UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Implementação de Técnicas de Replicação de Componentes de *Software* sobre a Plataforma Aberta CORBA

Dissertação Submetida à Universidade Federal de Santa Catarina
para Obtenção do Grau de Mestre em Engenharia Elétrica

Lau Cheuk Lung

Florianópolis, Maio de 1996

Implementação de Técnicas de Replicação de Componentes de *Software* sobre a Plataforma Aberta CORBA

Lau Cheuk Lung

Esta dissertação foi julgada para obtenção do título de

Mestre em Engenharia

especialidade **Engenharia Elétrica,**

área de concentração **Sistemas de Controle, Automação e Informática Industrial,**

e aprovada em sua forma final pelo Curso de Pós-Graduação.

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Enio Valmor Kassick, Dr.
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Carlos Alberto Maziero, Dr.
Co-orientador

Prof. Jean-Marie Farines, Dr. Ing.

Prof. Orlando Gomes Loques Filho, PhD.

Prof. Raimundo J. de Araújo Macêdo, PhD.

Aos meus pais Lau Shing Lit e Lau Yeung Pik Yee
e aos meus irmãos Lun e Jim.

**"O que é necessário não é a vontade de acreditar,
mas o desejo de descobrir, que é justamente o oposto."**

Bertland Russel

Agradecimentos

Aos Professores Joni da Silva Fraga e Carlos Alberto Maziero pela amizade, pela oportunidade, pelo trabalho de orientação e pelo apoio em todas as etapas deste trabalho.

Ao Professor Jean-Marie Farines pelo conhecimento passado na fase dos créditos e pelo apoio indireto neste trabalho e no projeto ASAP.

Aos membros da banca examinadora pela aceitação de participação e pelas críticas e comentários.

Ao meu amigo Frank Augusto Siqueira pelo companheirismo, pelas inúmeras contribuições e pela grande amizade que foram importantíssimos ao longo desta jornada.

Ao meu amigo Luiz Nacamura Jr. pela grande amizade e pelas várias discussões que tiveram influência direta neste trabalho.

Aos meus amigos que estiveram presentes desde o início dessa jornada: Augusto Loureiro, Max Mauro, Paulo James, Danielle Nishida, Ricardo Martins, André Leal, Cristiane Paim, Evânio, Marcelo Otte, Edvaldo Machado, Udo Fritzke, Olinto Furtado, entre outros.

Ao Professor Roberto Lavôr (UFAM) pelo incentivo.

Ao pessoal da CPGEEL, em especial ao Wilson, por todo auxílio dado.

À UFSC e à CAPES pelo suporte material e financeiro.

A todos meus amigos de baia...

A todos que ajudaram...

A Deus.

SUMÁRIO

Lista de Figuras.....viii

Lista de Tabelas.....ix

Resumo.....x

Abstract.....xi

Capítulo 1 - Introdução.....1

Capítulo 2 - Requisitos para Suporte a Grupo de Objetos em um Ambiente CORBA.....6

2.1 - Introdução.....6

2.2 - Arquitetura CORBA.....7

2.3 - Extensão dos Conceitos CORBA para Inclusão de Grupo de Objetos.....9

2.3.1 - Requisitos CORBA para Grupo de Objetos.....9

2.3.2 - Gerenciamento de Grupo.....10

2.3.2.1 - *Membership*.....11

2.3.2.2 - Serviços de Tratamento de Faltas.....11

2.3.3 - Comunicação de Grupo.....12

2.3.3.1 - Transparência de Grupo de Objetos.....13

2.3.3.2 - Implementação das Comunicações.....15

2.3.4 - Descrição das Interfaces de Grupo Servidor.....16

2.4 - Propostas de ORBs com Suporte para Grupo de Objetos.....17

2.4.1 - ISIS/C++.....17

2.4.1.1 - Modelo de Comunicação de Grupo ISIS/C++.....17

2.4.2 - RDO/C++.....18

2.4.2.1 - Modelo de Comunicação RDO/C++.....19

2.4.2.2 - Arquitetura do RDO/C++.....19

2.4.3 - Orbix+ISIS.....19

2.4.3.1 - Modelo de Grupo do Orbix+ISIS.....20

2.4.3.2 - Arquitetura Orbix+ISIS.....22

2.4.4 - Electra.....23

2.4.4.1 - Modelo de Grupo de Objetos.....24

2.4.4.2 - Arquitetura Electra.....26

2.5 - Conclusões do Capítulo.....27

Capítulo 3 - Técnicas de Replicação.....30

3.1 - Introdução.....30

3.2 - Classificação das Faltas.....30

3.3 - Técnicas de Replicação.....32

3.4 - Aspectos da Abordagem Réplicas Passivas.....33

3.5 - Aspectos da Abordagem Réplicas Ativas.....35

3.6 - Modelos de Réplicas Passivas e Réplicas Ativas.....37

3.6.1 - Modelo de Replicação *Coordinator-Cohort*.....37

3.6.2 - Modelo de Replicação *Viewstamp*.....39

3.6.3 - Modelo de Replicação *Líder/Seguidores*.....40

3.6.4 - Modelo de Replicação Ativa *Competitiva*.....42

3.6.5 - Modelo de Replicação Ativa *Cíclica*.....44

3.7 - Conclusões do Capítulo.....46

Capítulo 4 - Proposta do Modelo de Integração.....47

4.1 - Introdução.....47

4.2 - Modelo de Reflexão Computacional.....47

4.3 - Modelo Reflexivo para integração de Técnicas de Replicação em Sistemas Abertos.....49

4.4 - A Programação de Técnicas de Replicação em um Ambiente CORBA Segundo o Modelo de Integração.....51

4.5 - Especificação das Técnicas de Replicação no Contexto Reflexivo 54

4.5.1 - Replicação Ativa *Competitiva*.....54

4.5.2 - Replicação Ativa *Cíclica*.....58

4.5.3 - Replicação Ativa *Líder/Seguidores*.....59

4.5.4 - Replicação Passiva *Primário/Backup*.....61

4.5.5 - Considerações sobre as Especificações Apresentadas.....63

4.6 - Comparação com Outras Referências.....65

4.7 - Conclusão do Capítulo.....66

Capítulo 5 - Principais Aspectos do Modelo de Integração no Sistema Electra.....67

5.1 - Introdução.....67

5.2 - Implementação de Serviços Replicados no Electra.....67

5.2.1 - Caso Exemplo: Serviço Bancário.....68

5.2.2 - Mecanismos de Gerenciamento de Grupo e de Configuração no ORB/Electra.....71

5.2.3 - Limitações do Electra para as Implementações das Técnicas de Replicação e o Modelo de Integração.....73

5.3 - Implementações.....75

5.3.1 - Implementação da Técnica de Replicação *Competitiva*.....76

5.3.2 - Implementação da Técnica de Replicação *Cíclica*.....77

5.3.3 - Implementação da Técnica de Replicação *Líder/Seguidores*.....77

5.3.4 - Implementação da Técnica de Replicação *Primário/Backup*.....77

5.3.5 - Considerações da Implementação.....79

5.4 - Conclusões do Capítulo.....79

Capítulo 6 - CONCLUSÕES E PERSPECTIVAS.....81

Bibliografia.....85

Anexos.....90

1. Código da Implementação do Exemplo de Aplicação Bancária.....90

2. Código da Implementação dos Meta Controladores.....97

3. Transparências utilizadas na apresentação.....107

Lista de Figuras

Figura 2.1 A estrutura CORBA 2.0....8
Figura 2.2 Transparência de grupo servidor....14
Figura 2.3a Grupo com um membro centralizador....14
Figura 2.3b ORB responsável pela comunicação....14
Figura 2.4 Comunicação no estilo *Proxy/Dispatcher*....15
Figura 2.5 Arquitetura do RDO/C++....19
Figura 2.6 Modelo fluxo de eventos....22
Figura 2.7 Arquitetura do Orbix+Isis....23
Figura 2.8 Comunicação de grupo no Electra....25
Figura 2.9 Arquitetura do Electra....26

Figura 3.1 Grupo réplica passivas....33
Figura 3.2 Mecanismo de *checkpoint*....34
Figura 3.3 Grupo réplicas passivas com mecanismo de *log*....35
Figura 3.4 Grupo réplicas ativas....36
Figura 3.5 Modelo de replicação *Coordinator-Cohort*....38
Figura 3.6 Modelo de replicação....40
Figura 3.7 Modelo de replicação líder/seguidores....41
Figura 3.8 Modelo de replicação competitiva para falhas de temporização....42
Figura 3.9 Modelo de replicação cíclica para falhas de temporização....45

Figura 4.1 Reflexão computacional....48
Figura 4.2 Estrutura reflexiva para modelos de replicação....49
Figura 4.3 Estrutura do modelo sobre um suporte CORBA....50
Figura 4.4 Modelo de programação CORBA....52
Figura 4.5 Interface IDL do servidor replicado....53
Figura 4.6 Processo de construção da aplicação....54

Figura 4.7 Meta-controlador da replicação ativa *competitiva*.....55
Figura 4.8a Diagrama temporal da replicação *competitiva*.....56
Figura 4.8b *crash* do mais rápido.....57
Figura 4.9 Meta-controlador da replicação ativa *cíclica*.....58
Figura 4.10 *crash* da réplica privilegiada 1.....59
Figura 4.11 Meta-controlador da replicação *líder/seguidores*.....60
Figura 4.12 Diagrama temporal da replicação semi-ativa *lider/seguidores*.....61
Figura 4.13 Meta-controlador da replicação passiva *primário/backup*.....62
Figura 4.14 Diagrama temporal da replicação passiva *primário/backup*.....63

Figura 5.1 Modelo de programação Electra/CORBA.....68
Figura 5.2 Exemplo de descrição de interface de uma aplicação bancária.....70
Figura 5.3 *Skeleton* da aplicação bancária.....71
Figura 5.4 Mudança de *membership*.....72
Figura 5.5 Comunicação intra-grupo via ORB.....74
Figura 5.6 Formato da separação do código.....76
Figura 5.7 Replicação primário/*backup*.....78
Figura 5.8 *deadlock* causado pela réplica B.....80

Lista de Tabelas

Tabela 2.1 Comparação das plataformas.....28
Tabela 3.1 Quadro comparativo das três abordagens.....46

RESUMO

Os sistemas informáticos distribuídos têm se caracterizado ultimamente pelo aumento nas dimensões e pela heterogeneidade de seus componentes. Estes sistemas adotam o conceito de sistemas abertos, que busca padronizar as interfaces de seus componentes (*software e hardware*), garantindo uma melhor interoperabilidade entre estes. A programação distribuída aberta é recente e tem sido objeto de diversos padrões e modelos presentes na literatura. O modelo de programação distribuída orientada a objeto CORBA (*Common Object Request Broker Architecture*) é o resultado do trabalho de diversas companhias, integrantes do OMG (*Object Management Group*), no sentido de propor um padrão para permitir a integração de diferentes sistemas de programação e máquinas em ambientes abertos. Pelo fato das especificações CORBA não abordarem em abstrações de grupos de objetos (gerenciamento e comunicação) alguns protótipos e mesmo produtos de plataformas CORBA têm sido desenvolvidos oferecendo suporte para processamento de grupo como uma extensão ao padrão inicial.

Esta dissertação tem por objetivo avaliar a viabilidade do uso de uma plataforma aberta orientada a objetos no padrão CORBA na implementação de serviços replicados, visando aspectos de tolerância a falhas, disponibilidade e flexibilidade. As dimensões e a heterogeneidade em sistemas distribuídos dificultam a implementação de mecanismos para técnicas de replicação a partir de suportes de tempo de execução nestes sistemas. Neste sentido, uma estrutura reflexiva é introduzida no modelo de programação, permitindo que tanto a programação como a integração de diferentes técnicas de replicacão sejam facilitadas nestes sistemas. A abordagem de reflexão computacional adotada no modelo aumenta a flexibilidade permitindo, por exemplo, que se mude protocolos de replicação de acordo com o nível de tolerância a falhas desejado, sem que se tenha qualquer implicação sobre os códigos envolvendo algoritmos da aplicação.

ABSTRACT

Nowadays, distributed computing systems have been characterized by the grown in dimensions and by the heterogeneity of their components. These systems adopt the open systems concept, which standardizes the interfaces of their components (software and hardware) guaranteeing a better degree of interoperability among them. The open distributed programming is recent and has been the purpose of a set of standards and models found in the literature. The distributed object-oriented programmind model CORBA (Common Object Request Broker Architecture) is the result of the work of a group of companies, members of OMG (Object Management Group), with the common goal of propose a standard which allows integration of different programming systems and machines in the field of open systems. The CORBA standard doesn't deal with management and communication of object groups abstractions, but some prototypes and CORBA products were developed offering group processing support as an extension of the initial proposal.

The objective of this loork is to evaluate the viability of the adoption of an open object-oriented plataform built according to the CORBA standard to implement replicated services, dealing with features like fault-tolerance, disponibility and flexibility. The dimensions and the heterogeneity in distributed systems make more difficult the implementation of replication mechanisms using run-time supports in these systems. With this goal, a reflective structure is introduced in the programming model, allowing the integration and programming of different replication mechanisms be easily introduced in these systems. The reflective computational approach adopted in the model raises the flexibility, allowing, for example, that replication protocols be changed according to the desired level of fault-tolerance, without imposing changes in aplication-level algorithms.

CAPÍTULO 1
INTRODUÇÃO

Sistemas Abertos

Os sistemas informáticos distribuídos têm se caracterizado ultimamente pelo aumento nas dimensões e pela heterogeneidade de seus componentes. Estes sistemas adotam o conceito de sistemas abertos que padroniza as interfaces de seus componentes (*software e hardware*), garantindo assim, uma melhor interoperabilidade entre estes. Dentro deste contexto qualquer aplicação distribuída, através de um suporte adequado, deve ser capaz de interagir com as outras independentemente da máquina ou sistema operacional a qual está montado. A programação distribuída aberta é recente e tem sido objeto de diversos padrões e modelos presentes na literatura. Um bom exemplo é o modelo de programação distribuída orientada a objeto CORBA (*Common Object Request Broker Architecture*) resultado do trabalho de diversas companhias, integrantes do OMG (*Object Management Group*), no sentido de propor um padrão para permitir a integração de diferentes sistemas de programação e máquinas em ambientes abertos [OMG 93]. Além do modelo CORBA, no contexto de sistemas distribuídos abertos, várias arquiteturas e padrões têm sido propostos de maneira a permitir a interoperabilidade de diferentes componentes de *software* e *hardware*. O ODP - *Open Distributed Processing* - da ISO/CCITT [ISO 93], Arquitetura ANSA [Oskiewicz 93] e a plataforma DCE/OSF - *Distributed Computing Enviroment* - [OSF 92] são exemplos desses esforços. Entre estes padrões, apenas o modelo ANSA apresenta suporte para processamento em grupo.

O conceito de processamento de grupo tem sido introduzido em modelos de programação distribuída no intuito de dar suporte a aplicações que necessitam de alto desempenho, de disponibilidade de recursos quando da concorrência no uso compartilhado e de mecanismos para tolerância a falhas por razões de confiabilidade, além de possibilitar o uso em aplicações do tipo trabalho cooperativo (*groupware*). O padrão CORBA atualmente evolui no sentido de incorporar serviços de grupo. As especificações *Group Server* estão sendo desenvolvidas na OMG para inclusão na arquitetura OMA [Adler 95]. A proposta *Group Server* é similar à abordagem usada no ANSA, apresentando um elemento concentrador das comunicações de grupo, o que é um *handicap* no desempenho e na confiabilidade de um sistema.

Por outro lado, alguns protótipos e mesmo produtos de plataformas CORBA têm sido desenvolvidos oferecendo suporte para processamento de grupo. Em particular, podemos citar os ORBs (*Object Request Brokers*) RDO/C++, Orbix+Isis e Electra. Estas iniciativas se utilizam de ferramentas de baixo nível para aplicações distribuídas tais como o Isis e o Horus, que fornecem comunicação de grupo fundamentadas sobre a noção de disseminação confiável (*reliable broadcast*). As ferramentas citadas fornecem bases mais confiáveis que as soluções procuradas nas especificações de *Group Server* da OMG.

Modelos de Replicação

As técnicas de replicação são uma alternativa para que serviços continuem a ser fornecidos em sistemas distribuídos, mesmo em presença de nós falhos. A unidade de replicação ou réplica é um componente de software (módulo, objeto, processo, etc.), encapsulando dados identificados como estado da réplica. As réplicas são distribuídas entre diferentes nós da rede. A coordenação em uma replicação define como as diferentes réplicas devem interferir no processamento, no sentido de manter a consistência e a transparência do conjunto.

O objetivo deste trabalho é buscar um modelo de integração que facilite a programação de técnicas de replicação em sistemas abertos. A nossa intenção é usar de forma extensiva um suporte para processamento de grupo na programação destas técnicas. A implementação de

técnicas de replicação sobre um ambiente heterogêneo não é uma tarefa fácil. Outro aspecto a considerar são as dimensões envolvidas em um sistema aberto. O aspecto da flexibilidade na implementação das técnicas é um requisito relevante. Usualmente, as abordagens de implementação de técnicas de tolerância a faltas ou são efetuadas a partir de suporte de tempo de execução, ou usando bibliotecas de funções ou ainda integrando-as diretamente na programação da aplicação [Fabre 95]. A implementação de técnicas de tolerância a faltas a partir do suporte fornece um grau de transparência sobre os aspectos de coordenação da técnica usada a nível da aplicação. A desvantagem é que uma vez definidas as premissas de falhas e a técnica de replicação a ser usada, teremos definido (em tempo de configuração) um suporte de execução específico. Mudanças implicariam na necessidade de reconfiguração, criando um novo suporte. As abordagens de biblioteca e de linguagens para a implementação trazem os aspectos de coordenação a nível de programador, porém sem separá-los dos aspectos funcionais da aplicação, e portanto apresentam baixa flexibilidade..

Uma nova alternativa, adotada neste trabalho, é a introdução das técnicas de replicação a nível do modelo de programação usando os conceitos de reflexão computacional. Em termos gerais, o paradigma *reflexivo* torna possível uma aplicação controlar seu próprio comportamento atuando sobre si mesma. Para tanto, a parte funcional (nível *ba*se) é separada das partes não-funcionais (nível *meta*) de uma aplicação, ou seja, o nível *base* conterá os métodos e procedimentos da aplicação em si, enquanto a nível *meta* são tratadas as funções de controle e gerenciamento da aplicação. A reflexão computacional permite a independência dos códigos das réplicas em relação aos protocolos de coordenação, conduzindo a uma grande flexibilidade no sistema: mudar de técnica ou alterá-la para atender a graus de tolerância a falhas desejados podem resultar em simplesmente trocar os protocolos de coordenação a nível meta, sem implicar na alteração de algoritmos de aplicação, ou ainda em mudanças a nível de suporte de execução, o que seria difícil em sistemas abertos.

Este trabalho objetiva verificar a viabilidade do uso de uma plataforma aberta orientada a objetos no padrão CORBA na implementação de serviços replicados, visando aspectos de tolerância a falhas, disponibilidade e flexibilidade. Diante disso, foram feitos vários estudos

em literaturas específicas no intuito de definir uma solução mais adequada aos problemas citados acima. Os objetivos primários desta dissertação foram:

⇒ levantamento de um conjunto de conceitos e requisitos necessários para que o modelo de programação CORBA seja estendido para suportar a noção de grupo;

⇒ estudo das várias propostas de ORBs que incluem suporte para grupo encontrados na literatura e realizar uma análise comparativa à luz dos conceitos e requisitos identificados;

⇒ estudo de alguns modelos de replicação de componentes de *software* para diferentes tipos de falta;

⇒ propor um modelo de integração que ofereça uma maior flexibilidade na implementação de técnicas de replicação em sistemas abertos;

A dissertação está estruturada da seguinte maneira: no capítulo 2 é apresentado um estudo sobre conceitos e requisitos necessários para suporte ao processamento em grupo, em seguida são apresentados vários Orbs com suporte para grupo de objetos; no capítulo 3 é apresentado um estudo sobre modelos de replicação de componentes de *software*; no capítulo 4 é apresentada uma proposta de um modelo de integração para técnicas de replicação em sistemas abertos; no capítulo 5 é apresentada a implementação de técnicas de replicação usando o modelo de integração sobre uma plataforma CORBA. No capítulo 6 são apresentados as conclusões e perspectivas deste trabalho.

O trabalho de dissertação apresentado aqui faz parte de um projeto cooperativo de pesquisa PROTEM/CC, que tem como propósito construir um ambiente orientado a objetos que suporte aplicações distribuídas com requisitos de tolerância a falhas e tempo-real, em um contexto de sistemas abertos. Deste projeto, denominado ASAP (*Ambiente para Suporte a Aplicações Distribuídas Baseado em Objetos*), participam equipes de pesquisadores das instituições Universidade Federal de Santa Catarina (UFSC), Universidade Federal

Fluminense (UFF), Universidade de Campinas (Unicamp) e Universidade Federal do Ceará (UFC).

CAPÍTULO 2

REQUISITOS PARA SUPORTE A GRUPO DE OBJETOS EM UM AMBIENTE CORBA

2.1 Introdução

O conceito de grupo de processamento tem sido introduzido em modelos de programação distribuída no intuito de representar atividades de trabalho cooperativo (*groupware*), de possibilitar o aumento da disponibilidade de recursos quando da concorrência no uso compartilhado ou ainda, no processamento replicado por razões de tolerância a falhas.

A arquitetura CORBA é uma plataforma aberta definida através de um conjunto de especificações e de conceitos para que objetos distribuídos, em um ambiente de rede heterogêneo, possam ser acessíveis através de uma interface bem definida. No entanto, as especificações do CORBA inicialmente não apresentaram uma visão muito clara em termos de conceitos e de necessidades de suporte para a introdução da noção de grupo de objetos na programação distribuída aberta. Devido a isto, o padrão CORBA tem sido objeto de extensão em vários protótipos e mesmo produtos para garantir o suporte para grupo. Em particular, podemos citar os ORBs (*Object Request Broker*): RDO/C++ [ISIS 94], Orbix+Isis [IONA 95] e Electra [Maffeis 95a] que incluem a noção de grupo de objetos.

O objetivo deste capítulo é levantar um conjunto de requisitos desejáveis para o processamento em grupo em arquiteturas abertas. Queremos também rever neste contexto, alguns conceitos CORBA. Em seguida, apresentaremos uma descrição concisa dos ORBs de comunicação que incluem suporte para processamento em grupo de objetos; uma comparação

entre estes é feita à luz dos requisitos e conceitos identificados. O intuito deste estudo é verificar a adequação destas proposições de ORBs a modelos usuais de técnicas de replicações encontrados na literatura. A escolha de uma destas proposições de ORB e as necessidades de desenvolvimento para atender aos requisitos citados e fornecer suporte a implementação de diferentes modelos de replicação de componentes de *software* são alguns dos objetivos deste trabalho que coincidem com o projeto ASAP. A discussão sobre requisitos de grupo e os conceitos CORBA nos próximos itens deste capítulo está fortemente fundamentada no trabalho descrito em [ISIS 93] e [Liang 90].

2.2 Arquitetura CORBA

O OMG (*Object Management Group*) é uma organização formada por um grupo de mais de 500 empresas (tais como IBM, SUN, DEC, Microsoft, Novell, etc) com o objetivo de especificar um padrão aberto para a programação distribuída orientada a objeto chamado arquitetura CORBA. Esta arquitetura é um conjunto de mecanismos padronizados e de conceitos que se baseiam no modelo cliente-servidor em que objetos distribuídos encapsulam um estado interno e se fazem acessíveis através de uma interface bem definida. A estrutura CORBA é constituída por um ORB (*Object Request Broker*) que implementa abstrações e semânticas da comunicação entre objetos em um sistema distribuído. O ORB fornece os mecanismos de representação de objetos e de comunicação para que seja possível o processamento das requisições no lado servidor em um ambiente aberto. A figura 2.1 ilustra os componentes de um ORB.

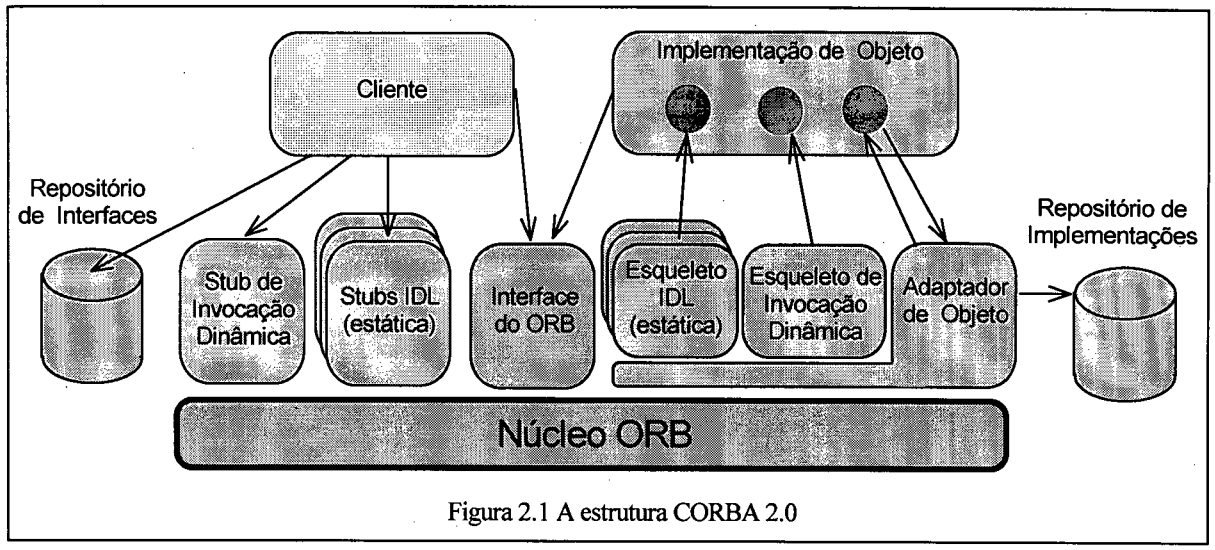


Figura 2.1 A estrutura CORBA 2.0

Em um ambiente CORBA, cada objeto tem sua interface padrão especificada em IDL (*Interface Definition Language*). O cliente pode requisitar serviços estaticamente através de *Stubs* IDL gerados no processo de compilação da especificação IDL da interface, ou construir um pedido usando o *Stub* de chamada dinâmica (DII) e o repositório de interfaces. Os dois casos citados de requisição são tratados usando as mesmas estruturas no lado servidor: uma requisição transferida através de um ORB determina a transferência de controle a uma implementação de objeto usando um *esqueleto* IDL, próprio ao adaptador do objeto que deve tratar a operação desejada. Na execução de uma requisição, a implementação de objeto pode acessar os serviços oferecidos pelo ORB através do adaptador de objeto básico (BOA). Os serviços oferecidos pelo BOA, geralmente apropriado para classes específicas de objetos, podem ser: geração e interpretação de referência dos objetos, invocação de métodos, ativação e desativação de objeto e implementações, mapeamento de referência do objeto para implementação, etc. A arquitetura CORBA permite também a implementação de diferentes adaptadores de objeto de acordo com as necessidades da implementação de objeto. A interface do ORB é a mesma para todas as implementações de objeto, independente do adaptador de objeto utilizado. As interfaces ORB e os adaptadores de objetos são disponíveis para o gerenciamento no modelo.

2.3 Extensão dos Conceitos CORBA para Inclusão de Grupo de Objetos

A noção de grupo determina uma associação em que seus membros apresentam uma relação abstrata comum, uma política interna comum e/ou uma regra de acesso comum [Lea 94]. Um grupo de objetos deve estar dentro destas condições de associação. Modelos de processamento em grupo são essenciais em aplicações distribuídas, contribuindo para requisitos de maior disponibilidade de recursos, tolerância a falhas, distribuição de carga, etc. Um grupo de objetos, como um objeto qualquer, encapsula um estado e se faz acessível através de um conjunto de métodos bem definidos, permitindo, por exemplo, a realização de serviços de transferência de estado entre réplicas do grupo. Requisitos de transparência de grupo podem ser colocados tanto a nível de cliente como de servidor: objetos simples e grupos de objetos não são distinguíveis externamente. Características como componentes reusáveis, presentes no paradigma objeto, devem se manter quando em relações de grupo.

Para que possamos usufruir das vantagens do processamento de grupo em modelos de programação orientado a objetos em ambientes abertos, é necessário definir mecanismos de suporte e avaliar a integração destes mecanismos na estrutura destes sistemas. Estes aspectos serão tratados na sequência deste capítulo.

2.3.1 Requisitos CORBA para Grupo de Objetos

As questões que se colocam neste contexto são: que requisitos de serviços de grupo desejamos ? Como integrar grupos de processamento em sistemas abertos construídos, fazendo uso de padrões e plataformas consagradas ? É evidente que os requisitos dependerão do modelo de processamento de grupo usado e estes terão reflexos em vários aspectos do suporte, envolvendo o gerenciamento e a comunicação de grupo (itens 2.3.2. e 2.3.3).

A integração de serviços baseados na noção de grupo em uma arquitetura CORBA pode se dar segundo diferentes abordagens. Podemos ter [ISIS 93]:

- um objeto do grupo como intermediário registrado no ORB, servindo de ponte nas comunicações de clientes CORBA, usuários do serviço do grupo, com os demais elementos do grupo. Os elementos do grupo não identificados no ORB formam no sistema o que poderíamos chamar de componentes de segunda classe. Neste caso, o elemento que serve de ponte entre o grupo e o ORB se constitui em um elemento central para a confiabilidade e o desempenho dos serviços do grupo.

- uma biblioteca no cliente fornecendo a habilidade de se comunicar com um grupo de servidores. Nesta situação a transparência no cliente é comprometida, pois o cliente deve ser reconectado ao grupo usando serviços de comunicação ponto-a-ponto através do ORB a cada mudança nos objetos do grupo (número de objetos no grupo). Estas bibliotecas operariam sobre o ORB tratando com a dinâmica do funcionamento do grupo.

Qualquer solução de suporte que não envolva a inclusão de abstrações de grupo em ORBs deve impor graus de desempenho baixos, além de comprometer as transparências de cliente e servidor. Então, é importante que grupo de objetos sejam suportados de forma transparente pelos ORBs como qualquer objeto e que interfaces de grupo sejam definidas em IDL. Este é o princípio básico que deve ser seguido no sistema ASAP no referente a grupo de objetos. A adição de novos mecanismos não devem causar nenhum impacto no desempenho de comunicações ponto a ponto através do ORB. Nos próximos itens discutiremos aspectos da integração do gerenciamento e da comunicação de grupo em arquiteturas abertas.

2.3.2 Gerenciamento de Grupo

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

[1] lista de membros ativos no grupo

2.3.2.1 *Membership*

Este é um dos principais mecanismos que um suporte a grupos de objetos precisa oferecer para garantir aplicações distribuídas robustas. O mecanismo é caracterizado por um algoritmo que atualiza as entradas e saídas (normais ou por falha) de objetos no grupo, mantendo uma lista de membros (*membership*) corrente. É necessário que a interface *Basic Object Adapter* (BOA) seja estendida com operações como *addmember* e *delmember* ou ainda que se crie uma outra interface *Group Object Adapter* (GOA) como um subtipo do BOA englobando o *membership* e outras operações de gerenciamento. Um *groupview* ou lista de membros é necessário pelo ORB para a coordenação de suas execuções no grupo e para implementar mapas de comunicação de grupo. Portanto, mudanças de *membership* devem ser comunicadas a todos os membros do grupo. Estas comunicações de mudanças devem ser recebidas na mesma ordem pelos componentes. O uso de serviços de disseminação confiável com ordem total é imprescindível para tal. As informações e o serviço de *membership* não precisam estar diretamente contido no ORB, mas ele deve ser o responsável pela notificação confiável destas informações.

2.3.2.2 Serviços de Tratamento de Faltas

Este serviço inclui a detecção de objetos com comportamento faltoso e o respectivo tratamento. Em grupos de objetos é necessário criar mecanismos para detectar a ocorrência de falha em cada membro do grupo prevenindo que o cliente fique esperando indefinidamente a resposta; isto pode ser obtido através de mecanismos de *timeout* e *keepalive*, que detectam faltas com semânticas de omissão e de *crash*. Um serviço deve ser responsável por coletar todas as informações de *timeout* e *keepalive* e produzir um fluxo único confiável e coerente de notificação de falhas. Na implementação deste serviço em um ORB é preciso decidir a classe de faltas a ser detectada. Além disso, a notificação de falha de um objeto precisa ser difundida a todos membros do grupo de forma totalmente ordenada para que estes tenham a mesma visão do objeto faltoso.

Serviços de transferência de estado são importantes no tratamento de faltas, na re-inserção de objetos no grupo. A transferência pode se dar através de uma cópia consistente do estado corrente de um objeto, de forma que um novo objeto entrando no grupo possa ao receber a cópia, tornar-se uma réplica idêntica às outras. Serviços de *log* e de *checkpoint* podem periodicamente copiar em disco estados de um objeto e requisições subsequentes ao grupo, caracterizando *checkpoints* e *logs* que poderiam ser usados na recuperação do grupo em situação de falhas em todos os membros. Este serviço deve ser implementado na interface BOA. O uso de serviços de *membership* disponíveis nesta interface devem ser usados no tratamento de elementos faltosos ou na re-inserção de um objeto onde a mudança de *membership* só se efetiva quando a transferência de estado tiver sido sucesso.

2.3.3 Comunicação de Grupo

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

- transmitir o pedido a um membro privilegiado ("objeto primário");
- enviar o pedido de forma aleatória a um membro do grupo;
- enviar o pedido a todos os membros do grupo (modelo Máquina de Estado)

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

- o primeiro resultado a chegar é passado ao cliente, os restantes são descartados;

- concatenar os resultados em sequências e enviar ao cliente. Esta alternativa é útil em modelos de processamento cujos membros fornecem alguma forma de processamento paralelo;

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

A escolha do mapeamento no envio de pedidos e da coleta de resultados deve ser especificada como parte da implementação do processamento replicado, e portanto estar armazenada no repositório de implementações [ISIS 93]. O ORB pode consultar estes mapas quando da ativação de um grupo. A execução dos mapas de envio e a coleta de resultados pelo ORB é importante para se obter os graus de transparência desejados.

2.3.3.1 Transparência de Grupo de Objetos

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

- Transparência do servidor: os objetos membros de um grupo servidor são vistos pelos seus clientes como se fossem um objeto servidor simples que implementa a mesma interface. O ORB, ao definir a execução de um mapa de envio para requisições e também a coleta de resultados, estará contribuindo na implementação do servidor.

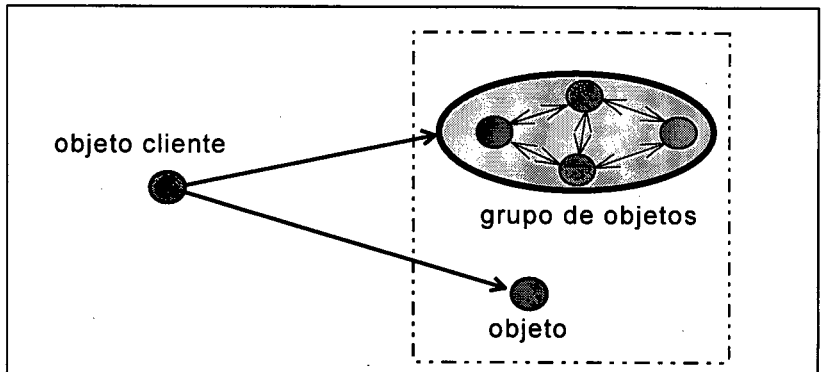


Figura 2.2 Transparência de grupo servidor

- Transparência do cliente: as requisições enviadas de um grupo ou de um objeto simples para um objeto servidor devem ser equivalentes e tratadas como as enviadas por um cliente único. Isto se consegue ou através do modelo de grupo utilizado no cliente em que um dos membros é o responsável pelas interações externas ao grupo (figura 2.3a); ou através de um mecanismo no ORB que capture as múltiplas requisições replicadas de um grupo cliente e as transforme em apenas uma emissão de requisição no servidor (figura 2.3b).

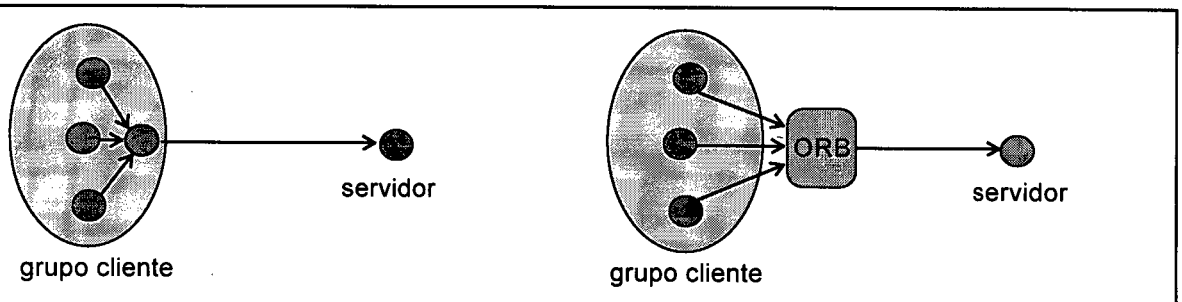


Figura 2.3a Grupo com um membro centralizador

Figura 2.3b ORB responsável pela comunicação

Estas formas de transparência nas comunicações são melhor suportadas utilizando mecanismos de *proxy* e *dispatcher* incluídos no ORB [Shapiro 86] [Hagsand 92]. A figura 2.4 mostra um modelo de implementação para comunicação de grupo usando estes mecanismos. Os *stubs* e *proxies* do lado dos clientes e os *dispatchers* e *skeletons* do lado dos objetos no grupo servidor são gerados a partir da compilação de uma especificação de interface de um grupo servidor. O *stub* residente no espaço de endereçamento de cada membro cliente oferece

uma interface de serviço idêntica à de ativação de um método de um objeto servidor simples local. O *proxy* é uma versão local do grupo de objetos servidor; a comunicação de um membro cliente com um grupo se dá através de seu *proxy*. O *proxy* distribui uma requisição de serviço entre os objetos do grupo servidor usando os serviços de um protocolo de comunicação (item seguinte). Em cada sítio receptor, o *dispatcher* localiza o membro do grupo servidor e passa a requisição ao *skeleton* que, por sua vez, ativa o método.

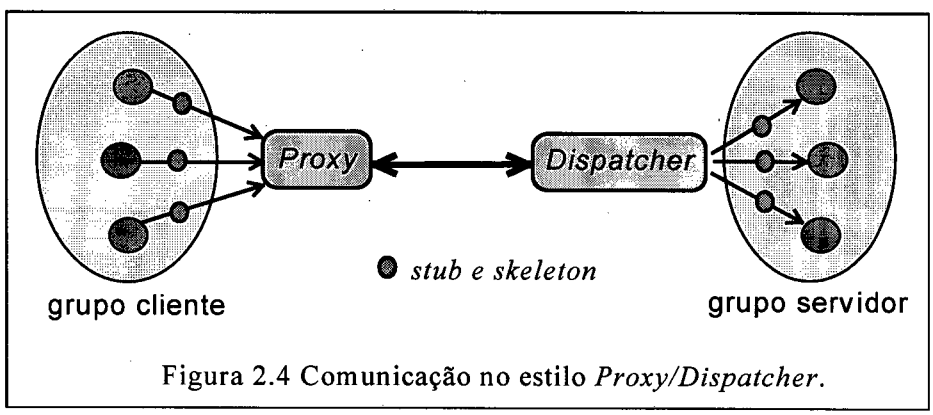


Figura 2.4 Comunicação no estilo *Proxy/Dispatcher*.

2.3.3.2 Implementação das Comunicações

A noção de grupo implica no uso de um identificador único para um conjunto de objetos que compartilham alguma característica comum, de modo que o grupo de objetos seja visto por outros objetos da aplicação como um objeto simples. A aplicação não precisa se envolver na expansão dos endereços de grupo dentro das listas de destinos. O ORB deve ser visto como canalizando pedidos e respostas de serviços, mediando interações através de protocolos apropriados. Usando um serviço de protocolo *multicast,* envolvendo primitivas de interação multi-ponto, em muito o desempenho das comunicações e a complexidade do ORB são melhorados. Se este protocolo *multicast* fornecer diferentes tipos de ordenação o ORB, além da simplicidade envolvida, poderá suportar diferentes modelos de replicações. Três tipos de ordem usualmente apresentados são ordem total, ordem causal e ordem FIFO [Dueñas 92]:

- Ordem total: assegura que múltiplas requisições emitidas por diferentes clientes serão recebidas na mesma ordem pelos membros objetos servidores. Este tipo de ordenação é importante na implementação do modelo de grupo Máquina de

Estado [Schneider90]. Neste modelo, o determinismo de réplica assegura que nas condições de garantia de ordenação total e de recepção de todas as requisições por parte dos membros, cada objeto (réplica ativa) pode atender sua fila de requisições sem necessidades de comunicação ou coordenação com os demais membros do grupo.

- Ordem causal: nesta ordenação as mensagens são recebidas obedecendo relações de precedência entre eventos. Esta ordenação previne erros de consistência em sequências de requisições relacionadas a eventos, mesmo quando estas requisições são emitidas por diferentes clientes.

- Ordem FIFO: assegura que as requisições de um cliente são recebidas em todos os membros do grupo na ordem de emissão. Ordenações FIFO em *multicast's* são normalmente usadas em comunicações assíncronas *one-way*, pois podem fornecer um mesmo tipo de ordenação de comunicação síncrona, sem implicar nos custos envolvidos nestas últimas com os *replies* a nível de aplicação.

Os mecanismos de comunicação não são visíveis ao cliente, e devem estar disponíveis para as interfaces de invocação estática ou dinâmica (DII) ou para o proxy gerado a partir da interface IDL do grupo através do ORB.

2.3.4 Descrição das Interfaces de Grupo Servidor

O conjunto de serviços que podem ser acessados pelos objetos clientes são especificados através da linguagem de definição de interface IDL. A IDL então serve para descrever o formato de cada chamada dos métodos oferecidos pelo objeto e dos parâmetros (atributos) necessários para efetuá-las. Em um arquivo de definição de interface de um grupo de objetos estão presentes todas as informações necessárias à geração do suporte necessário para que um cliente possa ativar um método do grupo. A IDL deve ser puramente declarativa, sem definições de variáveis ou estruturas algorítmicas. É desejável que a descrição de interface de grupo de objetos seja compatível com o modelo IDL do CORBA [OMG 94] e suportada pelo

ORB. Para isso, é necessário adicionar novos tipos de declarações para grupo de objetos ou considerar que qualquer implementação de objeto seja, por *default*, tratada como grupo de objetos, mesmo que seja um grupo contendo apenas um objeto membro. Assim, é possível que a IDL para grupo seja totalmente idêntica ao IDL do CORBA convencional, pois ao se compilar a IDL de grupo todas as abstrações de grupo seriam tratadas a nível de suporte, garantindo a total transparência da aplicação.

2.4 Propostas de ORBs com Suporte para Grupo de Objetos

Nos próximos itens deste capítulo serão descritas as propostas de suporte que seguem os padrões CORBA (exceto o ISIS/C++) e que incluem abstrações para processamento em grupo de objetos. Algumas destas propostas já se apresentam na forma de produtos (Orbix+Isis e o RDO/C++). O nosso intuito é comparar estes suportes na ótica de alguns dos requisitos levantados nos itens anteriores. Este estudo tomou como base o levantamento realizado em [Lau 95].

2.4.1 ISIS/C++

O ISIS/C++ [ISIS 91] é uma interface que abstrai os serviços da ferramenta ISIS convencional para permitir o desenvolvimento de aplicações distribuídas orientadas a objeto. A linguagem de definição de interface não é compatível com a IDL do padrão CORBA/OMG. A interface ISIS/C++ é implementado em C e C++ sobre as versões 2.1 e 3.0 do ISIS.

2.4.1.1 Modelo de Comunicação de Grupo ISIS/C++

O ISIS/C++ se baseia no modelo cliente/servidor, onde os serviços são implementados em grupos de objetos. Uma especificação de interface, definindo os serviços de um grupo de objetos, é usada no processo de compilação para gerar *stubs* e *proxies*, permitindo então a comunicação do cliente com os objetos servidores. Os *stubs* são integrados aos objetos-cliente e aos objetos membros do grupo por herança, permitindo então ao cliente dispor de serviços de grupo como uma extensão de linguagens orientadas a objeto.

Para realizar a comunicação com um objeto servidor, um cliente acessa uma versão local do objeto ou grupo de objetos remoto, chamado *proxy*, e a *stub* cliente associada oferece acesso ao serviço de grupo via *proxy*. Um *stub* servidor implementa uma interface de comunicação para o objeto servidor. Um objeto servidor herda as funcionalidades de comunicação do *stub* servidor.

Os métodos de comunicação de grupo no ISIS/C++ são fornecidos pelo *proxy*, e podem ter três variantes: `call` (comunicação síncrona - pedido/resposta), `Bcast` (comunicação assíncrona - *one way*), `request/collect` (comunicação assíncrona bi-direcional - pedido/resposta) e fazem uso dos protocolos de comunicação de grupo disponíveis no ISIS. No ISIS/C++ grupos de objetos são implementados usando grupos de processos ISIS e operando no modelo *sincronismo virtual* [Birman 91b]. Além disso, podem ser usados serviços ISIS de gerenciamento de grupo (*membership*, e transferência de estado) através da interface ISIS/C++.

2.4.2 RDO/C++

O RDO/C++ [ISIS 94] é um ambiente de desenvolvimento para aplicações distribuídas orientadas a objeto. Este ambiente oferece um suporte de execução que faz uso também dos serviços oferecidos pelo ISIS. O RDO/C++ pode ser considerado uma evolução do ISIS/C++, onde a sua linguagem de definição de interface é compatível com a IDL do padrão CORBA e onde, também, são oferecidas facilidades necessárias para implementação de aplicações distribuídas orientadas a objeto, permitindo a interação de objetos C++ dentro de um espaço de aplicação, executando-se em diferentes processos e mesmo em diferentes processadores.

A ferramenta RDO/C++ contém bibliotecas que simplificam o uso do ISIS em muitas aplicações C++ distribuídas. É capaz de suportar uma variedade de modelos distribuídos, tais como: grupos réplicas ativas (Máquina de Estado), grupos com processamento paralelo (um programa é particionado entre os membros por questões de desempenho ou de distribuição de carga) ou ainda grupos na forma de *Publisher/Subscriber* (item 2.4.3.1).

2.4.2.1 Modelo de Comunicação RDO/C++

Os mecanismos de comunicação são similares ao modelo *chamada de procedimento remoto* (RPC). Na comunicação de grupo, o RDO/C++ utiliza objetos *proxies* e *dispatcher*s que permitem a realização de chamadas de métodos em objetos ou grupo remotos. No desenvolvimento de uma aplicação do tipo cliente/servidor no RDO/C++, é utilizada a interface IDL do servidor. Na compilação desta descrição de interface é gerado um conjunto de classes C++ que implementa o servidor remoto com a interface definida. Nestas classes se incluem uma classe *proxy* usada no cliente e uma classe *dispatcher* para cada membro definido.

2.4.2.2 Arquitetura do RDO/C++

O modelo cliente/servidor implementado neste suporte é similar ao definido nos padrões CORBA/OMG. Mas o fato de estar fundamentado nos protocolos ISIS permite ao RDO/C++ ser aplicado na implementação de sistemas distribuídos que fazem uso de servidores replicados em diferentes modelos de grupo. A figura 2.5 apresenta de forma sucinta a estrutura de uma aplicação distribuída que faz uso do RDO/C++ como ORB.

Figura 2.5 Arquitetura do RDO/C++

2.4.3 ORBIX+ISIS

A ferramenta Orbix+Isis [IONA 95] é um produto comercial desenvolvido em conjunto entre as empresas ISIS Distributed Systems Inc. e IONA Technologies, Ltd. Como o anterior, é um suporte que permite um sistema ser construído como um conjunto de objetos interagindo

segundo o modelo CORBA/OMG. O Orbix+Isis simplifica o desenvolvimento e a integração de aplicações distribuídas tolerantes a faltas. Cada objeto tem uma interface bem definida e especificada em uma linguagem de definição de interface (IDL), compatível CORBA. O ORB neste caso é simplificado pelos recursos ISIS na implementação das abstrações de grupo. Mecanismos de *membership*, transferência de estado e comunicação *multicast* confiável com diferentes tipos de ordenação, fornecidos pelo Isis, são usados pelo ORB no suporte às aplicações Orbix+Isis.

2.4.3.1 Modelo de Grupo do ORBIX+ISIS

No modelo Orbix+Isis, os servidores formam um grupo de objetos C++ replicados ou associados onde a transferência das interações cliente/servidor é confiável. Grupos de objetos podem ser definidos em cima de dois estilos de execução: *processamento de grupo*[2] e *fluxo de eventos* (*event stream*).

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

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

[2] na literatura original o termo utilizado é *réplicas ativas*, utilizamos *processamento de grupo* por este ser bem menos restrito do que o apresentado no documento original.

comportamento do *proxy default* e adicionar a capacidade de enviar todos os resultados ao cliente.

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

- *Coordinator/Cohort*: neste estilo (capítulo 3), o *coordenador* executa a operação e então envia os resultados ao cliente e a todos os *cohorts* para que estes usem os resultados para a atualização de seus estados. Uma função *chooser* determina o *coordenador* para o processamento da requisição do cliente. Se o coordenador falha a função *chooser* é ativada automaticamente para a escolha do novo *coordenador*.

Neste estilo, *processamento de grupo*, o Orbix+Isis oferece ainda serviços de *membership*, transferência de estado e ordenação de requisições. Transferência de estado e *membership* são serviços chave para diferentes modelos de processamento de grupo, porque permite novos objetos se juntarem ao grupo e se tornarem réplicas exatas.

No estilo de execução "*fluxo de evento*", os clientes usando apenas comunicações assíncronas *one-way* enviam mensagens (eventos) para objetos membros de um grupo, chamados de "*receptores de eventos*". Este estilo de execução segue o modelo de grupo *publisher/subscriber*. Neste modelo, clientes enviam assincronamente mensagens ao *event stream* que mantém estes eventos e direciona os mesmos aos objetos *subscribers* (receptores de evento). Os receptores de evento podem se juntar ou deixar o grupo de *subscribers* em qualquer momento. A participação no grupo de um *subcriber* pode ser programada para a recepção de uma quantidade determinada de eventos. O estilo de execução *fluxo de evento* desacopla os clientes dos servidores, pois os objetos clientes, que mandam mensagens para o

fluxo de evento, não se preocupam se os receptores de eventos estão ou não ativos. A entidade *fluxo de evento* é replicada e tolerante a falta, implementada a partir de um grupo e se executando no estilo replicas ativas. Um número específico de eventos é salvo em um histórico de eventos para o caso de passar os eventos anteriores a um novo receptor de evento incluído no grupo. A figura 2.6 sintetiza este estilo de grupo:

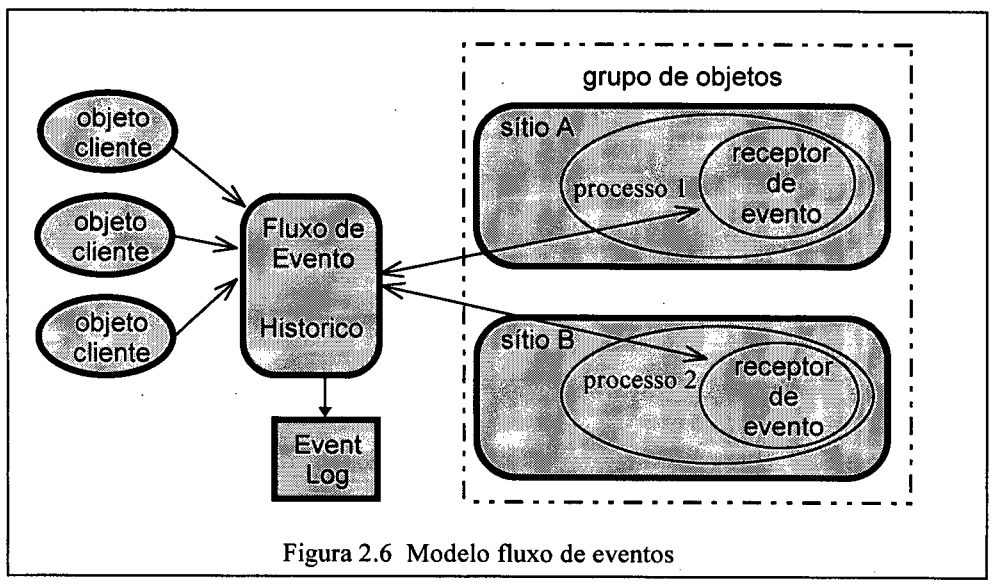


Figura 2.6 Modelo fluxo de eventos

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

2.4.3.2 Arquitetura ORBIX+ISIS

A arquitetura de um sistema fazendo uso do Orbix+Isis está apresentada na figura 2.7. O Orbix+Isis consiste de biblioteca de classes C++ e um suporte de execução que implementa as funcionalidades dos estilos de execução de grupo (*processamento de grupo* e *fluxo de evento*). Nesta estrutura convivem um ORB convencional (Orbix) e o Orbix+Isis com suporte para grupo. A camada de comunicação Orbix trata de comunicações ponto a ponto. A camada de comunicação Orbix+Isis usa as facilidades Isis no suporte para grupo permitindo então que se ofereçam as bases para aplicações distribuídas tolerantes a falhas.

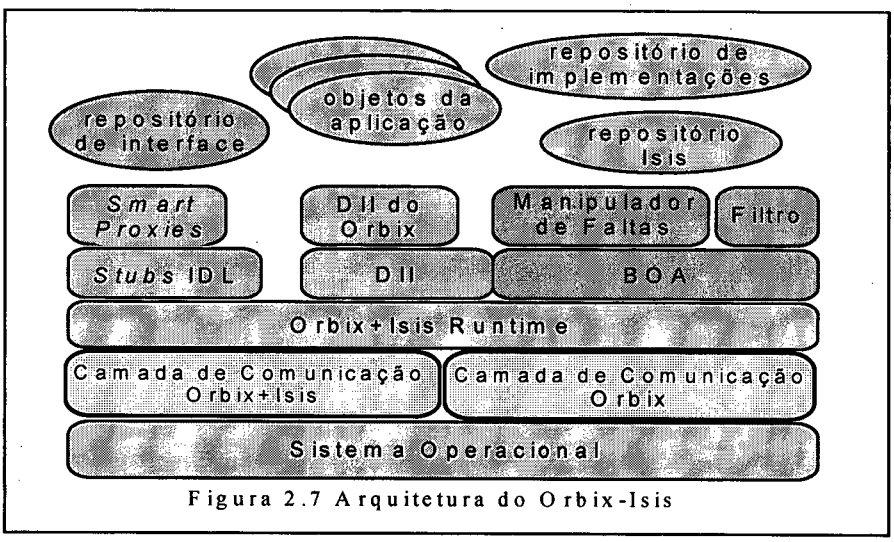


Figura 2.7 Arquitetura do Orbix-Isis

A linguagem IDL Orbix, compatível ao CORBA, foi estendida para permitir a geração de códigos requeridos para a construção de aplicações tolerantes a faltas no Orbix+Isis. As interfaces IDL de grupo no Orbix+Isis geram na compilação as estruturas necessárias para um dos estilos de execução no grupo servidor. Usando estas estruturas (*stubs, proxy*, etc..), o cliente conecta e comunica com um grupo de objetos como se este fosse um objeto Orbix simples. O repositório Isis (`IsR`) é uma hierarquia de arquivos do Orbix+Isis similar ao repositório de implementação do Orbix. Cada servidor Orbix+Isis tem seu arquivo no `IsR`, que define os procedimentos de ativação e de configuração do grupo de servidores. Os aspectos de desempenho e de estilo de execução do grupo são especificados no repositório Isis (IsR). As informações disponíveis nestes arquivos permitem, por exemplo, mudanças na aplicação envolvendo o estilo de execução de grupo, sem a necessidade de modificação em código ou de novas recompilações.

2.4.4 ELECTRA

O Electra [Maffeis 95b] é um *Object Request Broker* (ORB) compatível com o padrão CORBA [OMG 93], cuja arquitetura suporta também mecanismos de tolerância a falhas para aplicações distribuídas, através do uso dos conceitos de grupo de objetos. Para o desenvolvimento de aplicações distribuídas, este modelo combina os benefícios do padrão

CORBA com o poder de ferramentas de mais baixo nível, tais como: Isis [Birman 91b], Horus [Renesse 95], Transis [Amir 92], Consul [Mishra 93], Chorus [Chorus 92], entre outros. Na atual versão do Electra é possível torná-lo operacional sobre as plataformas de suporte Isis e Horus, mas, novos adaptadores podem ser desenvolvidos para outras ferramentas, o que o torna um sistema flexível.

O Electra é implementado em C++ e o mapeamento IDL para C++ está de acordo com o especificado pela OMG [OMG 94]. A IDL do Electra é idêntica às especificações da OMG, pois o modelo considera as implementações de objeto como grupo de objetos (ver item 2.3.4). Mecanismos de herança são possíveis desde que os membros do grupo sejam do mesmo tipo, instâncias da mesma interface ou pelo menos ter uma interface antecessora em comum, onde só as operações herdadas deste antecessor podem ser difundidas ao grupo.

2.4.4.1 Modelo de Grupo de Objetos

O modelo Electra, seguindo os padrões CORBA/OMG, permite que objetos possam ser implementados em diferentes linguagens de programação, sobre um número qualquer de máquinas. O Electra, como todo ORB, é o responsável pela implementação das semânticas de comunicação e independe das linguagens de programação ou das técnicas de implementação usadas na construção dos objetos. As comunicações no Electra podem se dar no estilo disseminação (*multicast*) confiável ou por comunicação *ponto-a-ponto*. O cliente faz uso de um mesmo modelo de invocação de método, independente se o servidor é um objeto simples ou um grupo. Estas invocações síncronas, assíncronas ou semi-síncronas[3] (*deferred-synchronously*), realizadas em interface estáticas ou dinâmicas no estilo CORBA, podem servir para emissões de *multicasts*. Dois modos de comunicação de grupo são disponíveis no Electra:

- Transparente: um grupo é visto como um objeto simples e altamente disponível, ao qual as requisições são submetidas como num ORB convencional ou seja, o cliente recebe só um resultado do grupo;

[3] é assíncrona até um tempo *t* e após este período o chamador se bloqueia até o retorno do resultado (síncrona).

- Não-transparente: permite o acesso, em uma invocação, dos resultados de cada membro individual do grupo de objetos.

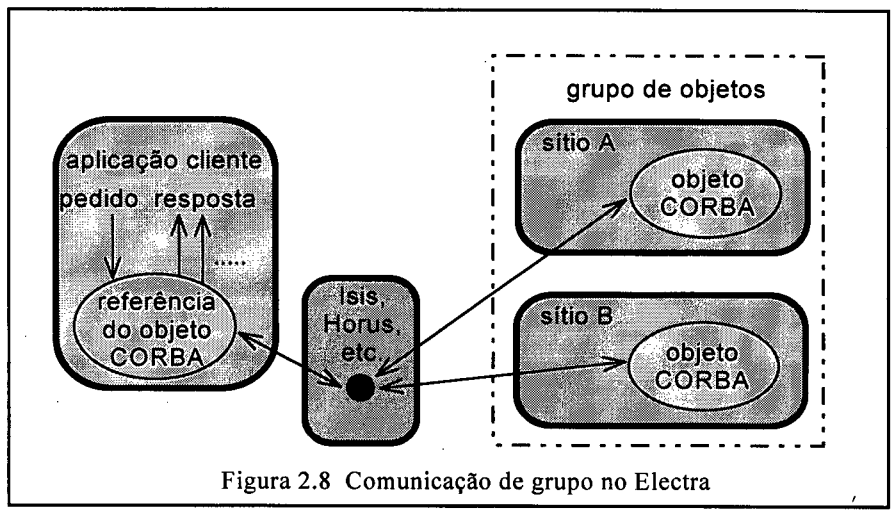


Figura 2.8 Comunicação de grupo no Electra

O Electra suporta grupos no modelo réplicas ativas, fazendo uso de ferramentas de mais baixo nível no fornecimento de serviços de *multicast* confiável e de gerenciamento de grupo (figura 2.8). A transferência de estado e o serviço *membership*, suportados por ferramentas como Isis, são disponíveis como operações Electra no *Basic Object Adapter* (BOA). Na interface do BOA do Electra foram ainda adicionados mecanismos para ativar um grupo de objetos e selecionar o protocolo de *multicast* a ser utilizado. A classe *environment* CORBA é utilizada para a seleção do estilo de invocação e principalmente, para passar exceções do servidor ao cliente. O modelo Electra acrescentou a esta classe três operações especiais, que são:

- `Call-Type`: utilizado pelo cliente para especificar a forma de invocação que pode ser síncrona (bloqueante), assíncrona (não-bloqueante) ou semi-síncrona.

- `Num_replies`: para especificar a forma de recepção das respostas de um grupo pelo cliente, que podem ser:

  - `All`: todas as respostas dos membros operacionais do grupo são coletadas;

- `Majority`: as respostas são coletadas até atingir a maioria;

- `Compare`: neste caso, o ORB coleta todas as respostas dos membros operacionais do grupo e as compara. A resposta mais frequente será enviada ao cliente.

- `Call_completed`: usado para verificar se uma invocação do tipo semi-síncrona foi completada.

2.4.4.2 Arquitetura ELECTRA

O Electra é um sistema flexível capaz de operar sobre vários tipos de ferramentas e sistemas operacionais que ofereçam algum suporte de grupo. A figura 2.9 mostra a arquitetura de um sistema fazendo uso do Electra.

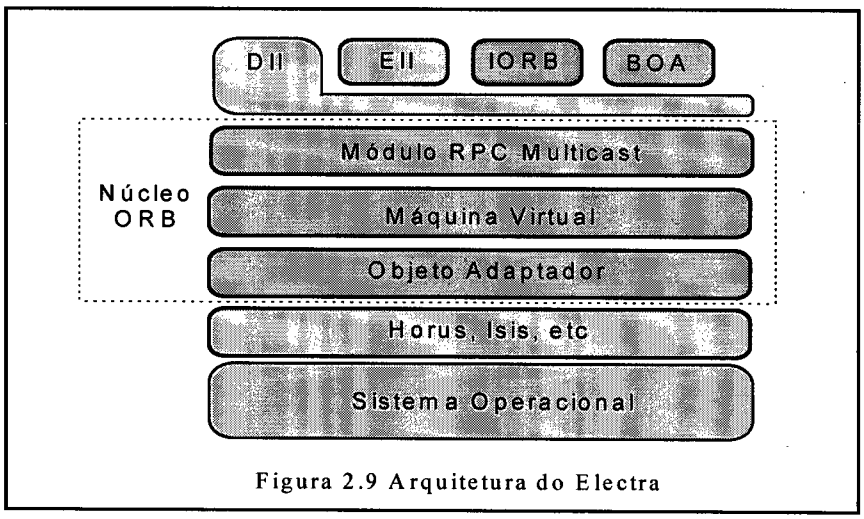


Figura 2.9 Arquitetura do Electra

A interface de invocação estática (EII), a interface ORB (IORB), o adaptador de objeto básico (BOA) e a interface de invocação dinâmica (DII) têm as mesmas funções das interfaces CORBA apresentadas na figura 2.1 deste texto. O núcleo do ORB está fundamentado sobre a interface de invocação dinâmica (DII) e o módulo RPC *multicast* que suporta RPC assíncrono

para grupo ou objeto simples. A interface *máquina virtual* é inserida nesta arquitetura com o objetivo de oferecer portabilidade sobre as diversas plataformas básicas possíveis. A função desta interface é abstrair o modelo CORBA das funcionalidades tais como, comunicação fim-a-fim, grupo de processos, *threads*, etc, oferecidas por estas plataformas de mais baixo nível. O objeto adaptador tem a função de mapear as operações da máquina virtual na interface da plataforma básica (API de baixo nível). O objeto adaptador é específico para a plataforma usada. Para que o Electra funcione em uma plataforma basta desenvolver um objeto adaptador e acrescentá-lo na arquitetura, sem a necessidade de fazer qualquer modificação aos códigos residentes acima dele. As atuais implementações de objeto adaptador para o Isis, Horus e Muts compreendem aproximadamente mil linhas de código C++ cada, demonstrando assim a fácil adaptabilidade destas ferramentas no suporte Electra para aplicações distribuídas.

2.5 Conclusões do Capítulo

As quatro propostas de ORBs estudadas foram desenvolvidas com vistas à programação distribuída orientada a objeto e suportam, igualmente, a noção de grupo de objetos. Todas estão fundamentadas no uso de serviços de ferramentas de mais baixo nível que ofereçam algum tipo de gerenciamento e comunicação de grupo.

Todas as propostas apresentam soluções similares para a integração do conceito de grupo em paradigmas a objeto para programação distribuída. Esta similaridade é mais acentuada nos casos do RDO/C++, do ISIS/C++ e do Orbix+Isis porque seus desenvolvimentos tiveram origem no mesmo grupo de pesquisa da Universidade de Cornell, USA. O RDO/C++ é uma evolução do ISIS/C++ no sentido que sua IDL é compatível com o padrão CORBA/OMG. As propostas do Electra e do Orbix+Isis apresentam também esta compatibilidade com o padrão de suas linguagens de descrição de interface.

Em termos de suporte de comunicação, como ORBs, o Electra, Orbix+Isis e o RDO/C++ tem uma certa compatibilidade com o modelo CORBA; sendo que, apenas os dois primeiros oferecem um adaptador de objeto básico (BOA) para auxiliar nas atividades de gerenciamento

de grupo de objetos. As documentações que possuímos não nos permitiram obter muitas conclusões sobre a compatibilidade das estruturas do RDO/C++ com o modelo CORBA.

A grande vantagem do Electra sobre as outras implementações é a sua possível facilidade na adaptação a diferentes plataformas básicas. Para sintetizar o estudo comparativo apresentamos a tabela abaixo:

	IDL compatível ao CORBA	*Apresentam uma compatibilidade ao ORB do CORBA*	*Suporte a grupo de objetos a nível CORBA*	*Oferece um BOA para grupo de objetos*	*Opera sobre vários suportes (flexibilidade)*
Electra	sim	sim	sim	sim	sim
Orbix+Isis	sim	sim	sim	sim	não
RDO/C++	sim	sim (?)	sim	não	não
ISIS/C++	não	não	não	não	não

Tabela 2.1 Comparação das plataformas

Serviços de grupo foram adicionados em plataformas ANSA (*Advanced Network Systems Architecture*) [Oskiewicz 93]. Estas abstrações de grupo permitem *multicast* de operações ANSA e gerenciamento de grupo. Similar ao Electra e ao Orbix, o modelo ANSA permite invocações em grupos com transparência e não transparência de servidor. Os *multicasts* podem se dar com ordenações FIFO e Total. As soluções de implementação no ANSA são muito simples: os mecanismos de ordem total e de *multicast* são centralizados em nós (membro do grupo) concentradores, o que constitui em possível ponto de falha e de baixo desempenho no sistema. As comunicações de grupo são realizadas usando repetidas mensagens em comunicações ponto a ponto e não pela exploração de facilidades de *hardware* e *software* na implementação de *multicast*. Um detalhamento maior do Modelo ANSA não foi objetivo deste texto porque este modelo não é compatível com o padrão CORBA.

Recentemente têm sido proposto que comunicação de grupo e tolerância a falhas sejam incluídos ao CORBA através da incorporação das abstrações *Group Server* na arquitetura OMA [Adler 95]. A API de gerenciamento de grupo da proposta é similar ao BOA do Electra.

Os resultados dos membros do grupo a serem retornados ao cliente são combinados, a exemplo do Orbix+Isis, com o uso de funções, permitindo alguns tipos de transparência de serviço. A proposta de *Group Server* segue a abordagem de implementação usada no ANSA, apresentando elemento concentrador na comunicação de grupo o que é um *handicap* negativo para o desempenho e a confiabilidade do sistema. Aspectos de transferência de estado e *membership* não estão sendo previstas nesta proposta.

CAPÍTULO 3

Técnicas de Replicação

3.1 Introdução

O objetivo deste capítulo é apresentar um estudo de técnicas de replicação de componentes de *software* em sistemas distribuídos. Serão explorados dentro deste estudo os aspectos e as características dessas técnicas de replicação segundo as abordagens réplicas ativas e réplicas passivas, visando a forma como cada abordagem busca garantir a consistência de estado. Na sequência, será apresentada uma seleção de modelos clássicos de processamento replicado, implementados em plataformas de sistemas distribuídos conhecidos, divididos segundo as duas abordagens.

3.2 Classificação das Faltas

A ocorrência de falhas em um sistema distribuído pode ser causada por erros de *hardware* (componentes do sistema ou a rede de comunicação), erros de *software* (no projeto) ou por causas externas ao sistema (ex.: queda de energia, inundação etc.). Se um serviço distribuído não está preparado para agir de acordo com a ocorrência de uma falha, como consequência disso, sérios prejuízos podem ser impostos aos usuários do serviço.

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

especificado. De acordo com a definição de [Laprie 90] uma falha ocorre quando um serviço oferecido pelo sistema não está de acordo com a especificação. Um **erro** é parte do estado do sistema que pode conduzir a falha do serviço especificado. A **falta** é definida como a causa fenomenológica de um erro no sistema. Podemos agrupar as falhas dentro de dois domínios: tempo e valor, e classificá-las, segundo [Cristian 94], da seguinte forma:

- Falhas por *crash* e de omissão: um componente que não responde a uma dada requisição, e no entanto responde a uma requisição subsequente caracteriza uma falha por omissão. Um exemplo deste tipo de falha de omissão é a perda de uma mensagem na rede de comunicação. Para falhas de *crash*, o componente não responde a mais nenhuma requisição subsequente. Há casos em que se atribui um grau de omissão *f* a um componente, quando esse limite é ultrapassado todas respostas subsequentes deverão ser omitidas, caracterizando uma falha de *crash*.

- Falhas por temporização: ocorre quando um componente responde a uma dada requisição com um valor correto mas fora do intervalo de tempo especificado, ex.: um processador sobrecarregado pode produzir um valor correto mas atrasado em relação a um intervalo especificado (falha por atraso). Vale ressaltar que uma falha de temporização só se caracteriza se a especificação do sistema impor restrições temporais ao componente.

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

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

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

3.3 Técnicas de Replicação

É de consenso geral que a utilização de técnicas de replicação em sistemas distribuídos contribuem fortemente para uma melhora na confiabilidade, um aumento da disponibilidade de recursos e possivelmente, um melhor desempenho do sistema.

O uso de replicação torna possível implementar serviços tolerantes a falhas através da noção de componente/serviço abstrato para o usuário (exibe a propriedade de um componente único mas que na realidade é formado por um conjunto de componentes replicados). A falha de uma réplica individual passa a ser considerada separadamente da falha do grupo como um todo, logo a confiabilidade do serviço passa então a depender da confiabilidade do grupo que oferece este serviço. Um protocolo de coordenação é necessário para assegurar a consistência de estado e a transparência do conjunto. Estes protocolos devem também ser responsáveis pelo controle da concorrência e pela recuperação em situação de falha parcial ou total das réplicas.

As abordagens chamadas *réplicas passivas* e *réplicas ativas* são capazes de mascarar falhas individuais de réplicas membros do conjunto. Contudo, as formas como isto é feito diferem fortemente. Na escolha de uma das duas abordagens deve ser considerados: o tipo de aplicação, a classe de falta que se deseja tolerar e as características do sistema distribuído.

3.4 Aspectos da Abordagem Réplicas Passivas

Nesta abordagem, somente um membro (primário) recebe, executa e responde as invocações dos clientes. As réplicas restantes do conjunto (*backups*) tem a função de substituir o membro primário caso este falhe. O cliente faz as requisições sem saber qual é o primário da replicação. Para assegurar que os estados dos membros se mantenham mutuamente consistentes, o primário envia periodicamente mensagens de *checkpoint* de seu estado para todos ou um número suficiente de *backups* (figura 3.1). Em situação de falha do primário é ativado um protocolo de eleição que seleciona entre os *backups* o novo primário, que reassume a execução da operação a partir do *checkpoint* mais recente. É importante ressaltar que nenhuma invocação é processada até que o novo primário seja eleito.

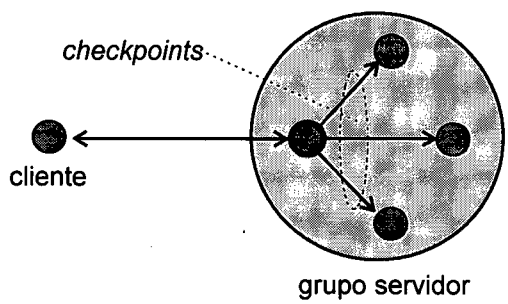


Figura. 3.1 Grupo réplicas passivas.

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

[1] o determinismo de réplicas introduzido por [schneider 90] implica que réplicas corretas, partindo do mesmo estado inicial e processando o mesmo conjunto de entradas, na mesma ordem relativa, devem produzir as mesmas saídas. O determinismo de réplicas é uma condição para a consistência de estados entre réplicas ativas.

A detecção de falha do primário é um outro fator que deve ser considerado. Nesta abordagem só é possível detectar falhas de *crash*, de omissão e alguns tipos de falhas de temporização. Para esses casos, podem ser utilizados mecanismos de *timeout e keepalive*. A detecção pode ser realizada pelos *backups*, por um mecanismo independente ou pelo próprio usuário do serviço replicado. *A* frequência com que é ativado o mecanismo de *keepalive* determina a rapidez da recuperação na falha do primário, mas se esta frequência for muito elevada pode influenciar no desempenho do servidor. Uma situação que pode ocorrer é o primário falhar logo após completar a execução de uma requisição e enviar o *checkpoint* aos *backups* sem, no entanto, responder ao cliente. Neste caso o cliente pode reenviar a mesma requisição. Mas para evitar que o novo primário processe-a novamente é necessário, através de um mecanismo de *retenção de resultado*, que ele reconheça a repetição da requisição e simplesmente retransmita o resultado retido [Birman 85].

O número de *backups* que recebem as mensagens de *checkpoint* do primário depende da aplicação e do sistema distribuído. Se o sistema é especificado para tolerar f nós faltosos, então é necessário que o *checkpoint* chegue em pelo menos f réplicas. As falhas dos *backups* não tem nenhuma influência sobre o sistema até atingir f réplicas, uma vez que até este valor é possível mascará-las. A frequência das operações de *checkpoint* pode diminuir o desempenho do serviço replicado. É possível, no entanto, diminuir a taxa de envio de *checkpoint*, por exemplo, a cada x requisições (figura 3.2). Esta solução é vantajosa em casos onde a ocorrência de falha do primário não for frequente. Nesta situação, quando o primário falha, o cliente terá que retransmitir as requisições a partir do último *checkpoint*, e portanto poderá ter que reavaliar as respostas do novo primário para que não ocorram repetições.

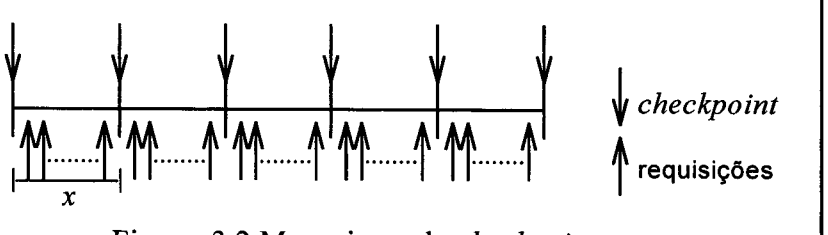


Figura. 3.2 Mecanismo de *checkpoint*.

Para evitar que o cliente tenha que retransmitir requisições em situação de falha do primário pode ser utilizado um mecanismo de *log* que guarde em disco mensagens de requisições entre intervalos de *checkpoint* (figura 3.3).

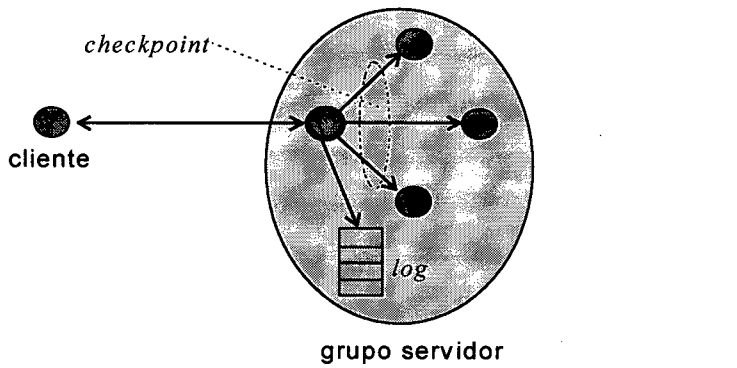


Figura. 3.3 Grupo réplicas passivas com mecanismo de *log*.

3.5 Aspectos da Abordagem Réplicas Ativas

Nesta abordagem (também conhecida como Máquina de Estado) todas as réplicas não faltosas do grupo são ativas: recebem, executam e respondem a todas requisições dos clientes (figura 3.4). Em relação à abordagem réplicas passivas o custo é maior, pois aloca mais recursos do sistema (memória, processador etc.) e o fluxo de mensagens no meio de comunicação é mais elevado. Por ter todas as réplicas ativas executando o mesmo conjunto de requisições é necessário que o conjunto de réplicas seja *determinístico*, caso contrário podem surgir possíveis inconsistências de estado entre as réplicas. Além disso, alguma forma de comparação de resultados passa a ser necessária, no sentido de retornar ao cliente um resultado. Algumas alternativas neste sentido são possíveis: o primeiro resultado a chegar é passado ao cliente; os resultados podem ser concatenados em sequência e enviados ao cliente; ou ainda, os resultados passam por um votador ou ajustador que seleciona o resultado mais frequente (maioria) para ser passado ao cliente.

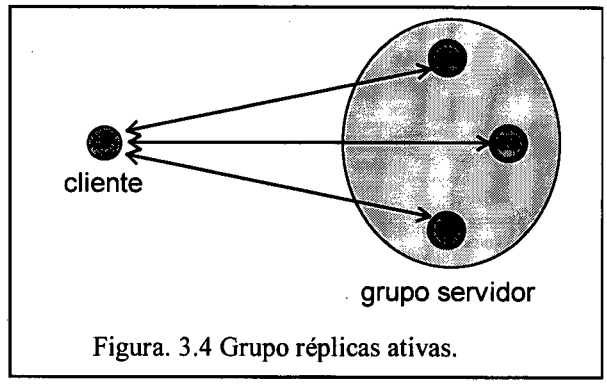


Figura. 3.4 Grupo réplicas ativas.

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

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

* Acordo: as réplicas não faltosas recebem as mesmas mensagens (requisições);

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

A abordagem réplicas ativas é capaz de tolerar todo espectro de faltas (*crach,* omissão, temporização, valor e arbitrária). Assumindo K como sendo o número de faltas de réplicas que o sistema pode tolerar, temos as seguintes especificações para tolerar as seguintes hipóteses de falhas:

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

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

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

- Falhas de temporização por atraso tem um tratamento idêntico ao do primeiro grupo. As falhas de temporização por antecipação correspondem ao segundo grupo e necessitam de mecanismos de voto majoritário [powell 91].

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

3.6 Modelos de Réplicas Passivas e Réplicas Ativas

Apresentaremos nos próximos itens, de forma sucinta, alguns modelos de replicação de *software*, segundo as duas abordagem (ativa e passiva), amplamente difundidas na literatura e implementadas em aplicações distribuídas.

3.6.1 Modelo de Replicação *Coordinator-Cohort*

Este modelo de replicação passiva é implementado usando o sistema ISIS [Birman 85] como suporte. As requisições emitidas por diferentes clientes são recebidas pelas réplicas do modelo. Uma réplica para cada replicação é nomeada *coordinator* (réplica ativa) enquanto que as outras operam como *cohort* (*backups*) prontas para substituir o *coordinator* em caso de falha. Cada *coordinator* pode atender a apenas um cliente. Devido ao modelo ser bastante complexo mecanismos de ordenação total e causal são necessários.

O gerenciamento do número de membros participantes (*membership*) é feito de forma que cada membro saiba quais outros membros funcionais participam do processamento de uma requisição (quem são os *coordinators* e os *cohorts*). A lista de membros é atualizada por um mecanismo de *membership*, que detecta a entrada ou a saída (normal ou por falha) de réplicas, e difunde para todos os membros por disseminação (*multicast*) confiável com ordem total (GBCAST). Requisições de diferentes clientes são enviadas ao grupo através de disseminação confiável com ordem total (ABCAST) assegurando que todas as réplicas funcionais recebam na mesma ordem relativa as requisições dos clientes.

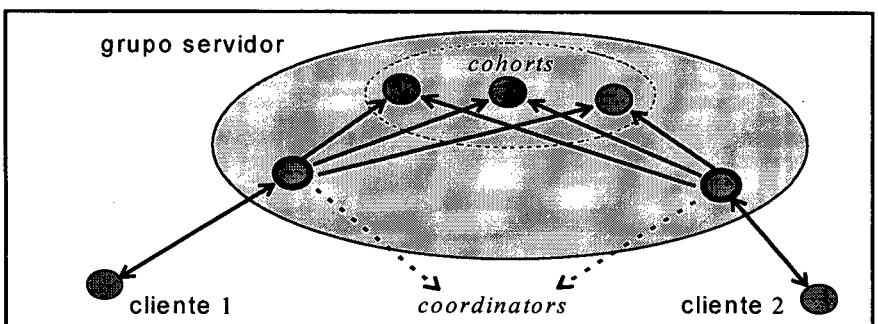


Figura. 3.5 Modelo de replicação *Coordinator-Cohort*

Um *coordinator* ao completar o processamento de uma requisição envia um *checkpoint* aos *cohorts*, para atualizarem seus estados, e depois responde ao cliente. O envio do *checkpoint* e da resposta para o cliente é feito através de disseminação confiável com ordem causal (CBCAST), garantindo que todos os *cohorts* atualizem seus estados antes que qualquer requisição futura seja executada.

Na situação de falha de um *coordinator* é feita a eleição de um novo dentre os *cohorts*. Devido ao fato de todos os membros do grupo terem uma mesma lista consistente e sempre atualizada de quais os membros operacionais, o protocolo de eleição não requer qualquer troca de mensagem entre os membros do grupo. Fazendo uso da lista e de uma mesma regra determinista, todos os membros chegam à mesma decisão de qual réplica será o novo *coordinator* para a requisição em questão. A regra determinista usada é baseada nos critérios: localização física do cliente (ex.: uma réplica que está no mesmo nó que o cliente tem maior

prioridade) e informação da taxa de balanceamento de carga (ex.: número de requisições por unidade de tempo).

3.6.2 Modelo de Replicação *Viewstamp*

Este esquema de replicação passiva está implementado no sistema Argus [Liskov 87a] [Liskov 88]. Objetos em Argus são denominados *guardians* e a unidade lógica de replicação é o *guardian*, a replicação é alcançada pela criação de grupo de *guardians*, que consiste de várias réplicas chamadas *cohort*, que comporta como uma entidade lógica única. O conjunto de *cohorts* formam um grupo de configuração e cada um sabe o grupo e a configuração à qual pertence. Dentre os *cohorts* um é designado primário (réplica ativa) e o restante de *backups*. Se o primário falha um novo entre os *backups* é eleito. A falha de qualquer membro do grupo (primário ou *backups*) é registrada na lista de membros (*view*) que contém a informação de quem é o primário e quem são os *backups*.

Para garantir a transparência de grupo qualquer falha de réplica deve ser mascarada. Este esquema é direcionado para falha por particionamento da rede. Cada membro envia mensagens de "are you alive" para todos a fim de detectar possíveis falhas, portanto mudanças de *membership* são detectadas através de trocas de mensagens entre as réplicas. Na figura 3.6 temos uma falha de comunicação entre o primário P1 (réplica *a*) e seus *backups* (*b, c, d, e*); o *view* do grupo antes da falha é $v1 = \{a: b, c, d, e, f\}$. Como os *backups* (com exceção de *f*) não conseguem se comunicar com o primário (ainda operacional) chegam ao consenso de sua falha e elegem um novo primário P2 (a réplica *b* da figura) com o view $v2 = \{b: c, d, e\}$. Por parte do primário P1, este executa o algoritmo de mudança de *view*, como não consegue se comunicar com {*b, c, d,* e}, não obtém o consenso da maioria (só o voto de *f*), fica inativo para executar qualquer requisição que porventura receba. Quando uma mudança de *view* tem sucesso todos os membros pertencentes a este novo *view* iniciam num mesmo estado (no mais recente *checkpoint*). O *checkpoint* é emitido aos backups com garantia que pelo menos a maioria receba, como na falha do primário é o consenso da maioria que vale, é garantido que na transição de *v*1 para *v*2 pelo menos um membro possua o mais recente *checkpoint* do primário antigo (P1). Esse será o novo primário.

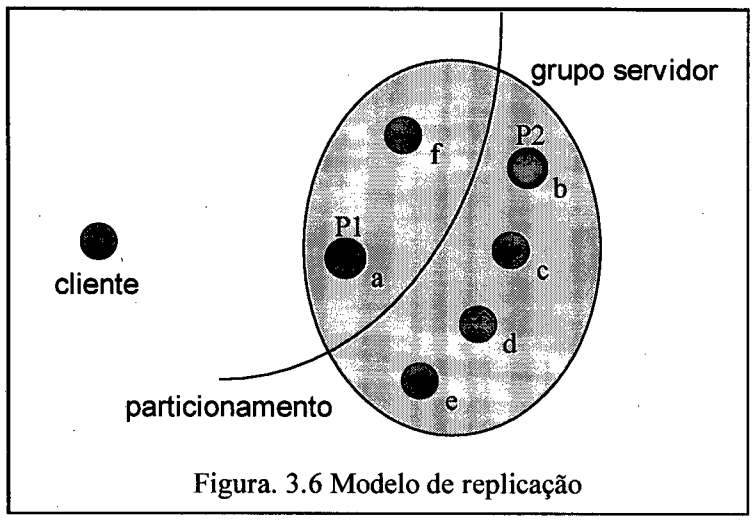


Figura. 3.6 Modelo de replicação

Para implementar o algoritmo apresentado se utiliza de artifícios do tipo *timestamp* e *view*. A associação destes dois artifícios é chamado de *viewstamp*, onde o *view* dá o número de membros do grupo e indica quem é o primário e o *timestamp* para marca a ordem lógica dos eventos.

3.6.3 Modelo de Replicação *Líder/Seguidores*

Este esquema de replicação ativa aproveita alguns mecanismos próprios da abordagem de réplicas passivas. A sua implementação se encontra no sistema Delta-4 [barret 90]. O modelo pode ser configurado para tolerar diferentes classes de faltas. Como foi afirmado anteriormente protocolos de replicação ativa requerem alguma forma de disseminação atômica que assegure que todos membros funcionais recebam um mesmo conjunto de mensagens e na mesma ordem relativa (os requisitos de acordo e ordenação para determinismo de réplica).

Neste modelo todas as réplicas são ativas, mas só a réplica líder responde as requisições do cliente e é responsável pelas decisões que afetam o determinismo das réplicas do conjunto. As decisões são propagadas do líder para seus seguidores via mensagem de sincronização (figura 3.7). Qualquer mensagem (de requisição ou de resposta) é enviada pelo líder aos seus seguidores logo que são geradas, e quando os seguidores geram as mesmas respostas que o líder o próprio sistema de comunicação se encarrega de descartá-las, o modelo é especificado para nós *fail-silent*. Se o sistema é especificado para tolerar falhas de valor o protocolo de

replicação ativa usa um mecanismo de votação na réplica *líder* que usa as respostas das várias réplicas para detectar possíveis falhas. A fim de simplificar a apresentação deste modelo trataremos apenas a tolerância à falha de *crash*, pois não necessita utilizar um mecanismo de votação.

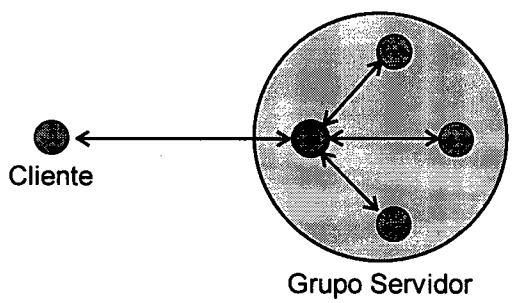


Figura. 3.7 Modelo de replicação lider/seguidores.

Na falha do *líder* um protocolo de eleição define o novo através de um *ranking* estático predefinido no momento da criação do grupo. A detecção de falha do *líder* e dos seus *seguidores* pode acontecer a partir:

* do sistema de comunicação quando tenta transmitir uma mensagem para seu destino;

* pelos *seguidores* que recebem periodicamente mensagens de "I am alive" do *líder*.

A grande vantagem deste modelo, além de oferecer artifícios de preempção para aplicações tempo-real, é que o líder pode iniciar o processamento de uma requisição assim que esta chega, ao contrário de outros modelos de replicação que precisam esperar por um acordo sobre a ordem das mensagens.

3.6.4 Modelo de Replicação Ativa *Competitiva*

No modelo de replicação competitiva todas as réplicas são ativas mas apenas uma responde a uma dada requisição de entrada. A principal característica deste modelo é a competição entre as réplicas: somente a mais rápida responde a requisição. A coordenação da técnica é distribuída. Cada réplica possui um controlador associado, responsável pela recepção, disseminação e comparação das mensagens, ficando a réplica correspondente dedicada ao processamento das requisições. Para garantir a consistência entre as réplicas, todas as mensagens entre réplicas são transmitidas via disseminações confiáveis com ordem total.

O modelo de replicação competitiva pode ser configurado para tolerar dois conjuntos de falhas [Powell 91]: *falhas de temporização*, envolvendo as semânticas de falha por *crash*, omissão e de temporização por atraso; e *falhas não controladas* (arbitrárias) que compreendem todo o espectro de falhas.

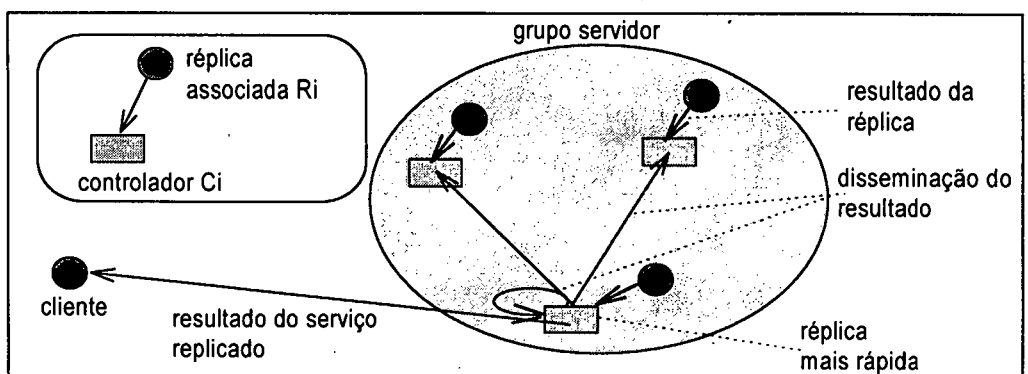


Figura 3.8. Modelo de replicação competitiva para falhas de temporização

A figura 3.8 ilustra de maneira simplificada a replicação competitiva sob premissa de falhas de temporização. Neste caso, considerando o modelo Cliente/Servidor com o servidor replicado, uma requisição do cliente difundida no grupo servidor, é recebida pelos controladores `Ci`, que as repassam às replicas `Ri` associadas. Ao receber o resultado do processamento de sua réplica, cada controlador verifica se já possui mensagem de outro controlador do grupo com o resultado do mesmo processamento. Na ausência de mensagem, o controlador concatena um identificador aos resultados, e difunde a mensagem resultante no

grupo de controladores. Se o controlador receber primeiro sua própria mensagem, descobre que sua réplica foi a mais rápida e assim é o responsável pelo envio da resposta ao cliente; caso contrário, a sua mensagem é descartada. Esse algoritmo garante que apenas uma réplica responde ao cliente, pois todas as mensagens difundidas no grupo são observadas por cada membro na mesma ordem relativa (disseminação confiável com ordem total).

Por fim, a disseminação de uma mensagem de *fim_de_processamento* após o envio dos resultados ao cliente, pelo controlador da replica mais rápida encerra o ciclo de processamento referente à requisição do cliente. Esta mensagem dá possibilidade que se monte estratégias para a detecção de falha do controlador da réplica mais rápida e a sua substituição por outro controlador no envio dos resultados ao cliente [Brito 95].

O protocolo citado acima mascara erros de temporização por atraso, de omissão e por crash. No que concerne o tratamento dos elementos falhos, dois procedimentos de detecção são previstos na literatura original [Powell 91]:

* é admitido um fraco acoplamento entre o controlador e a réplica. Neste caso, no controlador são mantidos mecanismos de *timeout* para detectar a falha da réplica associada;

* a replicação competitiva privilegia a réplica mais rápida e, por conseqüência, pode levar a um assincronismo muito grande no conjunto de réplicas. Esta dessincronização, é tratada realizando periodicamente um *rendez-vous,* onde todos os controladores difundem os resultados de suas réplicas entre si e o último a difundir envia o resultado ao cliente. Este *rendez-vous* é limitado no tempo de modo que possibilita também a detecção de controladores falhos.

Com premissas de falhas arbitrárias, o protocolo anterior deve ser acrescido de um mecanismo de comparação. Para tolerar f réplicas com falhas o grupo deve conter ao menos $2f+1$ réplicas, isto porque é necessário comparar as saídas das várias réplicas e obter-se êxito na comparação

de pelo menos de *f+1* respostas concordantes. No protocolo, ao receber a resposta de sua réplica, cada controlador anexa sua assinatura e difunde a mensagem resultante a todos os controladores do grupo. Devido à disseminação confiável com ordem total, as filas de mensagens dos controladores são idênticas e totalmente ordenadas. O consenso quanto à resposta é obtido quando um controlador do grupo receber e comparar com sucesso *f+1* respostas, incluindo a sua. O controlador que alcançou o limite *f+1* assinaturas comparadas e sua mensagem foi a (*f+1*)-ésima mensagem na comparação, envia a resposta ao cliente. As hipóteses de falhas deste protocolo incluem falhas de temporização por antecipação que não eram consideradas no caso do protocolo anterior.

3.6.5 Modelo de Replicação Ativa *Cíclica*

A principal característica deste modelo de replicação ativa é o uso de um mecanismo de passagem de bastão (*token*) entre os membros de um grupo replicado, onde a ordem de posse do bastão é definida segundo a sequência de um anel lógico (*token ring*). Como no modelo de replicação competitivo, cada réplica tem uma ent_cont[2] que cuida de todo o processo de tolerância a falha do sistema. Este modelo, conhecido também de *round robin*, também pode ser configurado para tolerar falhas de temporização e falhas não-controladas. Disseminação confiável com ordem total é necessária para garantir a consistência entre as réplicas.

A configuração para tolerar falhas restritas é basicamente a mesma que no modelo competitivo. A única diferença é que a ent_cont (entidade controladora) possuidora do bastão é o responsável pelo envio da mensagem para o cliente e pela disseminação para os membros restantes do grupo, quando feita a disseminação o bastão é passado para a outra ent_cont da sequência do anel lógico (figura 3.9). As ent_cont não possuidores do bastão devem armazenar as mensagens recebidas até que chegue a sua vez de ser o privilegiado. Assim sendo, ele pode descartar as mensagens armazenadas até a especificada no bastão (*last_message*). Outra característica marcante deste modelo é que na falha da ent_cont privilegiada o modelo é reconfigurado para o modelo competitivo.

[2] entidade controladora

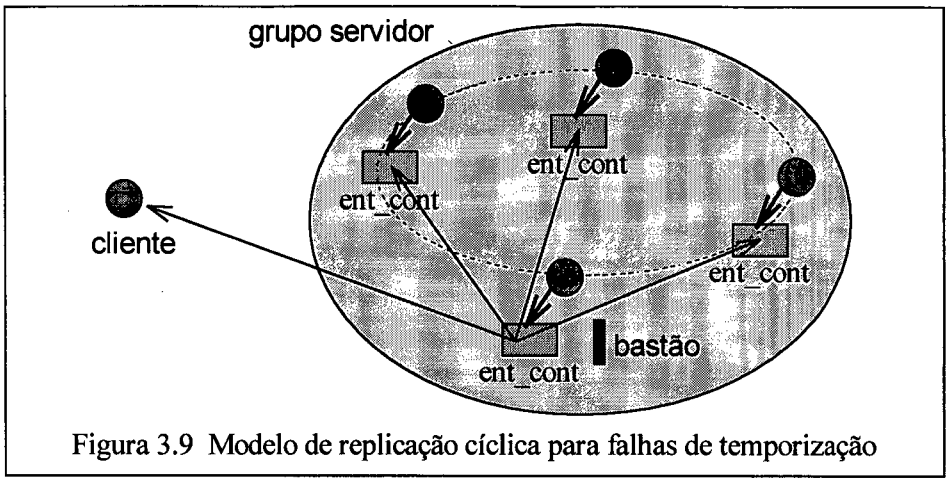


Figura 3.9 Modelo de replicação cíclica para falhas de temporização

A tolerância a falhas não-controladas neste modelo também é basicamente idêntica ao modelo competitivo. Quando a primeira ent_cont possuidora do bastão recebe uma mensagem da sua réplica controlada, ela anexa sua assinatura, emite por disseminação confiável a mensagem *claim* (mensagem original mais a assinatura) a todos os outros ent_cont do grupo ($2t$+1) e passa o bastão a próxima ent_cont definido no anel lógico. As subsequentes ent_cont's privilegiadas devem comparar a sua mensagem com as mensagens *claims* que receberam referentes ao mesmo processamento, e verificar se um total de t+1 mensagens são concordantes (mensagens iguais). No caso de êxito na comparação o possuidor do bastão envia a mensagem ao seu destino (ao cliente) e uma mensagem de *ack* (reconhecimento) as outras ent_cont determinando o fim da transação. No caso da comparação não atingir o limite de t+1 mensagens idênticas a ent_cont com o bastão simplesmente difunde sua mensagem *claim* e passa o bastão. Se até completar um ciclo do anel lógico não for alcançado o consenso da maioria (t+1 réplicas) é porque mais de t falhas de réplicas ocorreram no sistema, caracterizando uma falha no sistema como um todo. Se porventura ocorrer a falha da réplica privilegiada (perda do bastão), detectada através de um temporizador que marca o intervalo de tempo entre a recepção de uma mensagem original da réplica controlada e a recepção do bastão, o sistema é reconfigurado para o modelo competitivo.

3.7 Conclusões do Capítulo

Apresentamos neste capítulo uma introdução a alguns conceito do processamento replicado, um estudo sobre técnicas de replicação de componentes de *software*, sobre as abordagens réplicas ativas e réplicas passivas, enfatizando suas características na coordenação das replicas.

A principal vantagem das técnicas de replicação de componentes de *software* sobre as chamadas técnicas de tolerância a faltas fortemente acoplada (a nível do suporte de *hardware*) é o menor custo de se adaptar aplicações tolerantes a falhas a diferentes arquiteturas de computadores ou *update* destas. Como apresentado, as técnicas de replicação podem seguir as abordagens réplicas ativas e réplicas passivas, ou uma variante de ambas, réplicas semi-ativas (ex.: modelo *líder-seguidores*). A abordagem réplicas ativas tem um baixo *overhead* em situação de falha, pois todas réplicas são ativas e o mascaramento da falha de uma réplica é feito quase que instantaneamente. No sentido oposto, a abordagem réplicas passiva tem *overhead* maior, pois na falha do primário (ativo) um tempo é gasto para executar o protocolo de eleição do novo primário. A principal vantagem da abordagem réplicas passivas é a não necessidade do determinismo de réplicas (um custo menor em termos de suporte), pois apenas o primário é ativo e impõe seu estado sobre os *backups* através de *checkpoints* porém, esta abordagem tem a desvantagem de não tolerar classes de faltas menos restritivas. O quadro abaixo apresenta uma síntese comparativa destas três abordagens:

Técnicas de replicação	*overhead em situação de falha*	*Determinismo de réplica*	*Falhas arbitrárias*
Ativa	baixíssimo	necessário	tolera
Passiva	alto	desnecessário	não tolera
Semi ativa	baixo	necessário	não tolera

3.1 Quadro comparativo das três abordagens.

CAPÍTULO 4
PROPOSTA DO MODELO DE INTEGRAÇÃO

4.1 Introdução

Apresentamos neste capítulo a proposta de um modelo de integração usando o paradigma de reflexão computacional sobre uma plataforma aberta CORBA, no sentido de permitir a implementação de modelos de replicação de componentes de *software*. Além disto, introduzimos as especificações das técnicas de replicação *ativa competitiva, ativa cíclica, líder/seguidores e primário/backup,* segundo o modelo considerado e que terão suas implementações discutidas no próximo capítulo.

4.2 Modelo de Reflexão Computacional

O paradigma de reflexão computacional e utilizado em outros trabalhos para implementar algoritmos de gerenciamentos de aplicações do tipo tempo-real [Honda 92] [Furtado 95], programação concorrente [Takashio 92] e tolerância a faltas [Fabre 95], tem se mostrado uma solução bastante adequada para prover ao programador da aplicação flexibilidade nestas aplicações. Estas características permitem, por exemplo, que um sistema distribuído tolerante a falhas separe o código fonte da aplicação e o código fonte que contém os aspectos de controle responsáveis pela implementação da técnica de tolerância a faltas desejada.

Reflexão computacional é um modelo de programação em que um sistema pode analisar seu próprio comportamento e atuar sobre o mesmo. Aplicado à programação orientada a objetos o paradigma da reflexão computacional, através da abordagem meta-objetos [Maes 87], estrutura os objetos em dois níveis: nível base (*objeto-base*) e nível meta (*meta-objeto*). Cada

objeto-base ***x*** está associado a um meta-objeto ^***x***, que representa os aspectos estruturais e comportamentais de ***x***, estes aspectos podem ser gerenciados dinamicamente através de computações realizadas em ^***x*** (figura 4.1). As chamadas aos métodos do objeto-base são desviadas no sentido de ativar meta-métodos que permitem modificar o comportamento do objeto-base ou adicionar funcionalidades a seus métodos. Esta abordagem torna possível separar os aspectos funcionais e não-funcionais de uma aplicação, permitindo que os métodos e procedimentos da aplicação em si sejam tratados no nível base, enquanto o controle e o gerenciamento da aplicação são tratados no nível meta.

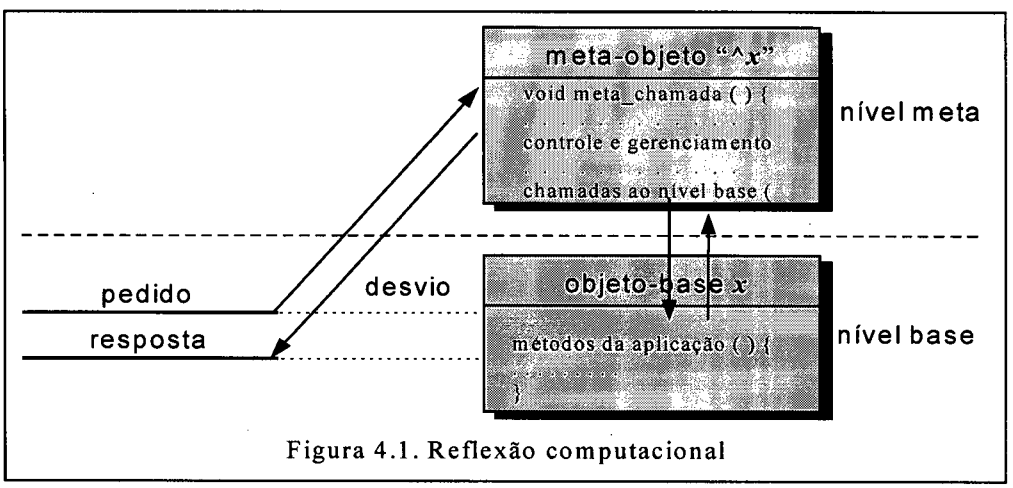


Figura 4.1. Reflexão computacional

Neste trabalho, a reflexão computacional é usada no sentido de desenvolver um modelo de integração para técnicas de replicação em ambientes abertos. O paradigma reflexivo nos permite atribuir a objetos-base as funcionalidades da aplicação replicada, enquanto meta-objetos executam os protocolos de coordenação entre réplicas. Isto permite o uso de diferentes técnicas de replicação com os objetos-base mantendo suas características, bastando para isso trocar os meta-objetos associados.

4.3 Modelo Reflexivo para Integração de Técnicas de Replicação em Sistemas Abertos

O objetivo deste item é propor um modelo de integração que facilite a programação de técnicas de replicação em sistemas distribuídos abertos. Este modelo faz uso de conceitos de reflexão computacional. A implementação das técnicas de replicação deverão fazer uso extensivo de suporte de grupo em uma plataforma CORBA.

A estrutura proposta deve incorporar os conceitos da técnica de replicação ao modelo reflexivo de processamento. Cada réplica é mapeada então sob a forma de um objeto-base. Os protocolos de coordenação da técnica serão implementados na forma de meta-objetos. A cada réplica (objeto base) estará associado um meta-objeto controlador responsável por aspectos da evolução do modelo e pelas interações no sentido de manter a consistência própria da técnica. A figura 4.2 mostra a estruturação reflexiva a ser adotada na implementação de técnicas de replicação.

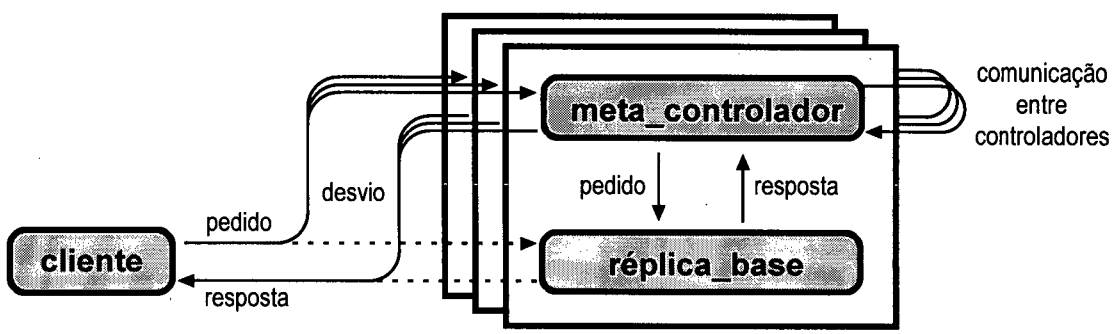


Figura 4.2. Estrutura reflexiva para modelos de replicação.

A figura 4.3 mostra como fica o modelo de integração da técnica de replicação no contexto CORBA. O acesso ao suporte fornecido por uma plataforma CORBA é disponível tanto do lado servidor quanto do lado cliente por entidades representadas como meta-objetos (cliente e servidor) e identificadas genericamente como meta-comunicação. Estas entidades na verdade nada mais são do que o conjunto de *stubs* no cliente e no servidor, *stubs* para comunicação das réplicas e o BOA (*Basic Object Adapter*) com o suporte para gerenciamento de grupo, gerados todos a partir da tradução da especificação IDL da interface de um objeto servidor. O tratamento de "objeto abstrato" dado ao meta-comunicação a nível de modelo segue a linha de

alguns autores [Hagsand 92] e tem o sentido de uma simples separação para maior clareza. Na verdade, estas interfaces são geradas pelo suporte CORBA como um conjunto de métodos que serão compostos com múltiplas heranças nos meta objetos cliente e servidor.

No modelo o cliente apresenta-se estruturado em um cliente-base que representa o comportamento da aplicação, e um meta-cliente, que não possui função ativa, mas que poderia ser usado no gerenciamento de um cliente replicado, ou para implementar mecanismos de tratamento de exceções no cliente. A estrutura de cada réplica do servidor é semelhante à do cliente: um objeto réplica-base, realizando o serviço que se deseja replicar, e um meta-controlador (ou meta-servidor), responsável pelo protocolo de coordenação da técnica de replicação.

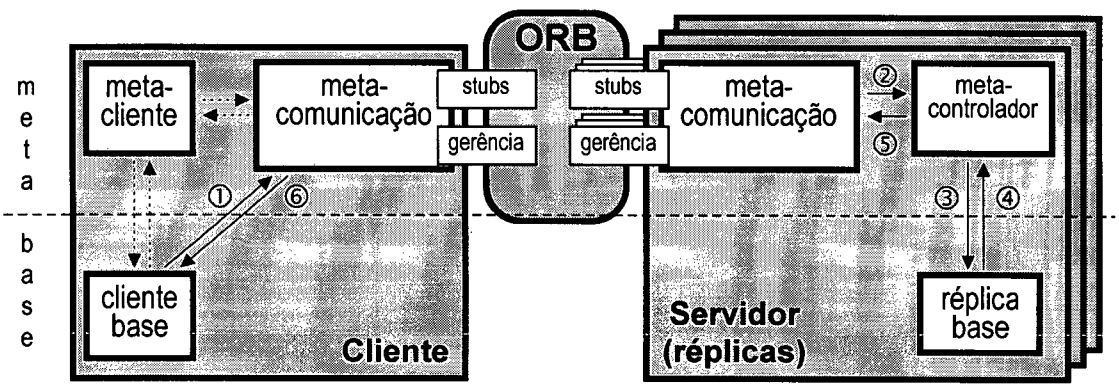


Figura 4.3. Estrutura do modelo sobre um suporte CORBA.

As setas numeradas na figura 4.3 indicam o trajeto normal de um pedido do cliente: o pedido emitido pelo cliente-base (①) é emitido usando o *stub* apropriado no meta-comunicação do cliente. Em cada réplica, o meta-comunicação através de um *stub* local recebe o pedido e o transfere ao meta-controlador (②), que então aciona a réplica local (③). Ao receber sua resposta (④), o meta-servidor executa o protocolo de coordenação, servindo-se do meta-comunicação para interagir com as outras réplicas. O processamento e as interações a nível de meta-controladores neste momento estão condicionadas pelo modelo de replicação utilizado. Após, a resposta é então devolvida ao cliente (⑤ e ⑥).

Este modelo pode ser usado em diferentes técnicas de replicação, as diferenças essencialmente se concentrarão nos meta-controladores do servidor replicado. Em algumas técnicas as entidades meta-comunicação podem ganhar funcionalidade além daquela de concentrar métodos de acesso aos serviços do suporte CORBA. Por exemplo, no uso de réplicas ativas com mecanismos de votador ou ajustador [Giandome 90], a implementação da votação ou ajuste é programada de maneira menos custosa no lado do cliente. A transparência pode ser conseguida neste caso, implementando estes mecanismos na entidade meta-comunicação do cliente que com a adição desta funcionalidade ganha as características de um objeto real.

A colocação da comunicação a nível meta é também realizado em [Fabre 95]. Isto é importante porque não prende a implementação das interações cliente/servidor a um tipo de modelo ou protocolo de comunicação. A estrutura é flexivel, podemos mudar os protocolos de comunicação sem alterar os códigos de nível base.

4.4 A programação de Técnicas de Replicação em um Ambiente CORBA Segundo o Modelo de Integração

Vimos no capítulo 2 a arquitetura CORBA e um conjunto de requisitos necessários para o suporte de grupos de objetos. O objetivo deste item é mostrar a programação de réplicas, segundo o modelo reflexivo em um ambiente CORBA. No CORBA a compilação de uma descrição de interface do servidor em IDL (*Interface Definition Language*) permite gerar todo o suporte necessário às comunicações entre cliente e servidor (figura 4.4). O servidor é implementado através de um conjunto de métodos da classe servidor, gerada na compilação da IDL, ficando o programador responsável pela inserção dos códigos de cada método da classe.

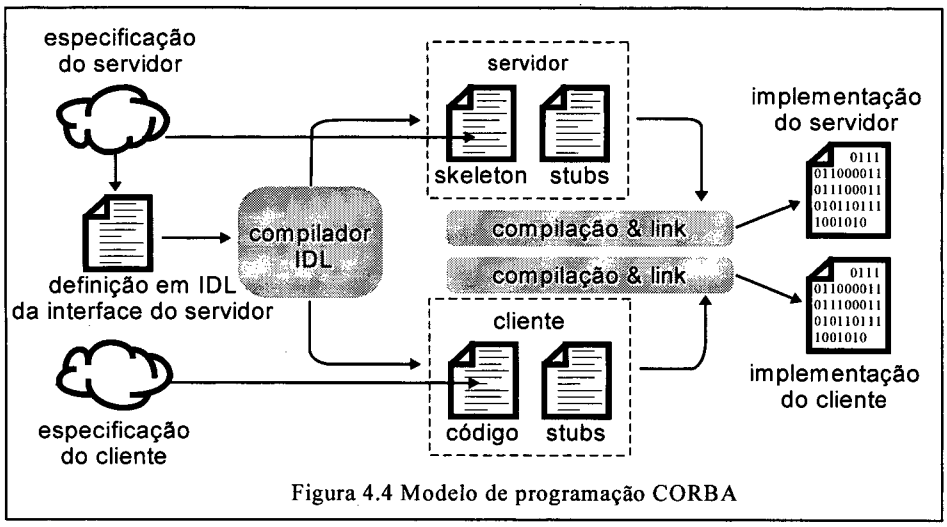


Figura 4.4 Modelo de programação CORBA

O modelo de integração proposto consiste em adicionar ao modelo de programação CORBA uma classe representando o nível meta do servidor (meta-controlador) e outra o nível meta do cliente. O primeiro passo para a implementação de uma aplicação replicada sobre uma plataforma CORBA é a descrição em IDL da interface do meta-controlador, a partir da especificação dos serviços oferecidos pelo servidor. No caso do modelo de integração, essa interface consiste da declaração de cada um dos métodos oferecidos pelo servidor ao cliente. Além destas, será necessário declarar os métodos que servem às interações entre as diferentes réplicas, referentes aos protocolos de coordenação da técnica utilizada.

A fim de maior clareza separamos o meta_controlador em duas interfaces: uma responsável pelas ativações aos métodos base e a outra para os métodos necessários ao gerenciamento da técnica de replicação utilizada. O código abaixo apresenta a IDL de ambas as interfaces CORBA do servidor replicado. A interface `meta_controlador_1` permite aos clientes o acesso aos serviços oferecidos pelo nível base, enquanto que a interface `meta_controlador_2` declara os métodos necessários às interações entre réplicas para a implementação de um determinado protocolo de replicação. Cabe ressaltar que ambas as interfaces são na verdade duas facetas de um mesmo servidor (ou, em nosso caso, de um mesmo grupo de objetos).

```
// IDL

interface meta_controlador_1 {
   // Descrição dos tipos de dados empregados

   // Descrição dos métodos do servidor
   boolean meta_método_1 (parâmetros);
   ...
   boolean meta_método_n (parâmetros);
};

interface meta_controlador_2 {
   // Descrição dos métodos do meta controlador
   boolean meta_controle (parâmetros);
   boolean difunde_id (in int id);
   boolean encerramento ();
   ...
};
```

Figura 4.5 Interface IDL do servidor replicado.

No processo de compilação das interfaces de uma aplicação, a plataforma CORBA gera automaticamente todo o suporte para a comunicação (*stubs*) entre as entidades envolvidas, incluindo também as funcionalidades para gerenciamento de grupos. O compilador também gera o arquivo contendo o "esqueleto" do código do servidor (declarações de variáveis e métodos). O programador fica então responsável pela descrição dos objetos-base da aplicação e de protocolos de coordenação, preenchendo os corpos dos métodos definidos nas interfaces dos meta-controladores. Com esse esquema de implementação, que está ilustrado na figura 4.6, os objetos-base cliente e servidor permanecem isentos de quaisquer atividades que não estejam ligadas à aplicação propriamente dita. Todos os aspectos relativos ao gerenciamento da replicação e às interações no contexto do CORBA ficam concentrados ao nível dos meta-controladores.

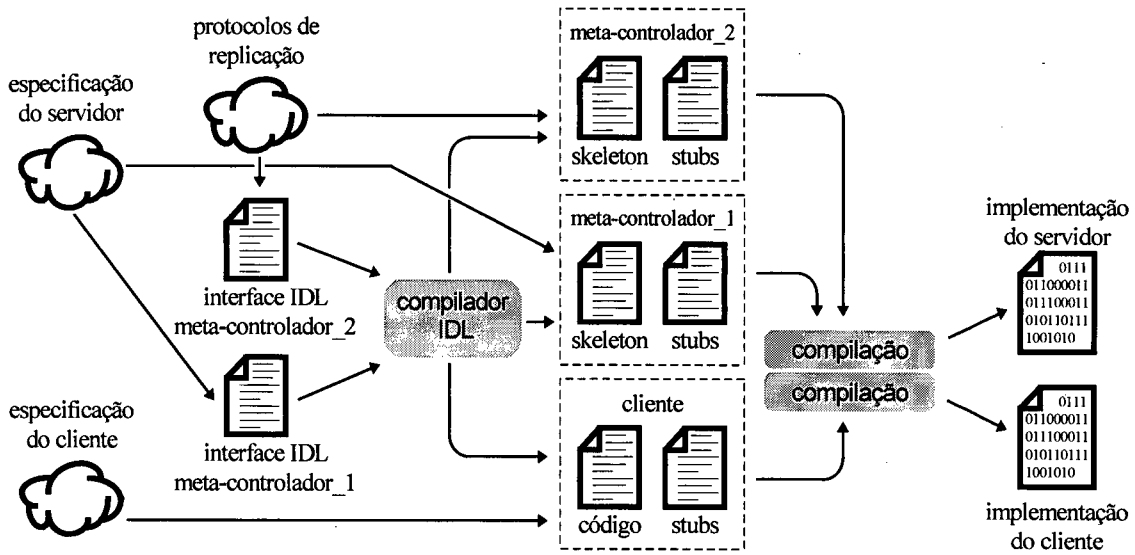


Figura 4.6 Processo de construção da aplicação

Na sequência é apresentada algumas técnicas de replicação estruturadas segundo o modelo reflexivo que é a base de nosso modelo de reflexão.

4.5 Especificação das Técnicas de Replicação no Contexto Reflexivo

Neste item serão apresentadas as especificações das técnicas de replicação: *ativa competitiva, ativa cíclica, primário/backup e líder/seguidores* de acordo com o modelo de integração apresentado. Como citado anteriormente, cada réplica é estruturada na forma de um objeto-base. Os meta-objetos correspondentes implementam os protocolos de coordenação da técnica de replicação. Nas técnicas citadas consideramos somente a hipótese de falhas por *crash.* Admitimos também um acoplamento forte entre o meta-controlador e a réplica associada, os erros gerados serão atribuídos a ambos: em hipótese de *crash*, controlador e réplica associada cessarão de executar.

4.5.1 Replicação Ativa *Competitiva*

Um pedido disseminado pelo cliente no grupo de réplicas é desviado aos respectivos controladores, que então se encarregam de interagir para implementar os protocolos de coordenação da replicação usada. As ações de um controlador são descritas no pseudocódigo da figura 4.7. A cada método do objeto-base é associado um meta-método no controlador,

responsável por sua ativação (ex: metodo_base_1 e meta_metodo_1, na figura citada). A partir da compilação da IDL apresentada na figura 4.5 é gerado o *skeleton* (já com as implementações inserida conforme a figura 4.7).

```
class meta_controlador_1{
        ... // declaração de variáveis

        method meta_método_1(parâmetros){
              método_base_1(parâmetros);
              meta_controle(parâmetros);
        };

        method meta_método_2(parâmetros){
              método_base_2(parâmetros);
              meta_controle(parâmetros);
        };
        ... // declaração dos demais meta-métodos

class meta_controlador_2{
        // implementação do meta-controle
        method meta_controle(parâmetros){
              primeiro := null;
              encerrou := false;
              meu_id := get_system_id();
              while not encerrou do
                    if (primeiro = null) then
                          grupo.difunde_id(meu_id);
                    end;
                    if (primeiro = meu_id) then
                          // primeira réplica a responder
                          return ; // retorna resposta ao cliente
                    else
                          if not encerrou then
                                grupo.encerramento();
                          end;
                    end;
              end;
        };
        method difunde_id(int id){
              if (primeiro = null) then
                    primeiro := id;   // id da réplica mais rápida
              end;
        };

        method encerramento(){
              if (primeiro ∈ membership) then
                    encerrou := true;
              end;
              primeiro := null;
        };
};
```

Figura 4.7 Meta-controlador da replicação ativa *competitiva*.

O método `meta_controle` implementa o protocolo de coordenação entre réplicas descrito no item 3.6.4. O comportamento básico do algoritmo consiste em iterar entre a escolha do emissor da resposta ao cliente (primeiro) e o procedimento de encerramento, até certificar-se de que a resposta foi efetivamente enviada (condição encerrou do laço `while`). A verificação do encerramento é simples: se após a difusão do método `encerramento` o emissor atual ainda estiver vivo (pertencer ao *membership* do grupo) então a resposta foi efetivamente enviada. Caso contrário, um novo emissor é escolhido e o processo se repete. Este procedimento elimina a necessidade de disseminação de uma mensagem de fim de processamento. No algoritmo, as ativações dos métodos `difunde_id` e `encerramento` são transmitidas a todas as réplicas do grupo, de maneira totalmente ordenada. Os diagramas temporais das figuras 4.8a e 4.8b mostram os procedimentos de cada réplica do grupo numa situação sem *crash* e outra com *crash* da réplica mais rápida, respectivamente.

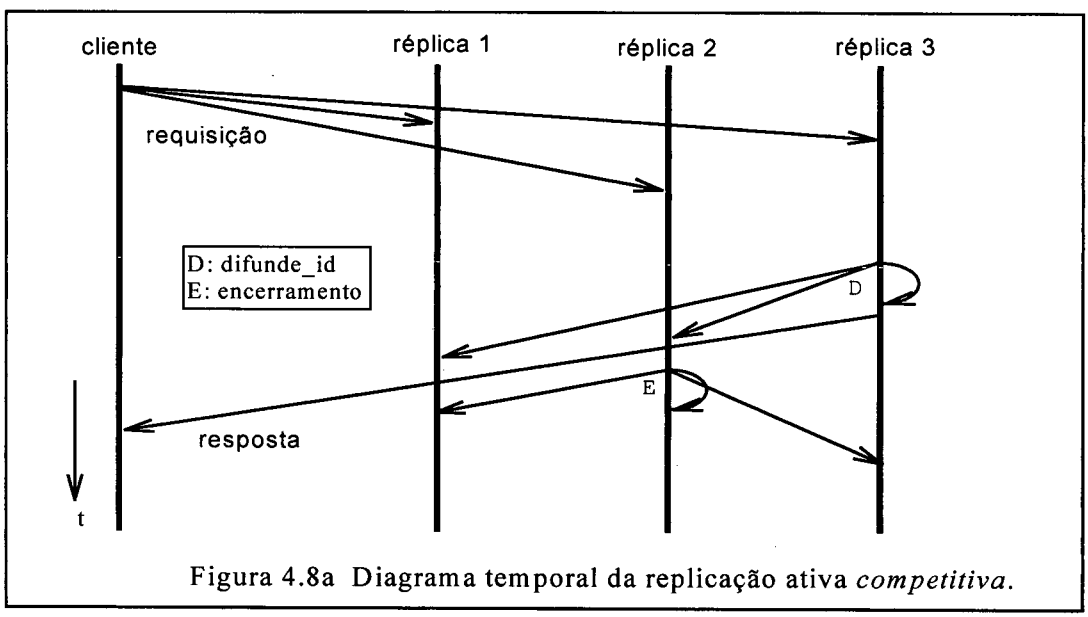


Figura 4.8a Diagrama temporal da replicação ativa *competitiva*.

Caso duas réplicas (ou mais) observem a condição (`primeiro = null`) verdadeira, ambas deverão difundir seu *id* ao grupo, mas, devido a garantia de ordem ordenação total (fila idêntica para todos membros) apenas uma réplica será identificada como a mais rápida.

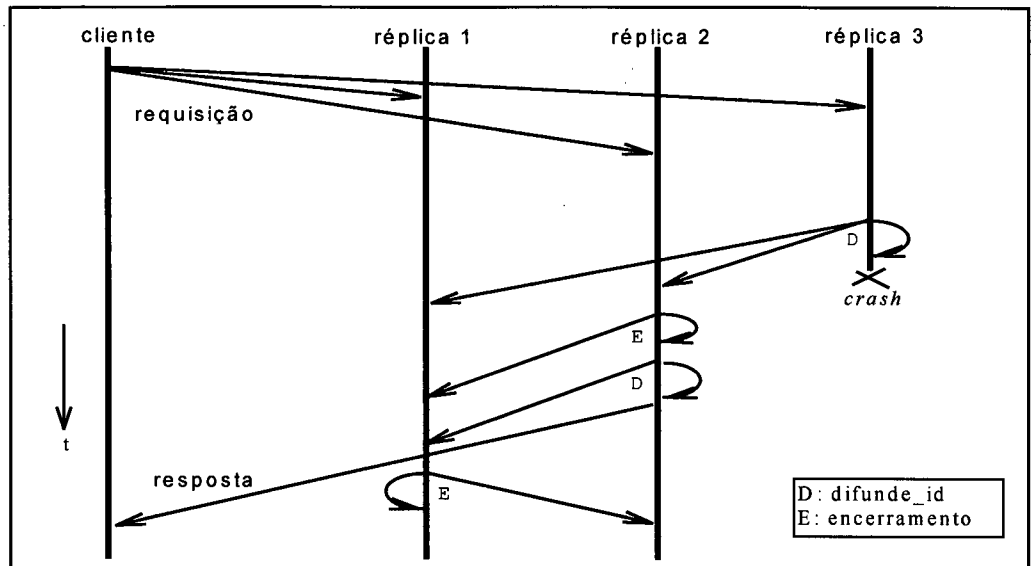


Figura 4.8b *crash* do mais rápido.

Tanto o modelo de replicação competitiva como o suporte usado privilegiam a réplica mais rápida, o que pode acarretar na dessincronização das réplicas mais lentas. A técnica de execução periódica do rendez-vous global proposta em [Powell 91] não é admitida nesta especificação por suas implicações de custo no desempenho do conjunto. A solução adotada é baseada na propriedade de sincronismo virtual [Birman 91], mantida pelo suporte de mais baixo nível usado neste trabalho. Desta forma, enquanto a réplica pertencer ao *membership* do conjunto ela terá as mesmas mensagens na mesma ordem das demais. Quando o *buffer* de entrada no suporte de comunicação, associado ao par controlador/réplica mais lenta, atinge sua capacidade limite o suporte retira a réplica do *membership*. Uma réplica pode detectar sua exclusão e reintegrar-se ao grupo através do método `view_change(view)`, que deve ser fornecido pela interface BOA do suporte CORBA e ativado automaticamente por ele a cada mudança no *membership*. A ativação deste método não é preemptiva, ocorrendo somente após o processamento do método corrente. O corpo do método `view_change` é definido segundo as características da aplicação. Assim sendo, em nossa especificação no corpo deste método executamos um teste sobre o *membership*, e se a réplica tiver sido excluída (`view.number` = 1), é ativada a primitiva BOA `join(group)`, que efetua a reintegração da mesma no grupo.

4.5.2 Replicação Ativa *Cíclica*

A especificação em pseudocódigo do modelo de replicação ativa cíclica é semelhante à anterior, com pequenas modificações. A identidade de uma réplica é estabelecida através da função `rank_system`, que determina um identificador único $id : 1 \rightarrow n$ a cada réplica do grupo, onde n é o número de membros do grupo. Este identificador é dinâmico, podendo ser atualizado a cada mudança no *membership* do grupo: se uma réplica com $id = k$ falhar (sair do grupo) todas as réplicas com $id > k$ terão seus identificadores decrementados (id <- id- *1*). A indicação da réplica privilegiada é feita através de um inteiro *priv*, incrementado (módulo n) a cada ativação de método, simulando assim um anel lógico. Quando o identificador *id* de uma réplica é igual ao valor de `priv` a este é dado o privilégio de retornar a resposta ao cliente, enquanto as demais réplicas realizam o procedimento de encerramento.

```
class meta_controlador_2{
        method meta_controle(parâmetros){
                encerrou := false;  semáforo := false;
                meu_id := rank_system();
                n := membership;
                while not encerrou do
                        if (meu_id = priv) then
                                grupo.pronto();
                                return; // privilegiado retorna resposta
                        else
                                wait(semáforo or timeout);
                                if not encerrou then
                                        grupo.encerramento();
                                end;
                        end;
                end;
                priv := (priv mod n) + 1;
        };
        method encerramento(){
                if (priv ∈ membership) then
                        encerrou := true;
                else
                        priv := priv + 1;
                        if (priv > membership) then
                                priv := 1;
                        end;
                end;
        };
        method pronto(){
                semáforo := true;
        };
};
```

Figura 4.9 Meta-controlador da replicação ativa *cíclica*

O método `pronto` é um artifício utilizado na especificação para que as réplicas não privilegiadas não iniciem o procedimento de encerramento antes que a réplica privilegiada termine de processar a requisição do cliente. Se um *crash* ocorrer na réplica privilegiada durante o processamento da requisição a condição do `wait(semáforo)` jamais será atendida gerando um *deadlock*, portanto acrescentamos uma outra condição no `wait(semáforo or timeout)`, onde o *timeout* representa o tempo máximo para o recebimento da mensagem `pronto` por parte da réplica privilegiada. A fim de diminuir a complexidade das comunicações, em caso de *crash* da réplica privilegiada o sistema não é reconfigurado para o modelo competitivo como descrito no item 3.6.5. A nova réplica privilegiada é definida pela próxima do anel lógico (figura 4.10).

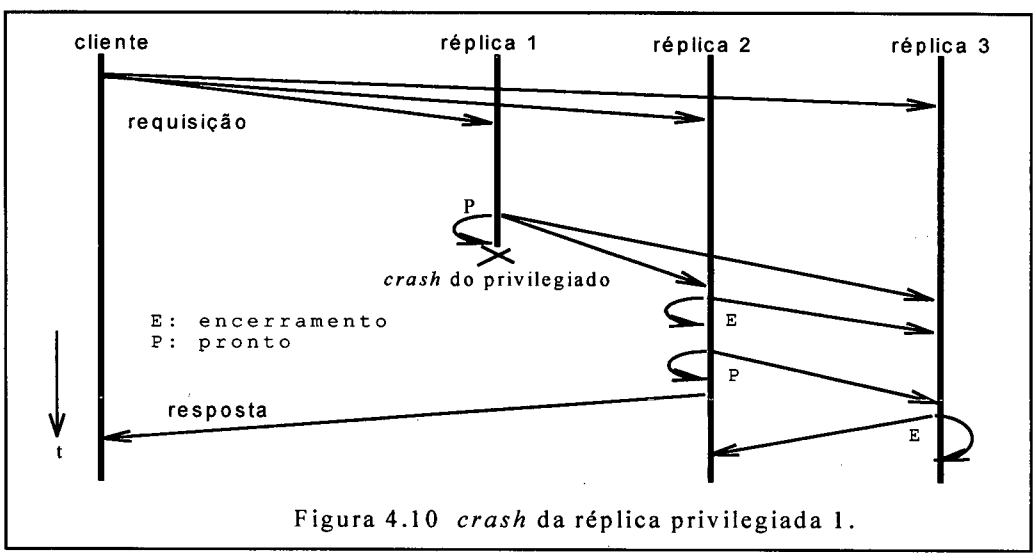


Figura 4.10 *crash* da réplica privilegiada 1.

Devido a cada réplica ter o direito de processar uma requisição do cliente a cada ciclo do anel lógico não há o problema de dessincronização das réplicas mais lentas. O mecanismo de reintegração de réplicas faltosas segue a mesma regra do item anterior.

4.5.3 Replicação Ativa *Lider/Serguidores*

Neste modelo de replicação todas as réplicas são ativas e executam o mesmo código, mas apenas a réplica líder é a responsável por todas decisões que afetam o determinismo de réplica (item 3.6.3). No pseudocódigo da figura 4.11, a indicação do líder é determinada pela réplica com `meu_id` igual a 1 (**`rank_system`** = 1), a mais antiga do grupo. Devido ao modelo

adotar este mecanismo de *rank* tem-se a vantagem da não necessidade de qualquer troca de mensagem entre réplicas para eleição do novo lider, caso o antigo falhe.

```
class meta_controlador_2{
        method meta_controle (parâmetros){
                encerrou := false;  semáforo := false;
                meu_id := rank_system();
                lider := 0;
                while not encerrou do
                        lider := lider + 1;
                        if (meu_id = lider) then
                                // sou o líder
                                grupo.resposta_do_lider(resposta);
                                return; // retorna resposta ao cliente
                        else
                                wait(semáforo or timeout);
                                if not encerrou then
                                        grupo.encerramento();
                                end;
                        end;
                end;
        };
        method encerramento(){
                if (lider ∈ membership) then
                        encerrou := true;
                end;
        };
        method resposta_do_lider(resposta){
                if (minha_resposta ≠ resposta) then
                        minha_resposta := resposta;
                        semáforo := true;
                end;
        };
};
```

Figura 4.11 Meta-controlador da replicação *líder/seguidores.*

Assumindo faltas de *crash* não há a necessidade de validação de uma mensagem, quando o líder termina de processar uma requisição envia a resposta aos seguidores, e em seguida ao cliente.

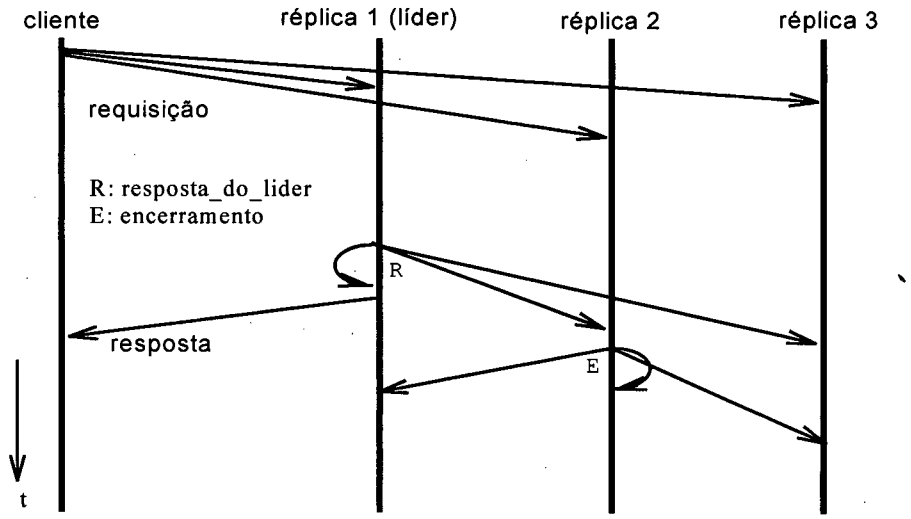


Figura 4.12 Diagrama temporal da replicação semi-ativa *líder/seguidores*

Após o líder entregar a resposta aos seus seguidores, a condição do `wait` é satisfeita (figura 4.11 e 4.12), e estes executam o procedimento de encerramento como descrito para o modelo competitivo (item 4.5.1). Na descrição original, o lider envia periodicamente mensagens de "I am alive" aos seus seguidores para detecção de *crash.* A nossa especificação tem a vantagem da não necessidade deste artifício. O cliente usa uma disseminação confiável para enviar ao grupo de réplicas a requisição. Na comunicação líder/seguidores basta mecanismos de ordem causal para garantir a consistência de estado das réplicas nas iterações em grupo.

4.5.4 Replicação Passiva *Primário/Backups*

Nas especificações apresentadas até agora, cada réplica ao ser ativada executa o método base solicitado, e ativa o método `meta_controle` para a realização do protocolo de coordenação especificado. Diferente dos anteriores, neste modelo de replicação apenas a réplica primária executa as requisições (método_base). Esta condição é satisfeita através do teste `meu_id` no início de cada meta_método, se igual a 1 a réplica é o primário e ativa o método base (figura 4.13). As outras réplicas (*backups*) atualizam seus estados com o *checkpoint* enviado pelo primário (método `transfere_estado`).

```
class meta_controlador_1{
        ... // declaração de variáveis
        method meta_método_1(parâmetros){
              meu_id := get_system_id();
              if (meu_id = 1) then
                    método_base_1(parâmetros);
              end;
              meta_controle(parâmetros);
        };

        method meta_método_2(parâmetros){
              meu_id := get_system_id();
              if (meu_id = 1) then
                    método_base_2(parâmetros);
              end;
              meta_controle(parâmetros);
        };
        ... // declaração dos demais meta-métodos
        // implementação do meta-controle

class meta_controlador_2{
        method meta_controle(parâmetros){
              encerrou := false;  semáforo := false;
              primário := 0;
              while not encerrou do
                    primário := primário + 1;
                    if (meu_id = primário) then
                          if (meu_id ≠ 1) then
                                // novo primário
                                ... // executa o método solicitado
                          end;
                          // checkpoint aos backups
                          grupo.transfere_estado(meu_state);
                          return;
                    else
                          wait(semáforo or timeout);
                          if not encerrou then
                                grupo.encerramento();
                          end;
                    end;
              end;
        };

        method encerramento(){
              if (primário ∈ membership) then
                    encerrou := true;
              end;
        };
        method transfere_estado(state){
              meu_state = state;
              semáforo := true;
        };
};
```

Figura 4.13 Meta-controlador da replicação passiva *primário/backup*.

Caso o primário falhe antes ou depois do *checkpoint,* um novo primário é eleito entre os *backups* para responder ao cliente. Isto é feito incrementando na variável `primário` no laço `while`. Neste ponto, o novo primário realiza a execução do método requisitado pelo cliente e após isso, transfere seu estado às réplicas *backups* restantes. O diagrama temporal apresentado na figura 4.14 sintetiza a descrição acima. O mecanismo de encerramento realizado pelas réplicas *backups* segue a mesma forma descrita para o modelo de replicação ativa competitiva (item 4.5.1).

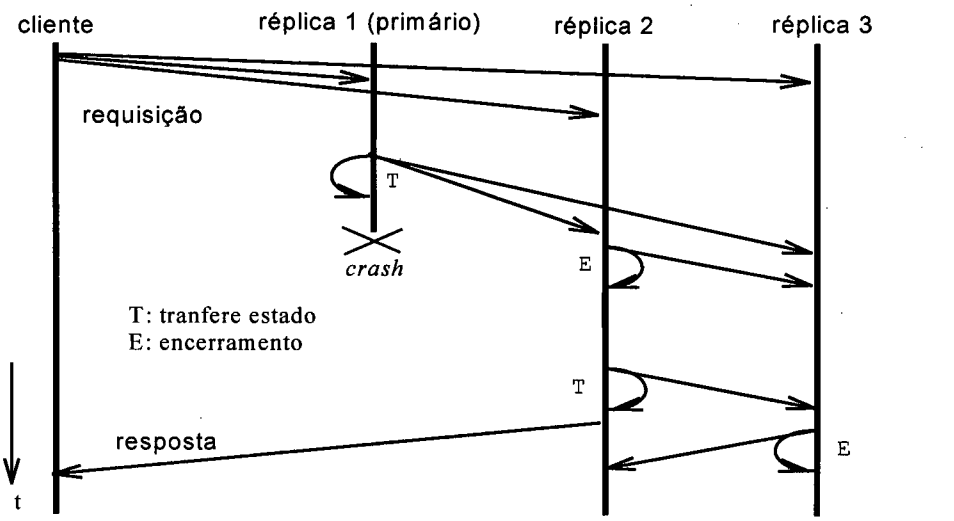


Figura 4.14 Diagrama temporal da replicação passiva *primário/backup.*

4.5.5 Considerações sobre as Especificações Apresentadas

Pode-se observar nas especificações apresentadas neste capítulo que houveram algumas diferenças em relação aos algoritmos originais das técnicas de replicação. Isso se deve ao fato de que as especificações originais seguem uma abordagem orientada a processos, enquanto a nossa, conforme os objetivos iniciais, segue a abordagem orientada a objetos. Em nossa abordagem, objetos só são ativos no momento em que processam uma ativação de método cessando após isto. O uso de mensagens de *fim de processamento*, de acordo com as descrições originais destas técnicas, para detecção de falha de uma réplica privilegiada, não pode ser facilmente implementada na abordagem orientada a objetos, pois assim que a réplica privilegiada enviar a resposta da ativação de um método para o cliente, sua execução encerra, impossibilitando o envio da mensagem de *fim de processamento* aos demais membros do

grupo. A solução foi adotar um meio alternativo em que a detecção de falha da réplica privilegiada é feita por pelo menos uma réplica do grupo (não privilegiada), com auxílio do suporte no fornecimento da lista de *membership* do grupo replicado. A lista de *membership* deve ser atualizada pelo BOA, do suporte CORBA, a cada entrada ou saída de réplicas no grupo.

Nos modelos *líder/seguidores* e *primário/backup*, à primeira vista parece não ser necessário o uso do método `encerramento` para verificar a falha da réplica principal. Mas, com uma análise mais detalhada, conclui-se que não basta a réplica principal enviar uma mensagem aos seus secundários para garantir que esta não falhou, pois ela pode falhar imediatamente após o envio desta mensagem (figura 4.14), sem devolver a resposta ao cliente.

Em relação às falhas que eventualmente podem ocorrer na evolução do sistema, as especificações realizadas, segundo o modelo de integração para as diferentes técnicas de replicações, prevêem medidas de recuperação do grau de replicação. Se o número de réplicas ativas no grupo cair abaixo de um limite preestabelecido, a réplica mais antiga toma a iniciativa de lançar novas réplicas, restabelecendo a população ideal. O código referente a estes procedimentos de recuperação é baseado num teste de *membership* (`view.number < quorum_mínimo`), e está inserido no corpo do método `view_change` (item 4.5.1).

Nossa abordagem de atualização de estado (*checkpoint*) das réplicas difere daquela proposta em [Fabre 95], na qual as atualizações de estado se dão através de meta-métodos fazendo *updates* em atributos públicos de suas réplicas associadas, com o uso exclusivo de protocolos de coordenação. Em nossa abordagem utilizamos mais primitivas de suporte e menos protocolos a nível de coordenação, simplificando a recuperação de estado. A recuperação de estado, em nosso sistema, está baseada na primitiva `join`, que deve ser oferecida pela interface BOA do suporte CORBA, e ativada através do método `view_change`.

4.6 Comparação com Outras Referências

O uso da reflexão computacional para implementações de técnicas de tolerância a falta tem sido objeto de proposições na literatura. Em [Fabre 95] usando uma arquitetura R2 as divisões meta-objetos para a coordenação e objetos-base para as funcionalidades de aplicação nas réplicas. A experiência em [Fabre 95] utiliza comunicações ponto-a-ponto. Os exemplos de técnicas de replicação especificados no texto são simples, envolvendo replicações ativas, passivas e semi-ativas. As implementações se utilizam da linguagem Open/C++ [Chiba 93]. A coordenação a nível meta é complexa pelo uso de comunicações ponto-a-ponto e nas especificações não são esclarecidos as técnicas implementadas para detecção de falhas (mesmo no caso de *crash*) de réplicas. Como exemplo citamos as falhas do líder ou do primário nos modelos semi-ativo e passivo onde não são especificados os mecanismos de detecção no texto. Creditamos às proposições descritas em [Fabre 95] a condição de um *framework* a ser seguido para implementações de técnicas de tolerância a falhas usando a computação reflexiva.

As descrições feitas em [Lisbôa 96] correspondem a especificações das técnicas N-versões [Avizienis 77] e bloco de recuperação [Randell 75] na forma reflexiva. Acreditamos que as experiências relatadas nessa referência se limitam as especificações citadas. Não existe a experiência de implementação.

O nosso modelo de integração é similar ao apresentado em [Fabre 95], com as diferenças que estendemos para o uso em sistemas abertos. Integramos o modelo reflexivo aos conceitos de uma plataforma CORBA. A reflexão computacional é estendida à noção de processamento de grupo. O modelo foi usado extensivamente para implementar diferentes técnicas de replicação com diferentes graus de complexidade conforme citado neste capítulo e também no próximo. O que é interessante na nossa experiência é que usamos suporte de grupo mesmo em replicações passivas e os meta-objetos (procedimentos de coordenação) se mantém similares aos de replicações ativas. O uso extensivo do suporte de grupo nos permite executar procedimentos de tratamento de faltas e da manutenção do grau de replicação da técnica. Estes procedimentos ou estão ausentes ou não muito claros nas propostas citadas acima.

4.7 Conclusões do Capítulo

Neste capítulo foi introduzido o modelo de integração usado para implementar servidores replicados em um sistema distribuído aberto. Este modelo de integração está fundamentado na reflexão computacional. E como vantagens, traz a nível de aplicação aspectos referentes a coordenação das técnicas de replicação, porém separados da parte funcional da aplicação.

O modelo proposto permite a integração na funcionalidade CORBA, sem comprometer os aspectos de separação e flexibilidade preconizados ao longo do texto. Na sequência foram mostrados aspectos referentes a programação no ambiente CORBA deste modelo reflexivo. Por fim, foram especificados várias técnicas de replicação segundo o modelo introduzido. Vale ressaltar que a utilização da reflexão computacional, permite que com a troca do meta-controlador se mude de técnica de replicação sem nenhuma implicação a nível das réplicas (funcionalidades da aplicação).

Conforme podemos também verificar nas especificações realizadas, os meta-controladores de diferentes técnicas de replicação são muito similares. A partir de um meta-controlador, com pequenas modificações pode se obter um outro que execute as funções de coordenação de outra técnica de replicação. A plataforma CORBA utilizada deve fornecer suporte de grupo. Mecanismos como disseminação de mensagem de forma confiável (*reliable multicast*), ordenação causal e total, gerenciamento de grupo (*membership*) e transferência de estado são imprescindíveis para implementação destas técnicas

CAPÍTULO 5
Principais Aspectos do Modelo de Integração no Sistema Electra

5.1 Introdução

O objetivo deste capítulo é demonstrar a viabilidade da implementação de técnicas de replicação de componentes de *software*, usando reflexão computacional, sobre uma plataforma CORBA com suporte para grupos. Apresentaremos neste capítulo as características da implementação do modelo de integração (CORBA + Reflexão Computacional), proposto no capítulo anterior, sobre o sistema Electra[1]. Iniciaremos o capítulo com um estudo sobre o Electra, abordando suas características e os procedimentos necessários para a implementação de serviços distribuídos. Em seguida, analisam-se sua viabilidade e limitações para a implementação do modelo de integração proposto. A partir disso, será formulado um modelo de implementação considerando as limitações do Electra. Por fim, apresentaremos a implementação de algumas técnicas de replicação, escolhidas no capítulo anterior, para demonstrar a viabilidade do modelo de integração proposto.

5.2 Implementação de Serviços Replicados no Electra

O propósito deste item é salientar aspectos do Electra que consideramos importante na programação de servidores replicados. O processo de desenvolvimento de uma aplicação em uma plataforma CORBA foi salientado no capítulo anterior em seus principais passos. A partir do pré-processamento do código de descrição de interfaces da IDL do Electra

[1] suporte CORBA disponível para nossas experimentações.

(compatível ao CORBA) é gerado um conjunto de códigos necessários para implementar as comunicações entre o cliente e o grupo de servidor [Maffeis 95c]. Na descrição de interface o programador da aplicação dispõe aos seus potenciais usuários da aplicação quais operações (métodos) estão disponíveis e como devem ser invocados. Para facilitar a compreensão, apresentamos um pequeno exemplo de implementação.

5.2.1 Caso Exemplo: Serviço Bancário

A aplicação escolhida é de um serviço bancário em que um usuário, após entrar com o número de sua conta e senha, pode acessar as operações de depósito, saque e verificação de saldo. A figura 5.1 sintetiza todo o processo para a geração do código executável do cliente e do servidor da aplicação. A partir da especificação da aplicação banco, o programador abstrai desta as operações que serão declaradas no arquivo de interface IDL.

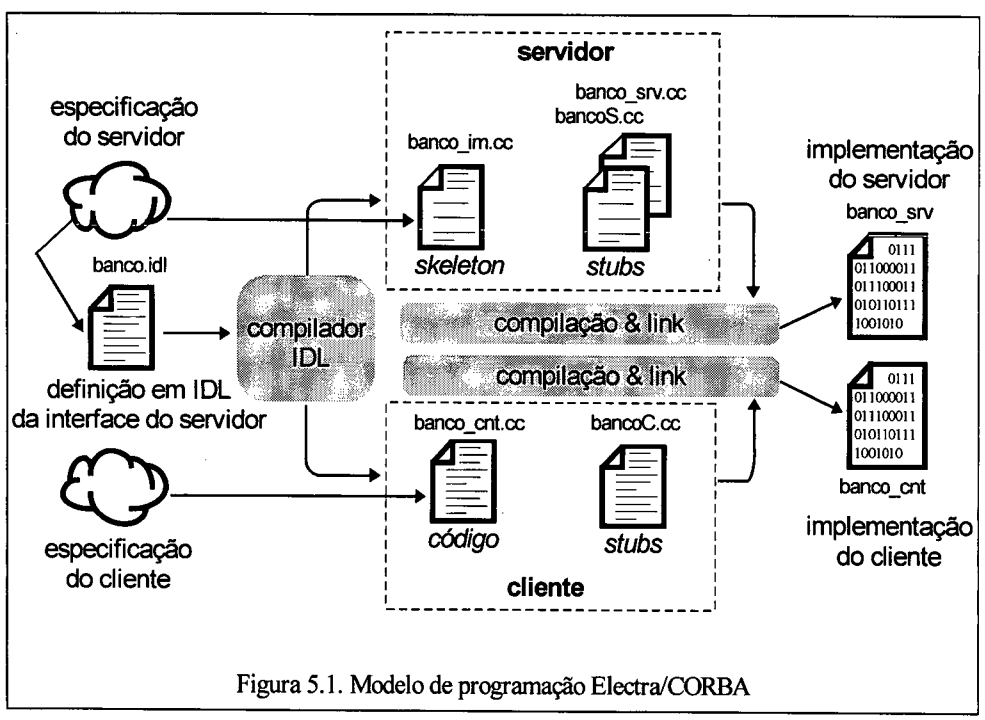


Figura 5.1. Modelo de programação Electra/CORBA

Os arquivos `banco_srv.cc` e `bancoS.cc` correspondem o inicializador do servidor (criação do grupo, geração de referência, etc) e a *stub* do servidor, respectivamente. O arquivo `banco_cnt.cc` é o inicializador do cliente, nele é inserido os procedimentos do cliente da aplicação.

A interface do servidor dispõe de quatro operações assim definidas:

- `deposito(in short valor_dep, in conta conta_corrente):`

  ⇒ realiza a operação de depósito através dos parâmetros de entrada (`in`) `valor_dep` e da estrutura `conta`;

- `saque(in short valor_saq, in conta conta_corrente);`
  `raises(LimiteExcedido):`

  ⇒ realiza a operação de saque através dos parâmetros de entrada `valor_saq` e da estrutura conta, o `raises` determina um retorno de exceção caso o limite de saque seja ultrapassado;

- `saldo(inout conta conta_corrente):`

  ⇒ realiza a operação de verificação de saldo através do parâmetro de entrada/saída (`inout`) da estrutura `conta`;

- `verif_dados(in conta conta_corrente);`
  `raises(SenhaErrada, ContaErrada):`

  ⇒ realiza a operação de verificação de dados (número da conta e senha) através do parâmetro de entrada a estrutura `conta`, o `raises` determina um retorno de exceção caso os dados estejam errados;

Na descrição de interface não é necessário inserir qualquer código referente a aplicação, isto será feito após o pré-processamento no compilador IDL. A descrição de interface desta aplicação é apresentada na figura 5.2.

```
//
//      banco.idl
//
exception SenhaErrada{};
exception ContaErrada{};
exception LimiteExcedido{};

struct conta {
  short senha;
  short numero;
  short saldo;
};
typedef sequence<conta> sequenciaDeContas;

interface banco {
   boolean deposito(in short valor_dep, in conta conta_corrente);
   boolean saque(in short valor_saq, in conta conta_corrente)
           raises(LimiteExcedido);
   boolean saldo(inout conta conta_corrente);
   boolean verif_dados(in conta conta_corrente)
           raises(SenhaErrada, ContaErrada);
};
```

Figura 5.2 Exemplo de descrição de interface de uma aplicação bancária.

Após o pré-processamento da descrição de interface acima, que faz o mapeamento da IDL para C++, o compilador IDL do Electra gera o código para as comunicações entre cliente/servidor (*stubs*) e o *skeleton* onde deverá ser inserido o código referente as operações do servidor (figura 5.3 e implementação dos métodos no anexo 1).

```
//
//   banco_im.cc
//
//   Contem a implementação do objeto especificado em banco.idl
//
//

#include <orb/CORBA.h>
#include "banco.hh"       // arquivo cabeçalho contendo a
                          // referência do objeto, declaração
                          // dos métodos da implementação, typedef,
                          // etc.

#include "banco_im.hh"    // arquivo cabeçalho do skeleton servidor

Boolean _im_banco::deposito(Short valor_dep, const conta&
conta_corrente, Environment& env)
{           };
Boolean _im_banco::saque(Short valor_saq, const conta&
conta_corrente, Environment& env)
{           };
Boolean _im_banco::saldo(conta& conta_corrente, Environment& env)
{           };
```

```
Boolean _im_banco::verif_dados(const conta& conta_corrente,
Environment& env)
{           };
void _im_banco::get_state(AnySeq& state, Boolean& done)
{           };
void _im_banco::set_state(const AnySeq& state, Boolean done)
{           };
void _im_banco::view_change(const View& view)
{           };
```

Figura 5.3 *Skeleton* da aplicação bancária.

Este exemplo nos próximos itens servirá de veículo para explicitar as implementações realizadas em nosso laboratório.

5.2.2 Mecanismos de Gerenciamento de Grupo e de Configuração no ORB/Electra

Como salientamos no capítulo 2, o Electra fornece acesso a serviços de gerenciamento de grupo ao programador da aplicação. Neste sentido estão as operações que tratam com transferência de estado e com *membership*.

Os métodos `get_state`, `set_state` e `view_change` são gerados automaticamente no arquivo *skeleton* (banco_im.cc); eles são mecanismos oferecidos pelo BOA para o gerenciamento de *membership* e transferência de estado entre réplicas. Quando uma réplica se junta a um grupo o sistema Electra ativa o método `get_state` na réplica mais antiga do grupo (*rank* = 1) e, após obter seu estado, ativa o método `set_state` na nova réplica, para transferir o estado obtido da réplica mais antiga. O estado (`state`) a ser transferido é definido pelo programador da aplicação dentro dos métodos `get_state` e `set_state`. O método `view_change` tem por função informar os objetos sobre mudanças no *membership*: quando há a entrada ou saída (normal ou por falha) de réplicas no grupo, o sistema ativa o método `view_change` em todas as réplicas do grupo informando o novo *membership* e o novo *rank* de cada uma (figura 5.4).

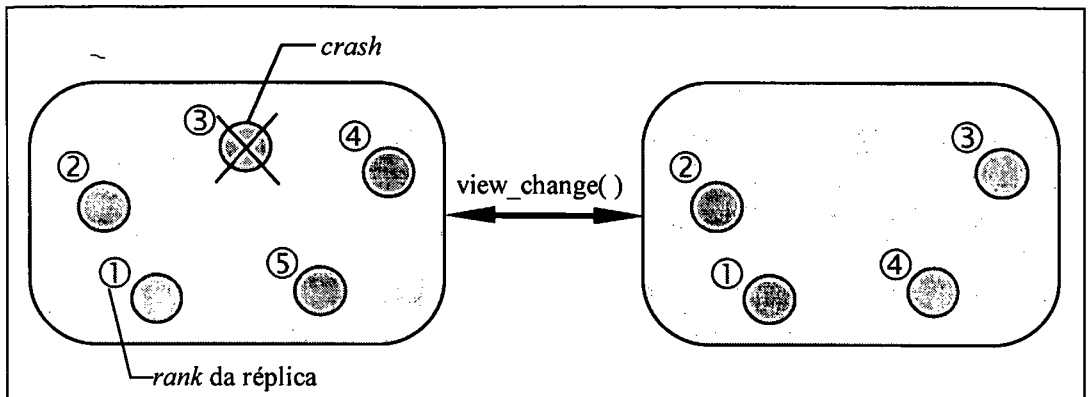


Figura 5.4 Mudança de *membership*

O Electra fornece dispositivos de seleção para os serviços de comunicação. Conforme exposto no item 2.4.4, o sistema Electra suporta comunicações por disseminação confiável (através do suporte Horus) e ponto-a-ponto, ambas com garantias de ordenação (fifo, causal e total).

A seleção da política de comunicação e de ordenação é feita diretamente no código contido no arquivo `banco_srv.cc` e passado ao BOA no momento da criação do grupo. A configuração da política de comunicação é feita de forma portável, o programador entra com os parâmetros determinando o tipo de rede (host local, WAN, LAN), o tipo da comunicação (ponto-a-ponto ou *multicast*) e o tipo de ordenação de mensagem desejado (fifo, causal e total). As opções *default* são[2]:

```
⇒ ProtocolPolicy endpoint_proto(Policy::eWAN, Policy::eEndpoint,
  Policy::eFifo);

⇒ ProtocolPolicy group_proto(Policy::eWAN, Policy::eGroup,
  Policy::eTotal);

⇒ ProtocolPolicy raw_group_proto(Policy::eLAN, Policy::eGroup,
  Policy::eUnorderedMsg);
```

[2] além destas o programador pode configurar a política de acordo com suas necessidades.

Quando em comunicação de grupo e retorno de resposta não transparente[3], é possível configurar o sistema para três tipos de recepção de resultados pelo cliente (função `num_replies`):

⇒ `IIOP::AllRep`: as respostas de todos os membros do grupo são retornadas ao cliente;

⇒ `IIOP::MajorityRep`: a resposta é retornada ao cliente quando a maioria das réplicas tiverem respondido;

⇒ `IIOP::CompareRep`: as respostas de todas as réplicas são coletadas pelo ORB e comparadas, a mais frequente e retornada ao cliente.

Na comunicação transparente o cliente recebe a resposta da réplica que completar primeiro a execução da tarefa, as restantes são descartadas pelo sistema. A execução distribuída pode ser configurada para que o servidor seja um grupo de réplicas ou apenas uma única réplica, como no CORBA convencional, direto na linha de comando:

⇒ host_1> banco_srv -c; -c (*create*): para criação de um grupo com referência banco;

⇒ host_2> banco_srv -j; -j (*join*): para aumentar o grau de replicação do grupo banco;

⇒ host_1> banco_srv -s; -s (singleton): para criação de réplica única;

Na criação de um grupo ou de uma réplica simples, o Electra gera uma referência do objeto, e que o cliente utiliza para fazer requisições ao grupo de objeto (ou objeto único), as novas réplicas utilizam para se juntar ao grupo.

5.2.3 Limitações do Electra para as Implementações das Técnicas de Replicação e o Modelo de Integração

A partir do que foi apresentado no item anterior e nos estudos do capítulo 2 e 4, podemos avaliar a viabilidade do sistema Electra no sentido de fornecer suporte a implementação do modelo de integração proposto. As necessidades iniciais para implementação do modelo eram

[3] o cliente realiza uma requisição e recebe a resposta de cada réplica do grupo.

a disposição de um ORB que fornecesse, além dos conceitos CORBA, um suporte para processamento em grupo de objetos (réplicas). No Electra, verificamos que este oferece os quatros principais mecanismo para o suporte desejado, a lembrar: comunicação por disseminação confiável de mensagem; ordenação fifo, causal e total; gerenciamento de *membership* e mecanismos de transferência de estado. Estes serviços são disponíveis e configuráveis pelo programador da aplicação.

Conforme mencionado, na ativação de um método do grupo de objetos, a *stub* cliente acessa a referência do grupo para realizar a invocação. No caso de um objeto do grupo acessar o grupo ao qual pertence, o processamento é feito de forma parecida, com a diferença de que o método solicitado é ativado, também, na própria réplica que a solicitou (Figura 5.5). Isto implica em concorrência, no interior de cada réplica, gerando a necessidade de um suporte de *multithreads*. Este suporte permite que múltiplas *threads* de controle se executem dentro de um objeto.

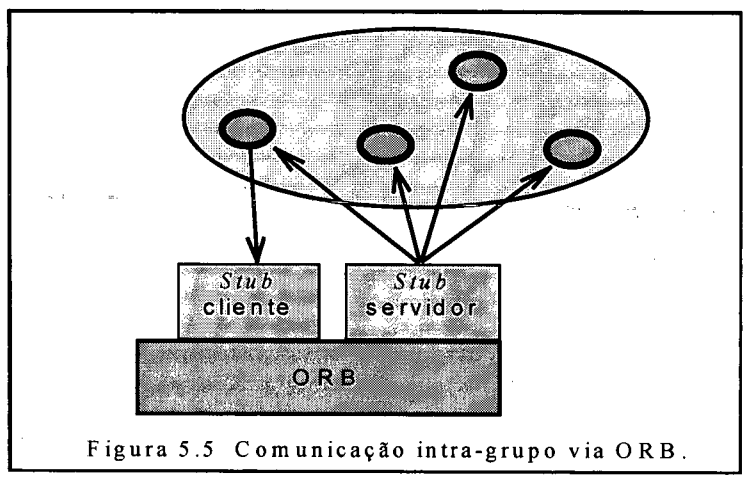


Figura 5.5 Comunicação intra-grupo via ORB.

Em nossas experimentações as necessidades de concorrência entre objeto-base e meta-objeto, no interior de um processo, são satisfeitas através de uma biblioteca de *threads* oferecida pelo suporte Electra. Entretanto, a versão atual (1.0) desta plataforma não suporta *threads* preemptivas, o que limita o grau de concorrência no tratamento de pedidos do cliente. O mesmo ocorre com as interações entre controladores. Na verdade, o modelo de programação do Electra não favorece comunicações entre réplicas (entre controladores). Devido a estas restrições, torna-se difícil implementar técnicas de replicação, o que nos forçou a busca de

soluções alternativas de implementação. A solução adotada consiste em separar as funcionalidades do meta-controlador em dois processos UNIX. Um trata das interações cliente/servidor e da ativação do método do objeto-base e outro que trata das comunicações entre réplicas (entre coordenadores de réplica).

Outra limitação encontrada é que o sistema Electra não oferece construções específicas de linguagem para suporte à *reflexão*, como por exemplo o desvio automático da ativação de um método no nível base para o respectivo meta-método. O uso de uma linguagem suportando reflexão, como é o caso de *Open C++* [Chiba 93], poderia eliminar este problema, mas neste caso o compilador IDL do ambiente CORBA usado deve ter um mapeamento para essa linguagem.

Devido a linguagem C++ não ter construções específicas para reflexão, a implementação de reflexão no CORBA é feita através da ativação direta ao nível meta. O cliente ativa um método da mesma forma que no CORBA convencional. Esta transparência é assegurada considerando que o *i*-ésimo `CORBA::meta_método_i(...)`[4] é o método reflexivo do método base - `método_base_i(...)`, onde está a implementação do método em si, isto é, o cliente necessitando de um método *i* ativa, via ORB, o `CORBA::meta_método_i(...)` que ativa localmente, sem uso do ORB, o `método_base_i (...)` (figura 5.6).

5.3 Implementações

Apresentamos neste item alguns aspectos de implementação das técnicas de replicação realizadas sobre a plataforma aberta Electra/CORBA, segundo o modelo de integração proposto no capítulo anterior.

[4] o prefixo CORBA significa que o método é ativado via ORB ao grupo.

5.3.1 Implementação da Técnica de Replicação *Competitiva*

Após a execução do `método_base_i`[5] solicitado, o segundo passo é a ativação local do método `meta_controle(...)` que inicia o protocolo de gerenciamento do grupo de réplicas, segundo a técnica de replicação utilizada. Dentro do método `meta_controle(...)` são realizadas algumas computações referentes ao protocolo de replicação e é ativado o método `difunde_id`, que tem a função de informar a cada réplica, naquela transação, quem primeiro lhe ativou. O método `difunde_id` ao ser ativado por uma réplica fornece a informação através de um contador que marca a colocação da réplica ativadora naquela transação.

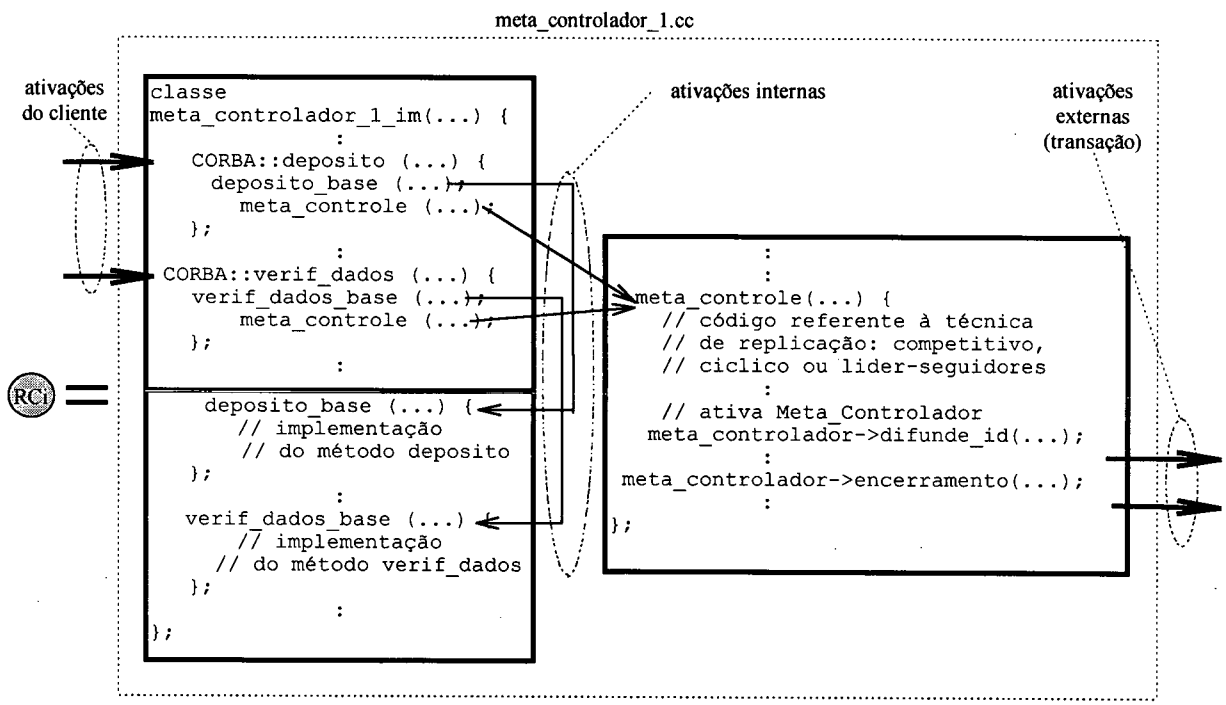


Figura 5.6 Formato da separação do código

Para a implementação de detecção de *crash* da réplica privilegiada as réplicas pertencentes ao grupo ativam o método `meta_controlador->encerramento(...)` para verificar se a réplica mais rápida ainda pertence ao grupo (está no *membership*), se sim a transação se dá por

[5] os métodos podem ser, por exemplo, os métodos da aplicação bancária.

terminada, caso contrário, a segunda réplica mais rápida responde ao cliente e assim, se repete o ciclo de detecção.

5.3.2 Implementação da Técnica de Replicação *Cíclica*

A implementação desta técnica segue o mesmo padrão do item anterior (figura 5.6). Neste caso, o mecanismo de passagem de bastão é implementado de forma virtual através de um contador. Após processar o método solicitado pelo cliente no nível base, é realizado o procedimento de definição da réplica privilegiada. Cada réplica do grupo possui um contador circulante. Este contador é incrementado a cada requisição do cliente e quando passa do *membership* do grupo, é retornado para um. Quando a identidade de uma réplica (*rank*) for igual a esse contador, a réplica é identificada como a privilegiada, com o direito de responder ao cliente da presente requisição. Antes de retornar a resposta, a réplica privilegiada ativa o método `encerramento` para inicio do ciclo de detecção de *crash*, como apresentado no modelo competitivo. Se for detectado o *crash,* a resposta é retornada ao cliente pela próxima réplica do anel lógico.

5.3.3 Implementação da Técnica de Replicação *Líder/Seguidores*

Neste modelo, a definição do líder é indicada pela réplica que tiver o *rank* igual a um. O *rank* é fornecido pelo suporte Electra e dinâmico - alterado a cada mudança de *membership*. A réplica líder ao processar a requisição do cliente ativa o método `encerramento`. A diferença entre este método de encerramento com o do modelo competitivo é que um dos parâmetros deste método é a resposta produzida pelo líder. Após o líder enviar sua resposta ao grupo e responder ao cliente, os seguidores ativam o método encerramento e que tem como retorno a resposta do líder. Conforme especificamos, se o líder falhar o seguidor escolhido para substituí-lo será a que tiver o *rank* igual a dois naquela transação (figura 5.4).

5.3.4 Implementação da Técnica de Replicação *Primário/Backups*

O mecanismo de detecção da réplica primário é o mesmo do modelo *líder/seguidores*, através do *rank*. Diferente do anterior, a definição da réplica primária é feita antes da ativação do

método base. Com isso, é garantido que apenas o primário ativa o método solicitado no nível base (figura 5.7). Nesta implementação separamos o meta_controle em dois métodos: meta_primário e o meta_backup, a primeira realiza os procedimentos do *primário* e a segunda os procedimentos do *backup*. O mecanismo de transferência de estado (*checkpoint*) se baseia nas primitivas do BOA - `get_state` e `set_state`. O primário, após processar o método solicitado pelo cliente, ativa o `get_state` que pega o seu próprio estado (pré-definido pelo programador) e deposita na variável `state`. Em seguida, o primário ativa o método encerramento que tem como um dos parâmetros a variável `state`. Este procedimento transfere o `state` do primário para o grupo e também confirma que não houve *crash* do primário. Após isto, os *backups* ativam o método encerramento para obter o `state` do primário, em seguida ativa a função `set_state` para atualizar seu estado em relação ao primário.

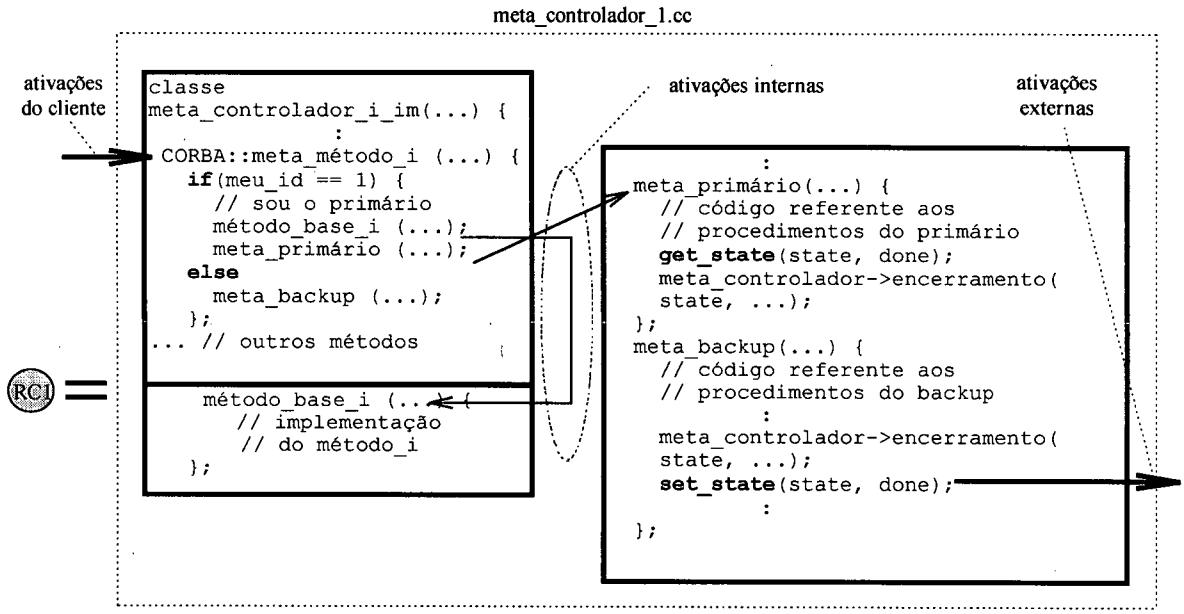


Figura 5.7 Replicação primário/*backup*.

Em caso de *crash* do primário a réplica substituta (*rank* = 2) executa o método solicitado e assume o papel de primário para transferência de estado nas outras réplicas. Isto é possível porque a requisição do cliente é recebida por todas réplicas do grupo (*multicast* confiável) e não somente pelo primário.

5.3.5 Considerações da Implementação

Nossas implementações foram efetuadas sobre um ambiente composto por um grupo de máquinas Sun Sparc com sistema operacional SunOs 4.1.x, interligadas por uma rede no padrão Ethernet. Para nossos estudos consideramos inicialmente a aplicação bancária descrita no início deste capítulo, composta por um servidor replicado e um ou mais clientes distribuídos sobre a rede. Atualmente uma aplicação distribuída para visualização de imagens MPEG está sendo desenvolvida, mas ainda não está operacional.
A execução da aplicação inicia com o lançamento de um servidor sobre uma das estações da rede, criando assim o grupo "banco". Outros servidores são lançados em outras estações da rede, juntando-se ao grupo já criado, até atingir-se o grau de replicação desejado. Uma vez instalado o grupo de servidores, clientes podem ser criados em diferentes postos do sistema, e estes podem acessar os serviços oferecidos pelo servidor. A ativação de um método por um cliente é passada para o suporte ORB/Electra, que efetua um *multicast* aos membros do grupo servidor.

Os testes para verificação do comportamento do servidor replicado em caso de falhas foram efetuados para cada uma das estratégias de replicação escolhidas para implementação. A ocorrência de falha em uma réplica foi simulada interrompendo-se a execução do processo que a implementa. Foi verificado que em todas as técnicas de replicação testadas o sistema foi capaz de se reconfigurar, determinando a nova réplica privilegiada para responder as requisições dos clientes. Também foi testada a adesão de novas réplicas ao grupo, com os resultados esperados. A recuperação automática do grau de replicação do grupo ainda esta sendo implementada, mas os primeiros resultados confirmam as expectativas.

5.4 Conclusões do Capítulo

As implementações apresentadas neste capítulo vieram no sentido de verificar a viabilidade de se implementar mecanismos de tolerância a falhas a nível da aplicação sobre a plataforma aberta CORBA utilizando um modelo reflexivo. As implementações foram feitas realizando as separações preconizadas pela reflexão computacional. Estas mesmas implementações

fazem uso extensivo das primitivas do suporte Electra, o que também não implica na perda de transportabilidade dos códigos da aplicação. Ou seja, mudanças no suporte se refletem apenas no meta-nível; os códigos que refletem as funcionalidades da aplicação não são afetados. Os resultados obtidos, embora as limitações do Electra, foram considerados satisfatórios.

Há a possibilidade de se implementar mecanismos de detecção de falhas menos restritivas, (temporização, valor etc). O Electra fornece um mecanismo de comparação das respostas enviadas pelos membros do grupo replicado (item 5.2), isto é feito no suporte de baixo nível (Horus, no caso). O mecanismo de *multithreads* que o Electra opera é o chamado *thread per process* (TPP), este modelo impõe que o servidor execute uma requisição de cada vez, do início ao fim, múltiplos objetos servidores que compartilham um mesmo processo não podem receber invocações concorrente. Em alguns casos este modelo pode atingir um *deadlock*, por exemplo: suponha que um servidor *A*, ao ser ativado pelo cliente, faça uma requisição a um servidor *B* e que durante o processamento desta requisição *B* ative *A*, isto provocaria um *deadlock* no sistema, pois *A* só pode processar uma requisição por vez (figura 5.8).

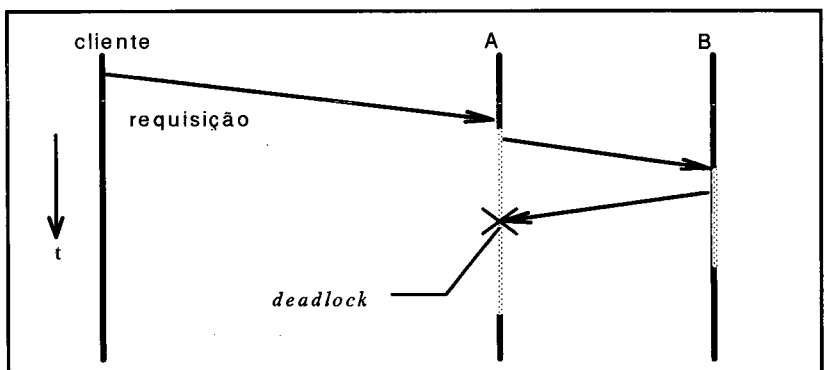


Figura 5.8 *deadlock* causado pela réplica B.

O mecanismo de *multithreads* que precisamos para implementar o modelo de integração proposto no capítulo anterior é o chamado *thread per request* (TPR). Com este mecanismo, uma nova *thread* é criada para cada invocação que chega no servidor e evita problemas de *deadlock* que podem ocorrer no TPP.

CAPÍTULO 6
CONCLUSÕES e PERSPECTIVAS

O trabalho de pesquisa realizado nesta dissertação de mestrado conseguiu atingir os principais objetivos apontados no início desta. O estudo dos requisitos para processamento em grupo de objetos sobre uma plataforma aberta CORBA permitiu-nos avaliar com exatidão os ORB's, uns na forma de protótipo e outros como produto comercial, no sentido da viabilidade destes para suporte ao desenvolvimento de técnicas de replicação de componentes de *software*. A escolha do sistema Electra para implementação se deu por ser um protótipo disponível livremente para instituições acadêmicas. No entanto, o suporte ideal para nossas experimentações, segundo os requisitos apontados, seria o produto comercial Orbix+Isis devido principalmente ao oferecimento do suporte *multithreads* - TPR, além dos essenciais suportes para processamento de grupo de objetos. Acreditamos que as especificações das técnicas de replicação feitas no capítulo 4 poderiam ter sido concretizadas com maior eficiência na plataforma Orbix+Isis. Mas, os resultados obtidos no Electra não devem ser desconsiderados, pois permitiu-nos ter uma visão mais concreta e uma experiência inicial sobre as características das implementações que seguem o modelo de integração.

CORBA

A escolha do modelo CORBA para nossas pesquisas se deve pelo fato deste ser um padrão amplamente difundido entre as empresas e universidades. De certa forma, o modelo de objetos distribuídos CORBA já garante alguma transparência e separação da implementação da aplicação em relação ao suporte, nos aspectos referentes a comunicação a ao gerenciamento do mesmo.

O ORB e o BOA são implementados na forma de classes que são compostas por múltiplas heranças de outras classes, que serão herdadas pelas classes cliente e servidor, no momento da compilação, para gerar o executável de ambos. Para ambos clientes e servidor o suporte de comunicação é transparente, as ativações são realizadas como se fossem localmente.

O modelo de integração

O modelo de integração proposto neste trabalho, pelas suas características reflexivas, ao separar aspectos de coordenação da própria construção das réplicas obtemos a flexibilidade sempre citada nas implementações reflexivas. Podemos com a troca de meta-controladores mudar de técnica de replicação ou alterá-la para atender premissas diferentes. As mudanças necessárias à substituição de técnicas de replicação no modelo de integração limitam-se à interface IDL dos meta-controladores e aos seus códigos, que implementam os protocolos de coordenação correspondentes.

Se considerarmos sistemas distribuídos abertos, com suas dimensões e características de heterogeneidade, o modelo apresentado, fundamentado sobre serviços de grupo fornecidos por uma plataforma CORBA, nos parece altamente adequado. Soluções usuais de implementações de replicações seguindo as abordagens de suporte de tempo de execução, de bibliotecas ou de linguagens se mostram inadequadas nestes sistemas considerando o caráter evolutivo dos mesmos. A desvantagem da abordagem de suporte de tempo de execução é que uma vez definida as premissas de falhas e a técnica de replicação a ser usada, teremos definido em tempo de configuração um suporte de execução específico, mudanças implicariam na necessidade de reconfiguração criando um novo suporte. As abordagens de bibliotecas de funções e de linguagem trazem os aspectos de coordenação a nível de programador da aplicação, porém sem separá-los dos aspectos funcionais da aplicação. A convivência dos mesmos em um mesmo código diminui o grau de reusabilidade do *software*. Mudanças implicariam em refazer os códigos de aplicação. As implementações de técnicas de replicação, segundo o modelo introduzido, com as características de reflexão e trazendo a nível de programador de aplicação, porém separados os aspectos funcionais dos de coordenação, se mostram mais flexíveis e propícios as necessidades de sistemas abertos distribuídos.

Demonstramos a viabilidade do modelo de integração através da implementação de protocolos de replicação a nível da aplicação, que originalmente eram implementados a nível do suporte de sistema distribuído (líder/seguidores em Delta-4[1], primário/*backup* em ISIS[2] etc.). Nesta implementação é interessante salientar que a separação dos código da aplicação e da coordenação da replicação permitiu obter a flexibilidade necessária ao desenvolvimento e implementação de diferentes modelos de replicação para tolerância a falhas em sistemas abertos.

No Âmbito do Projeto ASAP

Segundo a proposta original do projeto ASAP [ASAP 94], o objetivo principal era desenvolvimento de um ambiente para suporte de aplicações distribuídas, baseado em objetos, com mecanismos de suporte à aplicações tempo real e tolerância a falhas. Os estudos realizados a princípio atingiu os objetivos iniciais do ASAP referentes a suporte de grupo e implementação de técnicas de replicação.

Perspectivas

Com a realização deste trabalho, perspectivas de novos trabalhos poderão ser desenvolvidas para dar segmento a este:

- realização de medidas detalhadas sobre o desempenho das técnicas de replicação sobre a plataforma Electra;
- implementação do modelo de integração proposto sobre a plataforma Orbix+Isis, quando este estiver disponível;
- implementação de técnicas de replicação com suposição de faltas menos restritivas;

[1] sistema de suporte *runtime*
[2] baseado em bibliotecas de funções e primitivas básicas

- proposição de um mapeamento da IDL para uma linguagem com construções para reflexão (ex: Open C++) para desenvolvimento de um compilador segundo as especificações do padrão CORBA;

- desenvolvimento de aplicações mais complexas, do tipo multimídia (visualizador de animações em formato *MPEG*), para serem testados no modelo de integração implementado;

Bibliografia

[Adler 95] R. M. Adler, **"Group-Oriented Coordination Extensions to OMG's OMA/CORBA".** OMG presentation, San Jose, CA, June 26-29, 1995.

[Agha 93] G. Agha, S. Frolund, R. Panwar, D. Sturman, **"A Linguistic Framework for Dynamic Composition of Dependability Protocols"**, Proceedings of the DCCA-3, 1993.

[Amir 92] Amir, Y., Dolev, D., Kramer, S. and Malki, D., **"Transis: A Comunication Subsystem for High Availability"**. In 22nd International Symposium on Fault-Tolerant Computing, IEEE, July 1992.

[ANSA 94a] E. Oskiewicz, N. Edwards, **"A Model for Interface Groups"**, ANSA Phase III technical report APM.1002.01, Cambridge-UK, may 1994.

[ANSA 94b] G. Li, **"Distributed Real-Time Objects: Some Early Experiences"**. ANSA Phase III, APM.1231.00.01 draft 25 May 1994.

[Avizienis 77] T. Avizienis and L. Chen, **"On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution"**, Proc. COMPSAC &&, Chicago, pp. 149-155, Nov, 1997.

[Barret 90] P. A. Barret et al, "The Delta-4 Extra Performance Architecture (XPA)", Proceedings of FTCS-20, Newcastle upon Tyne, June 1990.

[Birman 85] K. Birman et al, **"Implementing Fault-Tolerant Distributed Object"**, IEEE Transactions on Software Engineering, June 1985.

[Birman 88] K. Birman, T. Joseph and F. Schmuck, **"ISIS - A Ditributed Programming User's Guide and Reference Manual"**, The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, March 1988.

[Birman 91a] K. P. Birman , R. Cooper e B. Gleeson, **"Programming with Process Groups: Group and Multicast Semantics"**, Technical Report Tr91-1185, Cornell University Computer Science Department, Ithaca, N.Y., December 1991.

[Birman 91b] K. P. Birman, **"The Process Group Approach to Reliable Distributed Computing"**, Technical Report Tr91-1216, Cornell University Computer Science Department, Ithaca, N.Y., July 1991. Submitted to Comm. ACM.

[Brito 95] O. F. G. Brito, O. G. Loques, "Técnicas de Replicação de Módulos no Ambiente RIO", VI Simpósio de Computadores Tolerantes a Falhas, Canela - RS, 1995.

[Chiba 93] S. Chiba, **"Open C++ Programmer's Guide"**, Technical Report 93-3, Department of Information Science, University of Tokio, 1993.

[Chorus 92] Chorus Systemes, **"Chorus Simulator v3 r4 - Programmer's Guide"**, 1992.

[Dueñas 92] L. T. R. Dueñas, **"Uma Proposta de Algoritmo Assincrono para Difusão Confiável Atômica"**, Dissertação de Mestrado, LCMI/UFSC, 1992.

[Fabre 95] J. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud and Z. Wu, **"Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming"**, Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena (CA), June 1995.

[Furtado 95] J. O. Furtado, **"Um Modelo e uma Linguagem para Aplicações Tempo Real"**, Exame de Qualificação ao Doutoramento, LCMI/EEL/UFSC, Outubro 1995.

[Fraga 96] J. S. Fraga, C. A. Maziero, Lau C. L. e O. G. Loques **"Serviços Replicados em uma Plataforma Aberta CORBA usando uma Abordagem Reflexiva"**, SBRC - II Workshop do Projeto ASAP, Fortaleza - CE, Maio 1996.

[Fraga 96] J. S. Fraga, C. A. Maziero, Lau C. L. e O. G. Loques **"Implementação de Serviços Replicados em um Ambiente Aberto usando uma Abordagem Reflexiva"**, II Workshop do Projeto ASAP, Fortaleza - CE, Maio 1996.

[Giandome 90] Giandomenico, F.L. and Strigini, **"Adjudicators or Diversity Redundant Components"**, in Proceedings of the 20th International Symposium on Fault-Tolerant Computing , Alabama, pg 114-123. 1990.

[Hagsand 92] O. Hagsand, H. Herzog, K. P. Birman e R. Cooper, **"Object-Oriented Reliable Distributed Programming"**, IEEE, 2nd International Workshop on Object-Orientation in Operational Systems, I-WOOOS/1992.

[IONA 94] Isis Distributed Systems Inc., IONA Techonologies, Ltd. **"An Introduction to Orbix+Isis"**, Jul. 1994. Doc.

[IONA 95] Isis Distributed Systems Inc., IONA Techonologies, Ltd. **"Orbix+Isis Programmer's Guide"**, 1995. Document D070-00.

[ISIS 91] Isis Distributed Systems Inc., **"ISIS/C++ Reference Manual and Tutorial"**, Nov. 1991.

[ISIS 93] Isis Distributed Systems Inc., Ithaca, N.Y, **"Object Groups: A Response to the ORB 2.0 RFI"**, April 1993.

[ISIS 94] Isis Distributed Systems Inc., **"RDO/C++ Tutorial Release 1.0.3"**, Apr. 1994.

[ISO 93] ISO/IEC 10746-3, **"ITU-TS Recommendation X.903: Basic Reference Model of Open Distributed Processing: Prescriptive Model"**, (2nd CD draft) June 1993.

[Jalote 94] P. Jalote, **"Fault Tolerance in Distributed System"**, PTR Prentice Hall, Englewood Cliffs, New Jersey 1994.

[Lau 95] Lau C. L., J. S. Fraga e C. A. Maziero **"Suporte a Grupos de Objetos em uma Plataforma CORBA"**, I Workshop do Projeto ASAP, Florianópolis - SC, Outubro 1995.

[Lea 94] D. Lea, **"Object in Groups"**, SUNY at Oswego/NY CASE Center, Submitted, ECOOP 94.

[Liang 90] L. Liang , S. T. Chanson e G. W. Neufeld, **"Process Groups and Group Communications: Classifications and Requirements"**, IEEE Computer, pp 56-65, February 1990.

[Liskov 87] B. Liskov, **"Implementation of Argus"**, Proceedings of the 11th ACM Symposium on Operating Systems Principles, November 1987.

[Lisbôa 96] M. L. B. Lisbôa, **"O Framework MOTF: Meta-Objetos para Tolerância a Falhas"**, II Workshop do Projeto ASAP, Fortaleza - CE, Maio 1996.

[Liskov 88] B. Liskov, **"Distributed Programming in Argus"**, Communications of the CACM, Mach 1988.

[Little 91] M. C. Little, **"Object Replication in a Distributed System"**, Ph.D. Thesis, Computing Laboratory of the University of Newcastle upon Tyne, September 1991.

[Maes 87] P. Maes, **"Concepts and Experiments in Computational Reflection"** , OOPSLA 87 Proceedings, pp. 147-156, October 1987.

[Maffeis 93] S. Maffeis, **"Distributed Programming Using Object-Groups"**, Technical Report Tr 93.38, University of Zurich Dept. of Computer Science., September 1993.

[Maffeis 95a] Maffeis, S. **"Adding Group Communication and Fault-Tolerance to CORBA",** In Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies (Monterey, CA, June 1995), USENIX.

[Maffeis 95b] Maffeis, S. **"Electra 1.0 Reference Manual"**, Dept. Computer System opf the Cornell University, Jul. 1995.

[Maffeis 95c] Maffeis, S. **"Electra 1.0 Programmer's Manual"**, Dept. Computer System opf the Cornell University, Jul. 1995.

[Mishra 93] Mishra, S., Peterson, L. L., and Schlichting, R. D. **"Consul: A Communication Substrate for Fault-Tolerant Distributed Programs"**. Distributed Systems Engineering Journal 1,2 Dez. 1993.

[OMG 93] Object Management Group, **"The Common Object Request Broker: Architecture and Specification"**, Revision 1.2, OMG Document, December 1993.

[OMG 94] Object Management Group, **"IDL C++ Language Mapping Specification"**, OMG Document 94-9-14, 1994.

[OSF 92] W. Rosenberry, D. Kenney and G. Fisher, **"Understanding DCE"**, O'Reilly & Associates, Inc. 1992.

[Oskiewicz 93] E. Oskiewics and N. Edwards, **"A Model for Interface Groups"** Tech. Rep. AR.002.01, ANSA, Architecture Projects Management Limited, Cambridge UK, 1993.

[Powell 91] D. Powell, **"Delta-4 Architecture Guide"**, Esprit II P2252, Delta-4 Phase 3, August 1991.

[Randell 75] B. Randell, **"System Structure for Software Fault Tolerance"**, IEEE Transaction on Software Engineering, June 1975, pg. 220-232.

[Renesse 95] Van Renesse, R. and Birman, K. P. **"Protocol Composition in Horus"**, In Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, Ago 1995.

[Schneider 90] F. B. Schneider, **"Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial",** ACM Computing Survey, 22(4):299-319, Dec 1990.

[Shapiro 86] M. Shapiro, **"Structure and Encapsulation in Distributed Systems: The Proxy Principle"**, 198-204 IEEE, 1986.

[Verissimo 92] P. Veríssimo e L. Rodrigues, **"Group Orientation: a Paradigm for Distributed Systems of the Nineties"**, 3rd IEEE Workshop on Future Trends of Distributed Computing Systems, Tape, Taiwan, April 1992.

[Vinoski 93] S. Vinoski, **"Distributed Object Computing with CORBA"**, C++ Report Magazine, July/August 1993

Anexo 1

```
//
//    banco_im.cc
//
//    Contem a implementacao do objeto especificado em banco.idl
//
//

#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"


Boolean _im_banco::deposito(Short valor_dep, const conta&
conta_corrente, Environment &)
{
     ULong i=0;

     cout << "->depositando..." << endl;
     _saldo_conta = _saldo_conta + valor_dep;
     contas.length(i+1);
     contas[i].senha = conta_corrente.senha;
     contas[i].numero= conta_corrente.numero;
     contas[i].saldo = _saldo_conta;
     return 1;
};


Boolean _im_banco::saque(Short valor_saq, const conta&
conta_corrente,
                Environment& env)
{
     ULong i=0;

     temp = _saldo_conta - valor_saq;
     if (temp < LIMITE)
          env.exception(new LimiteExcedido);
     else {
          cout << "->sacando..." << endl;
          _saldo_conta = _saldo_conta - valor_saq;
          contas.length(i+1);
          contas[i].senha = conta_corrente.senha;
          contas[i].numero= conta_corrente.numero;
          contas[i].saldo = _saldo_conta;
          int a;
          for(a=0; a<40000000; a++);
          get_state(state, done);
          return 1;
```

```
    }
};


Boolean _im_banco::saldo(conta& conta_corrente, Environment&
env)
{
    cout << "->consultando..." << endl;
    cerr << "----- meu rank: " << rank << endl;

    conta_corrente.saldo = _saldo_conta;
    return 1;
};


Boolean _im_banco::verif_dados(const conta& conta_corrente,
Environment& env)
{


    if (conta_corrente.numero > NUMERO1)
        env.exception(new ContaErrada);
    else
        if (conta_corrente.senha > SENHA1)
            env.exception(new SenhaErrada);
        else
        return 1;
};


void _im_banco::get_state(AnySeq& state, Boolean& done)
{
    cerr << "----pegando estado\n";
    state.length(3 * contas.length());
    for (ULong i=0; i < 3 * contas.length(); i+=3) {
        state[i]   <<= contas[i/3].senha;
        state[i+1] <<= contas[i/3].numero;
        state[i+2] <<= contas[i/3].saldo;
    }
    done = TRUE;
};


void _im_banco::set_state(const AnySeq& state, Boolean done)
{
    cerr << "-----setando estado\n";
    contas.length(state.length() /3);
    for (ULong i=0; i < state.length(); i+=3) {
        state[i]   >>= contas[i/3].senha;
        state[i+1] >>= contas[i/3].numero;
```

```
            state[i+2] >>= contas[i/3].saldo;
            _saldo_conta = contas[i/3].saldo;
        }
        done = TRUE;
};


void _im_banco::view_change(const View& view)
{
        cerr << "----- n. de membros: " << view.num_members() <<
endl;
        cerr << "----- meu rank: " << view.my_rank() << endl;
        cerr << "----- iname: " << view.iname() << endl;
        rank = view.my_rank();

};



//
//      banco_im.hh
//
//

class _im_banco : public virtual _sk_banco
{
private:
  #define LIMITE 0

  #define SENHA1 1111
  #define NUMERO1 1111

  short     _saldo_conta,temp;
public:
  // operacoes em tempo real:
  //
  _im_banco(const ProtocolPolicy& p = ::endpoint_proto): BOA(p)
{
        _saldo_conta = 0;
        temp = 0;
  };

        AnySeq state;
        Boolean done;
        short rank;

  Boolean deposito(Short valor_dep, const conta&
conta_corrente, Environment&);
  Boolean saque(Short valor_saq, const conta& conta_corrente,
                Environment&);
```

```
  Boolean saldo(conta& conta_corrente, Environment&);
  Boolean verif_dados(const conta& conta_corrente,
                Environment&);
  virtual void get_state(AnySeq& state, Boolean& done);
  virtual void set_state(const AnySeq& state, Boolean done);
  virtual void view_change(const View& view);
protected:
  sequenciaDeContas contas;
};



//
//
//   banco_srv.cc
//
//   Contem o esqueleto da aplicacao servidor.
//
//

#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"

electra_main(int argc, char **argv){
  ORB_ptr           orb = ::ORB_init(argc, argv,
"Electra_ORB");
  BOA_ptr           boa = orb->BOA_init(argc, argv,
"Electra_BOA");
  NamingContext_var      nc =
    NamingContext::_narrow(orb-
>resolve_initial_references("NameService"));
  ReferenceData     rd;
  InterfaceDef_ptr  pt =0;
  ImplementationDef_ptr dpt =0;
  banco_var         obj =0, group =0;
  ProtocolPolicy    e = ::endpoint_proto;
  ProtocolPolicy    g = ::group_proto;



  if(argc > 4){
    cerr << "Usar: " << argv[0] << " [-c|-j] [-s stack]\n";
    exit(1);
  };

  if(argc > 1){
    if((argv[argc-2][0] == '-') && (argv[argc-2][1] == 's')){
      e.stack(argv[argc-1]);
      g.stack(argv[argc-1]);
```

```
    };
  };
  if(argc == 1){
    cerr << "Criando singleton \"banco\"\n";
    _im_banco obj0(e);
    obj = obj0.create(rd, pt, dpt);
    nc->bind("banco", obj);

    cerr << "PRONTO\n";
    boa->impl_is_ready();
    return 0;
  };

  _im_banco obj0(e);
  switch(argv[1][1]){
  case 'c':
    cerr << "Criando grupo \"banco\"\n";
    obj = obj0.create(rd, pt, dpt);
    obj0.create_group(obj, g);
    nc->bind("banco", obj);

    obj0.join(obj);
    break;
  case 'j':
    cerr << "Juntando grupos \"banco\"\n";
    group = nc->resolve("banco");

    obj0.join(group);
    break;
  default:
    cerr << "Usar: " << argv[0] << " [-c|-j]\n";
    exit(1);
  };

  cerr << "PRONTO\n";
  boa->impl_is_ready();
  return 0;
};
```

```
//
//   banco_cnt.cc
//
//   Contem o esqueleto da aplicacao cliente
//
//

#include <orb/CORBA.h>
#include "banco.hh"

electra_main(int argc, char **argv){
  ORB_ptr           orb = ::ORB_init(argc, argv,
"Electra_ORB");
  Object_var        ov;
  NamingContext_var      nc =
    NamingContext::_narrow(orb-
>resolve_initial_references("NameService"));
  Environment       env;
  Long              ss,su,es,eu,ds,du;
  conta             cc_cliente;
  Short             op,valor_dep,valor_saq;
  Exception_ptr     exc;


  ov = nc->resolve("banco");
  banco_var obj0 = banco::_narrow(ov);

  cout << "Digite o numero da conta-corrente :";
  cin >> cc_cliente.numero;
  cout << "Digite o numero da sua senha :";
  cin >> cc_cliente.senha;
  obj0->verif_dados(cc_cliente, env);
  if (exc = env.exception()) {
     if (ContaErrada::_narrow(exc)) {
          cerr << "CONTA INVALIDA !!!!!!" << endl <<
               ERROR(exc) << endl;
          return 0;
     }
     else
          cerr << "SENHA INVALIDA !!!!!!" << endl <<
               ERROR(exc) << endl;
          return 0;
  }
  else
     cerr << "OK !" << endl;

  cout << "<1> Deposito" << endl;
  cout << "<2> Saque" << endl;
  cout << "<3> Saldo" << endl;
  cout << "<4> Fim" << endl;
```

```
  for(;;){
     os->timeGet(ss,su);
     cout << "opcao <1..4>";
     cin >> op;

     if (op==1) {
          cout << "Digite o valor do deposito :";
          cin >> valor_dep;
          if (obj0->deposito(valor_dep, cc_cliente, env))
             cerr << "Deposito feito com sucesso !!!" << endl;
          else
             cerr << "DEPOSITO FALHOU !!!" << endl
               << ERROR(exc) << endl;
     }
     else {
          if (op==2) {
             cout << "Digite o valor do saque :";
             cin >> valor_saq;
             obj0->saque(valor_saq, cc_cliente, env);
             if (exc = env.exception()) {
               if (LimiteExcedido::_narrow(exc))
               cerr << "LIMITE EXCEDIDO - SAQUE FALHOU !!!" <<
endl
                    << ERROR(exc) << endl;
             }
             else
               cout << "Saque feito com sucesso !!!" << endl;
          }
          else {
             if (op==3) {
               if (obj0->saldo(cc_cliente, env))
                  cerr << "Saldo : " << cc_cliente.saldo <<
endl;
               else
                  cerr << "CONSULTA DE SALDO FALHOU !!!" <<
endl
                    << ERROR(exc) << endl;
             }
             else
               return 0;
             }
     }
  }
  return 0;
  if(::is_nil(obj0))
     abort();
  return 0;
};
```

Anexo 2

```
//
// banco_im.hh
// You may modify this file as it will not be overwritten.
//

#include "/soft/electra/demo/modelos/controle/controle.hh"


class _im_banco : public virtual _sk_banco{
private:

  #define LIMITE 0

  #define SENHA1 1111
  #define NUMERO1 1111
  short   _saldo_conta, temp;

public:
  // the real operations:
  //

     mensagem msg;
     short rank, membros_atual, id_prm_conf, id_prm, x, col;
     controlador_var control;
     conta conta_corrente;
     Environment env;

  Boolean  deposito( Short valor_dep, const conta&
conta_corrente, Short& meu_id, Environment& );
  Boolean  saque( Short valor_saq, const conta& conta_corrente,
Short& meu_id, Environment& );
  Boolean  saldo( conta& conta_corrente, Short& meu_id,
Environment& );
  Boolean  verif_dados( const conta& conta_corrente, Short&
meu_id, Environment& );

  // state transfer, view changes:
  //
  virtual void get_state(AnySeq& state, Boolean& done);
  virtual void set_state(const AnySeq& state, Boolean done);
  virtual void view_change(const View& view);

 // metodo do nivel meta: ativadas via chamadas C++

  void meta_competitivo(Short& meu_id);
```

```
 // metodos do nivel base: ativadas via chamadas C++

  Boolean deposito_base(short valor_dep, conta conta_corrente,
Environment&);
  Boolean saque_base(short valor_saq, conta conta_corrente,
Environment&);
  Boolean saldo_base(conta& conta_corrente, Environment&);
  Boolean verif_dados_base(conta conta_corrente, Environment&);

  _im_banco(const ProtocolPolicy& p = ::endpoint_proto): BOA(p)
{
  _saldo_conta = 0;
  temp = 0;
};
protected:
  sequenciaDeContas contas;
  sequnciaDeContadores contador;

};

// meta_competitivo.cc
// You may modify this file as it will not be overwritten.
//
// This file contains the implementation of the object you
specified.
//

#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"


//////////////////////////Nivel Meta////////////////////////////


void _im_banco::meta_competitivo(Short& meu_id){

// gerenciamento do token

     msg.id = rank + 1;

     if(msg.id == 0)
          msg.id = 1;       // rank


     cerr << "--- meu identificador: " << msg.id << endl;

     meu_id = msg.id;

     control->controle(msg, col, id_prm, env);
```

```
     cout << "--- minha colocacao: " << col << endl;


     if(col == 1) {
          sleep(1);
          msg.val = 1;  // permisao para escrever no id_prm do
control
          control->encerramento(msg, id_prm, env);
     }
     else {
          sleep(2);
          msg.val = 0;
          msg.id = col;
          control->encerramento(msg, id_prm_conf, env);

          if(id_prm_conf != id_prm) {
              // falta do primeiro
              x = 1;
              while(id_prm != x) {

               if(x == membros_atual)
                    x = 1;
               else
                    x++;

               if(msg.id == x) {
                    sleep(4);
                    msg.val = 1;
                    control->encerramento(msg, id_prm, env);
               }
               else { // possivel erro
                    sleep(2);
                    control->encerramento(msg, id_prm, env);

                  };
              };
          };
     };

};
```

```
// meta_ciclico.cc
// You may modify this file as it will not be overwritten.
//
// This file contains the implementation of the object you
// specified.
//

#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"

//////////////////////////Nivel Meta//////////////////////////


void _im_banco::meta_ciclico(Short& meu_id){

// gerenciamento do token

     msg.id = rank + 1;

     if(msg.id == 0)
          msg.id = 1;

     cerr << "--- meu identificador: " << msg.id << endl;


     if(cont > membros_atual)// inicia cont quando ultrapassa o
          cont = 1;        // numero total de elementos no grupo

     if(msg.id == cont)     // define privilegio
          token = 1;
     else
          token = 0;

     meu_id = msg.id;

     cout << "--------------- token: " << token << endl;

     if(token == 1) {
          sleep(1);
          msg.val = 1;  //  permisao para escrever no id_prv do
control
          control->encerramento(msg, id_prv, env);
     }
     else {
          sleep(2);
          msg.val = 0;
          control->encerramento(msg, id_prv, env);
          x = cont;
```

```
                while(id_prv != x) {
                 // falta do privilegiado

                 if(x == membros_atual)
                      x = 1;
                 else
                      x++;


                 if(msg.id == x) {
                      sleep(1);
                      msg.val = 1;
                      control->encerramento(msg, id_prv, env);
                 }
                 else {
                      sleep(2);
                      control->encerramento(msg, id_prv, env);
                 };

                };
      };
      cont++;
      ULong i=0;
      contador.length(i+1);
      contador[i].cont_a = cont;

};
```

```
// meta_lid_seg.cc
// You may modify this file as it will not be overwritten.
//
// This file contains the implementation of the object you
// specified.


#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"


//////////////////////////Nivel Meta//////////////////////////


void _im_banco::meta_lid_seg(Short& meu_id){

// gerenciamento do token

     meu_id = rank + 1;

     if(meu_id == 0)
          meu_id = 1;
     cerr << "--- meu identificador: " << meu_id << endl;

     cont++;                    // contador de eventos
     if(cont == 10)
          cont = 0;

     if(meu_id == 1) {
          // sou o lider
          msg.id = cont;
          msg.val = 1;          // com permisao de escrita
          control->encerramento(msg, cont_prm, env);
     }
     else {
          // sou um dos seguidores
          sleep(2);
          msg.val = 0;          // sem permissao de escrita
          control->encerramento(msg, cont_prm, env);
          x = 1;
          while(cont_prm != cont) {
             // falha do lider
               x++;
               if(meu_id == x) {

                  msg.id = cont;
                  msg.val = 1;
                  control->encerramento(msg, cont_prm, env);
               }
```

```
                else {
                    sleep(2);
                    control->encerramento(msg, cont_prm, env);
                };
            };
        };
        ULong i=0;
        contador.length(i+1);
        contador[i].cont_a = cont;

};
```

```
// meta_pri_bac.cc
// You may modify this file as it will not be overwritten.
//
// This file contains the implementation of the object you
specified.
//

#include <orb/CORBA.h>
#include "banco.hh"
#include "banco_im.hh"

////////////////////////////Nivel Meta////////////////////////////

void _im_banco::meta_contador(short& meu_id){

        meu_id = rank + 1;

        if(meu_id == 0)
            meu_id = 1;
        cerr << "--- meu identificador: " << meu_id << endl;

        cont++;                    // contador de eventos
        if(cont == 4)
            cont = 0;

        ULong i=0;
        contador.length(i+1);
        contador[i].cont_a = cont;

};
```

```
void _im_banco::meta_primario(){

// gerenciamento do token

     sleep(1);
     msg.id = cont;
     msg.val = 1;        // com permisao de escrita
     get_state( state, done);
     control->encerramento(msg, cont_prm, state, env);
};


void _im_banco::meta_backup(short valor, short metodo, short
meu_id){

// gerenciamento do token

          sleep(2);
          msg.val = 0;        // sem permissao de escrita
          control->encerramento(msg, cont_prm, state, env);
          x = 1;

          while(cont_prm != cont) {
             // falha do primario
               x++;
               if(x == meu_id) {
                    // sou o novo primario
                    sleep(1);
                    cout <<"Novo Primario: "<< meu_id << endl;
                    switch (metodo) {
                       case 1:
                      valor_dep = valor;
                      deposito_base(valor_dep, conta_corrente,
env);
                      break;
                    case 2:
                      valor_saq = valor;
                      saque_base(valor_saq, conta_corrente,
env);
                      break;
                    case 3:
                      saldo_base(conta_corrente, env);
                      break;
                    case 4:
                      verif_dados_base(conta_corrente, env);
                      break;
                     };
                     msg.id = cont;
                     msg.val = 1;
                     get_state(state, done);
```

```
                    control->encerramento(msg, cont_prm,
state, env);
                }
                else {
                   sleep(2);
                   control->encerramento(msg, cont_prm, state,
env);

                };
          };
          set_state(state, done);
};
```

```
// controle_im.cc
// You may modify this file as it will not be overwritten.
//
// This file contains the implementation of the object you
specified.
//

#include <orb/CORBA.h>
#include "controle.hh"
#include "controle_im.hh"

void _im_controlador::controle( const mensagem& msg, Short& a,
Short& b, Environment& env ){
  // in msg, out a, out b.
  //
     first = first + 1;
     a = first;                // define colocacao

     if(a == 1) {
          id_prm = msg.id;     // id do primeiro
          cout << " ****************************** "<< endl;

     };
          cout << "-------- A ordem de chegada: " << msg.id <<
endl;
     b = id_prm;


};
```

```
void _im_controlador::encerramento( const mensagem& msg, Short&
c, Environment& env ){
  // in msg, out c.
  //
     if(msg.val == 1) {
          id_prv = msg.id;
          cout << endl;
     }
     else
          first = 0;

     c = id_prv;

     if(msg.id == id_prv)
          cout << msg.id << "|";
     else
          cout << " " << msg.id;
};

void _im_controlador::get_state(AnySeq& state, Boolean& done){
};

void _im_controlador::set_state(const AnySeq& state, Boolean
done){
};

void _im_controlador::view_change(const View& view){
};
```

Implementação de Técnicas de Replicação de Componentes de *Software* em uma Plataforma Aberta CORBA

Mestrando: Lau Cheuk Lung
Orientador: Joni da Silva Fraga
Co-orientador: Carlos Alberto Maziero

Laboratório de Controle e Microinformática - LCMI
Departamento de Engenharia Elétrica - CTC - UFSC

Roteiro da Apresentação

1. *Introdução;*
2. *Requisitos p/ Suporte a Grupo de Objetos no CORBA e ORBs disponíveis;*
3. *Técnicas de Replicação;*
4. *Modelo de Integração e Implementação;*
5. *Conclusão e Perspectivas;*

1.Introdução

- Técnicas de Tolerância a Falhas usando Replicação em Sistemas Abertos.
  - Técnicas de implementação de mecanismo para TF:
    - Suporte de execução
      - Falta de Flexibilidade
    - Bibliotecas de funções e linguagens de programação
      - Falta de Flexibilidade: transparência e separação

- Solução Adotada para Integração de Técnicas de Replicação em Sistemas Abertos:
  - Modelo Orientado a Objetos Distribuídos;
  - Padrão CORBA - Sistemas Abertos;
  - ORBs com suporte ao processamento em grupo de objetos;
  - Reflexão Computacional: o sistema pode atuar sobre si mesmo para modificar seu comportamento;
    - Proporciona a Flexibilidade -> separação e transparência desejada;

- Objetivos:
  - Levantamento de um conjunto de requisitos para o CORBA suportar a noção de grupo;
  - Avaliar as propostas de ORBs que incluem suporte para grupo;
  - Estudo de algumas técnicas de replicação;
  - Proposta de um modelo de integração para implementação de técnicas de replicação:
    - usando conceitos de reflexão computacional;

2.Suporte a Grupo de Objetos no CORBA e ORBs disponíveis

O modelo CORBA

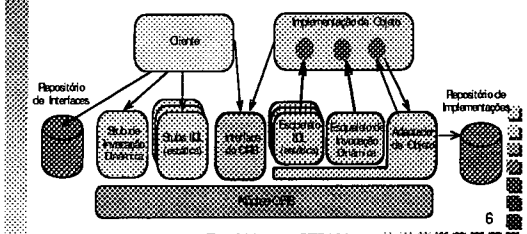


Figura 2.1 A estrutura CORBA 2.0

Modelo de Programação CORBA

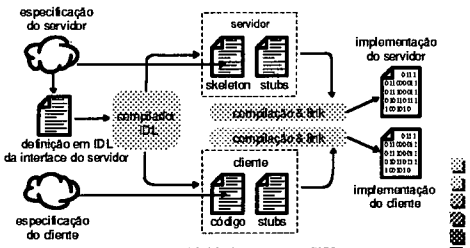


Figura 4.4 Modelo de programação CORBA

- Gerenciamento de Grupo:
  - Mecanismos para criação e ativação de grupo:
    - referência do grupo de objetos;
  - *Membership* (BOA);
    - mapas de comunicação;
  - Serviços de tratamento de faltas (ORB):
    - *timeout* e *keepalive*;
  - Serviço de transferência de estado (BOA):
    - *checkpoint* e *logs*;

- Comunicação de Grupo via ORB:
  - transmissão: primário, aleatória ou por disseminação;
  - recepção: primeiro resultado, concatenado em sequência ou votação;
- Transparência de grupo de objetos:
  - Interfaces: *Stubs*, *Skeletons*, *Proxies* e *Dispatcher*;
- Mecanismo de ordenação de mensagens
  - fifo, causal e total;

Propostas de ORBs com suporte a grupo de objetos

- ISIS/C++
  - interface para o ISIS;
  - stubs e proxies para comunicação de grupo;
- RDO/C++
  - IDL compatível ao CORBA;
- Orbix+ISIS
  - BOA para gerenciamento de grupo:
  - estilos de execução incorporados;
  - comunicação ponto-a-ponto e *multicast*;

- Electra
  - Combina o CORBA com ferramentas de baixo nível: Horus, Isis, etc. (Flexibilidade);
  - Comunicação ponto-a-ponto e *multicast* confiável;
  - Mecanismo de ordenação: fifo, causal e total;
  - BOA para gerenciamento de grupo:
    - *Membership*;
    - Transferência de estado;
  - Invocação síncrona, assíncrona e semi-síncrona;
  - Comunicação transparente e não-transparente;

- Comparações

	IDL compatível ao CORBA	ORB compatível acima/abaixo CORBA	Suporte a grupo e nível CORBA	BOA para gerenciamento de grupo	Opera sobre vários suportes
Electra	sim	não	sim	não	sim
Orbix+ISIS	sim	não	sim	não	não
RDO/C++	sim	sim(?)	sim	não	não
ISIS/C++	não	não	não	não	não

- Outras propostas:
  - *ANSA*;
  - *Group Server* (OMG);

3. Técnicas de Replicação

- Em sistemas distribuídos:
  - melhora da confiabilidade;
  - aumento da disponibilidade;
  - melhor desempenho do sistema;
- Protocolos de coordenação:
  - consistência de estado;
  - transparência;
  - Abordagens:
    - réplicas ativas;
    - réplicas passivas;
    - réplicas semi-ativas;

- Replicação Ativa:
  - determinismo de réplica -> acordo e ordenação;
- Replicação Passiva
  - primário/*backup*;
  - transferência de estado (*checkpoint*);
- Replicação Semi-ativa:
  - líder/seguidores;
  - acordo e ordenação -> réplica líder;

Comparação das abordagens

Técnicas de replicação	*Overhead* em situação de falha	Determinismo de réplica	Falhas arbitrárias
Ativas	baixíssimo	necessário	tolera
Passivas	alto	desnecessário	não tolera
Semi-ativas	baixo	necessário	não tolera

Quadro comparativo das três abordagens

Replicação Ativa Competitiva

- Coordenação da técnica é distribuída

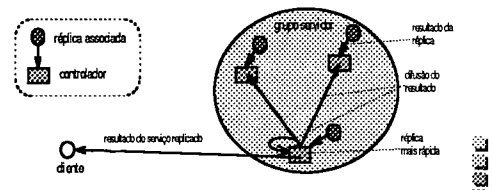


Figura 3.8. Modelo de Replicação Competitiva para Falhas de *crash*.

4. Modelo de integração e a implementação

- Simplificar a implementação de técnicas de replicação em sistemas abertos usando conceitos de reflexão computacional.
- Reflexão Computacional

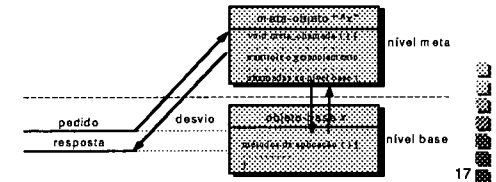


Figura 4.1. Reflexão computacional

Estrutura Reflexiva para o Modelo

- Separação dos aspectos não-funcionais e funcionais da aplicação;

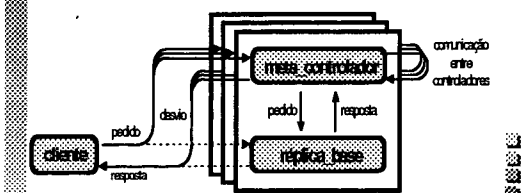


Figura 4.2. Estrutura reflexiva para modelos de replicação

Integração do Modelo Reflexivo ao CORBA

- As técnicas de replicação são introduzidas a nível do modelo de programação usando os conceitos de reflexão para integrar no ambiente CORBA.

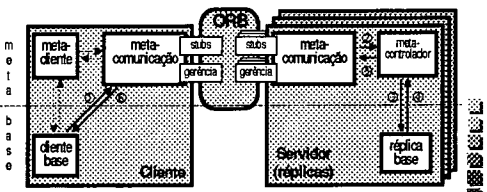


Figura 4.3. Estrutura do modelo sobre um suporte CORBA.


```
// IDL
interface meta_controlador_1
{
  // Descrição dos tipos de dados empregados

  // Descrição dos métodos do servidor
  boolean meta_método_1 (parâmetros);
  ...
  boolean meta_método_n (parâmetros);
};
interface meta_controlador_2
{
  // Descrição dos métodos do meta controlador
  boolean difunde_id (in int id);
  boolean encerramento ( );
};
```

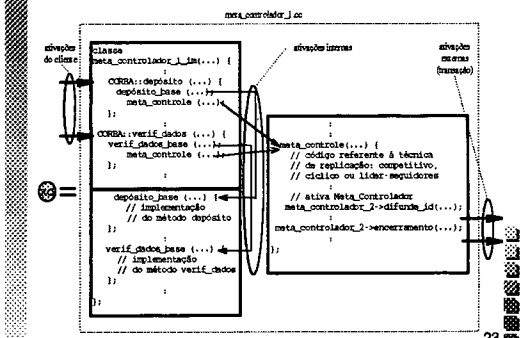
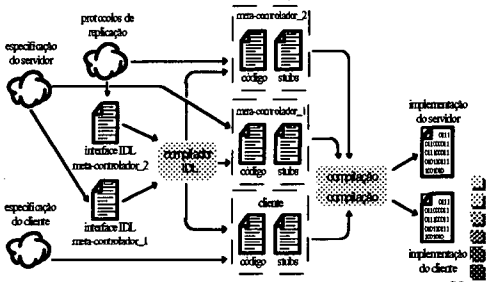



Figura 5.6 Formato da separação do código

Processo de Construção da Aplicação




```
class meta_controlador {
  // declaração de variáveis
  method meta_método_1 (parâmetros) {
    método_base_1 (parâmetros);
    meta_controle (parâmetros);
  };
  ... // declaração dos demais meta-métodos
// implementação do meta-controle
method meta_controle (parâmetros) {
  ;
  while not encerrou do
    if (primeiro = null) then
      grupo.difunde_id (meu_id);
    end ;
    if (primeiro = meu_id) then
      // primeira réplica a responder
      return ; // retorna resposta ao cliente
    else
      // not encerrou then
      grupo.encerramento() ;
    end ;
  end ;
}
method encerramento () {
  if (primeiro E membership) then
    encerrou := true ;
  end ;
  primeiro := null ;
}
```


Diagrama temporal

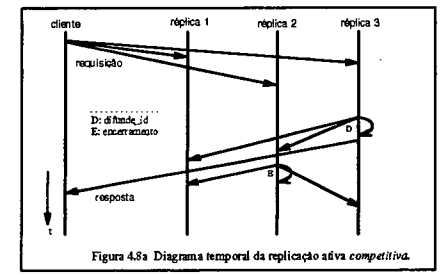


Figura 4.8a Diagrama temporal da replicação ativa *competitiva*.

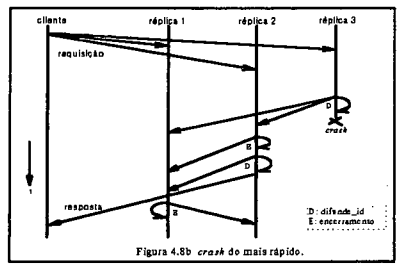


Figura 4.8b *crash* do mais rápido.

- Mecanismos de reintegração de réplicas;
- Medidas de recuperação do grau de replicação:
  - Primitivas do suporte:
    - *Membership*
      - primitivas do suporte: view_change(view), join(group) e leave(group);
    - Transferência de estado:
      - set_state(state) e get_state(state);

- As mudanças necessárias limitam-se à interface IDL dos meta-controladores e aos seus códigos;
- A implementação faz uso intensivo do suporte Electra, simplificando as necessidades de coordenação na técnica implementada;
- O uso de uma plataforma Electra/CORBA permite a implementação de aplicações sobre um suporte de execução heterogêneo (grupo de máquinas SunOS 4.X e Solaris, interligadas por uma rede local), facilitando os aspectos de interoperabilidade;

Dessincronização entre réplicas

- A técnica de execução periódica *rendez-vous global não é usada por suas implicações de custo e desempenho do conjunto;*
- Solução adotada:
  - Uso do conceito de *sincronismo virtual* garantido pelo suporte de mais baixo nível;
  - Quando o *buffer* de entrada do par meta_controlador/réplica_base mais lenta atinge sua capacidade limite o suporte retira a réplica do *membership;*

Considerações sobre as Implementações

- Implementadas as técnicas competitiva, cíclica, primário/secundário e de líder/seguidores;
- Estas técnicas foram testadas em pequenos exemplos de aplicação com simulação de falta de *crash*;
- A estrutura de integração proposta se mostrou bastante flexível, podendo comportar facilmente outros modelos de replicação.

Comparações com outras referências

- [Fabre 95]
  - Usa uma arquitetura R2 -> ponto-a-ponto;
  - Implementações feitas em Open/C++;
  - Não são esclarecidos os mecanismos de detecção de falhas;
- [Lisbôa 96]
  - Especificações para tolerância a falta de *software*
    - N-versões e bloco de recuperação;

Tolerância a Falhas

5.Conclusão e Perspectivas

- Estudo de requisitos para suporte a grupos e ORBs:
  - implementação de técnicas de replicação;
  - Orbix+ISIS;
- CORBA -> sistema aberto;
- A reflexão computacional permite a independência dos códigos das réplicas em relação aos protocolos de coordenação, o que é altamente desejável em *sistemas abertos*;

Tolerância a Falhas

- Flexibilidade no sistema: mudar de técnica ou alterá-la para atender a graus de tolerância a falhas desejados podem resultar em simplesmente trocar os protocolos de coordenação a nível meta;
- Viabilidade;

- ASAP -> suporte de grupo e tolerância a falta;

Tolerância a Falhas

Perspectivas

- Realização de medidas detalhadas sobre o desempenho das técnicas de replicação no Electra;
- Implementação do modelo de integração sobre o *Orbix+Isis*;
- Testes com técnicas de replicação com suposição de faltas menos restritivas;
- Mapeamento de IDL para uma linguagem reflexiva (Open/C++, Java RTR, etc.);
- Testes com aplicações mais complexas;