura Tolerante a Falhas Implementada

Package *source*

Class Diagrams

diagram source

Subpackages

package source.idl

Classes

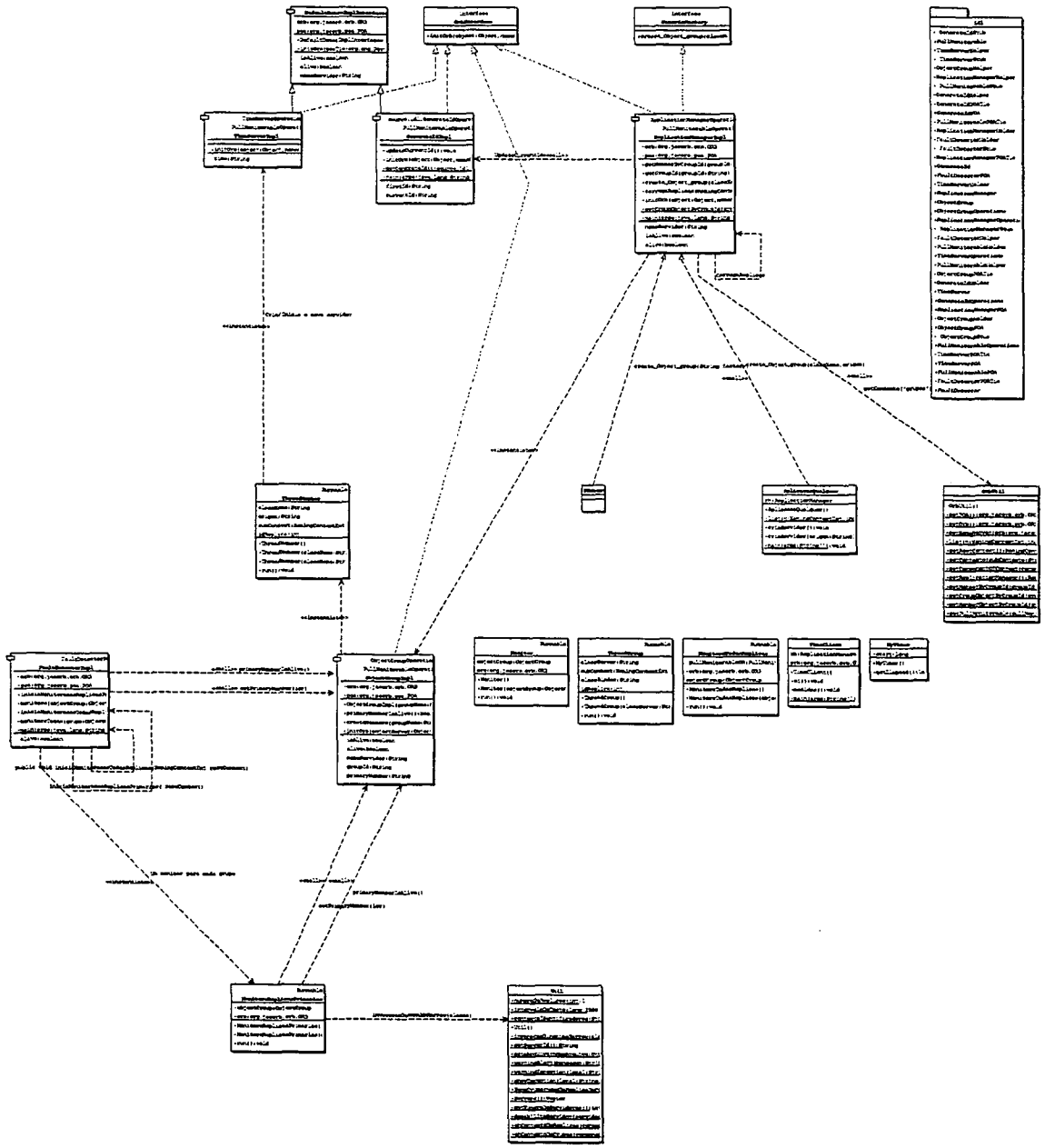
class source.AplicacaoQualquer
class source.DefaultDetecImplInterfaces
class source.FDStart
class source.FaultDetectorImpl
class source.GenerateIdImpl
class source.Monitor
class source.MonitoraReplicasPrimarias
class source.MonitoraTodasReplicas
class source.ObjectGroupImpl
class source.OrbUtil
class source.ReplicationManagerImpl
class source.ThreadGroup
class source.ThreadMember
class source.MyTimer
class source.TimeClient
class source.TimeServerImpl
class source.Util

Interfaces

interface source.GenericFactory
interface source.OrbInterface

7.1 Class Diagrams

Class Diagram *source*



Package Nodes

source.idl

Class Nodes

source.AplicacaoQualquer
source.DefaultDetecImplInterfaces
source.FaultDetectorImpl

source.FDStart
source.GenerateIdImpl
source.Monitor
source.MonitoraReplicasPrimarias
source.MonitoraTodasReplicas
source.MyTimer
source.ObjectGroupImpl
source.OrbUtil
source.ReplicationManagerImpl
source.ThreadGroup
source.ThreadMember
source.TimeClient
source.TimeServerImpl
source.Util

Interface Nodes

source.GenericFactory
source.OrbInterface

Package Node Detail

Package *source.idl*

Class Node Detail

Class *source.AplicacaoQualquer*

Esta classe é um exemplo de uma aplicação que deseja utilizar a infraestrutura tolerante a falhas. A partir desta classe o processo de criação das réplicas é iniciado.

Dependency Links

create_Object_group(className,origem) to **Class** ReplicationManagerImpl

link:
dependency

Stereotype:
call

Class *source.DefaultDetecImplInterfaces*

Esta classe é uma implementação padrão para as interfaces utilizadas pela infraestrutura, seu objetivo é facilitar o desenvolvimento de aplicações que desejem utilizar a infraestrutura tolerante a falhas proposta.

Class *source.FaultDetectorImpl*

Esta classe é responsável pelo procedimento de monitoração das réplicas, considerando as duas abordagens de monitoração propostas/consideradas por esta dissertação.

Extends:

FaultDetectorPOA

Dependency Links

iniciaMonitoracaoReplicasPrimarias(rootContext) to **Class** FaultDetectorImpl

link:

dependency

Um monitor para cada grupo to **Class** MonitoraReplicasPrimarias

link:

dependency

Stereotype:

instantiate

public void iniciaMonitoracaoTodasReplicas(NamingContextExt rootContext) to **Class** FaultDetectorImpl

link:

dependency

setPrimaryMember(ior) to **Class** ObjectGroupImpl

link:

dependency

Stereotype:

call

primaryMemberIsAlive() to **Class** ObjectGroupImpl

link:

dependency

Stereotype:

call

Class *source.FDStart*

Dependency Links

create_Object_group(String factory) to **Class** ReplicationManagerImpl

link:

dependency

Class *source.GenerateIdImpl*

Extends:
DefaultDetecImplInterfaces

Implements:
source.idl.GenerateIdOperations
OrbInterface
PullMonitorableOperations

Generalization links

to **Class** DefaultDetecImplInterfaces

Implementation links

to **Interface** OrbInterface

Class *source.Monitor*

Implements:
Runnable

Class *source.MonitoraReplicasPrimarias*

Implements:
Runnable

Dependency Links

invocacaoDinamicaNarrow(classe) to **Class** Util

link:
dependency

Stereotype:
call

setPrimaryMember(ior) to **Class** ObjectGroupImpl

link:
dependency

Stereotype:
call

primaryMemberIsAlive() to **Class** ObjectGroupImpl

link:
dependency

Stereotype:
call

Class *source.MonitoraTodasReplicas*

Implements:
Runnable

Class *source.MyTimer*

Class *source.ObjectGroupImpl*

Esta classe representa os grupos de objetos que são replicados.

Implements:
ObjectGroupOperations
OrbInterface
PullMonitorableOperations

Implementation links

to **Interface** OrbInterface

Dependency Links

to **Class** ThreadMember

link:
dependency

Stereotype:
instantiate

Class *source.OrbUtil*

Class *source.ReplicationManagerImpl*

Esta classe é responsável por criar os grupos de objetos, carregar réplicas em um host diferente caso o grupo já exista na rede, recuperar o grupo de objetos de um determinado contexto e inicializar e disponibilizar os serviços de gerenciamento de replicação para outros objetos que desejem utilizá-lo.

Implements:
ReplicationManagerOperations
OrbInterface
PullMonitorableOperations
GenericFactory

Implementation links

to **Interface** OrbInterface

to **Interface** GenericFactory

Dependency Links

to **Class** ObjectGroupImpl

link:
dependency

Stereotype:
instantiate

getContexto("grupos") to **Class** OrbUtil

link:
dependency

Stereotype:
call

CarregaReplicas to **Class** ReplicationManagerImpl

link:
dependency

UpdateCurrentId to **Class** GenerateIdImpl

link:
dependency

Stereotype:
call

Class *source.ThreadGroup*

Implements:
Runnable

Class *source.ThreadMember*

Esta classe representa a Thread utilizada para criação das réplicas dos grupos de objetos.

Implements:
Runnable

Dependency Links

Cria/Inicia o novo servidor to **Class** TimeServerImpl

link:
dependency

Stereotype:
instantiate

Class *source.TimeClient*

Class *source.TimeServerImpl*

Esta classe representa uma aplicação servidora exemplo que utiliza a infraestrutura tolerante a falhas proposta.

Extends:
DefaultDetecImplInterfaces

Implements:
TimeServerOperations
OrbInterface
PullMonitorableOperations

Generalization links

to **Class** DefaultDetecImplInterfaces

Implementation links

to **Interface** OrbInterface

Class *source.Util*

Interface Node Detail

Interface *source.GenericFactory*

Interface *source.OrbInterface*

Esta interface define o método necessário para disponibilização do objeto servidor via o ORB utilizado.

7.2 Class Detail

Class *source.AplicacaoQualquer*

```
public class AplicacaoQualquer
```

Esta classe é um exemplo de uma aplicação que deseja utilizar a infraestrutura tolerante a falhas. A partir desta classe o processo de criação das réplicas é iniciado.

Attributes

rm

Constructors

AplicacaoQualquer()

Operations

criaServidor(String)
criaServidor()
list(NamingContextExt, String)
main(String[])

Attribute Detail

rm

```
ReplicationManager rm
```

Constructor Detail

AplicacaoQualquer

```
public AplicacaoQualquer()
```

Operation Detail

criaServidor

```
public void criaServidor(String origem)
```

criaServidor

```
public void criaServidor()
```

list

```
public static void list(NamingContextExt n, String indent)
```

main

```
public static void main(String[] args)
```

Class source.DefaultDetecImplInterfaces

```
public class DefaultDetecImplInterfaces
```

Esta classe é uma implementação padrão para as interfaces utilizadas pela infraestrutura, seu objetivo é facilitar o desenvolvimento de aplicações que desejem utilizar a infraestrutura tolerante a falhas proposta.

Attributes

isAlive
NomeServidor

orb
poa

Constructors

DefaultDetecImplInterfaces()

Operations

getNomeServidor()
initOrb(org.omg.PortableServer.Servant, Object,
org.omg.CosNaming.NamingContextExt, String)
isAlive()
setIsAlive(boolean)
setNomeServidor(String)

Attribute Detail

isAlive

```
boolean isAlive = true
```

NomeServidor

```
private String NomeServidor
```

orb

```
org.jacorb.orb.ORB orb
```

poa

```
org.jacorb.poa.POA poa
```

Constructor Detail

DefaultDetecImplInterfaces

```
public DefaultDetecImplInterfaces()
```

Operation Detail

getNomeServidor

```
public String getNomeServidor()
```

initOrb

```
public void initOrb(org.omg.PortableServer.Servant poaTie, Object object,
org.omg.CosNaming.NamingContextExt nameServer, String nomeParaRegistro)
```

isAlive

```
public boolean isAlive()
```

setIsAlive

```
public void setIsAlive(boolean isAlive)
```

setNomeServidor

```
public void setNomeServidor(String nomeServidor)
```

Class *source.FaultDetectorImpl*

```
public class FaultDetectorImpl
```

Extends:

source.idl.FaultDetectorPOA

Esta classe é responsável pelo procedimento de monitoração das réplicas, considerando as duas abordagens de monitoração propostas/consideradas por esta dissertação.

Attributes

orb
poa

Operations

iniciaMonitoracaoReplicasPrimarias(NamingContextExt)
iniciaMonitoracaoTodasReplicas(NamingContextExt)
isAlive()
main(java.lang.String[])
monitoraTodas(ObjectGroup, PullMonitorable)
monitore(ObjectGroup)

Attribute Detail

orb

```
private org.jacorb.orb.ORB orb
```

poa

```
private org.jacorb.poa.POA poa
```

Operation Detail

iniciaMonitoracaoReplicasPrimarias

```
public void iniciaMonitoracaoReplicasPrimarias(NamingContextExt rootContext)
```

iniciaMonitoracaoTodasReplicas

```
public void iniciaMonitoracaoTodasReplicas(NamingContextExt rootContext)
```

isAlive

```
public boolean isAlive()
```

main

```
public static void main(java.lang.String[] args)
```

monitoraTodas

```
public void monitoraTodas(ObjectGroup grupo, PullMonitorable replica)
```

monitore

```
public void monitore(ObjectGroup objectGroup)
```

Class *source.FDStart*

```
public class FDStart
```

Class *source.GenerateIdImpl*

```
public class GenerateIdImpl
```

Extends:

source.DefaultDetecImplInterfaces

Implements:

source.idl.GenerateIdOperations
source.OrbInterface
source.idl.PullMonitorableOperations

Attributes

currentId
firstId

Operations

getCurrentId()
getFirstId()
getGenerateId()
initOrb(Object, org.omg.CosNaming.NamingContextExt, String, boolean)
main(java.lang.String[])
updateCurrentId()

Attribute Detail

currentId

```
private String currentId = "1921687106"
```

firstId

```
private String firstId = "1921687106"
```

Operation Detail

getCurrentId

```
public String getCurrentId()
```

getFirstId

```
public String getFirstId()
```

getGenerateId

```
public static source.idl.GenerateId getGenerateId()
```

initOrb

```
public void initOrb(Object object, org.omg.CosNaming.NamingContextExt
nameServer, String nomeParaRegistro, boolean chamadaInicial)
```

main

```
public static void main(java.lang.String[] args)
```

updateCurrentId

```
public void updateCurrentId()
```

Class source.Monitor

```
public class Monitor
```

Implements:

java.lang.Runnable

Attributes

objectGroup
orb

Constructors

Monitor()
Monitor(ObjectGroup)

Operations

run()

Attribute Detail

objectGroup

```
ObjectGroup objectGroup
```

orb

```
org.jacorb.orb.ORB orb
```

Constructor Detail

Monitor

```
public Monitor()
```

Monitor

```
public Monitor(ObjectGroup objectGroup)
```

Operation Detail

run

```
public void run()
```

Class source.MonitoraReplicasPrimarias

```
public class MonitoraReplicasPrimarias
```

Implements:

java.lang.Runnable

Attributes

objectGroup
orb

Constructors

MonitoraReplicasPrimarias()
MonitoraReplicasPrimarias(ObjectGroup)

Operations

run()

Attribute Detail

objectGroup

```
private ObjectGroup objectGroup
```

orb

```
private org.jacorb.orb.ORB orb
```

Constructor Detail

MonitoraReplicasPrimarias

```
public MonitoraReplicasPrimarias()
```

MonitoraReplicasPrimarias

```
public MonitoraReplicasPrimarias(ObjectGroup objectGroup)
```

Operation Detail

run

```
public void run()
```

Class *source.MonitoraTodasReplicas*

```
public class MonitoraTodasReplicas
```

Implements:

java.lang.Runnable

Attributes

objectGroup
orb
pullMonitorableOR

Constructors

MonitoraTodasReplicas()
MonitoraTodasReplicas(ObjectGroup, PullMonitorable)

Operations

run()

Attribute Detail

objectGroup

```
ObjectGroup objectGroup
```

orb

```
private org.jacorb.orb.ORB orb
```

pullMonitorableOR

```
private PullMonitorable pullMonitorableOR
```

Constructor Detail

MonitoraTodasReplicas

```
public MonitoraTodasReplicas()
```

MonitoraTodasReplicas

```
public MonitoraTodasReplicas(ObjectGroup objectGroup, PullMonitorable
objectReplica)
```

Operation Detail

run

```
public void run()
```

Class source.MyTimer

```
class MyTimer
```

Attributes

start

Constructors

MyTimer()

Operations

getElapsed()

Attribute Detail

start

```
private final long start
```

Constructor Detail

MyTimer

```
public MyTimer()
```

Operation Detail

getElapsed

```
public long getElapsed()
```

Class source.ObjectGroupImpl

```
public class ObjectGroupImpl
```

Implements:

source.idl.ObjectGroupOperations
source.OrbInterface
source.idl.PullMonitorableOperations

Esta classe representa os grupos de objetos que são replicados.

Attributes

GroupId
isAlive
NomeServidor
orb
poa
primaryMember

Constructors

ObjectGroupImpl(String, String)

Operations

createMembers(String, String)
getGroupId()
getNomeServidor()
getPrimaryMember()
initOrb(Object, org.omg.CosNaming.NamingContextExt, String, boolean)
isAlive()
primaryMemberIsAlive()
setIsAlive(boolean)
setNomeServidor(String)
setPrimaryMember(String)

Attribute Detail

GroupId

```
private String GroupId
```

isAlive

```
private boolean isAlive = true
```

NomeServidor

```
private String NomeServidor
```

orb

```
private org.jacorb.orb.ORB orb
```

poa

```
private org.jacorb.poa.POA poa
```

primaryMember

```
private String primaryMember
```

Constructor Detail

ObjectGroupImpl

```
public ObjectGroupImpl(String groupName, String origem)
```

Operation Detail

createMembers

```
public void createMembers(String groupName, String origem)
```

Este método é o responsável por criar as réplicas para o grupo recebido como parâmetro de acordo com o número de réplicas previamente definido. Será criado uma thread para criação de cada réplica.

groupName : Nome do grupo sob o qual serão criadas as réplicas.

origem : indicador de onde foi iniciado o processo de criação.

getGroupId

```
public String getGroupId()
```

getNomeServidor

```
public String getNomeServidor()
```

getPrimaryMember

```
public String getPrimaryMember()
```

initOrb

```
public void initOrb(Object objectServer, org.omg.CosNaming.NamingContextExt
nameServer, String nomeParaRegistro, boolean chamadaInicial)
```

Este metodo é o responsavel por disponibilizar o objeto para receber requisicoes.

object : Objeto a ser disponibilizado (no caso a própria classe)

nameServer : Contexto onde se encontra o servidor de nomes

nomeParaRegistro : Nome com o qual o servidor será registrado no serviço de nomes

chamadaInicial : Indicador se esta é a chamada de carga da classe (true) ou se a chamada é uma chamada para criação de réplicas (false).

isAlive

```
public boolean isAlive()
```

primaryMemberIsAlive

```
public boolean primaryMemberIsAlive()
```

Este método é o responsável por retornar true se o membro primário deste grupo estiver vivo.

setIsAlive

```
public void setIsAlive(boolean isAlive)
```

setNomeServidor

```
public void setNomeServidor(String nomeServidor)
```

setPrimaryMember

```
public void setPrimaryMember(String ior)
```

Class source.OrbUtil

```
public class OrbUtil
```

Constructors

OrbUtil()

Operations

getComponentOfContext(String, String)
getContexto(String)
getGroupObjectByGroupId(String)
getMemberByGroupId(String)
getMemberObjectByGroupId(String)
getNameServer(org.jacorb.orb.ORB)
getOrb()
getPOA()
getPullMonitorable(Object)
getReplicationManager()
getRootContext()
list(NamingContextExt, String)

Constructor Detail

OrbUtil

```
private OrbUtil()
```

Operation Detail

getComponentOfContext

```
public static org.omg.CORBA.Object getComponentOfContext(String component,
String contexto)
```

Throws:

org.omg.CosNaming.NamingContextPackage.NotFound

getContexto

```
public static NamingContextExt getContexto(String subContexto)
```

getGroupObjectByGroupId

```
public static ObjectGroup getGroupObjectByGroupId(String groupId)
```

retorna o Objeto CORBA que representa o grupo de Objetos para o grupo recebido como parametro (groupId)

getMemberByGroupId

```
public static String getMemberByGroupId(String groupId)
```

retorna a replica primaria valida para o grupo recebido como parametro (groupId)

getMemberObjectByGroupId

```
public static org.omg.CORBA.Object getMemberObjectByGroupId(String groupId)
```

retorna o Objeto CORBA que representa a primeira replica valida para o grupo recebido como parametro (groupId)

getNameServer

```
public static org.omg.CosNaming.NamingContextExt
getNameServer(org.jacorb.orb.ORB orb)
```

getOrb

```
public static org.jacorb.orb.ORB getOrb()
```

getPOA

```
public static org.jacorb.orb.ORB getPOA()
```

getPullMonitorable

```
public static PullMonitorable getPullMonitorable(Object pullMonitorable)
```

getReplicationManager

```
public static ReplicationManager getReplicationManager()
```

Throws:

Exception

getRootContext

```
public static NamingContextExt getRootContext()
```

list

```
public static void list(NamingContextExt n, String indent)
```

Class *source.ReplicationManagerImpl*

```
public class ReplicationManagerImpl
```

Implements:

source.idl.ReplicationManagerOperations
source.OrbInterface
source.idl.PullMonitorableOperations
source.GenericFactory

Esta classe é respon'sável por criar os grupos de objetos, carregar réplicas em um host diferente caso o grupo ja exista na rede, recuperar o grupo de objetos de um determinado

contexto e inicializar e disponibilizar os serviços de gerenciamento de replicação para outros objetos que desejem utilizá-lo.

Attributes

isAlive
NomeServidor
orb
poa

Operations

carregaReplicas(NamingContextExt)
create_Object_group(String, String)
getGroupId(String)
getGroupObjectByGroupId(String)
getMemberByGroupId(String)
getNomeServidor()
initOrb(Object, org.omg.CosNaming.NamingContextExt, String, boolean)
isAlive()
main(java.lang.String[])
setIsAlive(boolean)
setNomeServidor(String)

Attribute Detail

isAlive

```
private boolean isAlive = true
```

NomeServidor

```
private String NomeServidor
```

orb

```
private org.jacorb.orb.ORB orb
```

poa

```
private org.jacorb.poa.POA poa
```

Operation Detail

carregaReplicas

```
public void carregaReplicas(NamingContextExt namingContextExt)
```

namingContextExt : Contexto inicial para carga das réplicas.

Este método deve carregar as réplicas para cada grupo de objetos do contexto grupos. A informação referente a quais grupos estão replicados esta na própria lista de contexto como exemplificado abaixo, sendo que os contextos são os nomes seguidos do caracter "/". Assim sendo temos no exemplo abaixo os contextos : ReplicationManagerImpl, TimeServerImpl.

ReplicationManagerImpl/ ReplicationManagerImpl2_1921687106 ReplicationManagerImpl2_1921687107 ReplicationManagerImpl1_1921687106 ReplicationManagerImpl3_1921687106 ReplicationManagerImpl1_1921687107 ReplicationManagerImpl3_1921687107

TimeServerImpl/ TimeServerImpl2_1921687106 TimeServerImpl2_1921687107 TimeServerImpl1_1921687106 TimeServerImpl1_1921687107 TimeServerImpl3_1921687106 TimeServerImpl3_1921687107

grupos/ ReplicationManagerImpl_1921687106 ReplicationManagerImpl_1921687107 TimeServerImpl_1921687106 TimeServerImpl_1921687107

create_Object_group

```
public boolean create_Object_group(String className, String origem)
```

Cria o grupo de objetos replicados este grupo de objetos é representado pelo objeto "ObjectGroupImpl". Ao ser criado o grupo de objetos as replicas do mesmo também são criadas de acordo com o parâmetro número de réplicas. className : Nome do grupo de objetos a ser criado. origem : indicador de onde foi iniciado o processo de criação.

getGroupId

```
public String getGroupId(String groupId)
```

Este método retorna o IOR da primeira réplica "viva" do grupo recebido como parâmetro. groupId : Nome do grupo desejado que ira devolver o IOR da primeira réplica "viva".

getGroupObjectByGroupId

```
public static ObjectGroup getGroupObjectByGroupId(String groupId)
```

retorna o Objeto CORBA que representa o grupo de Objetos para o grupo recebido como parametro (groupId)

getMemberByGroupId

```
public String getMemberByGroupId(String groupId)
```

Este método retorna o IOR da réplica primária do grupo recebido como parâmetro. groupId : Nome do grupo desejado que ira devolver o IOR da sua réplica primária.

getNomeServidor

```
public String getNomeServidor()
```

initOrb

```
public void initOrb(Object object, org.omg.CosNaming.NamingContextExt
nameServer, String nomeParaRegistro, boolean chamadaInicial)
```

Este metodo é o responsavel por disponibilizar o objeto para receber requisicoes. object : Objeto a ser disponibilizado (no caso a própria classe) nameServer : Contexto onde se encontra o servidor de nomes nomeParaRegistro : Nome com o qual o servidor será registrado no serviço de nomes chamadaInicial : Indicador se esta é a chamada de carga da classe (true) ou se a chamada é uma chamada para criação de réplicas (false).

isAlive

```
public boolean isAlive()
```

main

```
public static void main(java.lang.String[] args)
```

setIsAlive

```
public void setIsAlive(boolean isAlive)
```

setNomeServidor

```
public void setNomeServidor(String nomeServidor)
```

Class source.ThreadGroup

```
public class ThreadGroup
```

Implements:

java.lang.Runnable

Attributes

classBinder
classServer
idReplica
subContext

Constructors

ThreadGroup()
ThreadGroup(String, NamingContextExt, String)

Operations

run()

Attribute Detail

classBinder

```
String classBinder
```

classServer

```
String classServer
```

idReplica

```
int idReplica
```

subContext

```
NamingContextExt subContext
```

Constructor Detail

ThreadGroup

```
public ThreadGroup()
```

ThreadGroup

```
public ThreadGroup(String classServer, NamingContextExt subContext, String
classBinder)
```

Operation Detail

run

```
public void run()
```

Class source.ThreadMember

```
public class ThreadMember
```

Implements:

java.lang.Runnable

Esta classe representa a Thread utilizada para criação das réplicas dos grupos de objetos.

Attributes

className
idReplica
origem
subContext

Constructors

ThreadMember()
ThreadMember(String, String, NamingContextExt, int)
ThreadMember(String, NamingContextExt, int)

Operations

run()

Attribute Detail

className

```
String className
```

idReplica

```
int idReplica
```

origem

```
String origem
```

subContext

```
NamingContextExt subContext
```

Constructor Detail

ThreadMember

```
public ThreadMember()
```

ThreadMember

```
public ThreadMember(String className, String origem, NamingContextExt
subContext, int idReplica)
```

ThreadMember

```
public ThreadMember(String className, NamingContextExt subContext, int
idReplica)
```

Operation Detail

run

```
public void run()
```

Class source.TimeClient

```
public class TimeClient
```

Attributes

orb
rm

Constructors

TimeClient()

Operations

main(String[])
medidas()
oi()

Attribute Detail

orb

```
org.jacorb.orb.ORB orb
```

rm

```
ReplicationManager rm
```

Constructor Detail

TimeClient

```
public TimeClient()
```

Operation Detail

main

```
public static void main(String[] args)
```

medidas

```
public void medidas()
```

oi

```
public void oi()
```

Class *source.TimeServerImpl*

```
public class TimeServerImpl
```

Extends:

source.DefaultDetecImplInterfaces

Implements:

source.idl.TimeServerOperations
source.OrbInterface
source.idl.PullMonitorableOperations

Esta classe representa uma aplicação servidora exemplo que utiliza a infraestrutura tolerante a falhas proposta.

Operations

getTime()
initOrb(Object, org.omg.CosNaming.NamingContextExt, String, boolean)

Operation Detail

getTime

```
public String getTime()
```

initOrb

```
public void initOrb(Object object, org.omg.CosNaming.NamingContextExt
nameServer, String nomeParaRegistro, boolean chamadaInicial)
```

Class *source.Util*

```
public class Util
```

Attributes

contextoIdentificadores
intervaloDeTeste
numeroDeReplicas

Constructors

Util()

Operations

desabilitaServidor(String, String)
ehContextoDeGrupos(String)
ehContextoDeReplicas(String)
getNumeroDeServidores()
getServerId()
invocacaoDinamicaNarrow(String)
NomeGrupo(String)
Servers()

setaArquivo(String)
showException(String, Exception)
warningAlert(String)
warningException(String, String)

Attribute Detail

contextoIdentificadores

```
public static String contextoIdentificadores = "zIdentificadores"
```

intervaloDeTeste

```
public static long intervaloDeTeste = 2500
```

numeroDeReplicas

```
public static int numeroDeReplicas = 1
```

Constructor Detail

Util

```
public Util()
```

Operation Detail

desabilitaServidor

```
public static void desabilitaServidor(String servidor, String invocadoDe)
```

ehContextoDeGrupos

```
public static boolean ehContextoDeGrupos(String componente)
```

ehContextoDeReplicas

```
public static boolean ehContextoDeReplicas(String componente)
```

getNumeroDeServidores

```
public static int getNumeroDeServidores()
```

getServerId

```
public static String getServerId()
```

invocacaoDinamicaNarrow

```
public static org.omg.CORBA.Object invocacaoDinamicaNarrow(String classe)
```

NomeGrupo

```
public static String NomeGrupo(String nomeDaReplicaDoGrupo)
```

Servers

```
public static Vector Servers()
```

setaArquivo

```
public static void setaArquivo(String nomeArquivo)
```

showException

```
public static void showException(String local, Exception e)
```

warningAlert

```
public static void warningAlert(String mensagem)
```

warningException

```
public static void warningException(String local, String mensagem)
```

7.3 Interface Detail

Interface *source.GenericFactory*

```
public interface GenericFactory
```

Operations

create_Object_group(String, String)

Operation Detail

create_Object_group

```
public abstract void create_Object_group(String className, String origem)
```

Interface *source.OrbInterface*

```
public interface OrbInterface
```

Esta interface define o método necessário para disponibilização do objeto servidor via o ORB utilizado.

Operations

initOrb(Object, org.omg.CosNaming.NamingContextExt, String, boolean)

Operation Detail

initOrb

```
public abstract void initOrb(Object object, org.omg.CosNaming.NamingContextExt
nameServer, String nomeParaRegistro, boolean chamadaInicial)
```