Universidade Federal da Bahia
Escola Politécnica / Instituto de Matemática
Programa de Pós-Graduação em Mecatrônica

SEMÍRAMIS RIBEIRO DE ASSIS

REPLICAÇÃO RECONFIGURÁVEL EM SISTEMAS DE TEMPO REAL

DISSERTAÇÃO DE MESTRADO

Salvador
2010

SEMÍRAMIS RIBEIRO DE ASSIS

REPLICAÇÃO RECONFIGURÁVEL EM SISTEMAS DE TEMPO REAL

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica da Escola Politécnica e do Instituto de Matemática, Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre.

Orientador: *Prof. Dr. Raimundo José de Araújo Macêdo*
Co-orientador: *Prof. Dr. Sérgio Gorender*

Salvador
2010

Sistemas de Bibliotecas - UFBA

Assis, Semíramis Ribeiro de.
Replicação reconfigurável em sistemas de tempo real / Semíramis Ribeiro de Assis. - 2010.
241 f. : il.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.
Co-orientador: Prof. Dr. Sérgio Gorender.
Dissertação (mestrado) - Universidade Federal da Bahia, Instituto de Matemática e Escola Politécnica, Salvador, 2010.

1. Tolerância a falha (Computação). 2. Arquitetura de computadores. 3. Sistemas operacionais distribuídos (Computadores). 4. Controle em tempo real. I. Macêdo, Raimundo José de Araújo. II. Gorender, Sérgio. III. Universidade Federal da Bahia. Instituto de Matemática. IV. Universidade Federal da Bahia. Escola Politécnica. V. Título.

CDD - 004.33
CDU - 681.3.014

TERMO DE APROVAÇÃO

SEMÍRAMIS RIBEIRO DE ASSIS

REPLICAÇÃO RECONFIGURÁVEL EM SISTEMAS DE TEMPO REAL

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia - UFBA, pela seguinte banca examinadora:

Prof. Dr. Raimundo José de Araújo Macêdo (Orientador)
Doutor em Ciência da Computação, University of Newcastle Upon Tyne, Inglaterra.
Professor da Universidade Federal da Bahia

Prof. Dr. Flávio Morais Assis Silva (Examinador PPGM)
Doutor em Ciência da Computação, Technische Universitat Berlin, Alemanha.
Professor da Universidade Federal da Bahia

Prof. Dr. Paulo Roberto Freire Cunha (Examinador Externo)
Doutor em Ciência da Computação, University of Waterloo, Canadá
Professor da Universidade Federal de Pernambuco

Salvador, 17 de Junho de 2010.

À meus pais, Hercílio Rodrigues de Assis (in memorian) e Albertina Maria Ribeiro de Assis, por todo apoio, confiança e dedicação.
A todos que acreditaram neste projeto de pesquisa.

AGRADECIMENTOS

Agradeço primeiramente à Deus, por ter me concedido a vida e condições para que eu pudesse concretizar mais esta vitória. Aos meus Orixás, por estarem sempre presentes me sustentando.

A meu pai, Hercílio Rodrigues de Assis (*in memorian*), por ter sempre me apoiado e me ensinado a ter o interesse pela área de mecatrônica e informática. A minha mãe, Albertina Maria Ribeiro de Assis, por estar sempre presente em todos os momentos de minha vida, e ter sempre me apoiado no que precisei.

A Raimundo José de Araújo Macedo, meu orientador, por ter me incentivado a buscar as soluções quando parecia não ter mais caminhos a seguir. E pelas várias frases "Força, siga em frente!!!". A persistência é tudo!!!

A Sérgio Gorender, meu co-orientador, por todos os debates, clareamentos de idéias, reuniões semanais e por toda ajuda prestada! Professor, realmente valeu!!! O desafio foi grande, mas valeu a pena!!!

A todas as pessoas que, em algum momento desta caminhada no mestrado, me ajudaram de alguma forma:

Aos colegas que participaram da longa jornada com final feliz em Sistemas Mecatrônicos;

A Ednaldo Ferreira Marques, por todo o apoio como revisor de texto, engenheiro eletricista e incentivador para uma mais breve conclusão deste projeto.

A Sandro Santos Andrade, por ter me ajudado com a configuração do ambiente do ARCOS, CIAO, TAO e ACE.

A Fabricio Mota Oliveira, por ter me dado uma importante dica para seguir o mestrado. Cara, valeu!!! Sou eternamente grata por isso!!!!!

A Alírio Sá, por todas as dicas dadas no inicio deste projeto, pelas conversas de madrugada no Gtalk para esclarecer as coisas.

A Ricardo Perrone da Silva, por ter sido uma peça chave na ajuda com o CIAO, TAO e com o próprio Linux!!! Pelas conversas, e-mails, debates.... Valeu mesmo, você é 10!!!!

Aos meus colegas da PRODEB pelo auxílio, compreensão e todo apoio para realização dos testes finais. Agredecimento especial para Fabrício Reis (vulgo Barriga) por toda ajuda para a migração do ARCOS e GesRep da máquina virtual para a máquina física, Eduardo Alex (vulgo Duca) pelas dicas e ajuda com a migração do projeto, Cristiana Sousa pelo empréstimo do cabo de rede e pela compreensão com os testes, Pedro Paulo de Melo Ruas pelo empréstimo do CD da última versão do Ubuntu e pela cópia feita, a Lívia Barreiros pela dica dada para o projeto, a Caterine Rendall Gonçalves por sempre me dar todo apoio, a Marília Pedreira pela compreensão quando precisei sair mais cedo. Gente, muito obrigada a todos!!

A todas as pessoas que trabalham ou trabalharam no LaSid, por terem sempre me ajudado de alguma forma. Agradecimentos especiais a Socorro, Rebeca, Cynthia, Carmem (por todo apoio na fase final do mestrado)e Carol, por facilitarem as coisas no decorrer do mestrado.

A todas as pessoas que trabalharam ou trabalham na Politécnica, por toda ajuda prestada. Agradecimentos especiais a Lúcia, por toda ajuda!

Aos meus amigos Laisa Sales Calmon, Tiago Roblêdo Damasceno (Spy), Cycero Tavares, Bárbara (Bia), Milena Sampaio Guerra, Luiza Abreu e Jeane Franco (Jeo) que entenderam o meu período de greve para festas, *shows*, aniversários... Gente, acabou!!!!!!!!!

A minha amiga Maiara Alves Bomfim por toda ajuda quando precisei em Sistemas Mecatrônicos.

Ao meu ex-chefe da VIVO, Jean Claude Santana, por ter me liberado quando eu precisei para a matéria de Sistemas Mecatrônicos. Realmente valeu a pena!!!!

Ao pessoal da CPM-Braxis, que me ajudaram nesta caminhada:

A Wagner Lima, por todas as nossas conversas e debates sobre meu projeto, que sempre me davam alguma idéia para a solução! Wagner, valeu!!!

A Cyro Salvatore, pela disponibilização do instalador da máquina virtual e pelas dicas. Isso foi essencial!

A Ivone Souza, Débora Costa e Rogério Lacerda por terem me compreendido quando precisei sair mais cedo do trabalho para ir à UFBA.

Às minhas gatinhas Ísis, Miu e Tina, pelos momentos de carinho e companhia nas longas noites programando.

A todos, o meu MUITO OBRIGADA!!!!!!!!!

Quem conduz e arrasta o mundo não são as máquinas, mas as idéias.

—VICTOR HUGO

RESUMO

A automatização do processo de produção de uma fábrica necessita de sistemas computacionais mais robustos, disponíveis e confiáveis. Uma falha em um dos *softwares* responsáveis pelo controle da planta de produção pode ocasionar graves prejuízos financeiros e perda de matéria prima. A inclusão de técnicas de tolerância a falhas nos sistemas computacionais se torna, então, algo fundamental para evitar maiores danos às indústrias na ocorrência de falhas nos sistemas citados.

A redundância de *software*, por meio da replicação, tem por objetivo aumentar a disponibilidade de partes ou de todo um *software* através da distribuição e gerenciamento das partes replicadas em diversas máquinas da rede, garantindo que a aplicação principal não sofra uma parada devido a um único ponto falho.

O trabalho proposto tem por objetivo o desenvolvimento e validação de um Gestor de Réplicas, denominado GesRep, que tem a finalidade de prover o serviço de replicação de componentes e o gerenciamento das réplicas criadas, através da técnica de replicação passiva.

As aplicações passíveis de utilizarem os serviços do Gestor são aquelas construídas com base na arquitetura de componentes, utilizando o *framework Component Integrated* ACE ORB (CIAO), baseado no padrão de comunicação distribuído CORBA.

Uma aplicação construída com base na arquitetura de componentes tem suas funcionalidades divididas em módulos distintos, também chamados componentes, o que facilita o reuso e incorporação de novas funcionalidades. A replicação, neste tipo de aplicação, é realizada pela duplicação dos módulos (componentes) essenciais para que, em caso de falhas, a aplicação se mantenha em funcionamento.

Aplicações de controle e supervisão são críticas por estarem diretamente ligadas ao

processo produtivo de uma fábrica, de modo que a ocorrência de falhas de *software* pode acarretar sérios danos a todo o processo produtivo. O *framework Architecture for Control and Supervision* (ARCOS), voltado para construção de aplicações do domínio supracitado e construído sobre a arquitetura de componentes, não possui nenhum serviço de tolerância a falhas para as aplicações construídas sobre sua estrutura. Deste modo, o Gestor de Réplicas proposto terá o objetivo de suprir esta necessidade, aumentando o nível de disponibilidade e, consequentemente, de confiabilidade das aplicações.

Os experimentos mostraram que o *overhead* de recuperação do sistema após a ocorrência de uma falha possui tempo satisfatório, não interferindo significativamente na performance de aplicações *soft-realtime* pertencentes ao domínio de controle e supervisão e reduzindo a parada do sistema por *crash*. Todo o processo de criação, monitoramento e recuperação das réplicas é feito de forma transparente ao usuário da aplicação.

Palavras-chave: Componentes, Replicação, Tolerância a falhas

ABSTRACT

Industrial production process automation requires increasingly more robust, available and reliable computational systems. A production process control software fault can cause serious financial damage and raw material loss. Thus, there is a need to include fault tolerance techniques in order to prevent such damage to industries caused by fault occurrences in the systems.

Software redundancy, achieved by the replication technique, increases the availability of parts or all of one application through the management and distribution of the replicated parts in various nodes on the network, guaranteeing that execution of the main application will not be suspended due to a single faulty host.

The purpose of this work is to design, implement and validate a Replica Management System, known as GesRep, which creates and manages component replicas using the passive replication technique.

Applications ready to use GesRep services are created on top of the component architecture, specifically based on the Component Integrated ACE ORB (CIAO) framework.

An application developed which is based on the component architecture has its functionalities distributed on different modules, e.g. components, facilitating code reuse and integration of new functionalities. Replication in this kind of application duplicates one or more essential components so that in case of failure of one of these the application continues to function.

Supervision and control applications are critical because of their close relationship with the production process, therefore a software fault occurrence can cause serious damage to the whole process. The Architecture For Control And Supervision (ARCOS) framework, designed to develop supervision and control applications based on the com-

ponent architecture, do not provide any fault tolerance mechanism to applications based on it. Thus, the proposed Replica Manager addresses this, increasing the level of availability and consequently the reliability of the applications.

Experiments showed a satisfactory recovery time after a component failure, with no significant interference in soft-real-time performance of the application and reducing crash failures in applications using GesRep. The whole creation, monitoring and recovery process of the replicas is carried out with no human intervention.

Keywords: Components, Replication, Fault Tolerance

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Sistemas Computacionais estão sujeitos a falhas de algum tipo. Em alguns casos, entretanto, condições críticas no emprego destes sistemas podem requerer que falhas ocorram com uma menor freqüência.

Na área de Tolerância a Falhas, a palavra chave que melhor descreve o aumento da confiabilidade de sistemas sujeitos a falhas é redundância. Dois tipos de redundância são bastante conhecidos e empregados nestes sistemas: temporal e espacial. Na temporal, procura-se repetir operações que falhem por motivos transientes; na espacial, visa-se dispor de um número maior dos componentes que estejam sujeitos a falhas.

A replicação é reconhecida como um caso típico de redundância espacial. No contexto dos sistemas distribuídos e tolerantes a falhas, replicar consiste em dispor de um número plural de componentes no sistema, em operação parcial ou total, e em comunicação entre si. Trata-se de uma técnica largamente empregada no desenvolvimento de sistemas tolerantes a falhas [41].

De acordo com [13], a replicação de objetos é uma combinação entre replicação de dados e de execução. Modelos de replicação de objetos são resultantes da combinação dos modelos de replicação de dados com os de execução. Os modelos de replicação de dados são focados em consistência e disponibilidade dos estados dos objetos, enquanto que os de execução se concentram em tolerância a falha das aplicações. Modelos de replicação de execução podem ser classificados como Replicação Passiva e Replicação Ativa. Outros modelos híbridos, entretanto, são também concebidos por outros autores, como pode ser visto mais adiante. Os modelos de replicação de dados podem ainda ser classificados como:

- Pessimistas, ou de consistência forte (*strong consistency*), que requerem consistência

imediata nas réplicas;

- De Inconsistência Controlada, onde a consistência das outras réplicas não precisa ser imediata;
- Otimistas, onde não há garantia temporal de consistência das réplicas.

São dois os requisitos para métodos de replicação para sistemas com requisitos temporais: determinismo de réplica e obedecer a requisitos temporais. Um dos métodos utilizado para este tipo de sistema é evitar que os recursos do sistema sejam esgotados (*exhaustion-safe*), ocasionando uma maior disponibilidade do serviço, e , com isso, diminuindo a probabilidade de falha geral no sistema [44].

Uma segunda técnica de replicação em tempo-real é a Replicação Proativa (*Proactive replication*), na qual, por meio de algumas informações do sistema, é possível determinar quando será feita a mudança de controle para uma nova réplica, ou quando será executada uma rotina de manutenção do sistema [34].

Um componente de *software* pode ser definido como um conjunto de objetos (classes) encapsulados, com o objetivo de prover um serviço bem definido, especificado em interfaces claras. Possui portas de entrada e saída, que indicam os serviços requisitados e fornecidos, respectivamente. Através destas portas, é possível uma conexão com outros componentes, de modo a montar uma aplicação completa [23].

Replicar um componente representa não só fazer uma cópia de suas classes, como também replicar suas conexões com outros componentes que compõem a aplicação. As políticas utilizadas para gerenciamento da replicação de componentes são, basicamente, os mesmos que os utilizados para replicação de objetos, vistas nos parágrafos anteriores neste capítulo.

As aplicações voltadas para controle e supervisão têm por finalidade a integração entre equipamentos eletro-mecânicos e soluções baseadas em computadores, como sensores e atuadores para coleta de dados e ativação de dispositivos, braços mecânicos para realizar determinadas funções, etc. Por esta característica de integração, este domínio de

aplicações é considerado crítico em sua disponibilidade, já que uma falha no sistema pode ocasionar a interrupção de toda a linha de produção da fábrica [2].

Considerando a criticidade acima citada, o presente trabalho tem por meta final aumentar a disponibilidade e confiabilidade de aplicações do domínio de controle e supervisão, reduzindo possíveis perdas financeiras ou de matéria-prima para a fábrica, através da técnica de replicação de componentes de *software*. O objetivo deste trabalho ora apresentado é especificar, implementar e validar um gerenciador de réplicas reconfigurável voltado para componentes de *software*, de modo que o processo de validação deste seja realizado sobre aplicações mecatrônicas do domínio de controle e supervisão. A reconfiguração se dá de modo a permitir que o número mínimo de réplicas em um grupo possa variar para mais ou para menos.

Alguns *frameworks* voltados para aplicações de controle e supervisão já foram desenvolvidos, como, por exemplo, o ARCOS [1], ESPRIT *OpenDreams* [24], um focado em controle de plantas pela Internet utilizando Java [7], dentre outros. Dentre os *frameworks* citados, após uma análise, foi visto que nenhum utiliza o serviço de replicação para componentes de *software*, podendo prejudicar a confiabilidade das aplicações que deles o tiverem como base para seu desenvolvimento.

Uma aplicação construída tendo por base a arquitetura de componentes de *software* possui o foco em reuso e em facilidade para adição de novas funcionalidades, sem que haja necessidade de alteração de toda a aplicação. Isso se deve ao fato que os componentes que compõem a aplicação são independentes, sendo interconectados através de suas portas de entrada e saída.

Para que um componente seja utilizado por uma aplicação, basta que esta satisfaça os requisitos necessários pelo componente (serviços requeridos) e se adeque aos serviços providos por este, disponíveis através da interface disponibilizada por este componente. O processo de adição de novo componente a uma aplicação já construída pode, desta forma, ser realizado sem que toda a aplicação seja alterada, o que facilita para o desenvolvedor da aplicação.

O Gestor de Réplicas proposto foi desenvolvido com base na arquitetura de componentes, atendendo aos requisitos de criação e gerenciamento das réplicas de componentes de uma aplicação, incluindo monitoramento destas réplicas e tomada de decisão em caso de falhas das mesmas. Para validação do funcionamento do Gestor, foi utilizado o *framework Architecture For Control and Supervision* (ARCOS), voltado para construção de aplicações mecatrônicas do domínio de controle e supervisão. Buscou-se, desta forma, avaliar o comportamento do Gestor em caso de falhas de réplicas do sistema, mostrando sua aplicabilidade à aplicações *soft-realtime* de controle e supervisão.

Esta dissertação se divide da seguinte forma: no capítulo 2 serão apresentados os conceitos de componentes e de uma arquitetura baseada em componentes, além de uma breve apresentação do *framework* ARCOS; o capítulo 3 traz o conceito de modelos de falhas, tipos de replicação e trabalhos correlatos; o capítulo 4 traz a apresentação do Gestor de Réplicas, sua infraestrutura, arquitetura e os resultados obtidos; e, finalmente, o capítulo 5 apresenta as conclusões e os trabalhos futuros.

CAPÍTULO 2

ARQUITETURAS BASEADAS EM COMPONENTES

Neste capítulo, serão apresentados os conceitos de componentes e de arquitetura baseada em componentes.

2.1 COMPONENTE

Um componente se caracteriza por um conjunto de objetos encapsulados capazes de prover um serviço bem definido. Possui interfaces claras que definem quais os serviços fornecidos (*provided*) e quais os requeridos (*required*), possibilitando o seu acoplamento a outros componentes, com a finalidade de montar uma aplicação [23, 17].

Os serviços que são utilizados por um componente são explicitados nas interfaces do tipo *required*, enquanto que os serviços fornecidos pelo componente são conhecidos nas interfaces *provided*.Com a utilização deste conceito de construção, é possível que sejam adicionadas ou removidas funcionalidades, sem ter a necessidade de reconstrução de toda a aplicação [23, 30].

Um componente pertence a um determinado modelo de componentes, tendo este último um *framework* de componentes, que irá ditar quais os serviços que serão implementados para dar a base ao modelo citado [30].

É ilustrado na figura 2.1 o relacionamento entre um componente, o modelo ao qual o mesmo pertence, e o *framework* de componentes, base pela qual os serviços que serão providos pelo componente final serão selecionados. A figura mostra as interfaces que estarão presentes no componente e servirão para sua utilização e conexão com as aplicações que desejarem fazer uso deste. A conexão entre dois componentes se faz através da satisfação de um contrato para utilização (interface que satisfaz contrato).

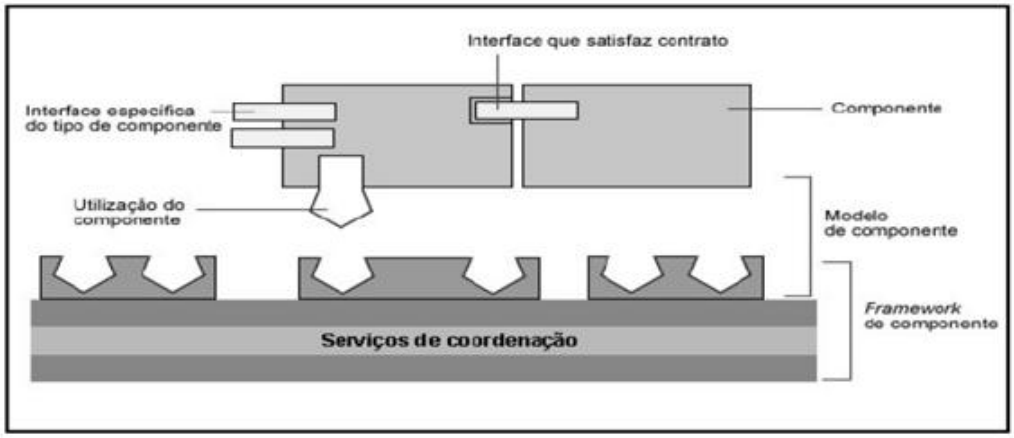

Figura 2.1. Relacionamento entre Componente, modelo de Componente e *Framework* de Componente [30].

Para que uma aplicação seja construída com base em componentes, alguns pontos devem ser observados, concordando com [30], que são:

- Encontrar componentes capazes de prover o serviço necessitado;
- Selecionar componentes que atendam os requisitos da aplicação;
- Adaptar os componentes selecionados à aplicação;
- Acoplar os componentes previamente adaptados à aplicação através da composição;
- Substituir versões antigas dos componentes da aplicação por versões atualizadas.

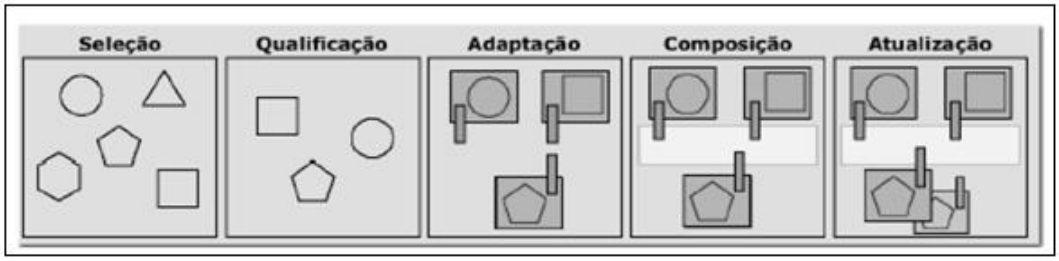

Figura 2.2. Desenvolvimento de aplicações utilizando componentes [30].

A figura 2.2 ilustra os passos necessários para o desenvolvimento de uma aplicação baseada em componentes, seguindo as fases acima descritas. Primeiro, selecionam-se os

componentes prováveis de atenderem às expectativas da aplicação. Em seguida, qualifica-se os componentes previamente escolhidos, separando aqueles que irão efetivamente compor o *software* desejado. O próximo passo será adaptar os componentes à aplicação, de acordo com os serviços que são providos pelos mesmos. Vem, então, a fase da composição entre os novos componentes adicionados e os já existentes. Por último, é feita a atualização dos componentes, onde versões mais novas substituem as mais antigas, dentro da aplicação.

2.2 ARQUITETURA BASEADA EM COMPONENTES

Um modelo de componentes é fundamentado numa arquitetura baseada em componentes, e possui regras e padrões definidos para a comunicação entre componentes de um mesmo domínio [30].

Ao referenciar arquiteturas baseadas em componentes, podem-se citar algumas importantes, que serão detalhadas neste capítulo, como: CORBA *Component Model* (CCM) [30] e *Enterprise Java Beans* (EJB) [25].

2.2.1 CORBA *Component Model* (CCM)

2.2.1.1 *Commom Object Request Broker Architecture* (CORBA)

CORBA é uma arquitetura voltada para a comunicação entre objetos distribuídos, de modo que estes objetos podem estar implementados em diferentes linguagens de programação e se situarem em diferentes plataformas [36].

Para permitir que esta comunicação seja efetuada, CORBA possui em sua arquitetura interna uma camada de nome *Object Request Broker* (ORB), sendo esta responsável por receber e processar as requisições vindas dos vários clientes. Assim, o ORB recebe a requisição, procura pela implementação correspondente e realiza a operação desejada sobre o objeto requerido [30].

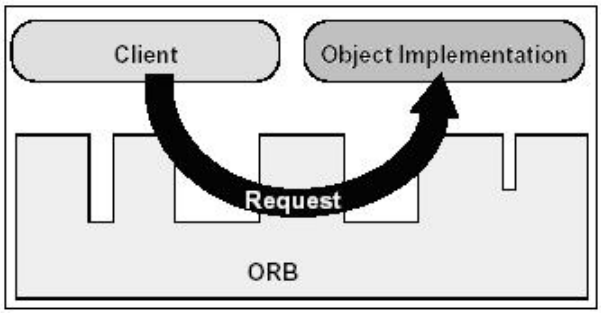

Figura 2.3. A Comunicação no CORBA [23].

A figura 2.3 ilustra a comunicação explicada no parágrafo anterior sobre o CORBA. Uma requisição de um cliente é primeiramente interceptada pelo ORB, que a repassa ao objeto responsável pelo seu processamento, recebendo a resposta deste e a enviando ao cliente.

A camada ORB é definida pelas suas interfaces, as quais irão definir as operações disponíveis, as específicas para um determinado tipo de objeto e as específicas para um estilo particular de implementação do objeto. Estas interfaces são definidas através da linguagem *Object Management Group Interface Definition Language* (OMG IDL), tendo como propósito definir os métodos e seus atributos necessários numa invocação feita por um objeto [30].

O conceito de cliente para os objetos CORBA é o mesmo conceito existente em uma arquitetura cliente-servidor, ou seja, a parte responsável por gerar requisições para um determinado objeto. Em CORBA, um cliente só terá acesso às assinatura dos métodos dos objetos por ele acessados. É importante citar que um determinado cliente só o é para um determinado objeto, de modo que a implementação de um objeto pode vir a ser cliente para outro objeto [30].

A implementação de um objeto irá determinar, além do código para seus métodos, os dados que serão retornados em sua instanciação. Para definir seu comportamento, outros objetos podem ser utilizados no desenvolvimento de um objeto específico. Sua codificação é independente da maneira como será feita sua invocação, assim como também não depende da implementação do ORB que o gerenciará [30].

2.2.1.2 Arquitetura CORBA

As estruturas necessárias na parte cliente unidas às necessárias na parte servidora, constituem a arquitetura do CORBA. O ORB é o responsável pela intermediação da comunicação entre cliente e servidor, repassando as requisições recebidas aos objetos apropriados para atendê-las, garantindo assim a transparência de localização do servidor [39].

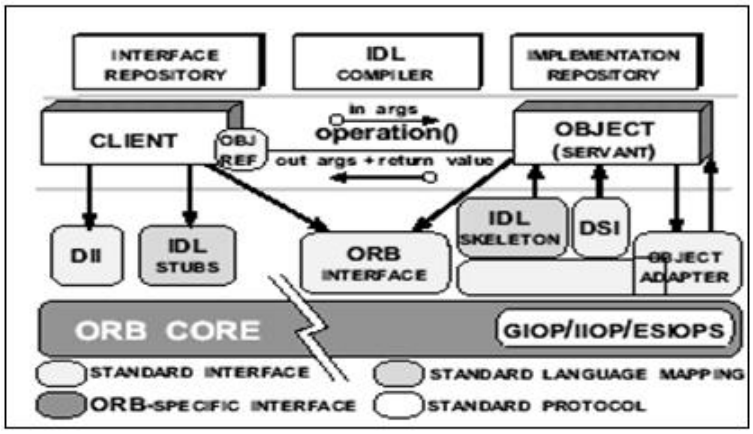

Figura 2.4. Arquitetura do CORBA [39].

A estrutura necessária para que esta comunicação ocorra é ilustrada na figura 2.4, e descrita a seguir, segundo [39]:

- **O *Stub* IDL do cliente** - Responsável pela interface estática para os serviços dos objetos. Dita como serão invocados os serviços no servidor pelo cliente. O cliente age como se fosse uma chamada local. Os serviços disponíveis são definidos através da interface IDL, da qual são geradas os *stubs* cliente e servidor. Deve-se ter um *stub* IDL para cada interface acessada pelo cliente. O código presente no *stub* serve apenas para realizar o *marshaling* (empacotamento de parâmetros necessários para a chamada de uma função) e o *unmarshaling* (desempacotamento dos parâmetros e do resultado da chamada);

- **Interface de invocação dinâmica (DII)** - Permite que métodos sejam conhecidos em tempo de execução. O CORBA possui algumas *Application Programming*

Interface's (APIs) responsáveis pela busca no servidor destes métodos, gerando seus parâmetros, efetivando as invocações a estes e retornando o resultado;

- **API's repositórias de interface** - Permite que a descrição de componentes de interface registrados seja modificada ou conhecida, assim como os métodos suportados pela interface e os parâmetros requeridos. Estas descrições são conhecidas como assinaturas de métodos no CORBA;

- **A interface ORB** - São APIs que referenciam serviços locais que podem ser acessados por uma determinada aplicação, como por exemplo, os serviços de conversão de caracter para *string* e vice-versa.

O lado servidor de uma aplicação CORBA possui a seguinte estrutura, segundo [39]:

- **O *Stub* IDL servidor** - Também conhecido como *skeletons*, responsáveis pela interface estática para cada serviço mantido pelo servidor. Assim como no lado cliente, este *stub* é gerado pelo compilador IDL;

- **A interface *skeleton* dinâmica (DSI)** - Responsável pela ligação dinâmica nos servidores que suportam chamadas a procedimentos de componentes que não possuem *skeletons* IDL (ou stubs). O DSI procura o valor do parâmetro na requisição recebida para identificar para qual método esta deverá ser repassada. Estes *skeletons* dinâmicos são úteis na interligação entre diferentes ORBs. O DSI é equivalente ao DII no lado cliente;

- **O adaptador de Objetos (*Object Adapter*)** - Situa-se no topo da comunicação ORB e tem por objetivo receber requisições de serviços. Provê o ambiente necessário para a instância de objetos do servidor, encaminhando as requisições a eles, além de lhes atribuir uma identificação de objeto (*Object ID*). As classes suportadas por um servidor são registradas através do adaptador de objetos no Repositório de Implementação (*Implementation Repository*). É possível que servidores possuam mais de um adaptador de objetos;

- **Repositório de implementação** (*Implementation Repository*) - Repositório *online* de classes, objetos instanciados e seus respectivos identificadores de um determinado servidor;
- **A interface ORB** - Assim como na parte cliente, esta interface é composta de APIs para serviços locais.

2.2.1.3 Especificação do CORBA *Component Model* (CCM)

O modelo de objetos do CORBA possui algumas limitações que, de acordo com [30], são:

- Não existência de um padrão para desenvolvimento e empacotamento de objetos;
- Falta de um mecanismo capaz de estender objetos CORBA, além da herança;
- Falta de definição de serviços obrigatórios.

As especificações CORBA não definem um método de desenvolvimento de objetos, o que ocasiona o primeiro item das limitações expostas acima. Para um projeto com uma grande quantidade de objetos, a existência de um padrão de empacotamento e distribuição destes é importante [30].

O segundo item da lista de limitações se dá devido à não possibilidade de criar uma nova interface sem ter que herdar de todas as outras já existentes [30].

Ainda de acordo com [30], é necessário que haja serviços de implementação obrigatória em um objeto distribuído para que se tenha um padrão bem definido na estrutura interna de um objeto. Isso justifica o terceiro item da lista de limitações.

Com o intuito de suprir estas limitações existentes no modelo do CORBA, foi desenvolvido o Modelo de Componentes do CORBA (CCM). Este modelo surge como uma forma de padronizar a criação de componentes, utilizando o CORBA como base (*middleware*). Os componentes, assim, são desenvolvidos através da definição de suas interfaces,

sendo implementados, em seguida, com o uso de ferramentas que dêem suporte ao CCM. O produto deste desenvolvimento é uma biblioteca que pode ser compartilhada (através de um arquivo .JAR ou .DLL), e que é executada com o auxílio de um servidor de componentes [30].

É possível desenvolver dois tipos de componentes através do CCM: básico e estendido. O básico oferece poucos mecanismos de conexão com outros componentes em relação ao tipo estendido, sendo exemplos das portas de comunicação oferecidas pelo segundo tipo [30]:

- **Faceta** - Representa os serviços providos pelo componente;
- **Receptáculo** - Representa os serviços requeridos pelo componente;
- **Produtor ou fonte de eventos** - Emitem (publicam) eventos para quem possa interessar ou para um canal de eventos;
- **Receptor ou consumidor de eventos** - Pontos através dos quais eventos podem ser recebidos;
- **Atributos** - Valores que facilitam a configuração do componente.

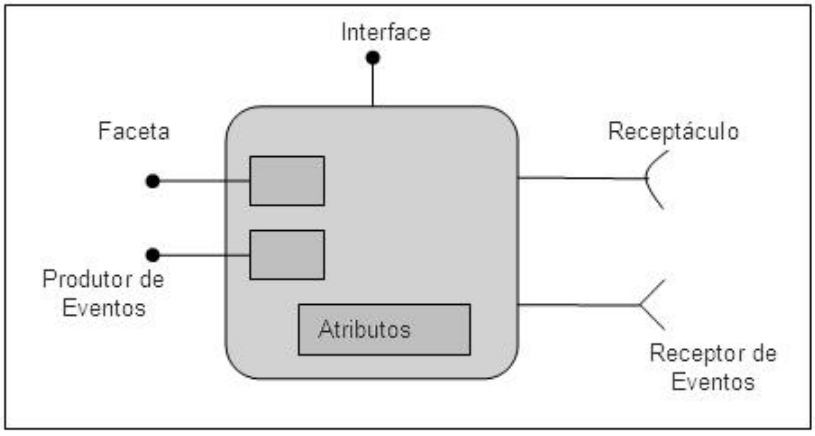

Figura 2.5. Exemplo de um componente estendido [30].

A figura 2.5 ilustra um exemplo de um componente do tipo estendido, mostrando suas portas de conexão, descritas acima.

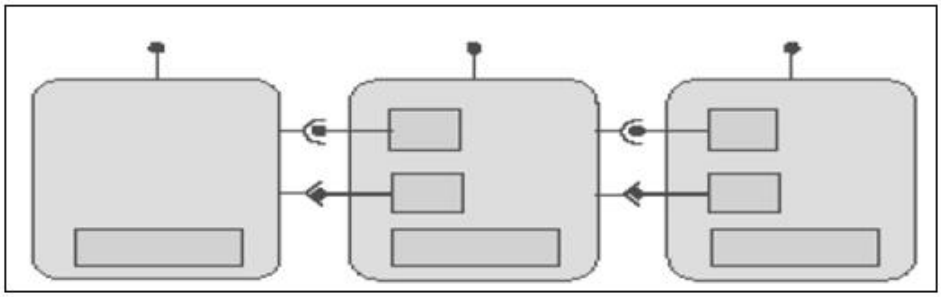

Figura 2.6. Exemplo de interconexão entre vários componentes [30].

A figura 2.6 ilustra um exemplo de interconexão entre diversos componentes através de suas portas de comunicação. A comunicação entre dois componentes distintos é feita através da conexão entre portas compatíveis, sendo que produtores são interligados com receptores e facetas com receptáculos.

Um componente CCM, de acordo com [30], funciona através de um *container*, responsável por gerenciar a atividade dos componentes a ele associados, por gerenciar o *Portable Object Manager* (POA) do componente, por interagir com o ORB e seus serviços, e com o próprio componente. Esta estrutura interna é ilustrada na figura 2.7.

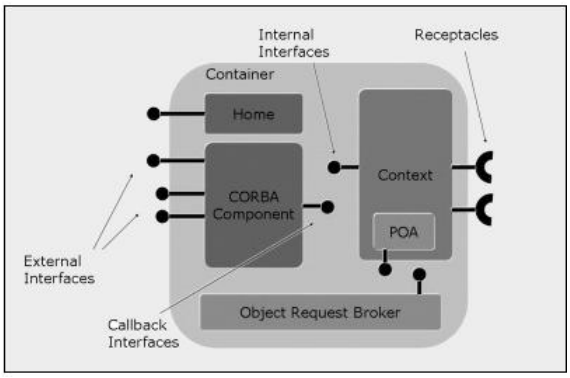

Figura 2.7. Estrutura interna de um componente CCM [42].

2.2.1.4 Interfaces *Component Implementation Definition Language* (CIDL)

A especificação CCM, por se basear na especificação CORBA, define que os serviços oferecidos por um determinado componente precisam ser descritos em arquivos inde-

pendentes de linguagem, denominados arquivos *Component Implementation Definition Language* (CIDL) [30].

Os arquivos IDL definem os módulos, interfaces e métodos pertencentes a cada interface definida, assim como define as portas de conexão requeridas e disponibilizadas [30].

Os arquivos CIDL são responsáveis por definir a estrutura e o estado da implementação de componentes. Através desta interface são gerados *stubs* e *skeletons*, responsáveis pela comunicação distribuída. A implementação do componente é feita através da extensão destes *skeletons* [30].

Um exemplo de arquivo IDL pode ser visto na figura 2.8.

```
1 //$Id: Hello_Base.idl 70613 2006-01-25 00:38:00Z dengg $:
2
3 #ifndef CIAO_HELLO_IDL
4 #define CIAO_HELLO_IDL
5
6 #include <Components.idl>
7
8 module Hello
9 {
10   interface ReadMessage
11   {
12     void texto();
13   };
14
15   interface trigger
16   {
17     void start ();
18   };
19
20 component Sender supports trigger,ReadMessage
21   {
22     provides ReadMessage texto;
23
24
25
26   };
27
28   home SenderHome manages Sender
29   {
30   };
31
32
33 };
34
35 #endif /* CIAO_HELLO_IDL */
```

Figura 2.8. Exemplo de interface IDL.

Na interface ilustrada na figura 2.8, é definido um módulo, na linha 8, com a fina-

lidade de modularização da aplicação. Nas linhas de 10 a 13 é definida uma interface, *ReadMessage*, que possui um método texto, sem parâmetros, que será responsável por imprimir uma mensagem no console.

As linhas 15 a 18 definem a interface *trigger*, necessária para a execução do componente. O componente *Sender* é definido nas linhas 20 a 26, e dá suporte às interfaces previamente definidas *trigger* e *ReadMessage*. Este componente irá fornecer o serviço texto, do tipo de interface *ReadMessage*.

Cada componente definido deverá ser associado a um *home*, que irá gerenciá-lo [26]. No exemplo acima, o *home* está definido nas linhas 28 a 30. Após a definição do arquivo IDL, um arquivo CIDL deve ser criado, de forma a gerar os *stubs* e *skeletons* necessários para execução do componente. Um exemplo de arquivo CIDL está ilustrado na figura 2.9.

```
1 //$Id: Sender.cidl 68081 2005-09-01 19:26:35Z wotte $
2
3 #ifndef SENDER_CIDL
4 #define SENDER_CIDL
5
6 #include "Hello.idl"
7
8 composition session Sender_Impl
9 {
10   home executor SenderHome_Exec
11   {
12     implements Hello::SenderHome;
13     manages Sender_Exec;
14   };
15 };
16
17 #endif /* SENDER_CIDL */
```

Figura 2.9. Exemplo de interface CIDL.

O arquivo ilustrado na figura 2.9 define a interação entre o componente e o *container* que irá executá-lo. São três os tipos de interação existentes [30]:

- *Service* - Indica que não existe armazenamento de estado nas invocações dos métodos do componente;

- *Session* - Indica que há armazenamento não persistente de estado nas invocações dos métodos do componente;
- *Process* - Indica que há armazenamento persistente de estado nas invocações dos métodos do componente. A identificação única do componente deve ser implementada pelo programador;
- *Entity* - Indica que há armazenamento persistente de estado e a identificação única do componente fica a cargo do *container*.

Estes tipos de interação são precedidos pela palavra *composition*, como está ilustrado na linha 8 da figura 2.9. As linhas 10 a 14 apresentam a definição de um executor para o *home* do componente definido no arquivo IDL.

2.2.2 *ADAPTIVE Communication Environment* (ACE)

O *framework* ACE é voltado para a construção de aplicações baseadas em serviços de comunicação em tempo real e aplicações com requisitos temporais [38].

São quatro os principais requisitos para um *framework* voltado para comunicação [38]:

- **Flexibilidade** - Importante para suportar novos tipos de mídias e padrões de tráfego;
- **Eficiência** - Requerida para prover uma maior eficácia para aplicações que dependem de requisitos temporais;
- **Confiabilidade** - Necessária para assegurar robustez das aplicações, tolerância a falhas e alta disponibilidade;
- **Portabilidade** - Necessária para reduzir o esforço de implantação de uma aplicação em sistemas heterogêneos.

O ACE provê funcionalidades e serviços que atendem aos requisitos acima citados. Sua construção foi feita em camadas, de modo que a complexidade e os conceitos possam ser divididos, facilitando seu entendimento. Seus componentes foram desenvolvidos para suportar a comunicação distribuída concorrente e os serviços. São os seguintes os componentes do ACE [38]:

- *Event demultiplexing components* - Através dos componentes *Proactive* e *Reactive* do ACE é possível responder a algum tipo de evento lançado pelas aplicações, como eventos de Entrada-Saída (ES), temporais, de sinais e eventos síncronos;
- *Service initialization componentes* - Os componentes *Connector* e *Acceptor* são os responsáveis pelas regras de inicialização ativa e passiva, respectivamente;
- *Service configuration components* - Este componente é responsável pela configuração das aplicações tanto no momento da instalação quanto em tempo de execução;
- *Hierarchically-layered stream components* - Componente útil para simplificar o desenvolvimento de aplicações voltadas para comunicação;
- ORB *Adapter components* - Capaz de prover integração transparente com aplicações CORBA *single* ou *multi-threaded* , através do ORB.

Figura 2.10. Estrutura do ACE [2].

A figura 2.10 mostra as camadas do ACE, dividindo-as em serviços implementados pelo *framework* e serviços que não são implementados, porém ficam disponíveis para implementação, de acordo com as necessidades da aplicação.

A parte de Adaptação ao Sistema Operacional é responsável pela comunicação entre interfaces específicas do sistema operacional e interfaces do ACE. Isso torna o ACE portável, já que as camadas superiores ficam independentes da plataforma utilizada. A camada dos *facades* ou empacotadores provê uma biblioteca orientada a objetos que é capaz de substituir as chamadas de sistema originais do sistema operacional em uso pelas chamadas implementadas no ACE. A camada de *frameworks* provê os serviços de *event demultiplexing components* e *service configuration components*. Esta camada facilita a construção de aplicações baseadas em camadas, facilitando a integração ou substituição de módulos nas mesmas. As camadas que não fazem parte do ACE são disponibilizadas com o intuito de fornecer às aplicações funcionalidades que, normalmente, são requisitadas, como, por exemplo, localização de objetos, *logging*, sincronização de relógios, dentre outras [30].

O ACE, apesar de prover os recursos supracitados, faz uso de um *middleware*, objetivando facilitar a implementação de aplicações que utilizem seus serviços fornecidos. Este *middleware* é denominado TAO e será descrito na próxima seção.

2.2.3 *The ACE ORB* (TAO)

O TAO é uma implementação otimizada da camada ORB do CORBA, tendo por base o ACE, e voltado para aplicações de tempo real. Possui um mecanismo de Entrada/Saída de alta performance e uma porta de comunicação *Asynchronous Transfer Mode* (ATM) [40].

O TAO surge como uma alternativa de utilização do CORBA para tempo real, tendo as seguintes características [40]:

- Identificar melhorias para o ORB padrão para atender aos requisitos de QoS;

- Determinar empiricamente as características necessárias para a construção de um ORB voltado a aplicações com exigências de qualidade de serviço fim a fim;
- Buscar estratégias que aumentem a performance dos mecanismos de Entrada/Saída, de maneira a diminuir a latência, garantir a taxa de transmissão e aumentar a confiabilidade fim a fim;
- Identificar os padrões de projeto necessários para construir, manter e estender aplicações que utilizem o ORB em tempo real.

O ACE/TAO está disponível em [14] e para sua utilização o usuário deve compilar cada classe conforme sua necessidade.

2.2.3.1 Arquitetura TAO

A arquitetura do TAO contém, além dos componentes do CORBA, um subsistema de Entrada/Saída, protocolo de comunicação e uma interface de rede, que contém [40]:

- **Subsistema de entrada/saída** - Envia e recebe requisições em tempo real através de uma porta de comunicação ATM ou placa-mãe (como VME ou compactPCI);
- **Escalonador em tempo real** - Determina a prioridade das requisições para o atendimento pelo ORB;
- **ORB** - Provê as características do ORB CORBA;
- **Adaptador de objeto** - Realiza a operação de *unmarshalling* das requisições e repassa-as aos serventes;
- *Stubs* **e** *skeletons* - Otimiza o processo de *marshalling* e *unmarshalling* no código gerado pelo compilador IDL do TAO;
- **Gerenciamento de memória** - Minimiza a alocação de memória dinâmica e a cópia de dados através do ORB;

- **API QoS** - Permite que os requisitos de qualidade de serviço sejam informados pelas aplicações e serviços CORBA, utilizando o modelo de orientação a objetos.

Estes componentes estão ilustrados na figura 2.11.

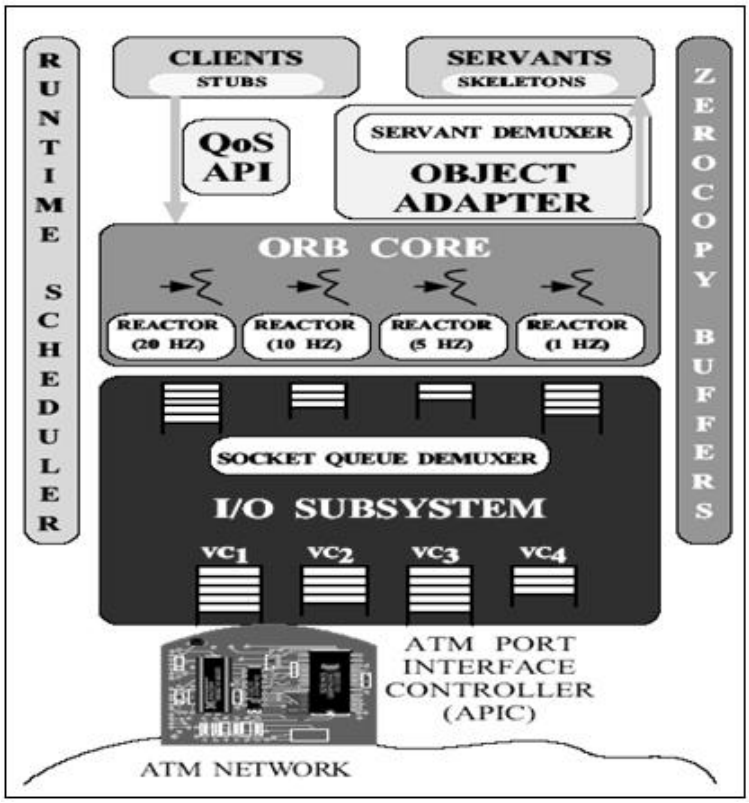

Figura 2.11. Arquitetura do TAO [40].

A figura 2.11 ilustra a estrutura interna do TAO, com a porta ATM de comunicação e o subsistema de Entrada/Saída, responsável por receber as requisições e fazer o processo de *marshaling* e *unmarshaling*, através do repasse das requisições aos objetos responsáveis por seus respectivos processamentos. Este processo de receber a requisição do cliente, repassar ao servidor para processamento, e retornar o resultado ao cliente caracteriza a função de um ORB. O fornecimento do serviço de QoS se dá através da API QoS.

2.2.3.2 Resolvendo limitações do CORBA no TAO

Conforme [40], para atender as aplicações de tempo real e suas exigências de qualidade de serviço, sistemas finais ORB devem definir políticas e mecanismos para especificar aplicações fim-a-fim com requisições QoS, reforço QoS em sistema operacional de tempo real e rede, protocolo de comunicação de tempo real e engenharia de protocolo eficiente e confiável, *demultiplexing* (capturar vários sinais e transformá-los em um só de maneira eficiente e confiável), camada de aplicação e gerenciamento de memória eficientes e confiáveis.

No CORBA 2.x não é possível definir especificação QoS. No TAO existe uma combinação da IDL OMG padrão com serviço ORB de suporte a QoS, resultando em IDL's de tempo real (RIDL), RT_OPERATION e RT_INFOSTRUCT, permitindo que informações de QoS sejam especificadas.

Subsistemas de I/O (Entrada/Saída) e interface de rede são integrados com o TAO. Um subsistema de I/O é responsável por intermediar acesso entre o ORB e o baixo nível de rede e recursos do Sistema Operacional. Através desse subsistema é possível reforçar atributos QoS pela rede. Ele é composto por uma porta de comunicação ATM de alta velocidade, subsistema de I/O de tempo real, serviço de escalonamento de tempo real e escalonamento em tempo de execução e controlador de admissão.

O TAO é formado por um protocolo de comunicação de tempo real confiável e eficiente através de melhorias na implementação do protocolo CORBA *General Inter-ORB* (GIOP). O GIOP no TAO especifica uma arquitetura de comunicação e concorrência que minimiza inversão de prioridade e um protocolo de transporte que permite o processamento e a comunicação de forma eficiente e confiável entre sistemas finais ORB heterogêneos. O GIOP especifica uma interface abstrata, concretizada em um protocolo de comunicação com conhecimento de certas requisições [40].

Para minimizar o *overhead* ao fazer a tradução de vários sinais em um sinal apenas, processo conhecido como *demultiplexing*, o TAO aplica uma função *hashing* perfeita para

otimização ativa no mapeamento das requisições do cliente diretamente para a operação no servidor no tempo específico. Esse esquema é aplicado quando as chaves *hash* são conhecidas antecipadamente.

Transformar dados do nível de aplicação é custoso e, para tornar mais eficiente e confiável, o TAO deixa o processamento da camada de aplicação para os *stubs* do lado cliente e os *skeletons* do lado do servidor, sendo estes gerados automaticamente por um compilador IDL de alta performance. Para reduzir as inconsistências entre os clientes *stubs* e os servidores *skeletons*, o compilador IDL do TAO suporta otimizações para diminuir o uso de memória dinâmica, atenuar a cópia excessiva de dados, trabalhando sempre que possível com blocos atômicos, e minimizar o *overhead* gerado nas chamadas de função. O projeto permite que a aplicação escolha entre gerar *stubs* e *skeletons* de forma interpretada ou compilada [40].

De acordo com [40], o gerenciamento de memória dinâmica causa um impacto significativo na performance devido ao *overhead* e fragmentação. Para tornar esse gerenciamento mais apropriado para aplicações de tempo real, o TAO é designado para minimizar e eliminar a cópia de dados nos múltiplos níveis de sistema final ORB, podendo utilizar a interface *Advanced Programmable Interrupt Controller* (APIC) e circuito virtual ATM para melhorar o sistema de gerenciamento de *buffer*.

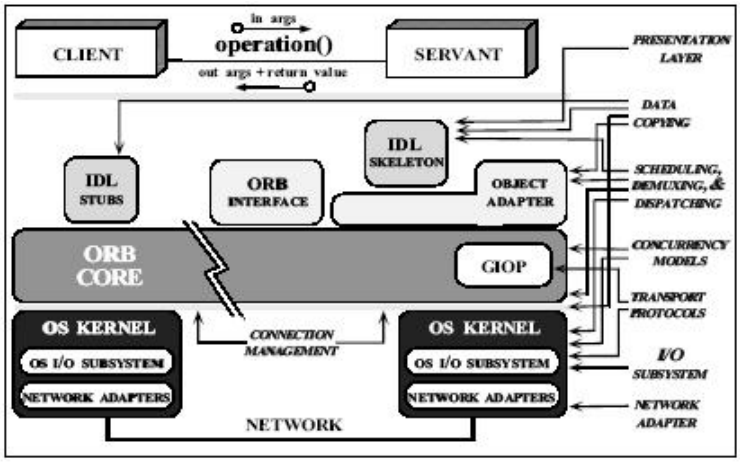

Figura 2.12. Otimizações necessárias para um ORB em tempo real [40].

A figura 2.12 ilustra as otimizações necessárias para converter um ORB tradicional em

um capaz de atender às requisições temporais das aplicações de tempo real. Políticas e mecanismos para prover qualidade de serviços, protocolos eficientes para comunicação em tempo real, demultiplexação e entrega de mensagens eficientes e previsíveis, camada de aplicação previsível e eficiente e gerenciamento de memória eficiente são as características principais apresentadas na figura citada.

2.2.4 *Component Integrated ACE ORB* (CIAO)

O CIAO é uma implementação do padrão CORBA CCM com o propósito de prover Qualidade de Serviço (QoS), através da extensão do ORB do TAO. A qualidade de serviço é executada através de pré-alocação de recursos de *hardware* (memória, disco rígido, recursos de CPU, banda de rede, dentre outros recuros [47, 28].

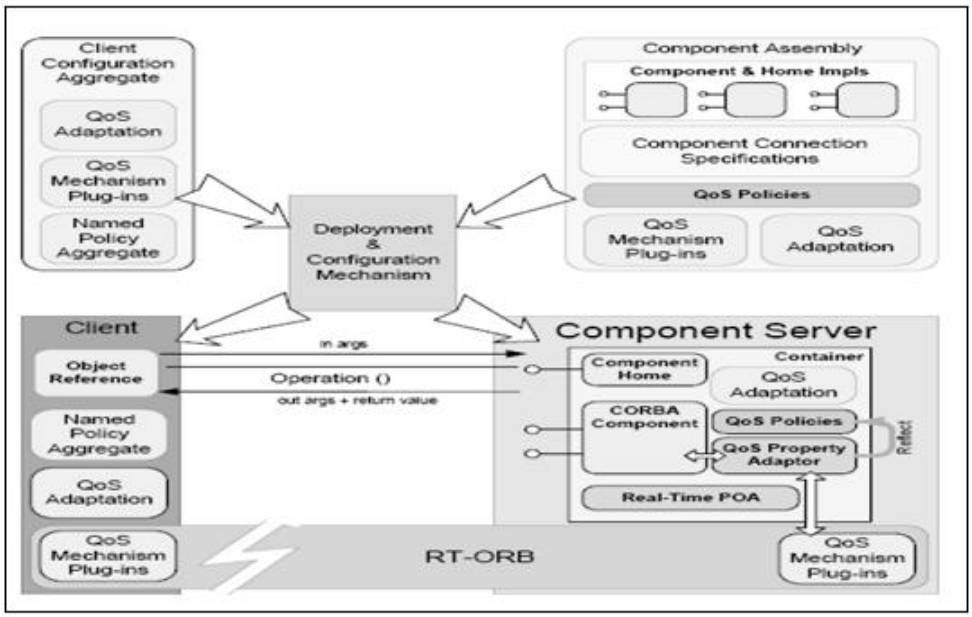

Figura 2.13. Arquitetura do CIAO [47].

A figura 2.13 ilustra a arquitetura do CIAO.

Para prover o QoS estático desejado, é necessário que alguns pontos do CCM sejam estendidos pelo CIAO, como [47]:

- *Component assembly* - Tem a função de descrever a forma como os componentes estão compostos no sistema. A extensão deste para o CIAO se dá com o propósito de prover QoS para as aplicações e para possuir a implementação dos mecanismos necessários para prover QoS;

- *Client configuration aggregates* - A configuração das especificações do lado cliente é importante para que muitas políticas referentes à qualidade de serviço sejam fornecidas. Esta configuração pode ser acoplada ao ORB CIAO de forma transparente;

- **QoS** *Aware containers* - Tem por finalidade gerenciar de forma centralizada as políticas de QoS disponibilizadas, além de interagir com os mecanismos requeridos pelo fornecimento da qualidade de serviço;

- **QoS** *Adaptation* - Provê uma maneira de adaptação dinâmica de qualidade de serviço.

No CIAO, os conceitos de serviços de QoS e componentes são tidos de modo separados, ou seja, a política de qualidade de serviço está acoplada diretamente ao *middleware* CIAO, não sendo dependente dos requisitos de qualidade de serviço requisitados pelos componentes e módulos que integram uma aplicação [28].

O CIAO fornece a possibilidade de inclusão de comportamento de tempo real para as aplicações, graças à extensão das funcionalidades de implementação de componentes e aplicações em tempo de execução do CCM. Esta característica facilita aos desenvolvedores especificar o comportamento de tempo real das aplicações, além de associá-lo aos componentes em várias partes de uma aplicação [28].

O CIAO, dentro de sua estrutura interna, possui o *Deployment and Configuration Engine* (DAnCE), que é uma implementação da especificação CCM com a finalidade de padronizar o processo de implantação e execução de aplicações baseadas em componentes [30].

O DAnCE é composto de objetos CORBA 2.x que são responsáveis pelo processo de implantação e execução, sendo os principais objetos [30]:

- *NodeManager* - Objeto executado em cada uma das máquinas que irão ser conhecidas pela aplicação, sendo responsável pela implantação da aplicação nos respectivos *hosts*. Este objeto cria outro objeto de nome *NodeApplicationManager* que, por sua vez, cria objetos do tipo *NodeApplication*, capazes de hospedarem os *containers* responsáveis pela execução de um componente CCM;

- *ExecutionManager* - Objeto responsável pela implantação da aplicação nos domínios de implantação. Um domínio é representado pelo conjunto de *hosts*, componentes e conexões de uma implantação;

- *DomainApplicationManager* - É invocado através do *ExecutionManager* e pode repartir o descritor de implantação em sub-partes, de modo a distribuí-las entre os *hosts* do domínio;

- *PlanLauncher* - Responsável pela leitura do descritor de implantação, geração de um plano de implantação da aplicação e pelo posterior envio ao *ExecutionManager* deste plano.

O DAnCE disponibiliza um serviço chamado *Redeployment and Reconfiguration* (ReDaC), que tem a finalidade de auxiliar na implantação de novos componentes e na reconfiguração dos já existentes, em tempo de execução, sem precisar que a aplicação seja interrompida [30].

A utilização do ReDaC pode ser feita via programação ou manualmente, através da execução do objeto *PlanLauncher*, utilizando um arquivo descritor de implantação diferente do utilizado previamente para implantar a aplicação. Para obter o plano de execução previamente implantado via programação, é necessário invocar o método *getPlan* do *ExecutionManager* passando como parâmetro o UUID do descritor de implantação desejado, fazer as alterações necessárias, e reimplantar este plano, utilizando o método *performRedeployment*, também do *ExecutionManager*, passando o plano de execução modificado como parâmetro.

Outra forma de obter o plano de execução desejado é, primeiramente, obter a referência do *DomainApplicationManager* responsável pela implantação da qual se deseja criar ou remover réplicas, através do método *getManager* do *ExecutionManager*, passando o UUID do plano como parâmetro, e, em seguida, invocar o método *getPlan* da classe *DomainApplicationManager*. Um exemplo de como isso é feito na prática está ilustrado na figura 2.14.

```
1 this->execution_manager_ = ::Deployment::ExecutionManager::_narrow (obj.in ());
2 Deployment::DomainApplicationManager_ptr DAM = this -> execution_manager_->getManager(uuid);
3 ...
4 ::Deployment::DeploymentPlan_var dp = DAM->getPlan();
5 ...
6 this->execution_manager_ -> perform_redeployment(dp);
```

Figura 2.14. Obtendo o plano de execução via programação.

A linha 2 ilustra a instanciação de uma variável do tipo *DomainApplicationManager*, que irá possuir a referência para o plano de implantação identificado pelo UUID passado como parâmetro. Na linha 4, o plano de execução do *manager* resolvido anteriormente (linha 2) é obtido invocando o método *getPlan* da classe *DomainApplicationManager*. Na linha 6, é feita a reimplantação do plano de execução, após as alterações necessárias, através da invocação do método *perform_redeployment* do *ExecutionManager*, passando o plano de execução alterado. Estas alterações podem compreender inclusão ou remoção de instâncias de componentes em *containers*, assim como adição ou remoção de conexões para estas instâncias alteradas.

Caso se deseje uma reimplantação manual de um plano de execução, é preciso invocar o objeto *PlanLauncher*, informando o plano de execução modificado, e o tipo de implantação como parâmetro (por exemplo, -r caso seja reimplantação ou -p para implantar uma montagem nova).

A figura 2.15 ilustra, de maneira simplificada, o processo de implantação de uma aplicação no CIAO. O ponto de partida é a inicialização do servidor de nomes (*NameService*), seguindo-se pela inicialização do *NodeManager* em cada um dos *hosts* que irão hospedar *containers* e componentes. Em seguida, o *ExecutionManager* é invocado, passando o arquivo descritor de implantação para o *PlanLauncher*, que irá transformá-lo

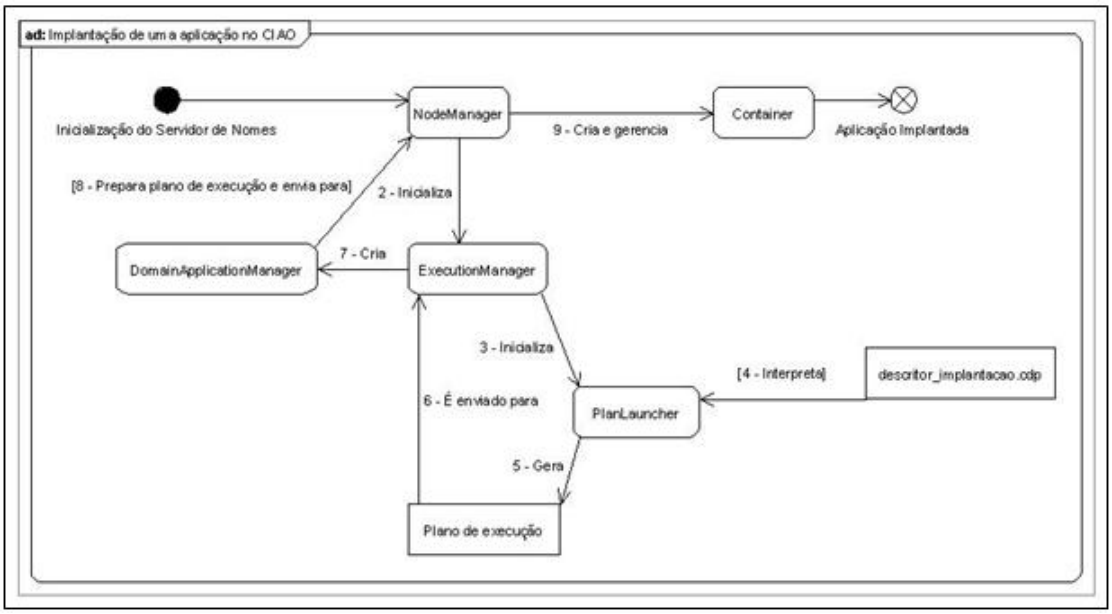

Figura 2.15. Implantação de uma aplicação no CIAO.

em plano de execução, enviando a referência a este plano de volta ao *ExecutionManager*. O passo seguinte será a criação do *DomainApplicationManager* pelo *ExecutionManager* para gerenciar o plano de execução criado, dividindo-o em subplanos e repassando as respectivas divisões aos *NodeManagers* apropriados, em cada *host*. O *NodeManager*, então, irá criar os respectivos containers para implantação final dos componentes de sua responsabilidade.

2.2.5 *Architecture for Control and Supervision* (ARCOS)

O *Architeture for Control and Supervision* (ARCOS) é um *framework* de componentes de tempo real voltado para a área de Controle e Supervisão (S&C) [1]. O ARCOS define interoperabilidade através das especificações dos seus componentes, para os três módulos básicos da indústria de S&C: aquisição de dados, controle e supervisão. Este *framework* foi desenvolvido sobre o padrão de comunicação entre objetos distribuídos CORBA, utilizando o módulo CORBA *Component Model* (CCM), e a variação do CCM para tempo real, CIAO.

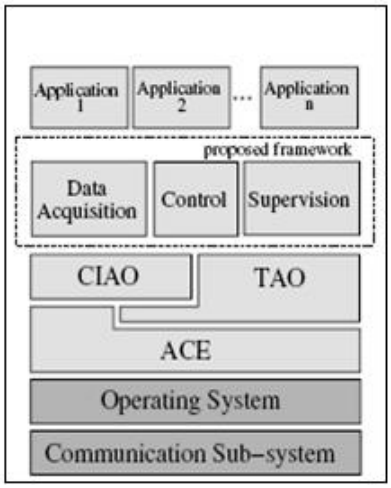

Figura 2.16. Arquitetura básica do ARCOS [23].

A figura 2.16 ilustra a arquitetura básica do ARCOS. Na parte mais baixa está o sistema operacional e o subsistema de comunicação. Na parte intermediária encontra-se o CIAO, o TAO e sua base arquitetural, o ACE. O ACE, juntamente com o TAO e o CIAO formam a base na qual o *framework* foi desenvolvido.

O ARCOS foi arquitetado pensando em três características importantes - reusabilidade, flexibilidade e interoperabilidade. Em se tratando de flexibilidade, a importância se deve a várias aplicações feitas para tempo real terem objetivos especificados para uma determinada classe restrita de aplicações. Quando se utilizou uma arquitetura baseada em componentes, buscou-se uma maior flexibilidade na comunicação entre componentes e seus relacionamentos [1].

A indústria de controle e supervisão requer uma comunicação do tipo muitos-para-muitos, sempre podendo ter a capacidade de acoplar novos módulos ao sistema. Devido a isso, a plataforma TAO foi escolhida, por prover serviço de eventos em tempo real, além da facilidade de comunicação entre os objetos do sistema.

Quando se decidiu pela criação de um *framework* pensou-se na reusabilidade, pelo próprio conceito de *framework*, no qual componentes genéricos para Controle e Supervisão foram desenvolvidos. Para aquisição de dados, foi utilizado o *Data Acquisition from Industrial Systems* (DAIS) que é um padrão CORBA e que funciona mapeando os dados

adquiridos através de algum dispositivo para interfaces CORBA.

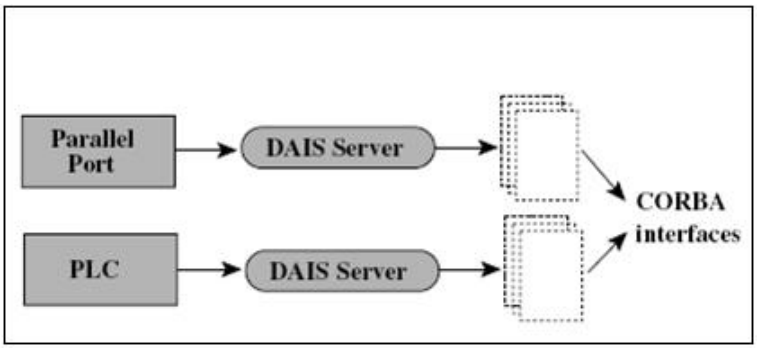

Figura 2.17. Funcionamento do DAIS [1].

A figura 2.17 ilustra o funcionamento do DAIS. Dados obtidos pelos periféricos do sistema, como porta paralela e *Program Logic Control* (PLC), são interceptados pelos servidores DAIS, que, por sua vez, os convertem para as interfaces CORBA definidas.

Para a parte de supervisão, existem, no ARCOS, duas aplicações: DAIS *monitor* e DAIS *browser*. A primeira tem por funcionalidade exibir o estado do servidor DAIS, além de ativar o serviço de eventos em tempo real, enquanto o DAIS *browser* é uma visualização genérica para qualquer servidor DAIS, permitindo ao usuário criar e gerenciar grupos de dados.

A parte de controle está em um componente de nome *ControlManager* que tem o papel de consumir, através do serviço de eventos, as mensagens produzidas pelo servidor DAIS, enviando mensagens aos atuadores também pelo serviço de eventos. A passagem das mensagens para os atuadores é feita pelo componente *DAISWRITER*.

2.2.6 *Fault Tolerant* CORBA (FT CORBA)

Muitas aplicações requerem mecanismos de tolerância a falhas. Para suprir esta necessidade em aplicações que utilizem o *middleware* CORBA, foi criado o padrão *Fault Tolerant* CORBA (FT CORBA). Este padrão está especialmente voltado para aplicações que tenham como requisito uma alta confiabilidade (*reliability*). O mecanismo de tolerância

a falhas incluído neste modelo foi a replicação de objetos, caracterizando a redundância. Este mecanismo facilita a configuração das réplicas e sua quantidade, assim como a disposição destas em diferentes máquinas (*hosts*) [31].

O padrão FT CORBA dá suporte às técnicas de replicação passiva, ativa, redirecionamento de requisições para outra réplica ativa (em caso de falha da primeira réplica requisitada), além de permitir que o usuário possa definir as propriedades necessárias de tolerância a falhas para cada réplica criada. Em caso de falhas ocorrerem em alguma das réplicas, o FT CORBA possui mecanismos de análise e detecção destes erros [31].

A replicação fornecida pelo padrão CORBA para tolerância a falhas funciona baseada no conceito de replicação e grupos de objetos (classes). Várias réplicas de um determinado objeto são criadas e gerenciadas como sendo um grupo. Além da referência individual de cada objeto existente no grupo, existe também a referência ao grupo, denominada de *Interoperable Object Group Reference* (IOGR). É esta referência que é disponibilizada pelos servidores replicados para utilização pelos clientes [32].

A figura 2.18 ilustra o processo de requisição feita por um cliente a um grupo de objetos replicados. A requisição, após ser recebida pelo grupo, é processada pelos membros do grupo e a resposta é, então, retornada pelo grupo ao cliente, como num processamento convencional. Esta definição de grupo permite a transparência ao cliente em relação à replicação dos objetos, pois, para este, há apenas um único objeto (o grupo).

Quando se refere à parte prática da utilização do FT CORBA, uma réplica pode ser criada através do método *create_object*. Para a criação de réplicas em diferentes *hosts*, é preciso invocar este método em cada um deles. Para a adição, remoção e criação de membros de um grupo deverão ser invocados os seguintes métodos, respectivamente: *create_member*, *add_member*, *remove_member*. Estes métodos permitem que a aplicação detenha controle sobre as operações descritas, porém, de forma não transparente ao usuário [32].

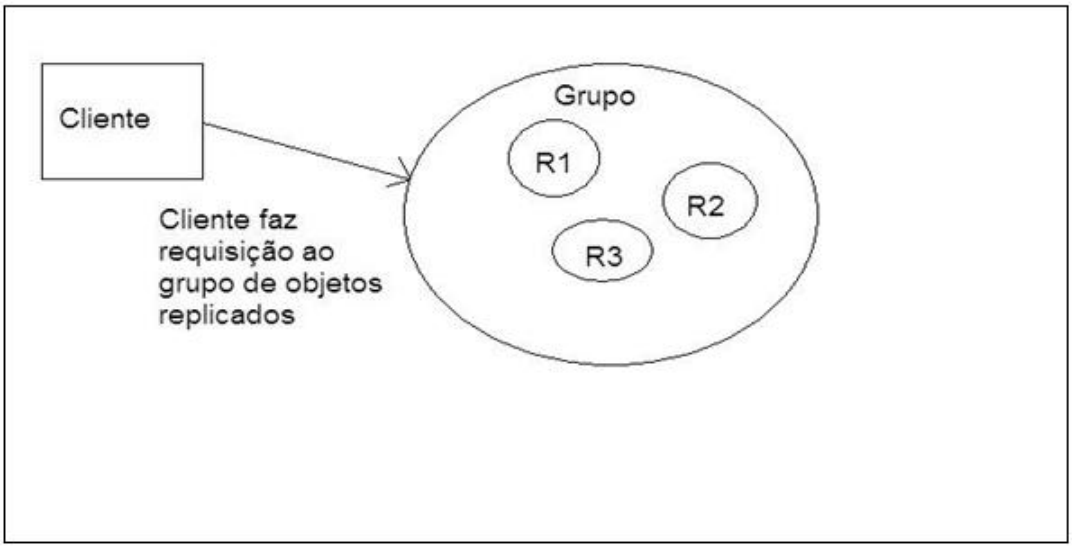

Figura 2.18. Requisição de um cliente a um grupo de objetos.

2.2.6.1 A prática da especificação FT CORBA

Enquanto que na teoria do FT CORBA a replicação é feita de maneira fortemente consistente, de modo que as réplicas permanecem sincronizadas mesmo na ocorrência de falhas, na realidade este tipo de replicação proposto só é concretizado quando há uma independência de falhas entre as máquinas envolvidas no sistema. Se cada máquina falhar de forma independente das outras, este modelo é aplicável, porém, caso uma falha em uma máquina ocasione falhas nas demais, o modelo proposto de replicação não será compatível, pois as réplicas não conseguirão se manter distintas umas das outras [29].

O padrão FT CORBA define algumas interfaces para os serviços fornecidos, que são [31]:

- *Replica management*;
- *Fault management*;
- *Recovery management*;
- *Fault-Tolerance property management*.

Uma aplicação que vá fazer uso dos recursos disponibilizados pelo FT CORBA terá, obrigatoriamente, que implementar todas estas interfaces citadas, embora a maneira pela qual implementá-las não é ditada na especificação. Outro ponto a ser observado é a questão do objeto a ser replicado. Pela padronização, é dito que apenas objetos do tipo servidor CORBA podem ser replicados, o que consiste em uma implementação específica, em uma linguagem também específica, de uma interface de servidor contendo métodos e estados internos. Na realidade o conceito de objeto pode diferir de ambiente para ambiente [29].

Em relação à existência de grupos de objetos replicados, na prática, existe um objeto considerado o mestre dentro de cada grupo. Caso este objeto apresente alguma falha, tanto ele quanto todos os demais objetos existentes no grupo serão removidos do sistema. Então, a visão de unidade de replicação passa de um objeto isolado para o grupo como um todo. O não determinismo é explicitamente proibido pelo FT CORBA. Isso se dá, devido ao risco que se corre em permitir o não determinismo entre as réplicas e haver alguma inconsistência devido à alguma falha na atualização de réplicas, como a atualização tardia que ocorre na replicação passiva.

Ainda segundo [29], apesar do padrão descrito nesta seção garantir a forte consistência entre réplicas, não há, na padronização, uma maneira explícita de como alcançar esta propriedade. Mesmo que haja determinismo por parte do servidor, para que aconteça esta característica entre as réplicas deste servidor é preciso que se tenha algum protocolo de ordenação total das requisições pelas réplicas processadas. Este tipo de ordenação requer algum protocolo de consenso para comunicação em grupo.

2.2.7 *Adaptive Fault-Tolerant Component Model* (AFT-CCM)

O conceito de tolerância a falhas adaptável é o de fornecer suporte à Qualidade de serviço (QoS) requerida pelas aplicações, de forma a ser configurada dinamicamente. O modelo AFT-CCM permite que as aplicações possam especificar sua necessidade de QoS - dentre os níveis baixo, médio ou alto. Com o propósito de poder adaptar dinamicamente

QoS às aplicações, o FT-CCM introduz em seu modelo o conceito de componentes lógicos, que são detectores de falhas. A partir da detecção de uma falha na aplicação, serão adotadas novas medidas para fornecer QoS de maneira a atender às novas exigências da aplicação [11].

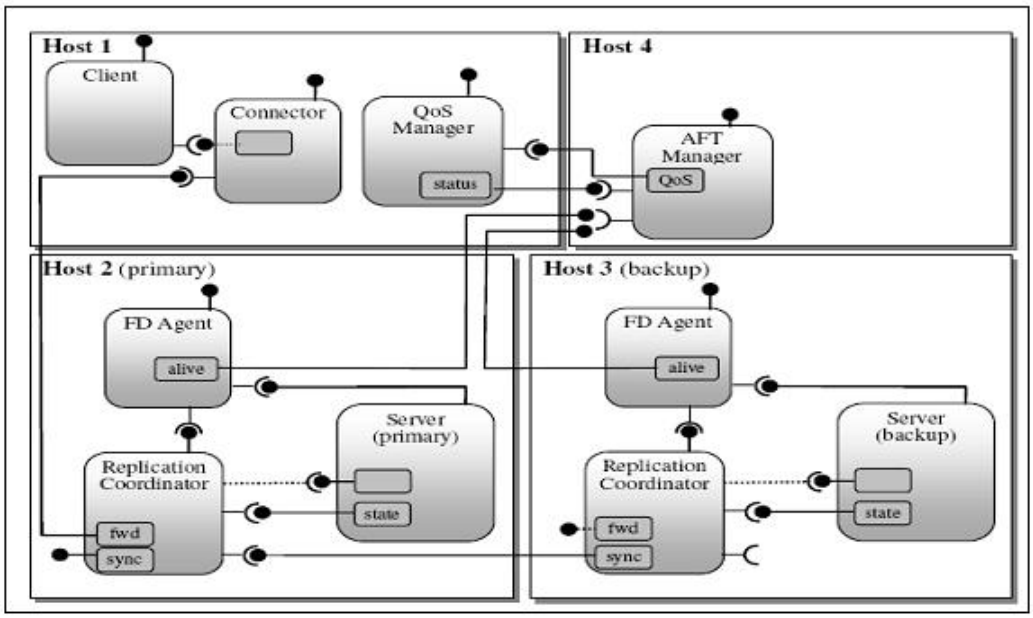

Figura 2.19. Visão do FT - CCM [11].

A figura 2.19 ilustra uma visão geral do modelo AFT - CCM. Nesta figura, os componentes estão dispostos em quatro *hosts* de um sistema distribuído. Estes componentes não funcionais são responsáveis pela configuração e monitoramento de componentes da aplicação, replicando estes componentes com uma técnica que vise prover a confiabilidade requerida.

Ainda na figura 2.19 é ilustrado um servidor primário (*Server (primary)*), que se encontra no *host* 2, tendo sua réplica localizada no *host* 3 (*Server (backup)*). No modelo AFT - CCM, o protocolo de coordenação da técnica de replicação utilizada é implementado em forma de um componente: o *Replication Coordinator*. Este coordenador tem a função, por exemplo, de atualizar as réplicas quando a replicação passiva é utilizada.

Os serviços providos por um componente se tornam acessíveis a um cliente através de um conector, que irá deixar de forma transparente qualquer mudança necessária na

configuração da aplicação. Para cada componente que necessite de tolerância a falhas, um componente denominado *Adaptive Fault-Tolerance Manager* (AFT *Manager*) é criado. Este componente é responsável pela definição da configuração atual da aplicação, que é baseada pelo QoS requerido e pela freqüência na ocorrência de falhas parciais. Cada componente da aplicação possui um FD *Agent* atrelado a ele, de forma que o FD *Agent* irá coletar informações de monitoramento e enviá-las ao AFT *Manager*. O FD *Agent* é capaz de detectar falhas do tipo *fail-stop* tanto para o *host* quanto para componentes [11].

A reconfiguração da aplicação se inicia quando os dados de monitoramento coletados pelo AFT *Manager* indicam que aquela qualidade de serviço inicialmente fornecida à aplicação já não mais se aplica às suas atuais necessidades. O AFT *Manager* utiliza componentes de detecção de falhas denominados *Fault Detector Agents* (FD Agent), que também estão ilustrados na figura 2.19, nos *hosts* 2 e 3 [11].

2.3 *ENTERPRISE JAVA BEANS* (EJB)

No cenário atual do mercado de produção de aplicações comerciais, uma tecnologia bastante utilizada é o *Enterprise Java Beans* (EJB), pois além de ser código aberto, pertencente à família da linguagem de programação mais utilizada atualmente, Java, existem também vários servidores de aplicação disponíveis que dão suporte a esta tecnologia. Quando o ORB entra em cena, fica responsável por toda a parte de gerenciamento das réplicas existentes [17].

Assim como acontece com o CCM, descrito em sessão anterior, o EJB é interligado a uma camada semelhante a um ORB capaz de receber, retransmitir e retornar a resposta de uma requisição feita por um cliente. Esta camada intermediária é responsável por toda a parte de gerenciamento das réplicas existentes.

Ainda segundo [17], são três os tipos de EJBs disponíveis na linguagem Java:

- *Entity beans* - Representa e manipula dados persistentes que são armazenados em

banco de dados relacionais;

- *Session beans* - Representa e manipula dados persistentes por apenas a presente sessão. Existe apenas um *session bean* por cliente. Se o estado deste *bean* for armazenado, é denominado *statefull*, caso contrário, *stateless*;

- *Message driven beans* - Fornece processamento assíncrono através de receptores de mensagens para o Java *Messaging Service* (JMS).

Um EJB acessa os serviços que são disponibilizados a ele por um *container*, que fica responsável pela intermediação entre as interações cliente - componentes. O estado de um *bean* pode ser gerenciado ou por ele mesmo ou pelo *container*, assim como as transações nas quais um *bean* está envolvido [17].

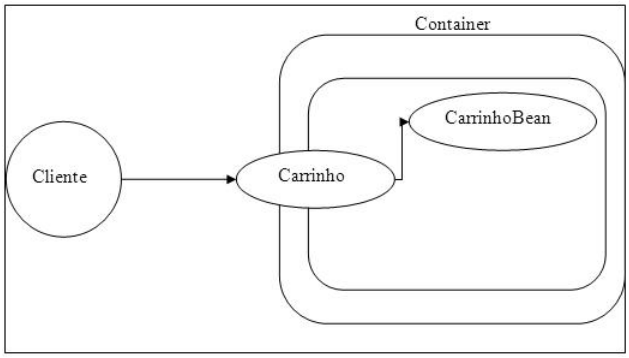

Figura 2.20. Exemplo de utilização de um *bean* [25].

Nesta ilustração 2.20 é apresentada uma interação entre um Cliente e um *SessionBean*. No exemplo, a aplicação é uma loja virtual, na qual o cliente obtém uma referência ao objeto Carrinho, o qual disponibiliza os serviços de uma loja. O carrinho pode, então, ser preenchido pelo cliente, sendo considerado como uma sessão do tipo *statefull*.

CAPÍTULO 3

MODELOS DE FALHAS E TÉCNICAS DE REPLICAÇÃO

Neste capítulo serão apresentados os conceitos e os modelos de falhas existentes, além das técnicas de replicação para sistemas com e sem requisitos temporais, e replicação para aplicações construídas com base na arquitetura de componentes. Em seguida, um comparativo entre as técnicas de replicação apresentadas será realizado.

3.1 MODELO DE FALHAS

Em um sistema distribuído, podem existir falhas em processos ou no canal de comunicação. Um modelo de falhas consiste na definição de como a falha pode ocorrer no sistema como um todo, facilitando o entendimento do efeito da falha, assim como seu tratamento [12].

Em geral, os seguintes tipos de falhas são assumidos num sistema para os seus componentes, quando se trata do desenvolvimento de sistemas tolerantes a falhas [41, 8]:

- **Falhas arbitrárias** - Também conhecidas como falhas bizantinas, descrevem uma classe de falhas cuja ocorrência de comportamento arbitrário ou malicioso pode levar o sistema a um funcionamento incorreto;
- **Falhas do tipo** *fail-stop* - O componente defeituoso comuta-se para um estado cuja detecção da falha seja possível. No contexto de serviços, estas falhas são evidenciadas por meio de paradas ou *crashes*, e comumente chamadas de falhas silenciosas. Estas falhas, segundo [20], podem ser sub-classificadas, de acordo com o comportamento da falha, em:

  - **Amnésia** - Na qual o componente reinicia em um estado pré-definido, independente das entradas recebidas antes da falha;
  - **Amnésia parcial** - Onde, no reinício do componente, uma parte do seu estado antes da falha é recuperado, enquanto que outra parte vai para um estado inicial pré-definido;
  - **Pausa** - O componente reinicia no mesmo estado em que estava antes da ocorrência da falha;
  - **Halting** - O componente, após a falha, nunca reinicia.

- **Falhas por omissão** - Um componente defeituoso do sistema omite a resposta a determinados dados de entrada;
- **Falhas temporais** - A entrega do resultado acontece em um período de tempo fora do previsto (antecipação ou retardo), mesmo que o resultado esteja correto;
- **Falhas de resposta** - O componente responde de maneira incorreta, sendo tanto pelo valor de resposta errado como pela transição de estado incorreta.

Um sistema é considerado tolerante a f falhas quando ele satisfaz a especificação de falhas na qual não mais que f componentes apresentem falhas em um determinado período de tempo observado, sendo este sistema constituído de um conjunto discreto de componentes [41].

Dentre os sub-tipos apresentados de falhas do tipo *crash*, os tipos de Pausa e *Halting* são classificados, de acordo com [8], como sendo sub-conjunto do tipo de falhas por omissão, devido ao comportamento dos mesmos.

As falhas temporais são ainda classificadas como falhas de desempenho, ainda conforme [20], pois podem ser tanto respostas adiantadas, quanto repostas atrasadas no tempo pré-definido para o recebimento das mesmas.

3.2 TÉCNICAS DE REPLICAÇÃO

Na literatura de Sistemas Distribuídos, três técnicas de replicação são bastante conhecidas e empregadas, que são: **Replicação Ativa**, **Passiva** e **Semi-Ativa (Líder - Seguidor)**. Também é proposto por [4] a **Replicação Semi-Passiva**, que busca adaptar a passiva para sistemas com requisitos temporais. Para a replicação voltada a sistemas de tempo real, temos, além das técnicas anteriormente citadas, a **Replicação Pró-ativa** e a replicação denominada *Resilient State-machine Replication*. Neste trabalho, as técnicas de replicação serão apresentadas da seguinte forma: primeiro serão abordadas as técnicas que podem ser aplicadas tanto para sistemas com requisitos temporais como para os sem estes requisitos e, em seguida, as técnicas específicas para sistemas de tempo real.

3.2.1 Sistemas Sem Requisitos Temporais

3.2.1.1 Replicação Ativa

Este tipo de replicação é também conhecido como *state-machine*, ou máquina de estados. A sua característica fundamental é o fato de que todas as réplicas trabalham sincronizadas, processando concorrentemente as requisições. Desta maneira, espera-se que todas elas adquiram o mesmo estado interno, e, por fim, atinjam o mesmo resultado. Os resultados gerados pelas réplicas em resposta à requisição são comparados de modo a se gerar um resultado final. Em geral, este passo é feito através de votação [43, 41, 20].

Este tipo de replicação é conhecido por máquina de estados, pois esta consiste em dois elementos: variáveis, que representam o estado interno do sistema; e comandos, que configuram as transições de estado da máquina. Os comandos podem ser vistos como as requisições de clientes. A abordagem de máquina de estados é um método geral e que se aplica bem para tolerância a falhas por meio de replicação. Por conta disso, muitos protocolos de replicação são baseados em máquinas de estados [41].

Três propriedades são definidas por [41] visando a determinar a consistência do sis-

tema quando se trabalha com replicação ativa. São elas:

- **Coordenação das Réplicas** - Todas as réplicas recebem e processam a mesma seqüência de requisições;
- **Concordância** - Todas as réplicas corretas recebem todas as requisições;
- **Ordem** - Todas as réplicas corretas processam as requisições que recebem na mesma ordem relativa.

Concordância

Considera-se que eventuais falhas de entrega por parte dos canais de comunicação devem ser evitadas ou resolvidas. Uma requisição enviada ao sistema, contendo um determinado valor, precisará que todas as réplicas respondam com o mesmo valor.

Assim, muitas abordagens são consideradas, tais como a retransmissão das mensagens por parte das réplicas.

As duas propriedades abaixo são estabelecidas em [41], na qual o protocolo de comunicação entre as réplicas deverá satisfazer para que exista concordância:

- **IC1** - Todos os processos *corretos* assumem o mesmo valor;
- **IC2** - Se o transmissor da mensagem é *correto*, então todas as demais réplicas assumem o seu valor.

Ordem

Garantir a ordem com que as mensagens são enviadas é de fundamental importância. Isto se revela, devido ao fato que nem sempre os canais de comunicação garantem que as mensagens serão todas elas entregues na mesma ordem em que foram solicitadas. Desta maneira, a causalidade é um fator de suma relevância para o processamento das mensagens pelas réplicas. Para isto, duas regras devem ser respeitadas, de modo que a causalidade seja garantida:

- Requisições enviadas de um cliente c serão processadas na ordem em que elas foram solicitadas;

- Se uma requisição r, feita por um cliente c, de alguma maneira pode ter causado uma requisição r' de um cliente c', então r deve ser processado antes de r'.

Uma grande vantagem da replicação ativa é que o tempo em que o sistema se encontra em falha, denominado tempo de falha (*fail-over*), no caso de ocorrerem falhas toleradas, pode ser considerado baixo ou inexistente. Uma desvantagem, porém, é que o custo (*overhead*) de se manter o sistema sincronizado a cada requisição é geralmente considerado alto, quando comparado com o baixo custo de outras técnicas baseadas em *checkpoints*, como a replicação passiva [43, 20].

Uma característica importante da técnica de replicação ativa é o fato de exigir determinismo por parte das réplicas, ou seja, as réplicas precisarão chegar a resultados idênticos em uma mesma ordem e dentro de um determinado intervalo de tempo, para que possam ser consideradas consistentes. Por exigir determinismo, esta técnica é capaz de suportar tanto falhas do tipo silenciosas (*fail-stop*) quanto arbitrárias (bizantinas) [43]. A relação entre o número de falhas toleradas e o número de réplicas dependerá do modelo de falhas assumido para o sistema [41].

Assumindo-se um sistema tolerante a f falhas, serão necessários:

- Para **falhas arbitrárias**, $2f + 1$ réplicas, para que se garanta que, por um processo de votação, a maioria seja considerada correta;

- Para **falhas silenciosas**, apenas $f + 1$ réplicas serão suficientes, já que apenas uma réplica ativa será necessária para que um resultado consistente seja obtido.

Ao assumir um modelo onde ocorram apenas falhas silenciosas, considera-se que o desempenho pode ser melhorado, pois os requisitos de comunicação podem ser simplificados [43].

A figura 3.1 ilustra a comunicação entre réplicas de um objeto usando a técnica de replicação ativa. B1, B2 e B3 representam instâncias de um mesmo processo ou serviço replicado, de quem A é cliente. C, por sua vez, cumpre o papel de serviço das instâncias de B. Note-se que B1, B2 e B3 assumem, ao mesmo tempo, o papel de serviço para A e de cliente para C.

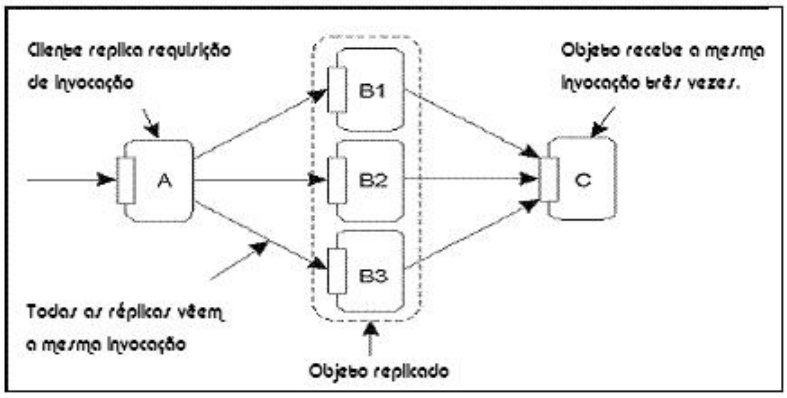

Figura 3.1. Replicação Ativa [15].

Conforme mostrado na figura 3.1, uma requisição de um cliente é enviada a todas as réplicas, que processam e enviam a requisição ao objeto. Um problema, entretanto, é a triplicação das requisições de B para C. A depender do modelo, esta situação pode levar a uma grave inconsistência, já que as respostas às requisições duplicadas chegam também duplicadas ao cliente A, que fez a requisição. Este cliente, caso dependa da resposta a este objeto para executar alguma operação, pode gerar uma resposta inconsistente, caso receba todas as respostas ao objeto B em tempos diferentes.

O problema apresentado na figura 3.1 é resolvido como apresentado na figura 3.2.

Como ilustrado na figura 3.2 em (a), uma requisição é enviada a todas as réplicas, porém é repassada a C uma única vez pelo coordenador do grupo de objetos B. Após ser recebida por todos os objetos do grupo C em (b), a requisição é processada e o resultado é retornado pelo coordenador do grupo C, que o retorna a todos os objetos do grupo B, que, por fim, o repassa ao Cliente através do coordenador do grupo B.

Os exemplos apresentados na figura 3.2 ilustram uma redução no número de men-

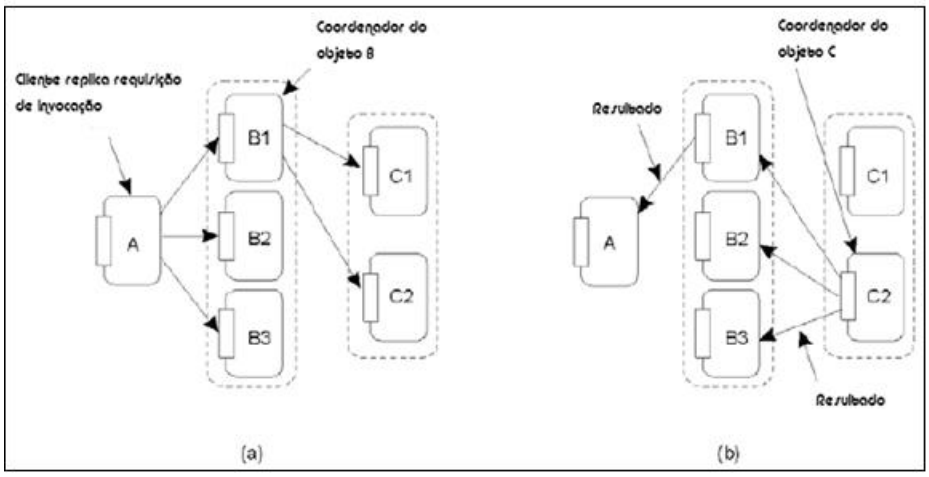

Figura 3.2. Resolvendo duplicidade de requisições na replicação ativa [15].

sagens enviadas entre o cliente e os objetos que irão processar a requisição, através de um centralizador para cada grupo de objetos. Esta solução evita a ocorrência da já citada possível inconsistência, gerada através da utilização da técnica ilustrada na figura 3.1.

Várias técnicas são abordadas em [41] que visam garantir consenso e ordem na replicação ativa, como através do uso de Relógios Lógicos, Relógios de Tempo Real Sincronizados, e Identificadores Gerados por Réplica.

3.2.2 Replicação Passiva

A replicação passiva é também conhecida como *Primary-Backup*, e se baseia no modelo mestre - seguidores. Nela, o sistema é composto por uma réplica primária (*primary*) ou líder que recebe e processa as requisições dos clientes. Diferentemente da técnica de replicação ativa, não há uma sincronização das réplicas quanto ao processamento das requisições dos clientes, mas sim um critério de comunicação entre a réplica lider e as seguidoras - normalmente com o uso de *checkpoints* - que garante a atualização e consistência do estado interno das réplicas. No caso de falha, uma réplica é ativada e continua a execução a partir do seu último *checkpoint* [20, 43, 5].

Enquanto que, na replicação ativa, a saída retornada ao cliente é concluída por meio

de votação, no caso da passiva apenas o líder responde às solicitações. Isto garante que não haja a obrigatoriedade de determinismo entre as réplicas, o que o torna um modelo bastante difundido para diversas aplicações [43].

Em caso de funcionamento normal do líder, sem ocorrência de falhas, esta técnica garante um menor *overhead* de processamento do que a replicação ativa, já que a sincronização feita por *checkpoints* é normalmente menos custosa do que a sincronização total das requisições de entrada. No caso de falhas, entretanto, aumenta-se o *overhead* devido à necessidade de intervenção para a escolha de um novo líder e continuidade do serviço [43].

Três métricas que definem bem os custos de um protocolo de replicação passiva são [5]: Grau de replicação (número de réplicas), Tempo de bloqueio (tempo de resposta em execução normal) e tempo de *Fail-over* (tempo de recuperação no caso de falha).

A figura 3.3 ilustra uma abordagem típica da técnica de replicação passiva. O processo se dá nos seguintes passos:

1) O cliente envia a requisição de gravação de dados para o *primary*;

2) A requisição é processada e uma mensagem de atualização é propagada às demais réplicas;

3) As réplicas executam a atualização e respondem ao coordenador;

4) Por fim, o *primary* envia resposta ao cliente.

Uma deficiência da técnica de replicação passiva está no não suporte a falhas arbitrárias (bizantinas), sendo aplicável a um modelo onde apenas serão toleradas falhas silenciosas [43]. Um modelo de sistema adequado para replicação passiva é assumido por [5], com ocorrências apenas dos seguintes tipos de falhas:

- **Falhas tipo** *crash* - Parada total no servidor, sem que ele seja capaz de se recuperar por si só;

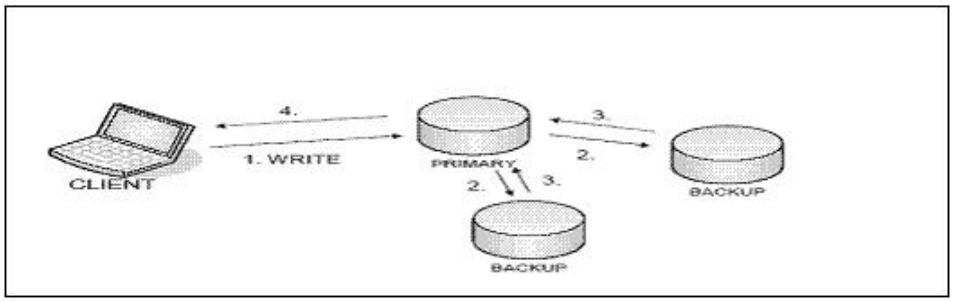

Figura 3.3. Replicação Passiva [16].

- *Crash* **com falhas no canal de comunicação** - Parada do servidor ou perda de mensagens no canal;
- **Omissão na recepção** - Embora um servidor continue operando, as mensagens são perdidas na recepção;
- **Omissão no envio** - Perda de mensagens no envio, sem a parada do serviço;
- **Omissão geral** - Ocorrência de omissão na recepção e no envio simultaneamente.

Em respeito às características citadas acima, um determinado protocolo pode ser considerado de replicação passiva ou não. Os critérios de replicação passiva são formalizados por [5] em quatro propriedades básicas, que aqui são resumidas em:

1) Para qualquer momento, no máximo um único servidor s será considerado primário;
2) Cada cliente i estabelecerá sua comunicação com um servidor $Dest_i$, para o qual fará requisições;
3) Se a requisição de um cliente chega a um servidor que não seja considerado o primário, então esta requisição não é considerada;
4) Existem valores finitos para k e Δ no sistema, onde k é o número máximo de ocorrências de falha de serviço, e Δ é a duração máxima de *fail-over*, ou seja, do tempo que o sistema levará para a recuperação, na ocorrência de uma falha.

Um protocolo simples de replicação passiva, capaz de tolerar uma única falha é apresentado por [5]. Este protocolo satisfaz as quatro propriedades esboçadas acima. Nele, considera-se a existência de dois servidores P_1 e P_2 - servidor primário e de *backup*, respectivamente - e os seguintes passos são descritos:

1) P_1 recebe a requisição do cliente, atualizando adequadamente o seu estado interno;

2) P_1 envia uma mensagem de atualização para P_2, a fim de que este atualize seu estado interno;

3) Independentemente de receber a confirmação da mensagem por P_2, P_1 envia a resposta da requisição para o cliente.

Alguns exemplos de protocolos de replicação passiva já existentes são: o Protocolo *Alsberg and Day*, o Tandem, o HA-NFS e o Protocolo Não-Bloqueante [5].

O Banco de dados em código aberto MySQL possui uma implementação para fins de replicação de dados, baseada em princípios de Replicação Passiva. O recurso chama-se MySQL *Replication*, estando disponível no servidor de banco de dados desde a versão 3.23 [27].

3.2.3 Replicação Semi-ativa

Este tipo de replicação surgiu como um híbrido entre a ativa e a passiva, buscando suprir deficiências encontradas nas duas técnicas, visto que a primeira requer determinismo, e a segunda possui alto tempo de latência em ocorrência de falhas [48]. Nesta técnica, também conhecida como líder - seguidores, todas as réplicas recebem e processam as requisições, com a diferença que uma delas é tida como líder, que irá tomar as decisões não determinísticas, sendo estas decisões impostas aos seguidores.

As réplicas podem ser atualizadas em dois momentos: na entrega de uma requisição, ou quando ela atinge um de seus pontos de preempção (pontos nos quais a réplica precise sofrer atualizações sem haver necessariamente uma requisição).

Realizar preempção em objetos com replicação ativa é bastante complexo, já que cada réplica deve sofrer preempção no mesmo momento de processamento. Este problema, entretanto, não acontece na replicação passiva, que pode sofrer preempção a qualquer momento, já que apenas uma réplica se encontra ativa e toda sua atividade é acompanhada de *checkpoints* [33].

O conceito de preempção neste modelo é direcionado à aplicações de tempo real, que possuem o conceito de precedência de computação. Ao selecionar uma mensagem para enviar aos seguidores, o líder também gera uma mensagem de sincronização, também a ser enviada, contendo uma identificação da mensagem. A preempção pode gerar um não determinismo entre as réplicas, que é contornada através da introdução de um pequeno *overhead*, devido à utilização de pontos de sincronização. A sincronização na preempção é atingida da seguinte forma [33]:

- Existe um contador que é incrementado a cada vez que uma réplica atinge um ponto de sincronização;
- O líder, ao receber uma mensagem, faz uma checagem para saber se é preciso ou não ser preemptado. Caso a resposta seja afirmativa, o contador é incrementado de 1, indicando o próximo ponto de sincronização, e uma mensagem de sincronização é gerada e repassada aos seguidores contendo o valor do contador incrementado;
- Um ponto de preempção precisa ser executado mais freqüentemente que o *delay* máximo de preempção permitido. Com isso, se faz necessário que a parte do código que não tenha preempção seja eficiente.

Neste modelo de replicação, os seguidores são do tipo *fail-silent*, o que significa que as mensagens são entregues às réplicas seguidoras imediatamente após serem geradas. Se alguma outra réplica gera a mesma mensagem, esta é descartada pelo sistema de comunicação [33].

Os seguidores devem estar um passo atrás do líder, ou seja, atrasados em uma mensagem de sincronização em relação ao coordenador para que o mecanismo de pontos de

sincronização funcione. Para que, as réplicas seguidoras mantenham a consistência na entrega das mensagens para as outras réplicas, é necessário que haja uma sincronização da visão das réplicas sobre o grupo. Com o propósito de evitar que os seguidores se atrasem demais em relação ao líder, mensagens de '*I'm alive*' ('Estou vivo') precisam ser periodicamente enviadas aos seguidores pelo líder [48, 33].

A escolha do líder é feita estaticamente, ficando este responsável por um determinado conjunto de seguidores (ou processos). Requisições diferentes podem trocar mensagens internas para interação entre elas, sendo a ordem de interação conhecida previamente pelo líder, garantindo assim o determinismo [48, 33].

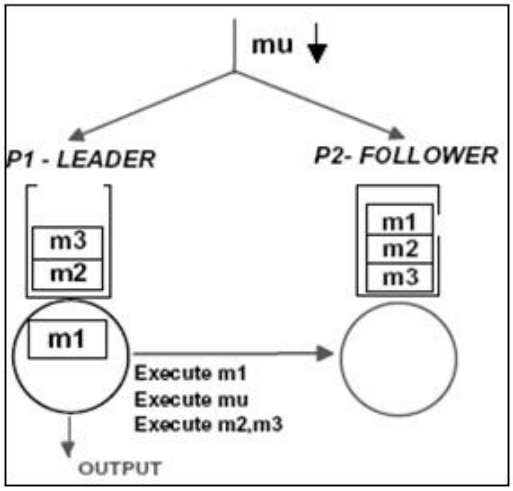

Figura 3.4. Replicação Semi- ativa [4].

A figura 3.4 mostra a replicação semi-ativa explicada anteriormente. Ao chegar uma requisição, ela vai tanto para o líder como para os seguidores, sendo apenas o líder a enviar a resposta ao cliente. A ordenação das mensagens é informada pelo líder.

Um exemplo de aplicação prática que utiliza fundamentos de replicação semi-ativa é o mecanismo *Network Database* (NDB), introduzido no banco de dados em código aberto MySQL. NDB garante aumento de confiabilidade e disponibilidade do banco de dados, possibilitando até 99,999% de disponibilidade do serviço (*fail-over* de aproximadamente 5 minutos por ano). O recurso está disponível no sistema desde a versão 5.0, para os sistemas operacionais Linux, Sun Solaris, IBM AIX, HP-UX e Mac OS X [26].

3.2.4 Replicação Semi - passiva

Na replicação passiva, caso o servidor primário (*primary*) falhe, o sistema entra em processo de votação para eleger o próximo primário, que vai então processar novamente a requisição que foi perdida no momento da falha, e reenviar a resposta ao cliente. Este tempo, no qual o sistema está fazendo o processo de reconfiguração, não passa despercebido pelo cliente, o que torna este tipo de replicação inviável para aplicações que requerem resposta dentro de um determinado tempo, que é o caso das aplicações com requisitos temporais.

A replicação semi-passiva busca manter os princípios da replicação passiva, de ter a requisição atendida por um servidor central (o *primary*) na ausência de falhas, além de preservar o não determinismo no processamento das requisições.

Propõe-se, ainda, a redução da latência como vantagem, no caso de falha do servidor central. Esta redução é alcançada através da remoção do serviço de grupo existente na replicação passiva tradicional, e na introdução do paradigma do *rotating coordinator* (coordenador rotativo) para a escolha do servidor primário [49].

No paradigma *rotating coordinator* é utilizado um protocolo de consenso denominado *Consensus with Deferred Initial Values* (Consenso com Valores Iniciais Definidos), que consiste em atualizar as réplicas (*backups*) de acordo com o valor escolhido pelo consenso. Este processo tem por objetivo decidir sobre o conteúdo da mensagem de atualização enviada às réplicas.

O consenso funciona da seguinte maneira: cada vez que um servidor receber uma requisição do cliente, um novo processo de consenso será iniciado com um valor inicial pelo servidor. Considera-se que somente o *primary* recebe e processa as requisições dos clientes, também somente ele enviará um valor inicial para o processo. Este valor é imposto como o valor de decisão, caso o servidor não falhe. Em caso de falha, outra réplica iniciará uma nova rodada com um novo valor inicial para o consenso. Em suma, o processo para escolher o novo coordenador para o consenso é o mesmo que decidirá o novo

primary. Os respectivos conjuntos das requisições recebidas e atendidas são mantidos pelo servidor, de modo que um novo processo de consenso começa quando o anterior termina e enquanto o resultado da subtração entre as requisições recebidas e atendidas não for igual a zero. Quando a rodada do consenso chega ao final para uma requisição recebida, esta passa para o conjunto das requisições atendidas.

A parte principal do algoritmo consiste na chamada à função que iniciará o consenso. Em um ambiente onde não há falhas ou suspeita de falhas dos processos, o funcionamento do algoritmo é o mesmo que a replicação passiva tradicional. A figura 3.5 ilustra este funcionamento.

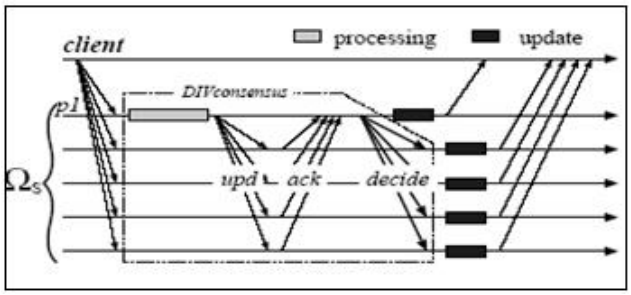

Figura 3.5. Funcionamento da técnica Semi-passiva sem falhas e/ou suspeitas de processos [49].

Na figura 3.5, o cliente envia uma requisição ao *primary*, que está indicado pelo processo P_1 para o número de consenso K. Após o recebimento desta requisição, P_1 fará o processamento necessário, tendo ao final deste um valor inicial para o consenso K. Este valor será enviado como um valor de atualização às outras réplicas (*backup*) através de envio *multicast*, esperando as mensagens de confirmação (*acks*) das réplicas. Após todos os reconhecimentos serem recebidos, P_1 decidirá pelo valor de atualização fazendo um novo envio *multicast* de decisão (*decide*), para as réplicas. Assim que a mensagem de decisão é recebida, os servidores atualizam seus estados e enviam a resposta ao cliente.

No caso da ocorrência de uma falha, ilustrado na figura 3.6, o funcionamento será da seguinte forma: o processo P_1, tido como *primary*, falha após processar a requisição recebida do cliente, mas antes de enviar a mensagem de atualização para as demais réplicas.

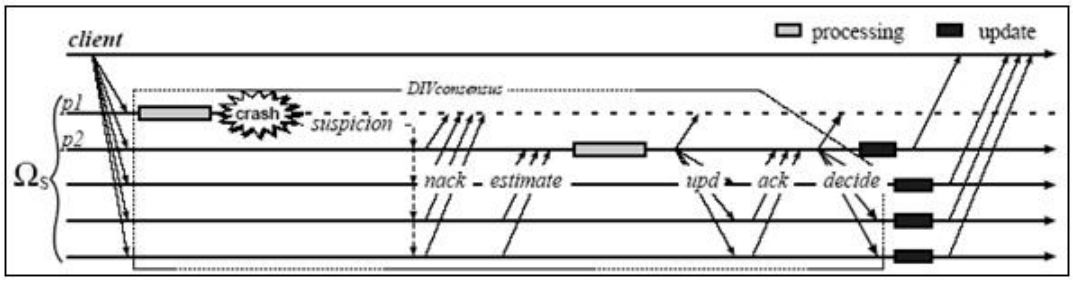

Figura 3.6. Semi-passiva no pior caso [49].

Após a suspeita de falha do *primary* pelos *backups*, um reconhecimento negativo será enviado a P_1, iniciando-se uma nova rodada. O processo servidor P_2 será o novo coordenador, e ficará esperando mensagens de estimativa - contendo um valor inicial para o consenso - da maioria dos servidores, o que evitará um re-processamento da requisição recebida. Como neste cenário P_1 falhou antes de enviar um valor de atualização para as réplicas, as mensagens de estimativas não possuem um valor inicial, obrigando o novo *primary* a processar a requisição do cliente novamente. Deste ponto em diante, os acontecimentos são os mesmos apresentados na figura 3.5.

3.2.5 Sistemas de Tempo Real

Nesta seção, irão ser apresentados métodos de replicação voltados para sistemas com requisitos temporais.

Ao mencionar a replicação para sistemas de tempo real, é necessário que seja introduzido previamente o conceito de Consistência de Réplicas. Por isso, antes de apresentar os métodos de replicação, será feita uma apresentação sobre o conceito de Consistência de Réplicas.

3.2.5.1 Consistência de Réplicas

Consistência de réplicas é uma relação existente entre objetos que fazem parte de um sistema de tempo real. Um conjunto de objetos de tempo real é dito possuir determinismo

de réplica se todos os membros deste conjunto possuem um mesmo estado externo visível, produzindo um mesmo resultado em pontos no tempo, sendo estes pontos pertencentes a um intervalo d de tempo, e sendo este tempo relativo a um mesmo relógio externo [18].

O determinismo ou consistência de réplica é necessário para que a tolerância a falhas seja implementada através de redundância ativa e para facilitar testes no sistema. Quando se tem mais de uma trajetória para uma mesma computação, podem ocorrer inconsistências de valores finais, e, com isso, erros podem ser cometidos pelo sistema. Então, se o determinismo for implementado, elimina-se a possibilidade de haver uma incoerência de resultado final.

Quando se fala em facilidade para testar a funcionalidade do sistema, é devido à propriedade do determinismo de produzir um mesmo resultado, no mesmo intervalo de tempo, para um conjunto de dados de entrada idênticos a todas as réplicas. Se todos os resultados forem divergentes, não existiria possibilidade de testar a corretude do sistema [18].

Na figura 3.7, tem-se um exemplo da importância da consistência de réplicas para um sistema real. Na figura, tem-se um sistema para avaliar que decisão tomar em um avião: se acelera ou reduz a velocidade dos motores. Cada motor possui seus próprios sensores e computadores. Dentre as decisões tomadas pelos três computadores existentes, a ação final a ser tomada será aquela decidida pela maioria.

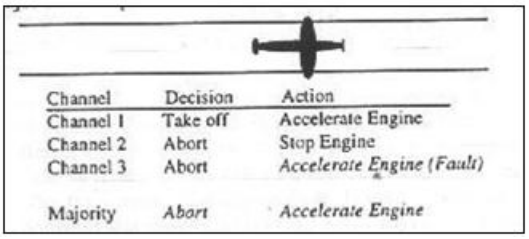

Figura 3.7. Necessidade de consistência de réplicas [18].

Interpretando os dados da figura 3.7, temos que o primeiro computador toma a decisão de levantar vôo (*take off*), acelerando o motor (*accelerate engine*), o segundo aborta a decolagem (*abort*), parando o motor (*stop engine*) e, o terceiro, aborta a decolagem,

porém, devido a uma falha no sistema, a ação tomada é de aceleração do motor. Então, já que o que prevalece é a decisão majoritária, será tomada erroneamente a decisão de acelerar motores. Se houvesse neste sistema o determinismo de réplicas, esta decisão equivocada não seria tomada, já que todas as réplicas iriam alcançar um mesmo valor final.

Existem algumas causas para a ocorrência de não - determinismo de réplicas [18]:

- **Entradas diferentes** - Uma conversão errada de valores analógicos para digital ou uma percepção errada de um operador de um sistema pode ser a causa deste fator. Se, numa conversão analógico-digital, um valor sofre a variação de um bit, isso já se torna motivo para que duas réplicas achem valores divergentes, e, consequentemente, entradas diferentes para um mesmo sistema sejam produzidas. Se dois operadores estão analisando um mesmo evento e este ocorre em ordem diferente para os dois trabalhadores, haverá também uma discordância de valores;

- **Diferença entre relógios locais nas réplicas e o progresso da computação** - Existem algumas ações em sistemas de tempo real que são disparadas em um determinado instante do tempo. Este tempo é referido ao relógio interno do sistema, que deve estar sincronizado com o relógio existente na máquina. Quando as freqüências entre estes dois relógios são diferentes, podem ocorrer divergências entre o tempo atual no computador e as tarefas executadas no sistema em certo ponto do tempo;

- **Desvio de oscilação** - A freqüência do relógio de um computador se dá devido a um oscilador físico: o cristal de quartzo. Como a dimensão entre dois cristais de quartzo são ligeiramente diferentes, não há dois cristais que tenham o mesmo desvio. Este desvio de tempo em duas réplicas distintas interfere em saídas que envolvem o tempo local da réplica;

- **Escalonamento preemptivo** - Ao utilizar escalonamento preemptivo, o momento em que o processamento for paralisado por uma interrupção pode ser diferente entre as réplicas, o que ocasiona uma visão diferente de estados do sistema nas diferentes

réplicas;

- **Características não-determinísticas de linguagem de programação** - Quando algum comando não-determinístico da linguagem de programação do sistema é utilizado, duas computações distintas podem tomar rumos diferentes em certo ponto. Isso ocasiona tomadas de decisões diferentes pelas réplicas;
- **Condições de corrida** - Um comando pode dar margem ao não-determinismo quando não se sabe onde este comando será interrompido. Por exemplo, um comando *wait* em C pode ser interrompido por mais de uma *thread* que pertença ao sistema;
- **Problema da comparação consistente** - Se duas versões diferentes são implantadas em réplicas distintas, é importante que a ordem das operações seja a mesma para evitar uma discordância no resultado final da operação, reduzindo o problema da diferença entre as versões implantadas.

Como não se pode utilizar o relógio local da réplica, pois já foi dito que sua utilização pode ocasionar inconsistência, a solução é utilizar um tempo global para todas as réplicas. Os eventos do sistema que são baseados em tempo utilizarão deste tempo global, evitando a inconsistência.

Um exemplo de como a utilização de tempos locais no sistema pode ser prejudicial está ilustrado na figura 3.8.

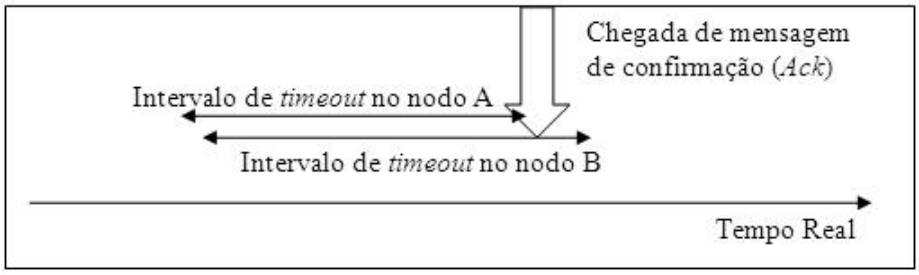

Figura 3.8. Utilização de Relógios Locais nas Réplicas [18].

Nesta figura 3.8, têm-se duas réplicas mandando mensagens uma para a outra, sendo que as duas estão esperando uma resposta de confirmação de recebimento. Se os relógios

locais são utilizados, ao invés de um tempo global do sistema, uma réplica pode receber a mensagem após o seu tempo determinado de *timeout*, o mesmo acontecendo com a réplica que espera a confirmação. Este problema pode ocasionar em tomadas de decisões diferentes nas duas réplicas, gerando uma inconsistência.

A respeito de dados de entrada, é preciso que haja uma concordância entre as réplicas para que o cenário de dados de entrada em um mesmo ponto no tempo seja o mesmo. A concordância no tempo é fundamental para que se tenha uma ordenação de eventos em todas as réplicas.

Outro fator importante é a implementação do *software* do sistema que deseja utilizar consistência de réplica com as estruturas de controle corretas. Para isso, uma implementação independente de dados de entrada e que possa ser validada independente da entrada é importante.

É importante que algoritmos que podem causar não-determinismo sejam evitados, como, por exemplo, sincronizações dinâmicas que podem levar a uma condição de corrida (*race condition*).

3.2.5.2 Replicação Proativa

Caracteriza-se replicação proativa aquela em que há uma antecipação à ocorrência de falhas no sistema. Esta antecipação é feita através da técnica de predição de falhas (*fault-prediction*). Com esta técnica de replicação o esforço para recuperação de uma falha é reduzido, pois uma rotina reativa para recuperação de falhas é utilizada antes que a falha aconteça de fato. Uma das vantagens deste tipo de replicação é que, se há o conhecimento pelo sistema que uma falha está prestes a acontecer, as requisições que chegarem dos clientes poderão ser encaminhadas para outras réplicas, evitando assim a perda da requisição e, consequentemente, sua retransmissão. Na recuperação proativa, existem duas coisas importantes a serem levadas em conta [34]:

- Quando executar a rotina de recuperação;

- Como propagar a notificação da recuperação proativa para os demais *hosts* do sistema.

Se o tempo de execução entre uma recuperação e outra for muito curto, pode haver um *overhead* adicional grande no sistema, o que negaria o uso deste método de replicação. Mas, por outro lado, se um tempo muito longo é definido entre uma execução e outra, o propósito da antecipação às falhas é perdido, já que pode haver mais falhas que o tolerado pelo sistema, o que faria apenas a utilização do método reativo [34].

A propagação da recuperação de falha para os outros nós no sistema depende muito da topologia adotada e da técnica de recuperação de falhas escolhida. O ponto ideal entre quando executar e como propagar a notificação está ilustrado na figura 3.9, e pode ser atingido através dos seguintes mecanismos de aperfeiçoamento (*tunning*): estratégia de notificação (*notification strategy*), estratégia de recuperação (*recovery strategy*), tempos de serviço (*service times*) e modelo de falhas (*fault model*).

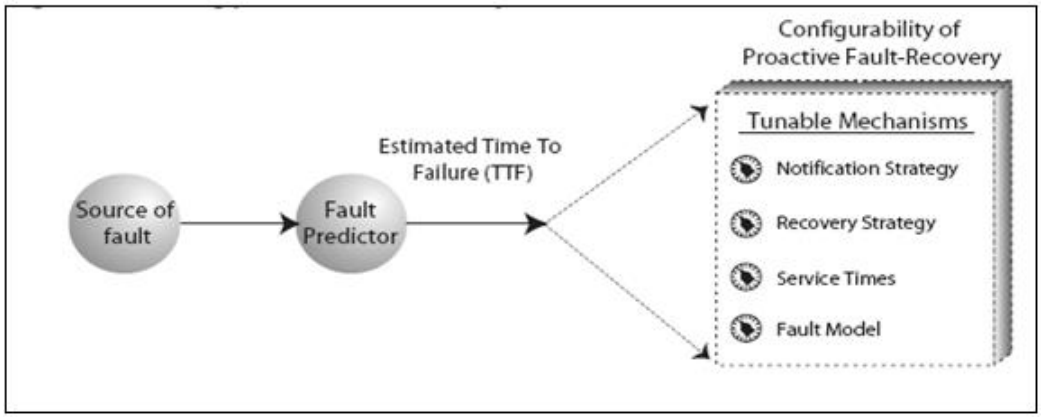

Figura 3.9. Balanço entre quando executar e como propagar a execução pro-ativa [34].

Para determinar quando executar a rotina de recuperação proativa, é necessário atentar para dois aspectos: (a) qual o ponto exato no tempo em que a recuperação deve ser iniciada; (b) quão freqüente a recuperação deve ser iniciada. Três parâmetros precisam ser determinados para que a recuperação proativa possa ser executada: o tempo entre falhas (*Time-to-Failure*) para o sistema, o tempo do pior caso de execução da requisição no servidor e o tempo de recuperação de falhas proativo. Uma vez determinadas estas

constantes, pode-se evitar o *overhead* desnecessário com execuções muito próximas umas das outras, por exemplo. Estes tempos são estimados levando em consideração falhas anteriormente ocorridas no sistema.

3.2.5.3 *Resilient State Machine Replication*

Esta técnica consiste em verificar periodicamente os recursos disponíveis no sistema de tempo real, executando uma manutenção de revitalização no sistema sempre que for necessário, evitando assim uma falha crítica. Quando falamos nesta técnica, precisamos estabelecer um número determinado de falhas possíveis a serem toleradas. Mas, em sistemas assíncronos, este número não é bem definido, o que leva a um conceito de janela de vulnerabilidade, na qual um número indefinido de falhas pode ser suportado pelo algoritmo. Exaustão de recursos em um sistema pode ser definida tanto como escassez de recursos físicos (memória, H.D., etc), quanto de recursos de *software*. Sendo assim, a exaustão de recursos é o estado no qual o sistema não possui os recursos necessários para o seu correto funcionamento [44].

Um sistema distribuído replicado geralmente se encontra em ambientes com pouca segurança, como a Internet, onde fica difícil determinar se uma das réplicas se encontra apenas lenta ou faltosa. Para que este problema seja suportado pelo algoritmo de detecção de falhas, uma rotina de rejuvenescimento do sistema é executada periodicamente,tornando-se possível garantir a não ocorrência de mais de um número X de falhas, sendo X um número qualquer determinado para segurança do sistema. Ao falar de exaustão de recursos, é necessário que exista uma medida a partir da qual se possa afirmar se um recurso se encontra escasso ou não para o sistema. Não podemos garantir o correto funcionamento de um sistema que está no limite dos recursos disponíveis [44].

Se um recurso é dito como *exhaustion-safe*, significa que o mesmo se encontra em um estado de não escassez, ou seja, de segurança para sua disponibilidade.

A recuperação proativa de um sistema é definida como um componente designado a

executar uma ação preventiva de recuperação de funções ou partes deste sistema. Este módulo ou componente, denominado *Proactive Recovery Wormhole* (PWR), se encontra dentro da réplica que se deseja aplicar a recuperação pro-ativa, porém separado do sistema operacional (em uma memória Eprom, por exemplo).

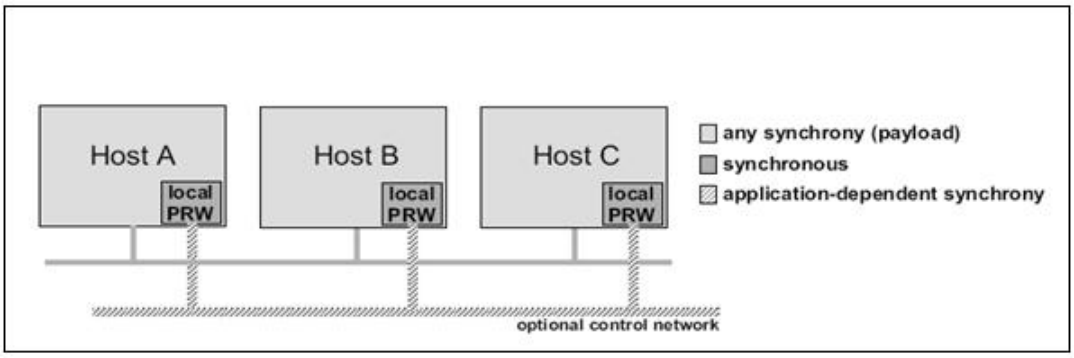

Figura 3.10. Arquitetura de um sistema usando PWR [44].

A figura 3.10 ilustra um sistema que está utilizando o módulo de PWR para recuperação pro-ativa. Cada uma das réplicas - os *hosts* da figura - possui seu próprio módulo PWR, separado do sistema operacional, que, de tempos em tempos, irá realizar uma rotina de checagem de procedimentos vitais ao sistema. O PWR executará uma rotina de checagem do sistema em um tempo bem definido. Caso a lista de checagem tenha falhado em algum ponto, o PWR pode desligar o sistema ou emitir algum aviso de alerta.

3.3 REPLICAÇÃO UTILIZANDO COMPONENTES

A replicação para componentes se dá com base na separação da interação entre aplicação e replicação. Esta separação é feita através de um modelo arquitetural, pois replicação e consistência estão relacionadas com características arquiteturais [46]:

- Replicação cria e distribui os componentes;
- Consistência gerencia as interações entre as réplicas (componentes).

Através da utilização de uma arquitetura de aplicação, é possível obter um maior controle sobre aspectos importantes para uma aplicação, como um maior gerenciamento de protocolo e o reuso de componentes e reconfiguração, coisas antes não possíveis de se obter com uma aplicação não estruturada em forma de arquitetura [46].

Uma aplicação, neste modelo explicado, é um conjunto de componentes interconectados através de pontos de entrada (acessados pela interface *required*) e pontos de saída (acessados pela interface *provided*).

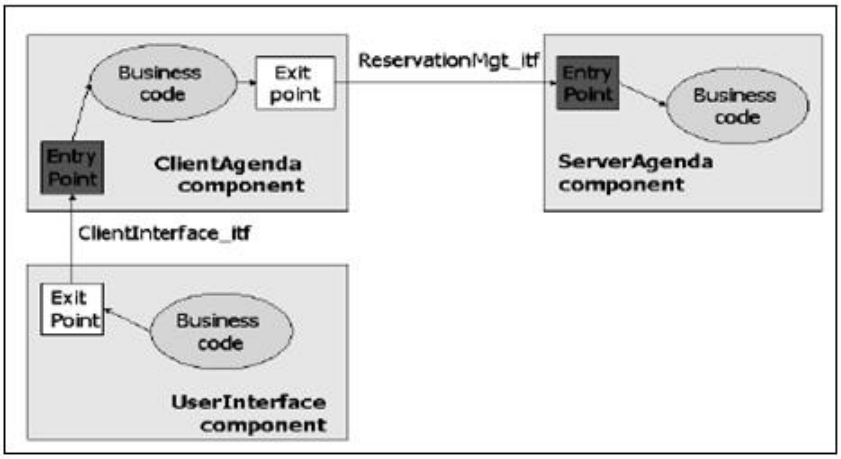

Figura 3.11. Aplicação baseada em componentes [46].

A figura 3.11 ilustra uma aplicação baseada em componentes, onde cada módulo do sistema é um componente interconectado a outro, completando o ciclo da aplicação. No exemplo acima, tem-se uma aplicação de agenda telefônica na qual o usuário, após conectar-se com um servidor através de uma interface gráfica, pode alterar, incluir ou excluir dados da agenda.

Como demonstrado na figura 2.6, capítulo 2, os componentes estão interconectados através dos respectivos pontos de entrada (*Entry point*) e pontos de saída (*Exit point*).

A interação entre os componentes pode ser classificada em três grupos, de acordo com [46]:

- **Dedicada à replicação** (*replication-dedicated*) - Este tipo de interação está associada com a criação de conexão, ações relacionadas com a criação de réplicas, sua

localização e destruição. Também fica responsável pela notificação de um grupo sobre a criação de uma nova cópia;

- **Orientada ao gerenciamento de estados** (*state-management oriented*) - Esta interação está associada à manipulação de estados durante a criação de uma nova réplica ou sua sincronização;
- **Dedicada à consistência** (*consistency-dedicated*) - Tem o intuito de prover corretude no provimento de funcionalidades fornecidas por uma nova cópia em sua sincronização. Esta corretude é checada de acordo com alguns critérios tidos como corretos.

Quando se fala de replicação baseada em componentes, os objetivos principais são [23]:

- **Reuso de código de replicação ou parte dele (componentes)** - Permite que a aplicação possa utilizar diferentes métodos de replicação, sem ter a necessidade de modificação do seu código de negócio;
- **Reuso de código de gerenciamento de replicação** - Permite que o mesmo protocolo de gerenciamento de replicação possa ser usado por diferentes aplicações, de acordo com suas necessidades.

3.3.1 Modelo de Replicação baseado em Componentes

O modelo de replicação baseado em componentes pode ser subdividido em três modelos: modelo de aplicação, modelo de protocolo de replicação e o modelo de composição [23].

- Modelo de Aplicação

Uma aplicação é definida, no modelo baseado em componentes, como um conjunto de componentes interconectados através dos seus serviços requeridos e fornecidos. Um

Gestor de Replicação se caracteriza por ser uma aplicação, por isso a justificativa para este tópico.

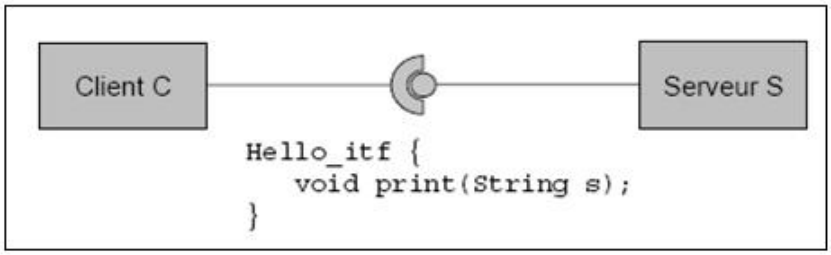

Figura 3.12. Exemplo de aplicação baseada em componentes [23].

A figura 3.12 ilustra um exemplo de aplicação baseada no modelo de componentes, onde um componente do tipo Cliente se conecta, através das interfaces dos serviços fornecidos e requeridos, ao componente do tipo Servidor.

- Modelo de Protocolo de Replicação

O modelo de protocolo de replicação define quais serão os tipos de componentes que irão efetivamente fornecer os serviços necessários para uma aplicação voltada ao gerenciamento de replicação. Uma possível divisão de componentes, proposta por [23], é:

- **Componente de cópia** - É a unidade de replicação responsável por copiar ou replicar um componente;
- **Componente cliente** - Representa um componente capaz de interagir com componentes replicados. Refletem a capacidade de modificação das cópias por seus clientes e da sincronização dos objetos replicados;
- **Componente de serviço** - Provê serviços genéricos, sendo independentes da semântica de um protocolo específico, podendo ser utilizado por diferentes protocolos;
- **Gerenciadores** - Gerencia a arquitetura do protocolo, conhecendo as instâncias dos componentes e suas conexões. Faz as ações necessárias para o correto gerenciamento das cópias, como criação de novas réplicas ou reconfiguração das mesmas.

- Modelo de Composição

Este modelo é responsável pela interconexão entre o modelo de aplicação e o de protocolo de replicação, ou seja, une o esqueleto da aplicação de gerenciamento de replicação com os componentes responsáveis pelo fornecimento dos serviços necessários para o funcionamento do *software*. O produto deste modelo não leva em consideração os fatores da aplicação, como consistência ou aspectos específicos do método de replicação escolhido. Seu único objetivo é por os dois módulos anteriores para trabalharem conjuntamente, compondo uma aplicação funcional [23].

Um exemplo deste modelo é relacionar uma simples aplicação *Hello World* com um protocolo de tolerância a falhas, onde este modelo será o responsável pela interação entre aplicação e protocolo.

3.4 COMPARATIVO ENTRE AS TÉCNICAS DE REPLICAÇÃO

Uma tabela comparativa entre as Técnicas de Replicação Ativa, Passiva e Semi-Ativa nos qualitativos de *overhead* do processamento de erro, não-determinismo da réplica e comportamento em falha arbitrária é apresentada por [43]. Aqui é apresentada uma tabela, adaptada de [43], acrescentando-se a técnica de Replicação Semi-passiva, com base em [49], e também um comparativo de *overhead* em funcionamento normal. O comparativo leva em consideração modelos pessimistas.

Tabela 3.1. Comparativo entre as Técnicas de Replicação, adaptado de [43, 49].

Técnica de Replicação	*Overhead* do processamento de erro	*Overhead* em funcionamento normal	Não-determinismo da réplica	Comportamento em falha arbitrária
Ativa	Mais Baixo	Alto	Proibido	Tolerado
Passiva	Mais alto	Mais Baixo	Permitido	Proibido
Semi-ativa	Baixo	-	Resolvido	Proibido
Semi-passiva	Baixo	-	Resolvido	Tolerado

O estudo sobre as técnicas levou à conclusão de que não há uma técnica sempre melhor do que as outras, mas sim uma técnica mais adequada para cada propósito, em termos

de custo de processamento, de número de réplicas e modelo de consistência requerido.

3.5 TRABALHOS CORRELATOS

Nesta seção serão apresentados alguns trabalhos correlatos ao trabalho ora proposto. Estes servirão como base para a modelagem e desenvolvimento do objetivo final desta dissertação: a implementação de um Gestor de Réplicas.

3.5.1 *Adaptive Quality of Service Availability* (AQuA)

O *framework Adaptive Quality of Service Availability* (AQuA) é voltado para o provimento das requisições de qualidade de serviço das aplicações que dele fizerem uso. Este *framework* utiliza *Quality Objects* (QuO) *runtime* para processar as requisições e torná-las disponíveis. Um objeto QuO tem por objetivo permitir que aplicações distribuídas orientadas a objeto possam especificar requisitos de QoS dinamicamente. Dentro do AQuA, o QuO tem por missão transmitir as necessidades de disponibilidade ao Proteus, que é responsável por prover mecanismos de tolerância a falhas no AQuA (replicação), além de votadores e monitores [35].

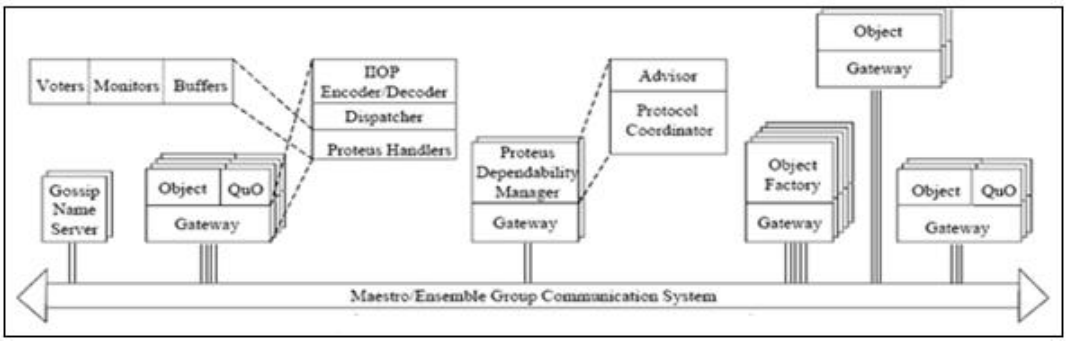

Figura 3.13. Arquitetura do AQuA [35].

A figura 3.13 ilustra a arquitetura do AQuA. Os componentes ilustrados nesta figura podem ser utilizados de acordo com a necessidade de cada *host* e de acordo com a disponibilidade requerida de cada um dos objetos remotos que por ele é utilizado. Um objeto

do tipo QuO é associado a outro objeto que gere requisições remotas. Cada QuO pode gerenciar um ou mais contratos QuO. Um QuO se comunica através de CORBA com o Proteus para obter a disponibilidade requerida de objetos remotos. A comunicação entre todos os componentes do AQuA é feita através de *gateways*, que traduzem as invocações CORBA para mensagens que são transmitidas via *Ensemble*.

A pilha de protocolo *Ensemble* tem a função de prover a comunicação entre processos baseado no modelo de sincronismo virtual. A replicação no AQuA é feita através de grupos de replicação, nos quais réplicas são inseridas ou retiradas (em caso de falhas). Cada grupo de replicação possui um líder, responsável por receber as requisições endereçadas ao grupo e repassá-las às demais réplicas. O resultado do processamento é, então, retornado a este líder, que fica incumbido de retornar esta ao cliente.

Na arquitetura do AQuA, existem duas técnicas de replicação implementadas: a ativa e a passiva. Apesar de todas as réplicas processarem as requisições recebidas no método ativo, existem diferentes esquemas de comunicação implementados: 1) Apenas o líder retorna o resultado da requisição; 2) Todas as réplicas retornam o resultado da requisição, sendo a primeira resposta selecionada; 3) Todas as réplicas respondem e uma eleição é feita. Quando existe o processo de votação, falhas de tempo, de valor e por parada (*crash*) podem ser toleradas [35].

A comunicação entre grupos de réplicas é feita através de grupos de conexão. Esta conexão acontece entre dois grupos de replicação que necessitam se comunicar. O principal objetivo de um grupo de conexão é prover as propriedades de comunicação em grupo (*multicast* atômico e ordenação total ou parcial de mensagens) quando há comunicação entre grupos de replicação [35].

3.5.2 Chameleon

Chameleon é uma infra-estrutura de *software* adaptável, capaz de prover níveis diferentes de disponibilidade em um ambiente distribuído. A arquitetura do Chameleon é

baseada no conceito de ARMOR - *Adaptive* (Adaptação), *Reconfigurable* (Reconfiguração) *and Mobile Objects for Reliability* (e dispositivos móveis para confiabilidade). ARMOR é classificado em três grupos, de acordo com [3]:

- **Gerenciadores** (*managers*) - Responsáveis pela monitoração de outros ARMORS e recuperação em caso de falhas em algum subordinado;
- **Demônios** (*daemons*) - Responsáveis pela ponte de comunicação entre os ARMORS em um determinado *host*. Suportam detecção de erros;
- **ARMOR comum** (*common ARMOR*) - Implementa técnicas específicas para prover necessidades pontuais das aplicações.

Qualquer rede que possua *hosts* não confiáveis pode se conectar ao ambiente do Chameleon. O funcionamento deste ambiente se dá da seguinte forma: ao ser inicializado, um gerenciador de tolerância a falhas, denominado *Fault Tolerance Manager* (FTM) é instalado em um nodo arbitrário da rede. Feito isso, um *deamon* é invocado para prover comunicação remota com outros *hosts*, além de um *heartbeat* ARMOR, que será responsável por detectar falhas em *hosts* remotos. O último passo é a criação de um *Backup* FTM, que será responsável por substituir o FTM em caso de falhas do mesmo. As configurações necessárias por cada aplicação são informadas ao Chameleon através de uma linguagem que este possa compreender. A política de provimento de tolerância a falhas será escolhida pelo FTM, de modo que as necessidades da aplicação possam ser atendidas. A replicação no Chameleon é feita através do auxílio dos seguintes Common ARMOR, de acordo com [3]:

- *HeartBeat* **ARMOR** - Invocado temporariamente pelo FTM para checar o *status* dos nodos do ambiente. Envia pequenas mensagens aos nodos, que devem responder ao recebimento destas mensagens. Um nodo é considerado faltoso caso não responda a um determinado número de mensagens;

- *Execution* **ARMOR** - Tem o papel de instalar a aplicação em um determinado *host*, monitorar sua execução e enviar o resultado do monitoramento para o gerenciador (FTM);

- *Checkpoint* **ARMOR** - Interage com o *Execution* ARMOR para, temporariamente, gravar pontos de recuperação (*checkpoint*) de uma aplicação rodando em um determinado *host*;

- *Voter* **ARMOR** - Utilizado para escolher um valor dentre um determinado grupo de valores de entrada. Este mecanismo é utilizado para detectar falha por valor;

- *Initialization* **ARMOR** - Responsável por coletar informações de *hardware* e *software* do *host* que irá hospedar a aplicação;

- *Fanout* **ARMOR** - Responsável por prover determinismo entre as réplicas do sistema. Este objetivo é alcançado através da distribuição consistente de dados entre as réplicas. Utiliza um protocolo de ordenação total e de *commit*, garantindo que todas as atualizações serão processadas na mesma ordem por todas as réplicas do sistema.

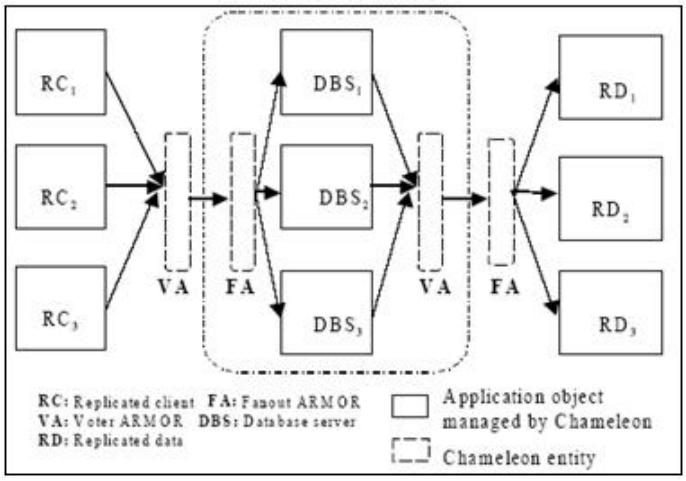

Figura 3.14. Exemplo de aplicação do Chameleon para replicação [3].

A figura 3.14 ilustra um exemplo de aplicação do Chameleon para prover replicação para uma aplicação de banco de dados. Neste exemplo, um cliente está tentando atualizar

um banco de dados replicado. Neste exemplo, são replicados servidor de banco de dados, as *querys* dos clientes e o próprio banco de dados. Para que os dados sejam recebidos e processados por todas as réplicas em uma mesma ordem, o *fanout* ARMOR entra em ação, mantendo a consistência e o determinismo de réplica.

3.5.3 Piranha

Piranha [22] é um serviço orientado a objetos, implementado em CORBA, que tem por finalidade monitorar a rede na qual esteja inserido, exibindo as falhas encontradas em uma interface gráfica. Além deste monitoramento, Piranha também é responsável por criar e gerenciar réplicas na rede.

Tem por base o ORB Electra, que provê funcionalidades de gerenciamento de grupo de objetos, *multicast* confiável, transferência de estado e sincronismo virtual. Através do Piranha, objetos, que são mantidos em grupos, podem ser replicados utilizando as técnicas de replicação passiva ou ativa. O aumento ou diminuição da disponibilidade de um objeto replicado é feito através do crescimento ou exclusão do número de réplicas daquele objeto no sistema.

Quando um objeto ou grupo de objetos é excluído por falha, novo objeto ou grupo é criado pelo Piranha em substituição ao que foi removido. Caso a máquina não tenha falhado, a criação dos novos objetos é feita na mesma máquina onde estes estavam anteriormente à falha. Caso contrário, o grupo ou o objeto será recriado em um novo *host* disponível. Outra funcionalidade provida pelo Piranha é a de atualização *on-line* de objetos. Este procedimento é feito sem que haja uma degradação de desempenho dos serviços providos pelo objeto ou grupo de objetos, levando em consideração que não há mudança na interface do mesmo [22].

3.5.4 *Replication Management System* (RMS)

O Sistema Gerenciador de Replicação (*Replication Management System* - RMS) [21] tem como objetivo replicar objetos em uma rede garantindo disponibilidade (*availability*) e performance, através do provimento de QoS para cada um dos objetos replicados.

O número e a localização das réplicas são parâmetros configuráveis pelo programador que deseje replicar objetos através do RMS. Com a capacidade de controlar dinamicamente as réplicas criadas, o RMS pode obter informações como carga na rede, interações com usuários, etc, à medida que a aplicação executa, possibilitando uma melhor performance que uma intervenção manual.

O RMS leva em consideração duas medidas de confiabilidade de um nodo (*host*): *Mean Time to Failure* (MTTF) e *Mean Time to Recovery* (MTTR). O primeiro se refere ao tempo entre falhas de um componente, enquanto que o segundo ao tempo necessário de recuperação de falhas. Estes dois valores podem ser obtidos através dos componentes do sistema ao passar do tempo e durante a execução da aplicação.

O RMS utiliza ambas técnicas de replicação passiva e ativa. A técnica de replicação passiva é utilizada para a replicação do próprio RMS em diversos *hosts*, evitando, assim, um único ponto de falhas, enquanto que a técnica de replicação ativa é utilizado na replicação e gerenciamento das réplicas criadas pelo RMS.

Um dos principais componentes do RMS é o *Monitor Daemon*, que tem por função monitorar e armazenar alguns dados importantes como hora e data, tráfego na rede, etc, em um intervalo periódico. Estes dados serão úteis na recuperação de algum nodo, momento que serão examinados e calculados os valores do MTTF e MTTR [44].Também será possível obter as informações sobre o tipo de falha sofrida pelo nodo (se *crash* ou um desligamento programado, por exemplo).

Outro componente existente no RMS é o *Dependency Tracker*, que é responsável por coletar todas as informações provenientes dos diversos *Monitor Daemons* e mostrar a relação de dependência entre os componentes existentes no ambiente distribuído. O

módulo *Placement Policy* é responsável pelas informações de número e localização das réplicas. Conflitos entre performance e disponibilidade de réplicas podem ser solucionados através de uma localização da réplica que possa maximizar a performance, dentro de um determinado limite.

A arquitetura do RMS está ilustrada na figura 3.15.

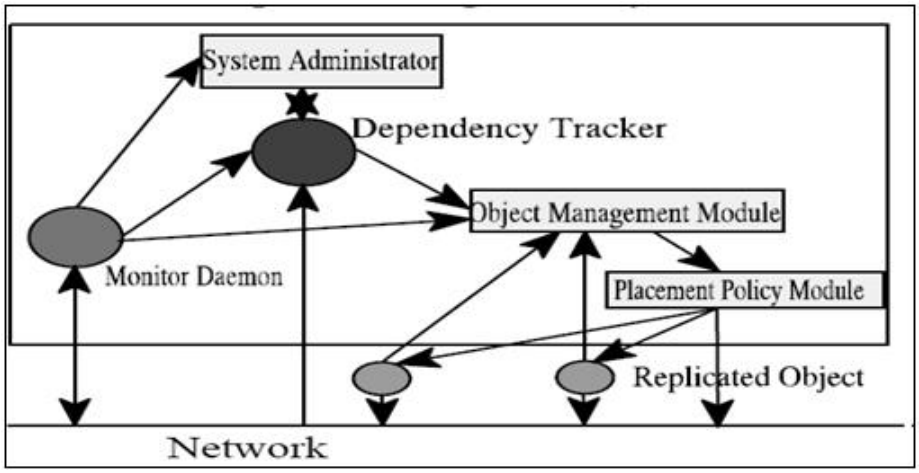

Figura 3.15. Arquitetura do gestor RMS [21].

A figura 3.15 ilustra a relação entre os componentes *Monitor Daemon*, *Depency Tracker* e os objetos replicados pelo RMS. Enquanto o *Monitor Daemon* é responsável por monitorar os objetos na rede, o *Dependency Tracker* recolhe os dados coletados pelo *Monitor Daemon*, gerenciando a relação de dependência entre os componentes existentes na rede. As informações coletadas pelo *Dependency Tracker* são, então, repassadas ao módulo gerenciador de objetos (*Object Management Module*), que irá gerenciar a política de replicação, recebendo também informação do módulo de política de localização (*Placement Policy Module*), que, por sua vez, é responsável pelas informações de quantidade e localização das réplicas.

3.5.5 *Distributed Object-Oriented Reliable Service* (DOORS)

DOORS [10] é um *middleware*, protótipo de implementação da especificação FT-CORBA [31], com o objetivo de prover tolerância a falhas a sistemas CORBA. Este

middleware provê a replicação de objetos CORBA, sendo composto por detectores de defeito e gerenciadores de replicação. Utiliza a técnica de replicação *warm_passive*, definindo uma réplica primária, como na replicação passiva, e pontos de checagem (*chekpoints*). O usuário pode selecionar a técnica de monitoramento a ser utilizada dentre a *heartbeat* ou *polling*.

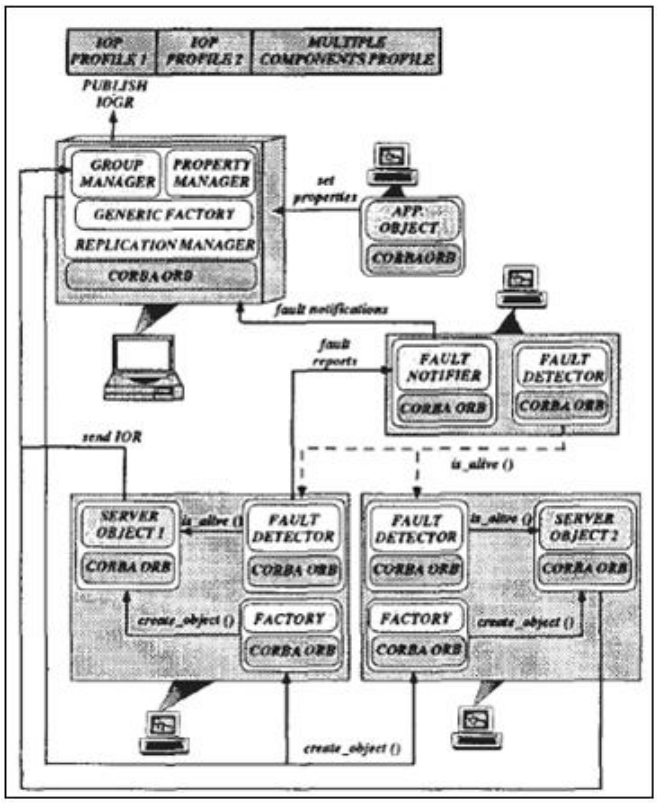

Figura 3.16. Componentes da arquitetura do *middleware* DOORS [10].

A figura 3.16 ilustra a arquitetura de componentes do DOORS. O componente *ReplicationManager* encapsula grande parte do gerenciamento de configuração e grupo, definidos pelo padrão FT-CORBA. Os componentes *FaultDetector* e *SuperFaultDetector* dão suporte à detecção de falhas hierárquica, além de agir como notificadores de falhas, propagando o aviso de detecção de falhas ao *ReplicationManager*. O *FaultDetector* é responsável pela detecção de falhas a nível de objetos, enquanto que o *SuperFaultDetector* se encarrega do monitoramento das falhas a nível de *hosts*.

O DOORS não implementa um notificador de falhas, sendo que as funcionalidades de

detecção e notificação de uma falha ficam a cargo dos detectores de falhas [10].

3.5.6 FTWeb

O FTWeb [37] é uma infraestrutura voltada para prover tolerância a falhas para aplicações baseadas em *webservices*, através da replicação de objetos CORBA, distribuídos em domínios diferentes. Assim como o DOORS, apresentado na subseção anterior, é baseado na especificação FT-CORBA. Organiza as réplicas em grupos e possui alguns parâmetros configurados via arquivo XML, como, por exemplo, a técnica de replicação selecionada, o estilo e intervalo de monitoramento das réplicas, o tempo máximo de resposta para uma requisição e o indicador de recuperação do serviço que utiliza a replicação. Implementa as técnicas de replicação ativa, *cold passive* e *hot passive*.

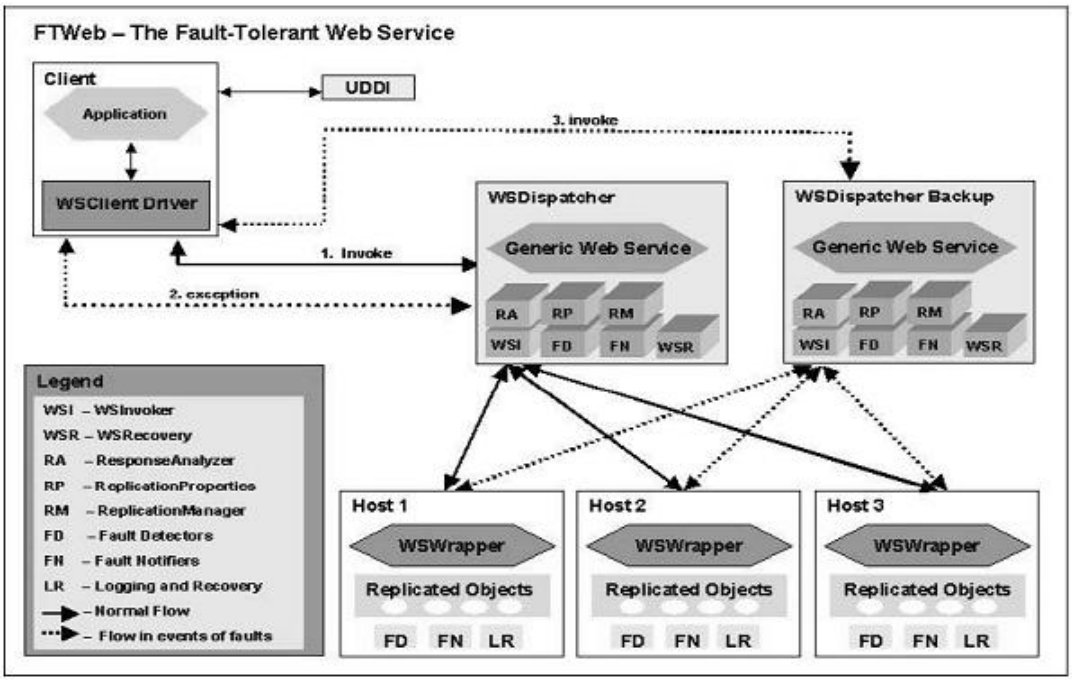

Figura 3.17. Arquitetura do FTWeb [37].

A figura 3.17 ilustra a arquitetura do FTWeb tanto em funcionamento normal do *WSDispatcher* quanto o fluxo na ocorrência de falhas deste componente. O componente *WSClient Driver* é responsável pela detecção de falhas no componente *WSDispatcher*

Engine e repasse do processamento das requisições para o componente *WSDispatcher Engine Backup*, localizado em um servidor separado. O objetivo desta visão de arquitetura é prover replicação de forma transparente à aplicação cliente, além de evitar que o componente *WSDispatcher Engine* se torne um ponto único de falhas no sistema. Em caso de falhas do *WSDispatcher*, o componente *WSClient Driver* irá transferir a requisição ao componente *WSDispatcher Engine Backup*, que irá invocar os serviços replicados.

Cada réplica possui um mecanismo de *log*, pelo qual é possível identificar o processamento ou não de uma requisição. Com o processamento com sucesso, a réplica retorna a resposta deste ao componente *WSDispatcher Engine Backup*.

O componente *ReplicationManager* estende as funcionalidades do gerenciamento de réplicas provido pelo FT-CORBA para *webservices*. O controle da adição e remoção de réplicas é feito dinamicamente e segue os parâmetros definidos no arquivo XML de configuração das propriedades de replicação. O FTWeb implementa a técnica de replicação ativa, na qual todas as réplicas não falhas recebem e processam as requisições do cliente.

CAPÍTULO 4

GESTOR DE RÉPLICAS (GESREP)

O presente trabalho se destina a propor e implementar um Gestor de Réplicas (GesRep) que irá fornecer o serviço de replicação de componentes à aplicações baseadas no modelo de componentes CIAO [33,34]. Como estudo de caso, será utilizado o *framework* ARCOS [1, 2], que atende à exigência anteriormente citada.

O propósito do GesRep é criar e gerenciar réplicas dos componentes de uma aplicação baseada em componentes, de modo que as réplicas mantenham consistência do estado interno entre si. O estado interno de um componente compreende as variáveis globais da(s) classe(s) executora(s) deste componente. O Gestor de Réplicas neste trabalho proposto, além destas funcionalidades básicas anteriormente citadas, disponibiliza a possibilidade de reconfiguração em tempo de execução da quantidade mínima de réplicas nos grupos inicialmente criados. Este Gestor de Réplicas é voltado para aplicações de tempo real *soft* (*soft real-time systems*).

A replicação vem, por sua vez, garantir uma maior disponibilidade do componente replicado, já que réplicas devem ser criadas de maneira distribuída na rede. Uma vez que ocorra falha da réplica que esteja fazendo o processamento das requisições, esta será substituída de modo transparente ao usuário da aplicação, garantindo a continuidade do processamento das requisições. Qualquer componente é passível de ser replicado pelo GesRep, contanto que o mesmo atenda às configurações necessárias para utilização do serviço de replicação, configurações estas que serão descritas nas seções a seguir.

Este capítulo será dividido da seguinte forma: a seção 4.1 apresentará a infraestrutura do GesRep; a seção 4.2 trará os detalhes para que uma aplicação construída sobre a arquitetura de componentes possa utilizar o serviço de replicação fornecido pelo GesRep; a seção 4.3 apresentará o processo de implantação do GesRep e da aplicação cliente no CIAO e, a seção 4.4, apresentará os experimentos e resultados obtidos.

4.1 INFRAESTRUTURA

4.1.1 Arquitetura

A arquitetura do GesRep é baseada em componentes, que são integrados entre si através de facetas e receptáculos. O Gestor foi totalmente construído em cima do conjunto ACE,TAO, CIAO, apresentado no capítulo 2, incorporando as características de tempo real herdadas do CIAO.

O Gestor de Réplicas se divide, basicamente, em dois módulos principais:

- **Administrador**, compreendendo os seguintes componentes:
  - **Réplica** - Responsável por criação e remoção de réplicas e atualização de conexões com outros componentes;
  - **Grupo** - Responsável por criação, atualização e remoção de grupos e seus membros.
- **Gestor de QoS**, compreendendo os seguintes componentes:
  - **GestorQoS** - Responsável por reconfigurar parâmetros do GesRep;
  - **Monitor** - Responsável por monitorar as réplicas criadas e tomar decisões em caso de falhas.

Além destes módulos e componentes descritos, compõe, também, a arquitetura do GesRep a interface Replicação, que deve ser estendida e suportada pela interface IDL do componente que será replicado, de modo a permitir que o GesRep realize operações básicas no gerenciamento das réplicas deste componente. Estas operações serão descritas mais detalhadamente a seguir, na subseção 4.1.1.3. As interfaces IDL dos componentes do GesRep estão apresentadas no Apêndice B.

O módulo Administrador e seus componentes serão detalhados na subseção 4.1.1.1, enquanto que o módulo Gestor de QoS e seus componentes serão detalhados na subseção

4.1.1.2. Os diagramas de classe, casos de uso e sequência do GesRep serão apresentados no Apêndice D.

Os serviços fornecidos pelo Gestor de Réplicas são:

- **Gerenciar réplicas e grupos de réplicas** - Incluindo funções de criação, remoção, atualização e sincronização de réplicas;
- **Reconfigurar grupos de réplicas** - Permite modificar de forma dinâmica a quantidade de réplicas mínimas necessárias para uma maior disponibilidade do sistema. Isso se faz útil em caso de expansão das réplicas para novas máquinas na rede ou redução das máquinas presentes na rede;
- **Detectar falhas em réplicas ou grupo de réplicas** - Disponibiliza uma forma de detecção de falhas por *crash* (parada).

Para que o GesRep possa funcionar para aplicações que sejam *soft real-time*, será necessário que os seguintes serviços estejam disponíveis no ambiente da aplicação:

- **Escalonador de tarefas de tempo real para sistemas do tipo** *soft real-time* - Permite que as tarefas sejam escalonadas no sistema de acordo com suas prioridades;
- **ORB tolerante a falhas** - Permite que requisições duplicadas sejam detectadas e descartadas.

A técnica de replicação implementada no GesRep é a passiva, de forma que apenas uma réplica (primária) recebe e processa as requisições, enviando seu estado interno periodicamente para que as réplicas *backup* possam ser sincronizadas, conforme explicado na subsessão 3.2.2, do capítulo 3.

A figura 4.1 ilustra a arquitetura interna do GesRep, incluindo seus componentes e a interação entre eles, além da interação com a réplica criada e sua ligação com o componente cliente que fará as requisições. O componente Grupo, pertencente ao módulo

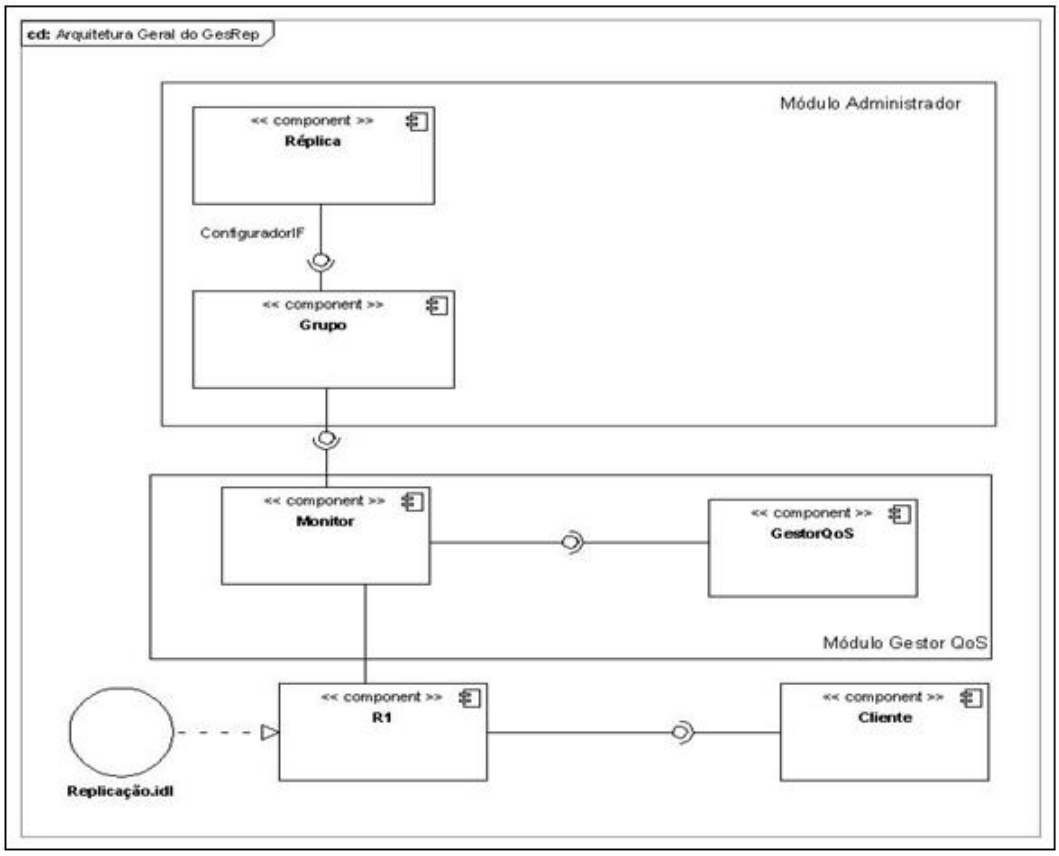

Figura 4.1. Arquitetura do GesRep.

Administrador, utiliza os serviços de criação e remoção de réplicas providos pelo componente Réplica, do mesmo módulo. O componente Monitor, pertencente ao módulo Gestor de QoS, utiliza dos serviços fornecidos pelo componente Grupo no monitoramento da réplica a qual está associado.

O componente que está sendo replicado, por sua vez, deve estender e suportar a interface Replicação, implementando assim seus métodos. Por fim, o componente Cliente estará conectado a apenas uma réplica por vez, sendo esta a réplica primária do grupo.

Seguindo-se os modelos de replicação para aplicações baseadas em componentes, apresentados no capítulo 3, o GesRep utiliza o modelo de aplicação, já que o conjunto dos componentes que compõem o Gestor de Réplicas iteragem de forma a compor uma aplicação; o modelo de protocolo de replicação, com componentes do tipo gerenciador, pela característica de criação e gerenciamento de réplicas; e o modelo de composição, visto que a

interação entre o protocolo de replicação e a aplicação é de suma importância para o provimento do serviço de replicação para componentes, produto fim do GesRep.

4.1.1.1 Módulo Administrador

Este módulo é o mais importante do GesRep, pois tem por responsabilidade prover as funcionalidades básicas de um Gestor de Réplicas, que é a criação e remoção de réplicas e o gerenciamento de grupos.

Possui dois componentes:

- **Réplica**, provendo as seguintes funcionalidades:
  - Criação e remoção de réplicas;
  - Atualização de conexões entre réplicas, para o caso de substituição da réplica primária;
  - Sincronização do estado interno das réplicas *backups* com a primária.
- **Grupo**, provendo as seguintes funcionalidades:
  - Criação e remoção de grupo de réplicas;
  - Adição de nova réplica a um grupo pré-existente;
  - Remoção de réplica pertencente a um grupo;
  - Informar quantidade mínima de réplicas no grupo;
  - Informar quantidade atual de réplicas no grupo;
  - Reconfigurar quantidade mínima de réplicas no grupo;
  - Salvar, persistentemente, a visão dos componentes de um grupo;
  - Retornar a referência da réplica primária.

O componente Grupo será o responsável por prover as funcionalidades básicas do GesRep ao componente Monitor, acessando as funcionalidades do componente Réplica.

O método para retornar o nome da réplica ativa é útil ao componente que está sendo replicado, caso a aplicação da qual o componente faz parte necessite se conectar a ele via Servidor de Nomes. O nome retornado será sempre o da réplica líder atual.

Os métodos de retornar quantidade mínima e atual de réplicas no grupo irão informar ao Monitor se é preciso criar uma nova réplica em substituição a uma que falhou, caso o número de réplicas atual do grupo esteja abaixo do mínimo.

O método de atualização das conexões de uma réplica é utilizado a cada momento que a réplica primária falha e uma nova réplica é eleita como nova líder. Este procedimento, no GesRep, funciona da seguinte forma: o grupo consiste numa seqüência de referências às réplicas, onde a primeira posição nesta seqüência indica a primária, sendo as demais posições as *backups*. Ao ocorrer uma falha da réplica primária, a próxima posição no grupo irá referenciar a nova réplica líder, sendo migradas para ela as conexões com componentes da aplicação que estavam apontando para a réplica falha, seguindo-se da remoção da referência à réplica falha do grupo ao qual pertencia. Sendo assim, apenas uma réplica estará conectada ao sistema por vez, e esta será responsável por receber, processar e retornar o resultado ao cliente.

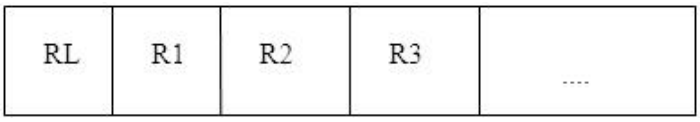

Figura 4.2. Representação interna de um grupo de réplicas no GesRep.

A figura 4.2 ilustra a representação de um grupo de réplicas no GesRep. Apenas as referências às réplicas são armazenadas em cada posição, sendo estas referências representadas pelo nome único com o qual as réplicas são registradas no servidor de nomes. A primeira posição (RL) representa a réplica primária (ou Réplica Líder), enquanto as demais as *backups*.

A figura 4.3 ilustra o relacionamento entre a aplicação cliente e o Gestor de Réplicas.

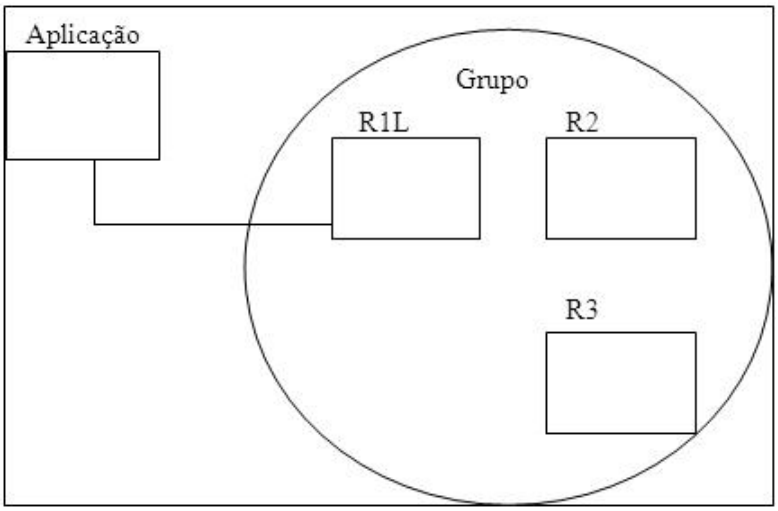

Figura 4.3. Relacionamento do GesRep com a aplicação cliente.

Após a inicialização do GesRep, da aplicação cliente e do procedimento de migração das conexões da réplica inicial, criada através do descritor da aplicação e fora do controle do GesRep, para a réplica primária, os antigos componentes que se conectavam ao componente alvo da replicação continuam conectados a ele, que agora é representado pela réplica primária do grupo (R1L). Este procedimento inicial é necessário pois o GesRep cria réplicas utilizando uma nomenclatura específica para registro no servidor de nomes e para controle das réplicas criadas e removidas, nomenclatura esta não presente na inicialização da aplicação cliente.

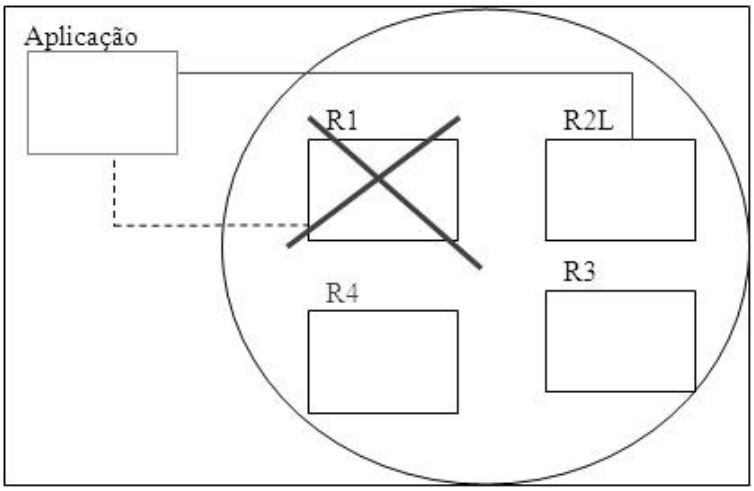

Figura 4.4. Comportamento do GesRep em caso de falha da réplica primária.

A figura 4.4 ilustra o comportamento macro do GesRep em caso de falha da réplica primária do grupo. Nesta situação, a conexão que estava sendo feita com a antiga réplica falha é transferida para a nova réplica primária do grupo, via manipulação do plano de execução da aplicação.

Para garantir a consistência entre as réplicas *backup*, o GesRep implementa o protocolo *reliable multicast* para envio das mensagens de sincronização. Este protocolo, nos trabalhos correlatos estudados, se econtra como um serviço disponibilizado pela infraestrutura (ORB) ao gerenciador de réplicas. Devido ao GesRep utilizar o ORB do TAO, que não dispõe do serviço de *reliable multicast*, ficou a cargo do GesRep implementar esta funcionalidade.

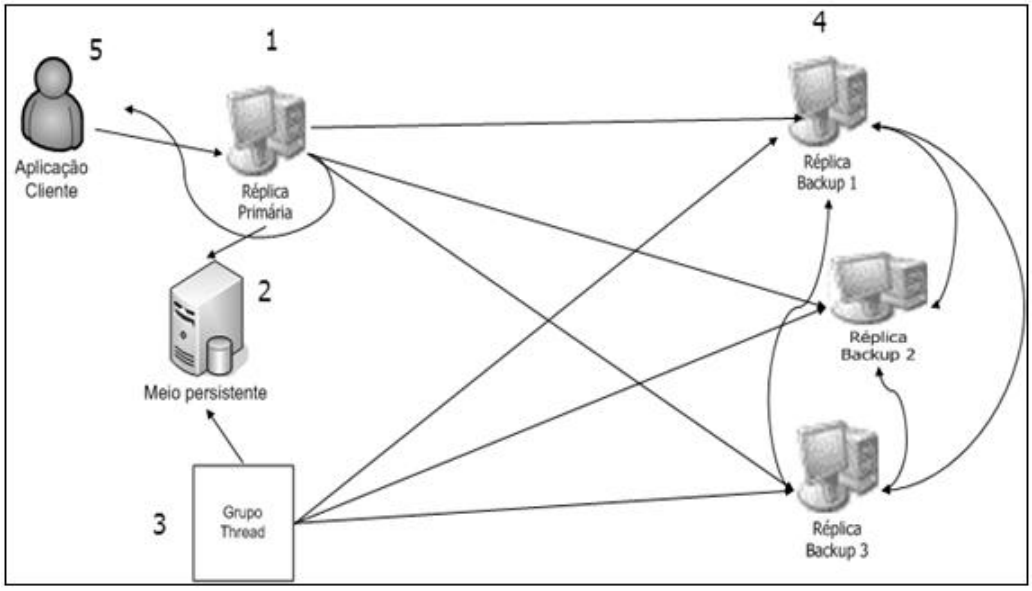

Figura 4.5. Política de Consistência de Réplicas no GesRep.

A figura 4.5 ilustra a política de consistência de réplicas no GesRep, que funciona da seguinte forma: o componente cliente faz uma requisição à réplica primária do grupo (1) que a processa e, antes que o resultado seja retornado ao cliente, este é salvo em meio persistente (2) e enviado via invocação do método recuperaEstado(), provido pela interface ReplicaçãoIF, para a primeira réplica *backup* do grupo (4). Esta réplica, por sua vez, envia a mensagem de atualização para as demais réplicas *backup* do grupo (4), que, ao receberem a mensagem também a enviam para as demais réplicas *backup* do

grupo, de forma que todas as *backups* recebem e enviam a mensagem de sincronização de e para todas as outras réplicas. Paralelo a isso, existe uma *thread* (Grupo*Thread*) que irá recuperar o estado salvo e enviar uma mensagem de sincronização para todas as réplicas *backup* do grupo (3), de maneira periódica, evitando que uma réplica recém integrada ao grupo permaneça desatualizada até a próxima requisição ser processada pela primária. O estado interno de um componente significa o valor de suas variáveis globais, presentes na(s) classe(s) executora(s) do mesmo.

O retorno do resultado ao cliente acontece após o envio da mensagem de atualização da réplica primária para as *backups* (5).

Para cada componente distinto que será replicado pelo GesRep, um novo grupo será criado, agrupando réplicas de um mesmo componente em um único grupo. Então, antes que uma nova réplica seja criada, uma pesquisa será feita dentre os grupos existentes, para certificar a existência prévia ou não de um grupo com membros do tipo da nova réplica criada. Caso o grupo exista, a nova réplica terá sua referência adicionada ao mesmo. Caso contrário, um novo grupo será criado para agrupar réplicas deste novo componente.

Este módulo representa o ponto de inicialização do GesRep, e recebe os parâmetros do componente a ser replicado via o arquivo gesRep.xml, que possui a estrutura ilustrada na figura 4.6.

```
<REPLICA>
        <UUID></UUID>
        <INSTANCIA></INSTANCIA>
        <IMPLEMENTATION></IMPLEMENTATION>
        <QTDMINREPLICAS></QTDMINREPLICAS>
        <NODENAME></NODENAME>
        <RECONFIGURACAO></RECONFIGURACAO>
        <TIPO_REPLICACAO></TIPO_REPLICACAO>
</REPLICA>
```

Figura 4.6. Estrutura do arquivo XML de configuração de componentes para serem replicados.

O bloco $< REPLICA >< /REPLICA >$ contém os parâmetros necessários para a criação de uma nova réplica de um componente. Para cada novo componente a

ser replicado, um novo bloco destes deve ser copiado no arquivo. Os parâmetros de configuração $< UUID >< /UUID >$, $< INSTANCIA >< /INSTANCIA >$, $< IMPLEMENTATION >< /IMPLEMENTATION >$ e $< NODENAME >< /NODENAME >$ são provenientes do arquivo descritor de implantação da aplicação (arquivo com extensão '.cdp'), e representam, respectivamente, os seguintes parâmetros do arquivo descritor: UUID, *Instance*, *Implementation* e *Node*, sendo necessário que o componente do qual se deseja criar réplicas esteja contido neste arquivo.

O bloco $< QTDMINREPLICAS >< /QTDMINREPLICAS >$ representa um número inteiro que indica a quantidade mínima de réplicas em um grupo. Também indica uma validação entre a quantidade de blocos $< REPLICA >< /REPLICA >$ informados e a quantidade de réplicas desejadas no grupo. Este parâmetro pode ser alterado em tempo de execução, bastando que o bloco $< RECONFIGURACAO >< /RECONFIGURACAO >$, que indica se o usuário deseja fazer uma reconfiguração do número de réplicas, seja alterado para 1. No padrão, este parâmetro é configurado como 0, indicando uma configuração inicial.

O bloco $< TIPO\_REPLICACAO >< TIPO\_REPLICACAO >$ indica a técnica de replicação escolhida pelo usuário para o gerenciamento das réplicas. O GesRep possui implementada apenas a técnica de replicação passiva, representado pelo número 0 (zero) para este parâmetro, mas novas técnicas de replicação poderão ser adicionadas futuramente, sendo representadas por outros valores numéricos.

Para cada réplica a ser criada, um novo bloco $< REPLICA >< /REPLICA >$ deverá ser construído, retirando apenas o bloco $< TIPO\_REPLICACAO >$, que será colocado apenas no último bloco de réplicas do grupo, indicando o término da configuração para aquele componente.

Este arquivo de configuração é lido por uma classe, denominada CriarReplicas, que irá desmembrar os valores de cada parâmetro e dar início ao processo de gerenciamento de réplicas do GesRep, ao invocar o método criarGrupo do componente Grupo.

4.1.1.2 Módulo Gestor de QoS

Este módulo é responsável pelo monitoramento das réplicas criadas e pela reconfiguração do GesRep. Possui dois componentes, que são:

- **Monitor** - Responsável pelo monitoramento das réplicas criadas. Funciona também como um detector de falhas do tipo *crash* e cada monitor criado será associado a uma e somente uma réplica. Caso a réplica venha a falhar, o monitor irá detectar a falha e remover a referência desta do grupo de réplicas, se autodestruindo em seguida. O monitoramento da réplica associada é feito através de uma *thread* que, periodicamente, envia mensagens de "está vivo?" à réplica alvo do monitoramento. Esta *thread* também é removida em caso de falhas;
- **GestorQoS** - Responsável pela reconfiguração dos parâmetros do grupo de réplicas no GesRep. Para o protótipo deste trabalho, apenas o parâmetro de quantidade mínima de réplicas no grupo está sendo reconfigurável.

O monitoramento de uma réplica é feito através do envio de mensagens de "está vivo?" para a mesma, de forma que, para que esta mensagem possa ser enviada, é preciso se conectar à réplica via Servidor de Nomes (*Name Service*) e invocar o método estaVivo, fornecido pela interface ReplicacaoIF. Caso a máquina na qual a réplica está instalada ou o *NodeManager* do *host* estejam falhos por *crash*, a conexão não será possível, e uma exceção será lançada pelo CORBA. Ao receber esta falha, o monitor entenderá que o *container* ou a máquina que hospeda a réplica falhou, executando a rotina de recuperação da quantidade mínima de réplicas no grupo. Falha em réplica primária requer migração de conexão para a primeira réplica *backup* do grupo, que se tornará nova líder, e criação de uma nova réplica *backup* no nodo *MainNode* para recuperar a quantidade de réplicas no grupo. Falha em réplicas *backups* requer apenas criação de uma nova réplica para manter o número mínimo das réplicas no grupo.

Os componentes do tipo monitor são criados apenas no nodo principal do sistema, denominado *MainNode*, que se caracteriza por hospedar o servidor da aplicação, servidor

do GesRep, *ExecutionManager* e servidor de nomes. Devido ao CIAO não ter tido sua construção voltada para atender a requisitos de tolerância a falhas, não é possível replicar o *ExecutionManager* numa mesma rede. Para evitar que o GesRep se torne um ponto único de falhas, é necessário que haja duas redes distintas executando o conjunto aplicação Cliente e GesRep, de modo que, com falha em uma rede, esta seja tolerada a partir da execução normal do sistema na outra rede.

O parâmetro de quantidade mínima de réplicas no grupo, que é definido pelo usuário no momento da configuração do componente a ser replicado no arquivo XML, poderá ter seu valor alterado pelo operador para mais ou para menos, contanto que este número seja superior a 1 (um). A teoria da política de replicação passiva informa que, para que *f* falhas sejam toleradas, é preciso ter *f+1* réplicas, o que exclui a existência de um grupo com apenas uma réplica.

As modificações de ambiente que serão levadas em consideração pelo componente Monitor são:

- **Falha de réplica em um grupo** - Caso ocorra uma falha em uma ou mais réplicas de um grupo, réplicas serão criadas até que o número mínimo seja atingido. Caso o grupo fique vazio após a remoção das réplicas falhas, este será removido e novo grupo deverá ser criado, contendo o mesmo número de réplicas do grupo anterior no nodo *MainNode*, para substituição;

- **Aumento ou diminuição do número de réplicas no grupo** - Caso a aplicação tenha a necessidade de uma alteração no número de réplicas de um determinado grupo, réplicas serão criadas ou destruídas para fazer a reconfiguração. Esta ação é tomada pelo usuário, ao alterar a quantidade de réplicas no arquivo XML de configuração do GesRep.

4.1.1.3 A interface Replicação

Com o objetivo de permitir que o GesRep possa ter acesso a algumas informações do componente e possa realizar o gerenciamento das réplicas, foi criada uma interface denominada Replicação.

Esta interface não está associada a nenhum dos módulos descritos anteriormente, e deve ser estendida e suportada pelo componente que deseje utilizar do serviço de replicação fornecido pelo GesRep.

Sua estrutura e seus métodos são apresentados na figura 4.7.

```
interface ReplicacaoIF
{
  string salvaEstado();
  long recuperaEstado(in string estadoSalvo,in string vGrupo);
  long estaVivo();
  void setTipoReplicacao(in long tipoReplicacao);
  long getTipoReplicacao();

};
```

Figura 4.7. Estrutura da interface Replicação.

Os métodos salvaEstado e recuperaEstado têm a finalidade de, respectivamente, salvar o estado interno da réplica e recuperar o estado interno da mesma. Como o GesRep não tem o conhecimento interno das variáveis do componente, estes métodos são úteis para que o usuário possa decidir quais as variáveis do componente que está sendo replicado terão seus estados salvos para posterior sincronização. Em uma aplicação, as variáveis importantes para guardar o estado são aquelas que possuem o escopo global e que não fazem parte da interligação do componente com outros, como variáveis do tipo ORB ou *Context* (criadas automaticamente pelo novo componente quando instalado no *container*).

O método estaVivo é responsável pela mensagem de "está vivo?" enviada pelo componente Monitor. Caso a réplica tenha tido alguma falha, ao invocar o método de *ping* a aplicação lançará uma exceção, o que representa a retirada da réplica do grupo e da posterior destruição do monitor responsável por ela, como já explicado anteriormente.

Para que esta exceção lançada seja controlada pelo GesRep, foi necessário que a classe *DomainApplicationManager_impl* do CIAO fosse alterada.

Os métodos setTipoReplicacao e getTipoReplicacao irão informar à réplica qual a técnica de replicação escolhida pelo usuário.

4.2 UTILIZAÇÃO DO GESREP POR UMA APLICAÇÃO BASEADA EM COMPONENTES

Para que uma aplicação baseada em componentes possa fazer uso do GesRep, algumas configurações devem ser feitas pelo componente alvo, de modo que o Gestor de Réplicas possa ter um mínimo de informações sobre o componente para realizar o serviço de replicação do mesmo.

No que se refere ao componente em si, serão necessárias as seguintes modificações em seus arquivos:

- Na interface IDL, o componente deve estender e suportar a interface Replicação do GesRep, como exemplificado na figura 4.8;
- No arquivo descritor de implantação, de extensão .CDR, deve-se incluir uma instância do componente a ser replicado, direcionada para cada *host* nos quais terão réplicas distribuídas;
- No arquivo MPC do componente em questão, deve-se fazer as seguintes modificações:
  - No projeto do executor (exec), deve-se colocar em libs a linha "Replicacao_stub";
  - No projeto do servant (svnt), deve-se colocar em libs a linha "Replicacao_svnt" e "Replicacao_stub".

A figura 4.9 ilustra um exemplo do arquivo MPC do componente *DAISSimulatedCarProvider*, do ARCOS, configurado para ser replicado pelo GesRep. Este arquivo é

```
interface ReplicacaoIF
{
  string salvaEstado();
  long recuperaEstado(in string estadoSalvo,in string vGrupo);
  long estaVivo();
  void setTipoReplicacao(in long tipoReplicacao);
  long getTipoReplicacao();

};
```

Figura 4.8. Interface configurada para utilizar o serviço de replicação.

apresentado na íntegra no Apêndice C.

```
project(ARCOSDAISSimulatedCarProvider_DnC_svnt) : ciao_servant_dnc {
  avoids += ace_for_tao
  after +=  OMGDAIS_DnC_svnt \
            ARCOSDAISDAGroup_DnC_svnt \
            ARCOSDAISDANode_DnC_svnt \
            ARCOSDAISProviderBase_DnC_svnt \
            ARCOSDAISSimulatedCarProvider_DnC_stub
  sharedname  = ARCOSDAISSimulatedCarProvider_DnC_svnt
  libs    += ARCOSDAISSimulatedCarProvider_DnC_stub \
             OMGDAIS_DnC_stub \
             OMGDAIS_DnC_svnt \
             ARCOSDAISDAGroup_DnC_stub \
             ARCOSDAISDANode_DnC_stub \
             ARCOSDAISDAGroup_DnC_svnt \
             ARCOSDAISDANode_DnC_svnt \
             ARCOSDAISProviderBase_DnC_stub \
             ARCOSDAISProviderBase_DnC_svnt \
             Replicacao_stub \ Replicacao_svnt
```

Figura 4.9. Exemplo de arquivo MPC configurado para utilizar o GesRep.

A relação entre o ARCOS, o GesRep e uma aplicação construída sobre o *framework* ARCOS é ilustrada na figura 4.10. O ARCOS e o GesRep se encontram no mesmo nível para a aplicação cliente, já que esta se beneficiará da facilidade de construção fornecida pelo *framework* e precisará configurar o(s) componente(s) de maior relevância para seu funcionamento para ser(em) replicado(s) pelo GesRep, além de configurar o arquivo XML do Gestor de Réplicas.

Os passos de configuração de um componente que deseja utilizar o serviço de replicação do GesRep estão ilustrados na figura 4.11.

Caso o componente esteja em fase de construção inicial, é preciso compilar o mesmo após os passos de configuração acima descritos, utilizando-se dos comandos abaixo:

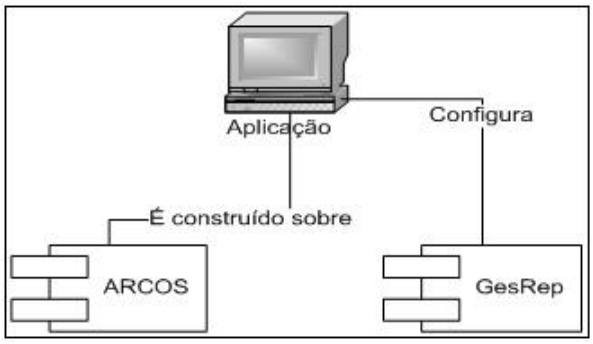

Figura 4.10. Relação entre aplicação cliente, ARCOS e GesRep.

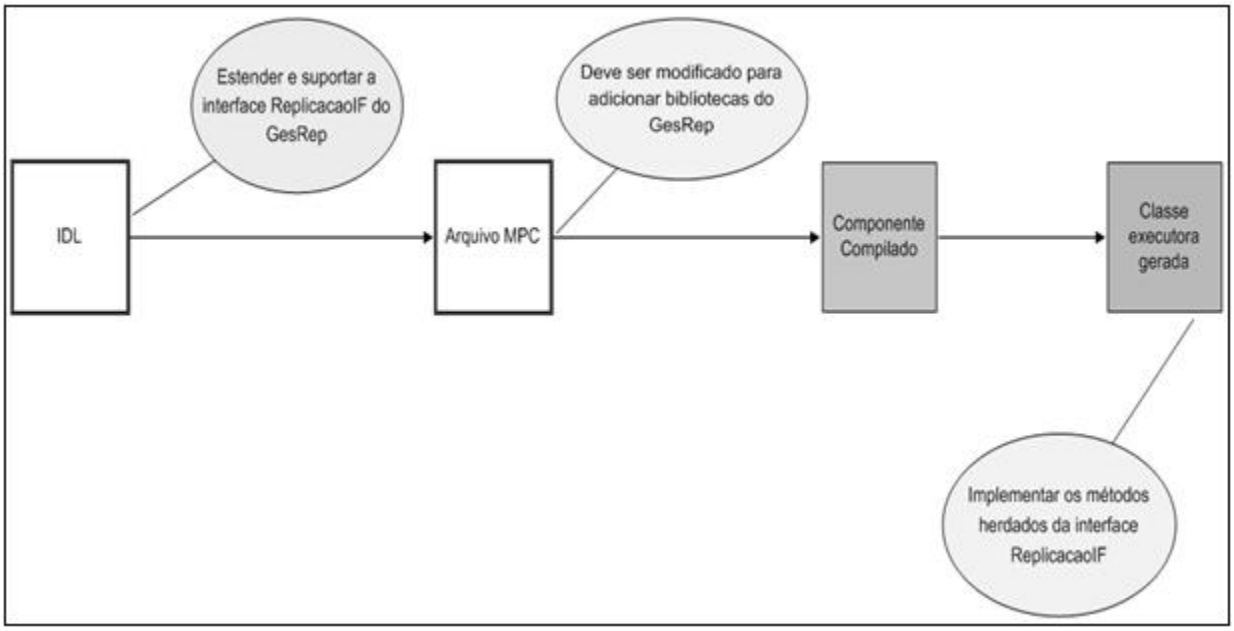

Figura 4.11. Passos para configuração de um componente que irá ser replicado pelo GesRep.

- **mwc.pl -type gnuace**, para compilação da interface IDL;
- **Make** , para compilação do componente.

Após a geração da classe executora, será necessário implementar os métodos herdados da interface Replicação. A implementação dos métodos é padrão para todos os componentes que utilizem do GesRep, exceto os métodos salvarEstado e recuperarEstado, pois são específicos para cada componente.

O passo seguinte é a configuração do arquivo XML, que está localizando dentro da

pasta Réplica, na estrutura de pastas do GesRep.

Após todas as etapas acima descritas terem sido feitas, o último passo é a inclusão de uma instância para cada *host* a ter réplicas distribuídas no arquivo descritor de implantação da aplicação, conforme ilustrado na figura 4.12.

```
364 <instance id="ARCOS-DAISSimulatedCarProvider-idd">
365     <name>ARCOS-DAISSimulatedCarProvider-idd</name>
366     <node>SemaNode</node>
```

Figura 4.12. Instância do componente a ser replicado nos *hosts* possíveis de replicação.

A linha 366 da figura 4.12 ilustra o nome do *host* que deve ser alterado em cada instância presente no arquivo descritor de implantação. Para cada *host* mapeado no arquivo *NodeManagerMap* uma nova instância deve ser criada no arquivo descritor. O arquivo descritor de implantação do GesRep está apresentado no Apêndice A.

4.3 IMPLANTAÇÃO DO GESREP E DA APLICAÇÃO CLIENTE NO CIAO

Para que uma aplicação cliente seja implantada no CIAO, os seguintes passos devem ser seguidos:

- O Servidor de nomes (*name service*) deverá ser inicializado;
- Os respectivos *NodeManagers* em cada *host* da rede deverão ser inicializados;
- O *script* de incialização da aplicação cliente é, então, invocado, inicializando-a;
- Após o passo anterior, o *Execution Manager* do CIAO será inicializado, dando início ao processo de implantação dos componentes em seus respectivos *containers*;
- O componente *Plan Launcher* do CIAO será inicializado, tendo o objetivo de ler o arquivo descritor de implantação da aplicação cliente e traduzí-lo em um plano de execução;

- Este plano de execução, por sua vez, será enviado ao *Execution Manager*, de forma a ser subdividido em planos de execução menores, que serão distribuídos pelos diversos nodos na rede para implantação dos respectivos componentes de forma distribuída;

- As diversas subdivisões do plano de execução são enviadas aos diversos *NodeManagers* distriuídos na rede;

- O componente *DomainApplicationManager* do CIAO é, então, invocado por cada *NodeManager* com o intuito de gerenciar os componentes futuramente implantados nestes *hosts*;

- E, por último, um *container* é criado em cada *host*, concluindo o processo de implantação dos componentes sob responsabilidade dos respectivos nodos.

Após os passos anteriormente citados, o *script* de incialização do GesRep deve ser invocado, de modo a inicializar o serviço de replicação do componente alvo da aplicação cliente. É importante citar que, anteriormente ao processo de invocação dos respectivos *scripts* de inicialização da aplicação cliente e GesRep, o processo de configuração do componente e do GesRep precisa estar concluído.

4.4 EXPERIMENTOS E RESULTADOS OBTIDOS

4.4.1 Experimentos

Para os experimentos com o GesRep, foram utilizadas cinco máquinas em uma rede local (LAN) isolada, sem acesso à Internet, de velocidade 100Mbps, contendo as seguintes configurações:

- Máquinas 1,2,3 e 4 - *Desktop*, com Sistema Operacional (SO) Linux, distribuição Ubuntu 9.04, processador Intel Core 2 Duo de 2.00 GHz, 1 GB de memória RAM, com os respectivos nomes Ufba1, Ufba2, Ufba3 e Ufba4;

- Máquina 5 - *Notebook*, com SO Linux, distribuição Ubuntu 9.04, 1 processador Intel Centrino de 1,86 GHz e 1 GB de RAM, de nome Sema-Laptop.

Foi assumida a premissa que a rede é confiável e que não há perda de mensagens.

O componente do ARCOS selecionado para replicação foi o ARCOS DAIS *Simulated Car Provider*, que é responsável por prover os dados de velocidade (*speed*) e acelerador (*throttle*) na simulação de um carro partindo da inércia e ganhando movimento até atingir uma determinada velocidade constante.

Uma outra aplicação de teste mais simples utilizada para validação do funcionamento do GesRep foi um contador intermitente, onde o objetivo final do teste era garantir a continuidade do contador, mesmo após ocorrência de falhas da réplica primária. Como resultado deste experimento simples, foi observada a continuidade do contador, sem interrupção mesmo após a falha da réplica primária do grupo.

Os testes e verificação do desempenho do GesRep foram realizados em cenários de, no mínimo, duas réplicas e, no máximo, dez réplicas, o que permitiu verificar a extensabilidade do GesRep. Outra justificativa para este número é que, em cenário real de replicação, poucas réplicas são criadas, de modo que uma quantidade de dez réplicas se mostra além do valor normalmente utilizado em um cenário real.

Foram avaliados o tempo de criação de um grupo de réplicas, o tempo de recuperação de uma réplica em caso de falha, o tempo de sincronização das réplicas de um grupo e o de sincronização de uma réplica individual, além do tempo gasto para o monitoramento de uma réplica individual.

Para cada um dos resultados acima obtidos, os valores do desvio padrão, média aritmética e mediana foram calculados, levando em consideração uma quantidade de 60 amostras por experiência realizada.

A dispersão de dados é o grau no qual dados numéricos tendem a dispersar-se em torno de um valor médio. Uma das medidas de dispersão de dados é o desvio padrão, que representa o desvio de cada um dos números de uma amostragem em relação à média

do grupo. Então, quanto menor for este valor, menor será o grau de dispersão dos dados coletados em relação à média calculada [45].

Para os resultados obtidos no GesRep, o cálculo do desvio padrão representa o nível de variação do tempo obtido para uma amostra de um experimento em relação à média de tempo obtida para o experimento realizado. Quanto menor for o valor do desvio padrão, menor será o distanciamento dos valores em relação ao valor médio, o que indicará um comportamento mais constante do GesRep.

O cálculo da média aritmética é útil para que o valor do tempo médio para cada experimento possa ser conhecido.

O valor da mediana equivale ao ponto central ou a média aritmética dos dois valores centrais da faixa de valores coletadas nos experimentos realizados com o GesRep. Geometricamente falando, o valor da mediana representa o valor da abscissa que divide a amostragem de valores em duas metades iguais [45].

4.4.2 Resultados Obtidos

Os resultados obtidos com os experimentos realizados com o GesRep foram coletados em microssegundos (μs), porém, para facilitar a leitura, foram convertidos para segundos, com uma precisão de 5 casas decimais.

Os tempos que foram avaliados estarão apresentados nas tabelas a seguir.

A tabela 4.1 apresenta os tempos, em segundos, obtidos com a criação de um grupo completo de réplicas. Como o desvio padrão tem um valor bastante inferior ao da média, assumimos que os vários experimentos tiveram um comportamento constante, com valores de tempo muito próximos.

Observamos também que o tempo para criação de um grupo de réplicas cresce com o aumento no número de réplicas no grupo. Entretanto, este crescimento só é superior a 0,4 segundos com o aumento de 3 para 4 máquinas e réplicas. Nas demais variações, este valor varia entre 0,1 e 0,4 segundos.

Tabela 4.1. Tempo de criação de um grupo de réplicas, em segundos.

Tempo de Criação - Grupo de Réplicas				
Num. Máquinas	Num. Réplicas	Desvio padrão	Média	Mediana
2	2	0,04284	0,27842	0,28693
3	3	0,07426	0,68027	0,70160
4	4	0,04439	1,32662	1,33359
5	5	0,07785	1,77005	1,75854
5	6	0,05355	1,88385	1,87942
5	7	0,06435	2,07954	2,08425
5	8	0,06471	2,28458	2,27395
5	9	0,05988	2,50607	2,51085
5	10	0,06916	2,83483	2,83804

O resultado do experimento de sincronização de todas as réplicas de um grupo é apresentado na tabela 4.2.

Tabela 4.2. Quantidade de réplicas e resultados de tempo para a sincronização de todas as réplicas de um grupo.

Sincronização de Grupo			
Qtd. Réplicas	Desvio Padrão	Média	Mediana
2	0,00133	0,00543	0,00586
3	0,00009	0,00883	0,00884
4	0,00035	0,01229	0,01234
5	0,00156	0,01958	0,01868
6	0,00104	0,01401	0,01415
7	0,00229	0,02615	0,02694
8	0,00608	0,02092	0,01815
9	0,00173	0,03239	0,03216
10	0,00279	0,03926	0,03871

A tabela 4.2 apresenta os resultados de tempo obtidos com a sincronização de um grupo de réplicas. Novamente, observamos que o desvio padrão apresenta um valor consideravelmente inferior ao da média, caracterizando experimentos consistentes, com valores constantes. O valor da média também cresce com o número de réplicas, sendo que, de 5 para 6 réplicas, e de 7 para 8 réplicas, houve uma redução no tempo médio de sincronização. Não foi encontrada uma explicação para esta redução ocorrida.

Nesta tabela 4.3 estão apresentados os cálculos do tempo de recuperação de uma réplica do tipo primária e do tipo *backup*. Na recuperação de um réplica primária, é necessário que haja a migração das conexões que envolviam a réplica faltosa para a

primeira réplica *backup* do grupo, que passará a ser a nova primária, e a criação de uma nova réplica *backup*, garantindo a manutenção do número mínimo de réplicas no grupo. A operação de recuperação em caso de falha de uma réplica *backup* equivale à criação de uma nova réplica *backup* para manutenção do número mínimo de réplicas no grupo.

Tabela 4.3. Tempo de recuperação de réplicas pelo GesRep.

Tempo de Recuperação de Réplica			
Tipo	Desvio Padrão	Média	Mediana
Backup	0,03946	0,17876	0,20046
Primária	0,03937	0,16078	0,15600

O resultado apresentado na tabela 4.3 indica o tempo de recuperação para réplicas do tipo *backup* e primária, inserindo uma alta carga de processamento nas máquinas e alto tráfego de mensagens na rede. A alta carga de processamento foi alcançada através da execução simultânea de vários programas nas máquinas envolvidas, além do processamento de disco de áudio. O tráfego na rede foi aumentado através da execução de três cópias simultâneas de todo o sistema operacional de uma máquina para outra.

Tabela 4.4. Tempo de recuperação de réplicas pelo GesRep, na presença de alta taxa de processamento e alto tráfego na rede.

Tempo de Recuperação de Réplicas na Presença de Stress de Processamento e alto tráfego na Rede.			
Tipo	Desvio Padrão	Média	Mediana
Backup	0,03055	0,23720	0,23739
Primária	0,02601	0,24003	0,23752

Se comparadas as tabelas 4.3 e 4.4, verifica-se que, mesmo na presença de carga significativa de processamento pela máquina e de alto tráfego na rede, o desempenho do GesRep não é significativamente alterado, mantendo um comportamento adequado para o funcionamento da aplicação que dele faça uso, que significa a continuidade do serviço fornecido pelo componente que está sendo replicado.

Verificou-se a continuidade do funcionamento da aplicação utilizada para teste, mesmo com falha da réplica primária. Esta disponibilidade se dá devido ao tempo mínimo de recuperação de falha na réplica primária, apresentado nas tabelas 4.3 e 4.4.

Os tempos de criação de um grupo completo de réplicas indicam que o processo de criação do grupo e inicialização das atividades do GesRep é feito de forma répida, se comparado com a criação de apenas uma réplica e com o tempo que o ReDaC leva para reimplantar um plano de execução.

Os tempos de sincronização de um grupo de réplicas mostram que, mesmo para um grupo com uma maior quantidade de réplicas, estas são sincronizadas quase que simultaneamente, levando apenas fração de segundo para que todo o processo seja completo.

A tabela 4.5 representa os valores calculados para o tempo de sincronização do estado interno de uma réplica individual com o estado interno da réplica primária do grupo. Os valores estão apresentados em segundos. Por esta tabela conclui-se que o processo de sincronização das réplicas de um grupo possui bom desempenho, levando em consideração o baixo tempo obtido e que o GesRep é voltado para aplicações não críticas de tempo real, de modo a não comprometer o desempenho do processamento das requisições pela aplicação cliente.

Tabela 4.5. Tempo de sincronização de uma réplica individual pelo GesRep.

Tempo de Sincronização Individual de Réplica		
Desvio padrão	**Média**	**Mediana**
0,0008	0,00265	0,00236

4.5 COMPARATIVO GESREP X GESTORES ESTUDADOS

Os gestores apresentados como trabalhos correlatos possuem características próprias, de acordo com o domínio de aplicações ao qual se destinam, como tempo real, adaptabilidade, provisionamento de qualidade de serviço, dentre outros.

O Gestor de Réplicas (GesRep) proposto neste trabalho busca unir as principais características no gerenciamento de réplicas e operações básicas (criação, remoção e atualização) com conceitos de requisitos temporais, onde poderão existir réplicas que necessitem cumprir requisitos temporais e outras sem estes requisitos.

Uma diferença entre o AQuA e o GesRep é que o primeiro possui um sistema de sincronismo virtual, no qual todas as réplicas se comunicam entre si através de *gateways*. No caso do GesRep, as réplicas não se comunicam entre si, ficando esta funcionalidade a cargo do processamento interno do GesRep, ao repassar o estado armazenado da réplica primária para as demais réplicas do grupo.

Outra diferença é relacionada à política de retorno de resultado ao cliente. No AQuA existem três tipos: apenas o líder retorna o resultado ao cliente, todas as réplicas retornam o resultado ao cliente e o primeiro deles é selecionado, todas as réplicas retornam resultado e uma votação é feita. No GesRep, existe apenas a primeira forma de retorno, já que o mesmo trabalha com o conceito de grupos de réplicas, ficando este papel para a líder do grupo.

Em relação ao Chameleon, o GesRep possui a semelhança no que diz respeito ao monitoramento periódico das réplicas através de mensagens do tipo "Está vivo?", que são respondidas pelas réplicas que as receberem.

O Piranha possui o conceito de grupo de réplicas utilizado pelo GesRep, porém a diferença entre ambos é que no primeiro as réplicas são objetos (classes), enquanto que no segundo o conceito de réplica é um componente e suas conexões.

Entre o GesRep e o RMS, pode-se citar a semelhança dos parâmetros passíveis de serem configurados, como, por exemplo, o número das réplicas no sistema. No RMS existem os conceitos de MTTF e MTTR, objetivando garantir maior confiabilidade e disponibilidade das réplicas, porém estes conceitos não são utilizados pelo GesRep. Outro ponto semelhante entre os dois sistemas é a redundância do próprio gerenciador de réplicas, evitando ponto único de falhas. A diferença neste sentido é como esta redundância acontece: enquanto no RMS o gerenciador é replicado dentro da própria rede, porém em máquinas diferentes, no GesRep a redundância acontece com redes distintas, ficando então todo o conjunto aplicação cliente e GesRep duplicado. Um cliente poderá ter acesso às duas redes, de modo que, quando o GesRep falhar em uma rede, a continuidade do serviço aconteça através da outra rede.

Assim como no FTWeb, o GesRep controla a quantidade de réplicas no grupo através de um arquivo de configuração XML, que pode ser redefinido em tempo de execução pelo usuário da aplicação. Outros parâmetros também são controlados via arquivo XML pelo GesRep, como a técnica de replicação utilizada. O intervalo de monitoramento das réplicas, diferentemente do FTWeb, é fixo no GesRep, assim como o tempo de resposta para as requisições não está presente nos parâmetros de configuração.

A arquitetura do GesRep provê o componente Monitor como detector, notificador e responsável pela tomada de decisão em caso de falhas. Na arquitetura do DOORS, dois componentes separados são responsáveis pela detecção e tomada de decisão em caso de falhas. Existem notificadores de falhas distintos para objetos e *hosts* no DOORS, enquanto que no GesRep estes dois tipos de falhas são detectados pelo mesmo componente. A técnica de monitoramento das réplicas, ao contrário do DOORS, não pode ser selecionada no GesRep, sendo fixa a *heartbeat*.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Esta dissertação apresentou a proposta e implementação de um Gestor de Réplicas reconfigurável, voltado para replicação de componentes de *software* e suas conexões, e para aplicações mecatrônicas do domínio de controle e supervisão, sendo estas construídas com base na arquitetura de componentes e na implementação do CORBA CCM, CIAO. A técnica de replicação é de suma importância para garantir uma maior disponibilidade da aplicação que dela faça uso, de modo a evitar um ponto único de falhas para suas partes essenciais. Este capítulo irá apresentar as conclusões obtidas com o GesRep e as possibilidades de extensão do projeto em trabalhos futuros.

5.1 CONCLUSÕES

Com o crescimento de aplicações voltadas para sistemas mecatrônicos e, tendo em vista a necessidade destas aplicações se manterem o maior tempo possível em funcionamento, com possíveis graves conseqüências em caso de parada de uma planta de produção, por exemplo, é importante que alguma técnica de tolerância a falhas seja incorporada a estas aplicações, ocasionando uma maior disponibilidade e confiabilidade das mesmas.

A técnica de replicação é uma importante aliada para aplicações do domínio acima referido, pois permite que partes ou toda uma aplicação seja mantida replicada em mais de um ponto na rede, assumindo o controle em caso de falha da parte replicada e garantindo um aumento de disponiilidade de tais aplicações.

Os gestores de réplicas existentes atualmente são voltados para replicação de objetos de uma aplicação, que consistem em classes ou métodos individuais. Com o desenvolvimento do GesRep, fica disponível um Gestor de Réplicas voltado para componentes de uma aplicação construída sobre a arquitetura de componentes, suprindo, também, as

necessidades de tolerância a falhas de aplicações mecatrônicas.

Um componente de *software* é um conjunto de classes capazes de fornecer um determinado serviço específico, com portas específicas para conexão com outros componentes. Replicar um componente significa, então, replicar as classes (objetos) deste componente, assim como suas conexões com os demais componentes da aplicação e o estado interno deste componente.

Para validação do Gestor de Réplicas desenvolvido, foi utilizado o *framework* ARCOS, voltado para construção de aplicações mecatrônicas do domínio de controle e supervisão, e que não possuía o serviço de tolerância a falhas acoplado. Um dos componentes essenciais para o funcionamento do *framework* foi replicado, através do GesRep, e experimentos de desempenho do Gestor de Réplicas foram realizados.

Com os resultados dos experimentos, foi possível concluir que os tempos de resposta do GesRep foram satisfatórios, não apresentando atrasos relevantes na execução das aplicações construídas através do ARCOS.

O CIAO, por não ter sido construído levando em consideração aspectos de tolerância a falhas, possui algumas limitações, que poderiam acarretar a existência de um ponto único de falhas no projeto do GesRep, pois não é possível que mais de um *ExecutionManager* seja inicializado em uma mesma rede, impossibilitando a replicação do GesRep e sua infraestrutura. Para contornar esta limitação, é preciso que existam duas redes separadas executando a aplicação cliente e o GesRep, de forma que o cliente para esta aplicação enxergaria as duas redes como dois pontos de fornecimento do processamento desejado.

Assim, em caso de uma falha na máquina servidora de uma das redes, o serviço provido ao cliente não seria interrompido, sendo assumido pela outra rede disponível.

Os objetivos iniciais do Gestor de Réplicas foram, então, alcançados com os resultados apresentados no capítulo 4, seção 4.4, subseção 4.4.2. As metas principais eram: criar, destruir, sincronizar e gerenciar réplicas através de monitoramento; manter a consistência entre as réplicas criadas e reconfigurar a quantidade mínima de réplicas em um grupo, de acordo com o número de *hosts* presentes na rede. O GesRep precisaria ter uma

desempenho aceitável de recuperação do grupo de réplicas em caso de falhas, de modo que o serviço de replicação se tornasse transparente para a aplicação que dele estivesse fazendo uso. Este desempenho aceitável se reflete nos resultados do cálculo do desvio padrão, necessitando que este valor seja pequeno em comparação com os valores da média e mediana.

O GesRep permite que novas funcionalidades sejam a ele adicionadas, como extensão e melhoria das funcionalidades disponíveis através de acoplamento de novos componentes ou do aperfeiçoamento dos componentes existentes. Novas técnicas de replicação podem ser adicionadas e, para permitir isso, foi adicionado no arquivo de configuração XML do GesRep uma indicação para o código da técnica de replicação a ser utilizada.

Qualquer aplicação com base na arquitetura de componentes pode ter seus componentes replicados através do GesRep, já que o Gestor de Réplicas foi construído de forma genérica e desacoplada de qualquer aplicação.

Após o início e andamento deste projeto, o serviço ReDaC foi removido do CIAO pelo grupo responsável por sua manutenção. Como o andamento do projeto se encontrava em estágio bem avançado, não foi possível fazer uma migração de solução, descartando o ReDaC, até porque o serviço fornecido por esta ferramenta é bastante custoso e passível de erro caso seja feito sem seu auxílio. Segundo o grupo de desenvolvimento do CIAO, este serviço pode vir a ser reincorporado ao mesmo no futuro, com aperfeiçoamentos.

5.2 TRABALHOS FUTUROS

Como trabalhos futuros, é possível implementar novas técnicas de replicação, além da técnica de replicação passiva já implementada. A cada nova técnica de replicação implementada, deve-se atribuir um valor inteiro positivo à mesma.

O módulo GestorQoS pode ser aperfeiçoado para inclusão de novos parâmetros para reconfiguração do GesRep, como a utilização de variáveis do sistema operacional para determinar a quantidade mínima de réplicas possíveis para um determinado *host* de forma

automatizada. Assim, a quantidade mínima de réplicas em um grupo passaria a ser calculada pelo GesRep, conforme a configuração do *host* alvo.

As réplicas criadas pelo GesRep podem ser destruídas e migradas para um *host* diferente em caso de degradação de desempenho do primeiro *host* ou em caso de queda do mesmo.

O componente Monitor pode ser aperfeiçoado, de forma a detectar não apenas falhas por parada, mas também outros tipos de falhas. Além disso, a freqüência de monitoramento das réplicas pode ser alterada em tempo real, de acordo com as condições de desempenho das máquinas onde as réplicas estejam instaladas. Lembrando que um tempo curto de monitoramento pode criar um grande *overhead* na rede e de processamento, enquanto que um tempo grande demais pode não detectar falhas assim que elas aconteçam.

Outro trabalho importante é o aperfeiçoamento da solução de criação e destruição de réplicas sem a necessidade de utilização do ReDaC, pertencente ao CIAO, de forma que versões mais recentes do conjunto ACE+TAO+CIAO possam ser utilizadas.

APÊNDICE A

DESCRITOR XML DE IMPLANTAÇÃO DO GESREP

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<Deployment:deploymentPlan
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">

  <label>GesRep-DeploymentPlan</label>
  <UUID>GesRep_UUID0001</UUID>

 <realizes>

   <label>GesRep-realizes</label>
   <UUID>2c56c1cd-d7c7-46d5-9afc-ef096db6ff90</UUID>
   <specificType></specificType>
   <supportedType></supportedType> <!--IDL:GesAdministrador/ReplicaIF:1.0 -->
   <port>
    <name>replica_receptaculo</name>
    <specificType>
     IDL:GesAdministrador/ReplicaIF/ProviderBaseFacet:1.0
    </specificType>
    <supportedType>
     IDL:GesAdministrador/ReplicaIF/ProviderBaseFacet:1.0
```

```
    </supportedType>
    <provider>false</provider>
    <exclusiveProvider>false</exclusiveProvider>
    <exclusiveUser>true</exclusiveUser>
    <optional>false</optional>
    <kind>SimplexReceptacle</kind>
  </port>

    <port>
      <name>grupo_receptaculo</name>
      <specificType>
       IDL:GesAdministrador/GrupoIF/ProviderBaseFacet:1.0
      </specificType>
      <supportedType>
       IDL:GesAdministrador/GrupoIF/ProviderBaseFacet:1.0
      </supportedType>
      <provider>false</provider>
      <exclusiveProvider>false</exclusiveProvider>
      <exclusiveUser>true</exclusiveUser>
      <optional>false</optional>
      <kind>SimplexReceptacle</kind>
    </port>

  <port>
      <name>monitor_faceta</name>
      <specificType>
       IDL:GestorQoS/MonitorIF/ProviderBaseFacet:1.0
      </specificType>
      <supportedType>
```

```
       IDL:GestorQoS/MonitorIF/ProviderBaseFacet:1.0
      </supportedType>
      <provider>false</provider>
      <exclusiveProvider>false</exclusiveProvider>
      <exclusiveUser>true</exclusiveUser>
      <optional>false</optional>
      <kind>SimplexReceptacle</kind>
    </port>

 </realizes>


<implementation id="Teste">
    <name>Teste</name>
    <source/>
    <artifact>Teste_exec</artifact>
    <artifact>Teste_stub</artifact>
    <artifact>Teste_svnt</artifact>
  </implementation>

  <implementation id="Replica">
    <name>Replica</name>
    <source/>
    <artifact>Replica_exec</artifact>
    <artifact>Replica_stub</artifact>
    <artifact>Replica_svnt</artifact>
  </implementation>

<implementation id="Grupo">
```

```
    <name>Grupo</name>
    <source/>
    <artifact>Grupo_exec</artifact>
    <artifact>Grupo_stub</artifact>
    <artifact>Grupo_svnt</artifact>
  </implementation>

<implementation id="Monitor">
    <name>Monitor</name>
    <source/>
    <artifact>Monitor_exec</artifact>
    <artifact>Monitor_stub</artifact>
    <artifact>Monitor_svnt</artifact>
  </implementation>

<implementation id="Gestor">
    <name>Gestor</name>
    <source/>
    <artifact>Gestor_exec</artifact>
    <artifact>Gestor_stub</artifact>
    <artifact>Gestor_svnt</artifact>
  </implementation>

 <instance id="TesteInstance">
    <name>TesteInstance</name>
    <node>MainNode</node>
    <source/>
    <implementation>Teste</implementation>
    <configProperty>
```

```
      <name>RegisterNaming</name>

      <value>
        <type>
         <kind>tk_string</kind>
        </type>
        <value>
          <string>TesteAdministrador</string>
        </value>
      </value>

    </configProperty>
 </instance>

<instance id="TesteInstance1">
    <name>TesteInstance1</name>
    <node>TresNode</node>
    <source/>
    <implementation>Teste</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
         <kind>tk_string</kind>
        </type>
        <value>
          <string>TesteAdministradorTres</string>
        </value>
```

```
      </value>

    </configProperty>
</instance>

<instance id="TesteInstance0">
    <name>TesteInstance0</name>
    <node>SemaNode</node>
    <source/>
    <implementation>Teste</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
         <kind>tk_string</kind>
        </type>
        <value>
          <string>TesteAdministradorSema</string>
        </value>
      </value>

    </configProperty>

</instance>

<instance id="ReplicaInstance">
    <name>ReplicaInstance</name>
    <node>MainNode</node>
```

```
    <source/>
    <implementation>Replica</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
         <kind>tk_string</kind>
        </type>
        <value>
          <string>ReplicaAdministrador</string>
        </value>
      </value>

    </configProperty>


  </instance>

<instance id="GrupoInstance">
    <name>GrupoInstance</name>
    <node>MainNode</node>
    <source/>
    <implementation>Grupo</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
```

```
          <kind>tk_string</kind>
        </type>
        <value>
          <string>GrupoAdministrador</string>
        </value>
      </value>

    </configProperty>


  </instance>

<instance id="MonitorInstance">
    <name>MonitorInstance</name>
    <node>MainNode</node>
    <source/>
    <implementation>Monitor</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>MonitorAdministrador</string>
        </value>
      </value>
```

```
    </configProperty>


  </instance>
  <instance id="GestorInstance">
    <name>GestorInstance</name>
    <node>MainNode</node>
    <source/>
    <implementation>Gestor</implementation>
    <configProperty>
      <name>RegisterNaming</name>

      <value>
        <type>
         <kind>tk_string</kind>
        </type>
        <value>
          <string>GestorAdministrador</string>
        </value>
      </value>

    </configProperty>


  </instance>

  <connection>
    <name>Replica_Grupo</name>
    <internalEndpoint>
```

```
      <portName>replica_receptaculo</portName>
      <kind>Facet</kind>
      <instance>ReplicaInstance</instance>
    </internalEndpoint>
    <internalEndpoint>
      <portName>replica_receptaculo</portName>
      <kind>SimplexReceptacle</kind>
      <instance>GrupoInstance</instance>
    </internalEndpoint>
  </connection>

 <connection>
    <name>Monitor_Grupo</name>
    <internalEndpoint>
      <portName>grupo_receptaculo</portName>
      <kind>Facet</kind>
      <instance>GrupoInstance</instance>
    </internalEndpoint>
    <internalEndpoint>
      <portName>grupo_receptaculo</portName>
      <kind>SimplexReceptacle</kind>
      <instance>MonitorInstance</instance>
    </internalEndpoint>
  </connection>

  <connection>
    <name>Gestor_Monitor</name>
    <internalEndpoint>
      <portName>monitor_faceta</portName>
```

```
      <kind>Facet</kind>
      <instance>MonitorInstance</instance>
    </internalEndpoint>
    <internalEndpoint>
      <portName>monitor_faceta</portName>
      <kind>SimplexReceptacle</kind>
      <instance>GestorInstance</instance>
    </internalEndpoint>
  </connection>

 <artifact id="Grupo_stub">
    <name>Grupo_stub</name>
    <source/>
    <node/>
    <location>Grupo_stub</location>
  </artifact>

  <artifact id="Grupo_svnt">
    <name>Grupo_svnt</name>
    <source/>
    <node/>
    <location>Grupo_svnt</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
```

```
          <string>create_GesAdministrador_GrupoHome_Servant</string>
        </value>
      </value>
    </execParameter>
  </artifact>

  <artifact id="Grupo_exec">
    <name>Grupo_exec</name>
    <source/>
    <node/>
    <location>Grupo_exec</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_GesAdministrador_GrupoHome_Impl</string>
        </value>
      </value>
    </execParameter>
  </artifact>

 <artifact id="Monitor_stub">
    <name>Monitor_stub</name>
    <source/>
    <node/>
    <location>Monitor_stub</location>
```

```
  </artifact>

  <artifact id="Monitor_svnt">
    <name>Monitor_svnt</name>
    <source/>
    <node/>
    <location>Monitor_svnt</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_GestorQoS_MonitorHome_Servant</string>
        </value>
      </value>
    </execParameter>
  </artifact>

  <artifact id="Monitor_exec">
    <name>Monitor_exec</name>
    <source/>
    <node/>
    <location>Monitor_exec</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
```

```
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_GestorQoS_MonitorHome_Impl</string>
        </value>
      </value>
    </execParameter>
  </artifact>

 <artifact id="Replica_stub">
    <name>Replica_stub</name>
    <source/>
    <node/>
    <location>Replica_stub</location>
  </artifact>

  <artifact id="Replica_svnt">
    <name>Replica_svnt</name>
    <source/>
    <node/>
    <location>Replica_svnt</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_GesAdministrador_ReplicaHome_Servant</string>
```

```
        </value>
      </value>
    </execParameter>
  </artifact>

  <artifact id="Replica_exec">
    <name>Replica_exec</name>
    <source/>
    <node/>
    <location>Replica_exec</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_GesAdministrador_ReplicaHome_Impl</string>
        </value>
      </value>
    </execParameter>
  </artifact>

  <artifact id="Gestor_stub">
    <name>Gestor_stub</name>
    <source/>
    <node/>
    <location>Gestor_stub</location>
  </artifact>
```

```
<artifact id="Gestor_svnt">
  <name>Gestor_svnt</name>
  <source/>
  <node/>
  <location>Gestor_svnt</location>
  <execParameter>
    <name>entryPoint</name>
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>create_GestorQoS_GestorHome_Servant</string>
      </value>
    </value>
  </execParameter>
</artifact>

<artifact id="Gestor_exec">
  <name>Gestor_exec</name>
  <source/>
  <node/>
  <location>Gestor_exec</location>
  <execParameter>
    <name>entryPoint</name>
    <value>
      <type>
        <kind>tk_string</kind>
```

```
        </type>
        <value>
          <string>create_GestorQoS_GestorHome_Impl</string>
        </value>
      </value>
    </execParameter>
  </artifact>

  <artifact id="Teste_stub">
    <name>Teste_stub</name>
    <source/>
    <node/>
    <location>Teste_stub</location>
  </artifact>

  <artifact id="Teste_svnt">
    <name>Teste_svnt</name>
    <source/>
    <node/>
    <location>Teste_svnt</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_TesteAdm_TesteHome_Servant</string>
        </value>
```

```
      </value>
    </execParameter>
  </artifact>

  <artifact id="Teste_exec">
    <name>Teste_exec</name>
    <source/>
    <node/>
    <location>Teste_exec</location>
    <execParameter>
      <name>entryPoint</name>
      <value>
        <type>
          <kind>tk_string</kind>
        </type>
        <value>
          <string>create_TesteAdm_TesteHome_Impl</string>
        </value>
      </value>
    </execParameter>
  </artifact>

</Deployment:deploymentPlan>
```

Este é um exemplo de arquivo descritor de implantação para o GesRep. Este exemplo contém o componente Teste, utilizado para testes do GesRep. Este componente tem por finalidade iniciar um contador contínuo, de forma que, mesmo com a falha da réplica primária, o contador não seja interrompido.

O componente Teste está configurado para ser replicado em três *hosts* diferentes,

sendo eles *MainNode*, SemaNode e TrêsNode. Para cada *host* a ser utilizado, uma nova instância do componente replicado deve ser criada no respectivo *host*.

APÊNDICE B

INTERFACES IDL DOS COMPONENTES DO GESREP

1) **Módulo Administrador**

- **Componente Réplica**

```
#ifndef REPLICA_IDL
#define REPLICA_IDL
#include <Components.idl>

module GesAdministrador
{
  interface ReplicaIF
  {
    //interface do modulo administrador - parte inicial

    void addReplica(
     in string uuid,
     in string instanceName,
     in string nodeName,
     in string implementation_name,
     in string ns_name_register);
    void removeReplica(
     in string uuid,
     in string instanceName);
    void removeReplicaInstancia(
     in string uuid,
```

```
      in long numInstancia);
    void atualizaConexoes(
     in long numIdInstanciaOld,
     in long numIdInstanciaNew);
    void atualizaConexoesUuid(
     in string uuid ,
     in long numIdInstanciaOld,
     in long numIdInstanciaNew);
    long getNumInstanceReplica(
     in string uuid,
     in string idReplica);

  };


component Replica supports ReplicaIF
  {
provides ReplicaIF replica_receptaculo;

  };

  home ReplicaHome manages Replica
  {
  };


};

#endif
```

- **Componente Grupo**

```
#ifndef GRUPO_IDL
#define GRUPO_IDL
#include "../Replica/Replica.idl"
#include <Components.idl>
#include <orb.idl>

module GesAdministrador
{
  interface GrupoIF
  {
    //interface do modulo administrador - gerenciamento de grupo de replicas

    void criarGrupo(
     in string host,
     in long qtdMinReplicas,
     in string uuid,
     in string instanceName,
     in string nodeName,
     in string implementation_name,
     in long tipoReplicacao);
    void removerGrupo(
     in string host,
     in string idGrupo,
     in string uuid);
    void addReplicaGrupo(
     in string idGrupo,
     in string idReplica);
```

```
        void removerReplicaGrupo(
         in string idGrupo,
         in string idReplica,
         in string uuid);
        long pesquisarGrupo(
         in string idGrupo);
        long getQtdMinReplicasGrupo(
         in string idGrupo);
        long getQtdAtualReplicasGrupo(
         in string idGrupo);
        void reconfigurarGrupo(
         in string idGrupo,
         in long tempoVida,
         in long qtdMinReplicas);
        long getMaxNumNomeReplica(
         in string idGrupo);
        void atualizaConexoesReplica(
         in string nomeGrupoReplica,
         in string nomeReplica,
         in string UUID);
        void salvaGrupo(
         in string idGrupo);
        string getNomeReplicaAtiva(
         in string idGrupo);

      };


    component Grupo supports GrupoIF
```

```
  {

uses ::GesAdministrador::ReplicaIF replica_receptaculo;
provides GrupoIF grupo_receptaculo;

  };

  home GrupoHome manages Grupo
  {
  };


};

#endif /* CIAO_HELLO_IDL */
```

2) **Módulo Gestor de QoS**

- **Componente Monitor**

```
#ifndef MONITOR_IDL
#define MONITOR_IDL

#include "../Grupo/Grupo.idl"
#include <Components.idl>

module GestorQoS
{
  interface MonitorIF
  {
```

```
    //componente de monitoramento de replicas

     void monitorarReplica();
    void associarReplica(
     in string grupoReplica,
     in string uuid,
     in string nsName,
     in string nodeName,
     in string instanceName,
     in string implementationName,
     in long tipoReplicacao);
    string getNomeReplica();
    void reconfigurarGrupo(
     in string idGrupo,
     in long qtdMinReplicas);

  };


component Monitor supports MonitorIF
  {

uses ::GesAdministrador::GrupoIF grupo_receptaculo;
provides MonitorIF monitor_faceta;


  };

  home MonitorHome manages Monitor
```

```
  {
  };


};

#endif
```

- **Componente Gestor**

```
#ifndef GESTORQOS_IDL
#define GESTORQOS_IDL

#include "../Monitor/Monitor.idl"
#include <Components.idl>

module GestorQoS
{
  interface GestorIF
  {
    //interface do modulo gestorQoS - gerenciamento de QOS

    void reconfigurarGrupoReplica();


  };


component Gestor supports GestorIF
  {
```

```
    uses ::GestorQoS::MonitorIF monitor_faceta;



      };

      home GestorHome manages Gestor
      {
      };


    };

    #endif
```

3) **Interface Replicação**

```
#ifndef REPLICACAO_IDL
#define REPLICACAO_IDL

#include <orb.idl>


  interface ReplicacaoIF
  {
    string salvaEstado();
    long recuperaEstado(
     in string estadoSalvo);
    long estaVivo();
```

```
        void setTipoReplicacao(
         in long tipoReplicacao);
        long getTipoReplicacao();

      };

    #endif
```

APÊNDICE C

EXEMPLO DE ARQUIVO MPC (*MAKE PROJECT CREATOR*) PARA UTILIZAR O GESREP

```
project(ARCOSDAISSimulatedCarProvider_DnC_stub): ciao_client_dnc {
  avoids += ace_for_tao
  after += OMGDAIS_DnC_stub \
   ARCOSDAISDAGroup_DnC_stub \
   ARCOSDAISDANode_DnC_stub \
   ARCOSDAISProviderBase_DnC_stub
  sharedname = ARCOSDAISSimulatedCarProvider_DnC_stub
  idlflags += -Wb,stub_export_macro=ARCOSDAISSIMULATEDCARPROVIDER_STUB_Export \
    -Wb,stub_export_include=ARCOSDAISSimulatedCarProvider_stub_export.h \
    -Wb,skel_export_macro=ARCOSDAISSIMULATEDCARPROVIDER_SVNT_Export \
    -Wb,skel_export_include=ARCOSDAISSimulatedCarProvider_svnt_export.h \
  -I${TAO_ROOT}/orbsvcs/orbsvcs/
  -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
  -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
  -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
  -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
  -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/

  dynamicflags   = ARCOSDAISSIMULATEDCARPROVIDER_STUB_BUILD_DLL
  includes += ${TAO_ROOT}/orbsvcs/orbsvcs/
   ${TAO_ROOT}/orbsvcs/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
```

```
   ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/

  libs += OMGDAIS_DnC_stub \
  ARCOSDAISDAGroup_DnC_stub \
  ARCOSDAISDANode_DnC_stub \
  ARCOSDAISProviderBase_DnC_stub

  IDL_Files {
    ARCOSDAISSimulatedCarProvider.idl
  }
  Source_Files {
    ARCOSDAISSimulatedCarProviderC.cpp
  }

  Header_Files {
    ARCOSDAISSimulatedCarProviderC.h
  }

  Inline_Files {
    ARCOSDAISSimulatedCarProviderC.inl
  }
}
project(ARCOSDAISSimulatedCarProvider_DnC_svnt) : ciao_servant_dnc {
  avoids += ace_for_tao
  after +=  OMGDAIS_DnC_svnt \
    ARCOSDAISDAGroup_DnC_svnt \
    ARCOSDAISDANode_DnC_svnt \
```

```
    ARCOSDAISProviderBase_DnC_svnt \
    ARCOSDAISSimulatedCarProvider_DnC_stub
  sharedname  = ARCOSDAISSimulatedCarProvider_DnC_svnt
  libs    += ARCOSDAISSimulatedCarProvider_DnC_stub \
     OMGDAIS_DnC_stub \
     OMGDAIS_DnC_svnt \
     ARCOSDAISDAGroup_DnC_stub \
     ARCOSDAISDANode_DnC_stub \
     ARCOSDAISDAGroup_DnC_svnt \
     ARCOSDAISDANode_DnC_svnt \
     ARCOSDAISProviderBase_DnC_stub \
     ARCOSDAISProviderBase_DnC_svnt \
     Replicacao_stub \ Replicacao_svnt

  idlflags  += -Wb,stub_export_macro=ARCOSDAISSIMULATEDCARPROVIDER_STUB_Export \
    -Wb,stub_export_include=ARCOSDAISSimulatedCarProvider_stub_export.h \
    -Wb,skel_export_macro=ARCOSDAISSIMULATEDCARPROVIDER_SVNT_Export \
    -Wb,skel_export_include=ARCOSDAISSimulatedCarProvider_svnt_export.h \
   -I${TAO_ROOT}/orbsvcs/orbsvcs/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/

  dynamicflags = ARCOSDAISSIMULATEDCARPROVIDER_SVNT_BUILD_DLL
  cidlflags -= --
  cidlflags += -I${TAO_ROOT}/orbsvcs/orbsvcs/
   -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
```

```
 -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
 -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
 -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
 -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/ --

includes += ${TAO_ROOT}/orbsvcs/orbsvcs/
 ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
  ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
  ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
 ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
 ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/

CIDL_Files {
  ARCOSDAISSimulatedCarProvider.cidl
}

IDL_Files {
  ARCOSDAISSimulatedCarProviderE.idl
}

Source_Files {
  ARCOSDAISSimulatedCarProviderEC.cpp
  ARCOSDAISSimulatedCarProviderS.cpp
  ARCOSDAISSimulatedCarProvider_svnt.cpp
}

Header_Files {
  ARCOSDAISSimulatedCarProviderEC.h
  ARCOSDAISSimulatedCarProviderS.h
```

```
    ARCOSDAISSimulatedCarProvider_svnt.h
  }

  Inline_Files {
    ARCOSDAISSimulatedCarProviderEC.inl
    ARCOSDAISSimulatedCarProviderS.inl
  }
}

project(ARCOSDAISSimulatedCarProvider_DnC_exec) : ciao_component_dnc {
  avoids += ace_for_tao
  after     += ARCOSDAISSimulatedCarProvider_DnC_svnt
  sharedname = ARCOSDAISSimulatedCarProvider_DnC_exec
  libs      += ARCOSDAISSimulatedCarProvider_DnC_stub \
               ARCOSDAISSimulatedCarProvider_DnC_svnt \
               OMGDAIS_DnC_stub \
               OMGDAIS_DnC_svnt \
               ARCOSDAISDAGroup_DnC_stub \
               ARCOSDAISDANode_DnC_stub \
               ARCOSDAISDAGroup_DnC_svnt \
               ARCOSDAISDANode_DnC_svnt \
               ARCOSDAISProviderBase_DnC_stub \
               ARCOSDAISProviderBase_DnC_svnt \
         Replicacao_stub

  dynamicflags   = ARCOSDAISSIMULATEDCARPROVIDER_EXEC_BUILD_DLL
  includes += ${TAO_ROOT}/orbsvcs/orbsvcs/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/
   ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/
```

```
    ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/
    ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/
    ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/

  IDL_Files {
  }

  Source_Files {
    ARCOSDAISSimulatedCarProvider_exec.cpp
  }

  Header_Files {
    ARCOSDAISSimulatedCarProvider_exec.h
  }
}
```

Este é um exemplo de arquivo MPC de um componente configurado para ser utilizado pelo GesRep.

No projeto *ARCOSDAISSimulatedCarProvider_DnC_svnt*, em libs são incluídas as bibliotecas Replicacao_stub e Replicacao_svnt, da interface Replicação do GesRep.

No projeto *ARCOSDAISSimulatedCarProvider_DnC_exec*, em libs é incluída a biblioteca Replicacao_stub.

As libs do GesRep irão ser incluídas no conjunto de bibliotecas do componente que será replicado no momento da compilação do mesmo, que teve sua interface estendida da interface Replicação.

APÊNDICE D

DIAGRAMAS

Este apêndice apresenta os diagramas concebidos para facilitar o entendimento e desenvolvimento do GesRep, sendo eles diagramas de caso de uso, sequência e classes, acompanhados das devidas explicações.

D.1 DIAGRAMAS DE CASO DE USO

A função de um diagrama de caso de uso é ilustrar as funcionalidades atendidas por um determinado sistema. Um caso de uso é composto por atores, que irão interagir diretamente para a realização de uma determinada funcionalidade no sistema, sendo a relação entre estes atores ilustrada também no diagrama. Uma outra finalidade deste tipo de diagrama é apresentar os atores externos do sistema de uma forma fácil [19].

A figura D.1 ilustra o diagrama de casos de uso do GesRep. O diagrama apresenta como atores a classe CriarReplica, os componentes Grupo, Monitor, Réplica, GestorQoS e o componente que está sendo replicado pelo GesRep, e as *threads* Grupo*Thread* e Monitor*Thread*.

Os casos de uso serão descritos detalhadamente a seguir:

- **Componente Réplica cria uma nova réplica no sistema**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao iniciar sua atividade, ou ao receber uma solicitação de criação de réplicas proveniente do componente Grupo, irá criar uma nova réplica do componente solicitado.

- **Gestor de Réplica Remove Réplica do sistema**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao detectar uma falha do tipo crash em uma réplica criada, irá remover esta réplica do sistema e do grupo de réplicas. Caso o grupo venha ficar vazio com a remoção, este também será removido. Este método se adequa mais ao componente Monitor, pois este é criado sempre na máquina principal (MainNode). Réplicas de componentes que falharam são removidas em caso de falha do NodeManager no qual estão instaladas, porém a máquina continua ativa.
- **Gestor de Replicação Remove réplica do sistema pelo número da instância**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao iniciar suas atividades, irá remover a réplica inicial criada pelo sistema do componente alvo, que não está sendo gerenciada pelo Gestor. Esta remoção será feita através do número da instância do componente no arquivo descritor de implantação da aplicação à qual o componente alvo pertença.
- **GesRep atualiza conexões de réplica**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao remover a réplica ativa e que está fazendo o papel de primária, irá eleger como nova líder do grupo de réplicas a próxima réplica do grupo, migrando as conexões anteriormente feitas com a réplica a ser removida para a nova réplica líder.

- **GesRep atualiza conexões de componente com UUID**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, através do identificador único do descritor de implantação da aplicação que fará uso do Gestor, o UUID, e da identificação do componente a ter as conexões atualizadas, deverá migrar as conexões de um componente que estará sendo removido para o componente que será o alvo da atualização.
- **Componente Grupo cria novo grupo de réplicas no sistema**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O Gestor de Réplicas, ao criar uma réplica, irá verificar se existe ou não um grupo previamente criado para aquele tipo de componente replicado. Caso não exista, um novo grupo será criado.
- **Componente Grupo adiciona nova réplica a grupo já existente**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O Gestor de Réplicas, ao criar uma réplica de um componente que já possui um grupo pré-existente, irá adicionar esta nova réplica ao grupo encontrado, sem ter necessidade de criação de novo grupo.
- **Componente Grupo exclui réplica de grupo**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;

  - **Descrição** - O Gestor de Réplicas, após detectar alguma falha em uma réplica de um determinado grupo, irá excluir a mesma, através de manipulação do plano de implantação da aplicação e, em seguida, remover a referência a ela do grupo ao qual estava associada.

- **Componente Grupo exclui Grupo de Réplicas**;

  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O Gestor de Réplicas, ao detectar que todas as réplicas de um grupo falharam, irá remover todo o grupo e suas réplicas.

- **Componente Grupo pesquisa existência de grupo de réplicas**;

  - **Tipo** - Secundária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao receber uma solicitação para criação de nova réplica, irá pesquisar a existência prévia de um grupo do tipo da réplica solicitada. Caso o grupo exista, a referência a este grupo será retornada para que a referência para a nova réplica seja inserida no mesmo.

- **Componente Grupo recebe solicitação de reconfiguração de grupo**;

  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao receber a solicitação de reconfiguração do parâmetro de quantidade mínima de réplicas em um grupo, irá adequar o grupo solicitado à nova realidade, criando ou removendo réplicas para que o novo número mínimo seja alcançado. Este número deve ser superior a 1.

- **Componente Grupo salva visão interna de grupo**;

  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O GesRep, ao ter a visão de um determinado grupo modificada, irá armazenar a mesma em modo persistente, permitindo que a réplica ativa na replicação passiva possa ter o conhecimento das demais réplicas do grupo. A visão do grupo de réplicas se altera nos seguintes casos: remoção de réplica de grupo, com inserção ou não de nova réplica; inicialização do GesRep e término da criação inicial do grupo de réplicas.

- **Componente Grupo retorna o nome da réplica ativa**;
  - **Tipo** - Secundária;
  - **Ator** - Componente Replicado;
  - **Descrição** - O GesRep, para permitir que o componente cliente do componente que está sendo replicado possa se conectar a ele via servidor de nomes, retorna o nome da réplica primária a este componente cliente.

- **Componente grupo recebe solicitação para atualizar conexões de réplica**;
  - **Tipo** - Secundária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O componente grupo, ao receber uma solicitação para atualizar a conexão de um determinado componente, repassa esta solicitação ao componente réplica, para que as conexões possam ser atualizadas.

- **Componente Gestor recebe solicitação para reconfigurar quantidade mínima de réplicas de grupo**;
  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica, usuário da aplicação cliente;

  - **Descrição** - O GesRep, ao receber uma solicitação do usuário para reconfigurar a quantidade mínima de réplicas em um grupo, irá repassar esta configuração ao componente Grupo para que a quantidade de réplicas seja ajustada.

- **Monitor monitora réplica a ele associada**;

  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - O componente Monitor irá, periodicamente, enviar mensagens de ping à réplica a ele associada para verificar se houve alguma falha por parada do NodeManager ou da máquina. Caso houve alguma falha, o Monitor responsável deverá solicitar a remoção da réplica e, caso o número de réplicas no grupo após a remoção seja menor que o número mínimo de réplicas para o grupo, solicitar a criação de novas réplicas até que estes números se igualem.

- **Réplica é associada ao monitor**;

  - **Tipo** - Primária;
  - **Ator** - Gestor de Réplica;
  - **Descrição** - Uma réplica, ao ser criada pelo GesRep, terá um novo componente monitor criado que deverá receber informações relativas à réplica que será monitorada. A partir do recebimento destas informações, o monitor se encontrará apto a realizar o monitoramento da réplica.

- **Reconfiguração de grupo de réplica**.

  - **Tipo** - Secundária;
  - **Ator** - Gestor de Réplica, usuário da aplicação cliente;
  - **Descrição** - O monitor, ao receber a solicitação para reconfigurar um grupo de réplicas vinda do componente Gestor, irá repassar esta solicitação ao componente Grupo para que a mesma possa ser processada.

D.2 DIAGRAMAS DE SEQÜÊNCIA

Segundo [19], um diagrama de seqüência ilustra os eventos gerados pelos atores externos, num determinado caso de uso, sua ordem e a interação entre sistemas. Nesta seção, serão apresentados os diagramas de seqüência para as principais funcionalidades do GesRep.

No diagrama ilustrado na figura D.2, a aplicação, que está fazendo uso do Gestor de Replicação, requisita ao sistema a criação de nova réplica. Esta requisição é passada, primeiramente, ao componente Grupo, que irá pesquisar a existência de grupo de réplicas para o tipo da réplica solicitada e, caso não encontre o grupo, irá criar um novo. Após a criação do grupo, as réplicas serão criadas através do componente Réplica, que, além das réplicas e suas conexões, criará também os respectivos monitores. Cada monitor deve ser associado a apenas uma réplica, sendo esta associação feita através por requisição do componente Grupo ao componente Monitor. Após a réplica e seu monitor serem criados, o componente grupo irá adicionar a referência da réplica criada ao novo grupo.

Caso o grupo de réplicas já exista para o tipo de componente solicitado, o procedimento se iniciará com a criação das réplicas e seus monitores pelo componente Réplica, associação dos monitores às réplicas pelo componente Grupo e Monitor e posterior associação das referências das réplicas ao grupo pré-existente.

Outra maneira de criação de réplicas é através da ocorrência de falha detectada pelo componente Monitor na réplica de sua responsabilidade. Caso ocorra falha no *NodeManager* ou *host* que hospeda uma réplica, o monitor considerará esta réplica como falha e ela terá sua referência removida do grupo de réplicas. Após a remoção, caso o número de réplicas que restou no grupo seja menor que o número de réplicas que é tido como mínimo, uma solicitação para criação de nova réplica será feita até que o mínimo de réplicas seja atingido. Esta solicitação será encaminhada do componente Monitor ao componente Grupo, que irá repassá-la ao componente Réplica para o processamento.

Existem duas possibilidades para remoção de réplicas no GesRep:

1) Componente Monitor detectou falha do tipo parada, enviando solicitação de remoção da referência da réplica do grupo de réplicas ao componente Grupo que irá processar esta requisição e, em seguida, enviando solicitação de remoção do componente Monitor que estava responsável pela réplica falha, sendo o componente Grupo responsável por repassar esta solicitação para o componente Réplica que processa a requisição;

2) Foi detectada uma reconfiguração em que o número mínimo de réplicas foi reduzido em relação à quantidade mínima anterior. O componente GestorQoS irá repassar a solicitação de reconfiguração para o componente Monitor Administrador (que não está ligado a nenhuma réplica sendo o responsável por gerar os Monitores para associar à réplicas criadas), com o novo mínimo de réplicas, e este irá repassar a solicitação ao componente Grupo. Caso a quantidade de réplicas precise ser ajustada para mais ou para menos, o componente Grupo repassará as requisições de criação ou remoção de réplicas ao componente Réplica.

D.3 DIAGRAMA DE CLASSES

O diagrama de classes do projeto proposto neste documento foi feito por módulos, levando em consideração o módulo Administrador e módulo Gestor de QoS. A explicação de cada diagrama será feita após sua apresentação.

Neste grupo apresentado pela figura D.4 existem as seguintes interfaces:

- **Grupo** - Responsável pela parte de operações sobre grupos criados pelo Gestor. Engloba criação e remoção de referências de réplicas em grupo, gerenciamento dos membros do grupo, dentre outras operações relativas a um grupo de réplicas. Se comunica com o componente Réplica para criação e remoção de réplicas;

- **Replica** - Responsável pelas operações sobre réplicas, incluindo criação, remoção e atualização das conexões de uma réplica.

As classe que implementam estas interfaces são:

- **Replica_exec.cpp** - Representa a implementação do componente Réplica, tendo como atributo o countConexao, representando o número das novas conexões criadas;
- **Grupo_exec.cpp** - Representa a implementação do componente Grupo.

Neste grupo apresentado pela figura D.5 existem as seguintes interfaces:

- **Monitor** - Responsável pelas funções de monitoramento de réplicas. Se conecta com o componente Grupo para criação de novas réplicas e remoção de réplicas;
- **Gestor** - Responsável pela reconfiguração do Grupo de Réplicas. Se conecta ao componente Monitor Administrador (componente mestre, do qual os demais monitores são gerados e que não está associado a nenhuma réplica), para monitoramento de uma reconfiguração.

As classes que implementam estas interfaces são:

- **Monitor_exec.cpp** - Representa a implementação do componente Monitor e possui como atributo a variável tm, representando a thread que será responsável pelo monitoramento da réplica associada a este componente;
- **Gestor_exec.cpp** - Representa a implementação do componente Gestor, possuindo como atributo a variável tm, representando a thread responsável pela verificação periódica de uma reconfiguração.

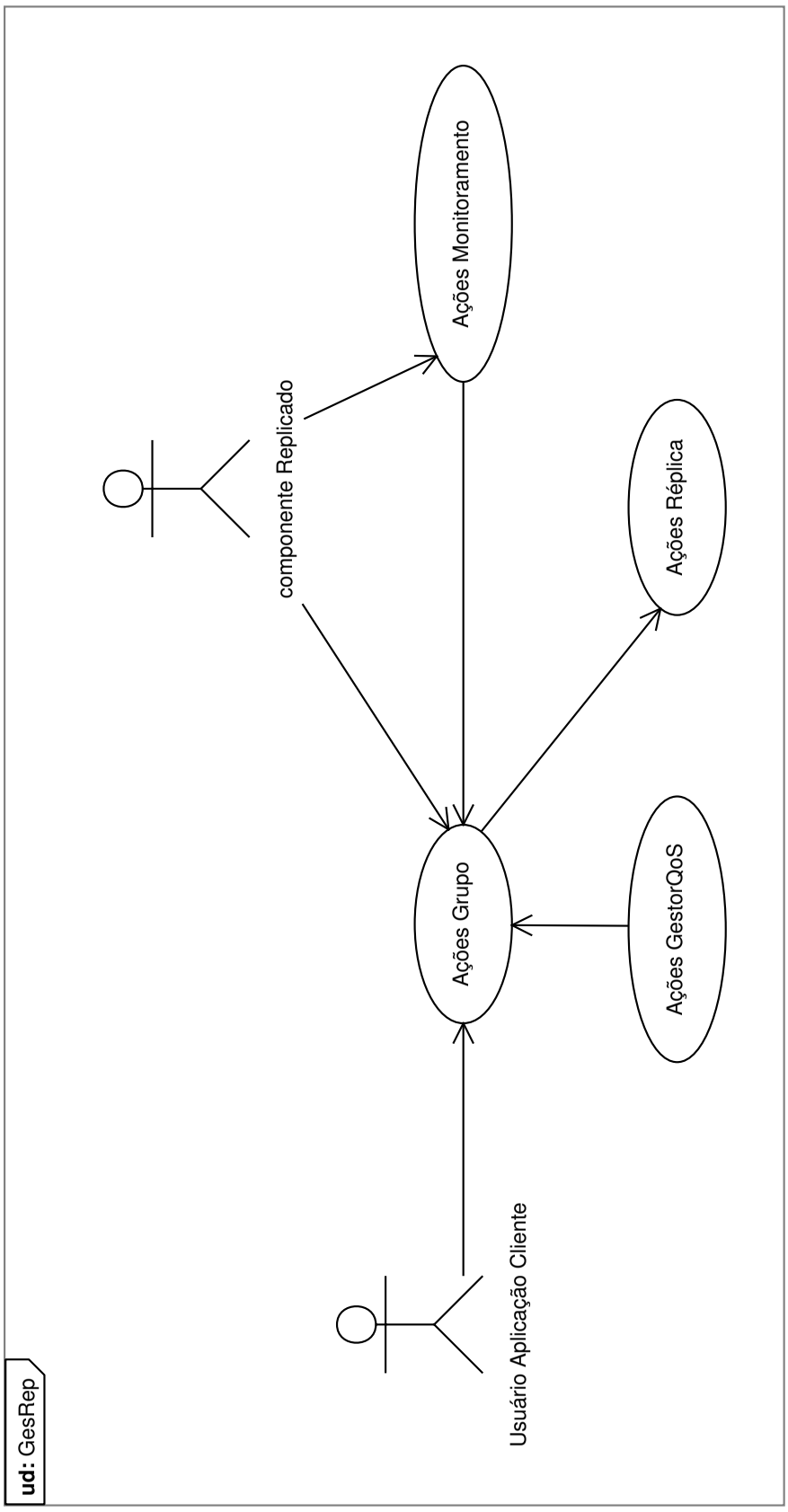

Figura D.1. Diagrama de caso de uso do GesRep.

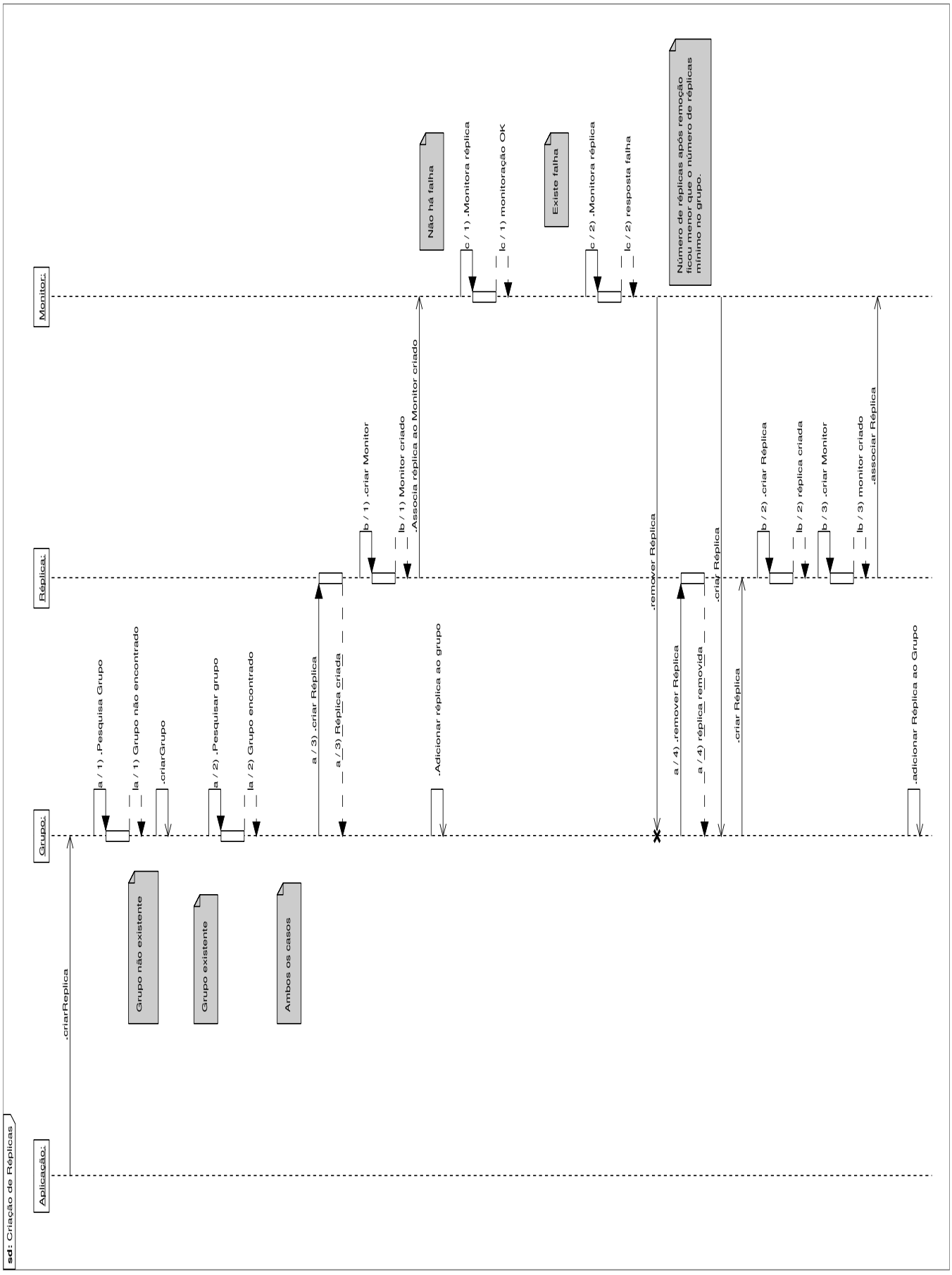

Figura D.2. Diagrama de sequência do caso de uso de Criação de Réplicas.

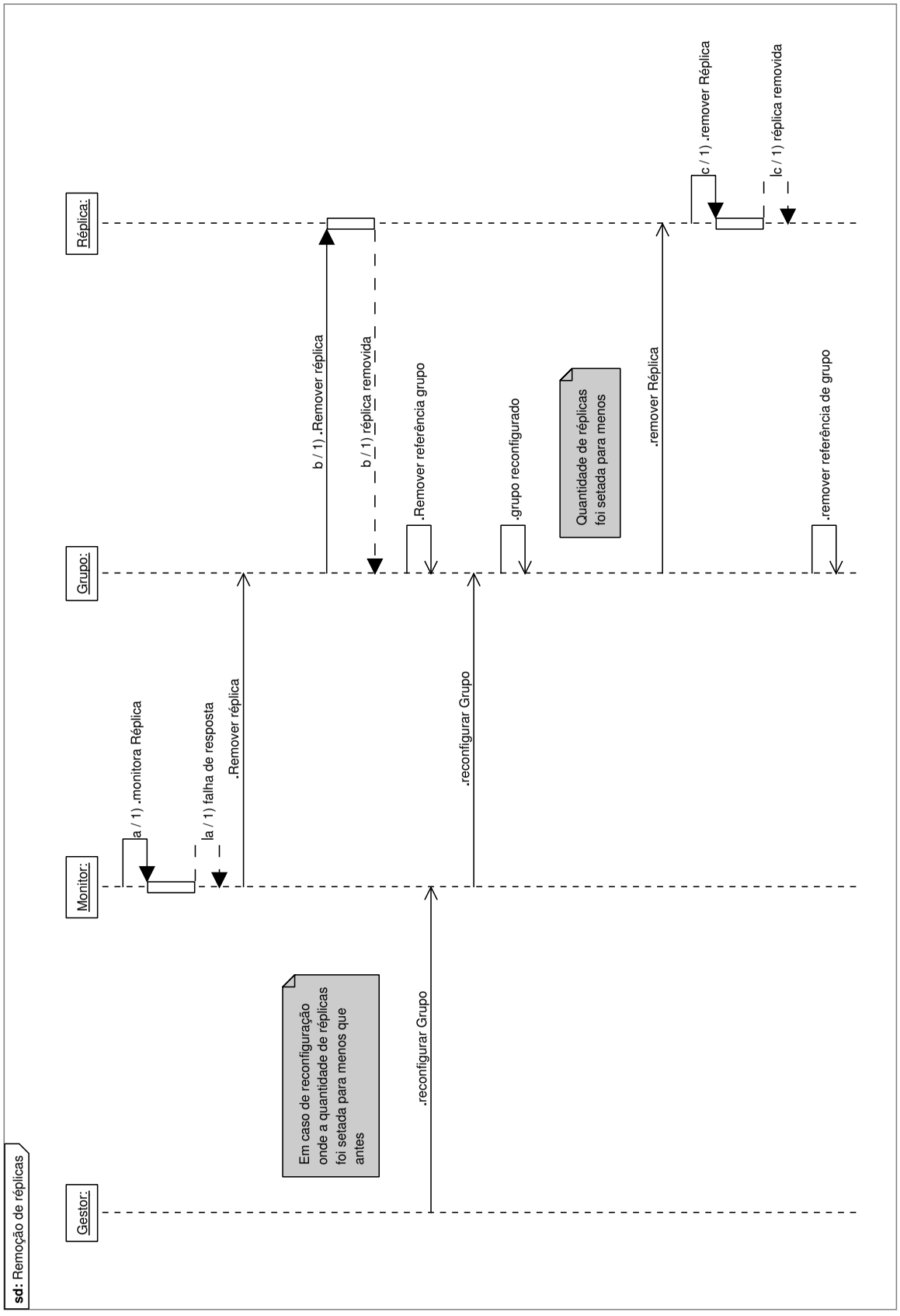

Figura D.3. Diagrama de sequência do caso de uso de Exclusão de Réplicas.

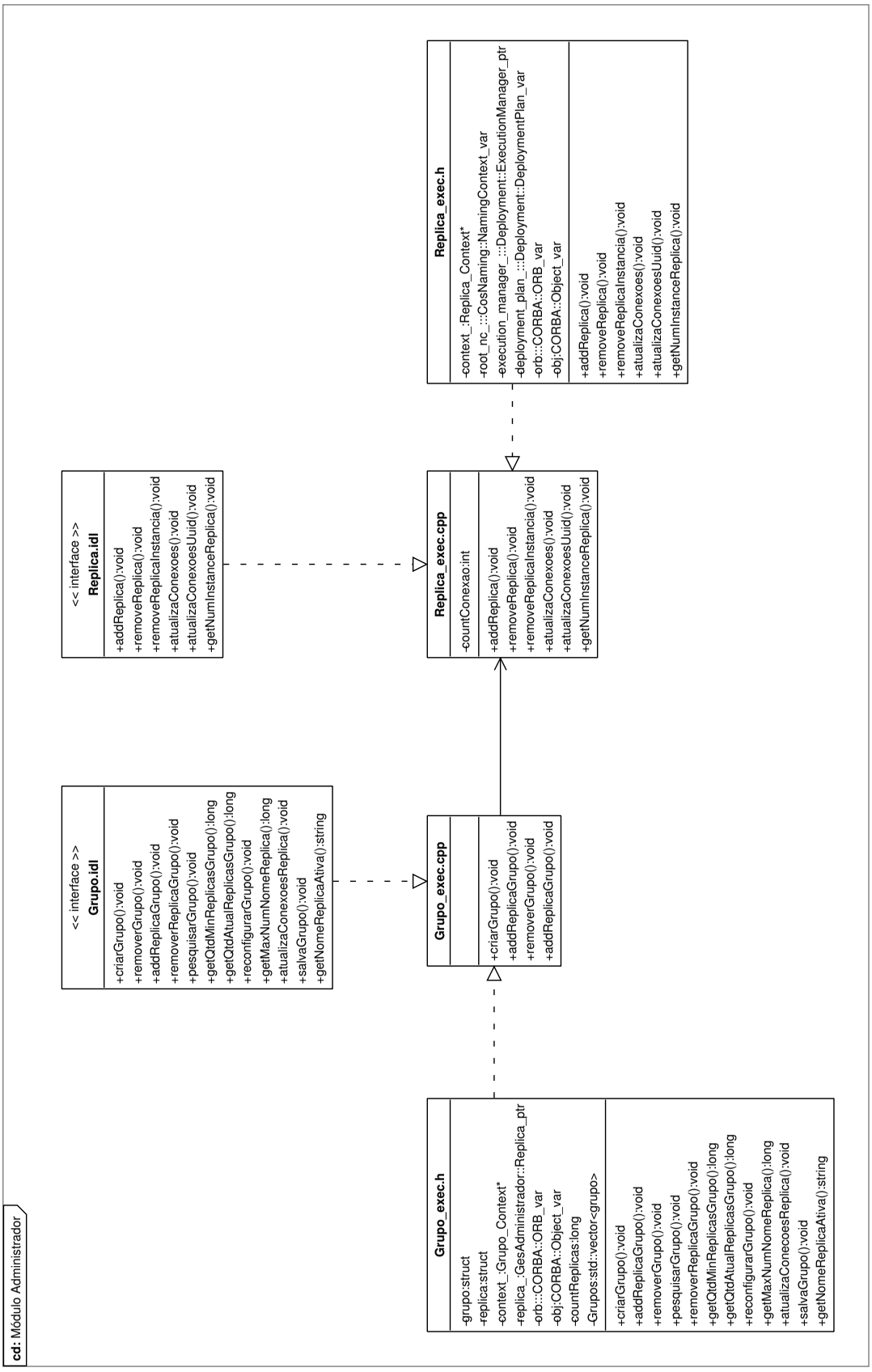

Figura D.4. Diagrama de classes do módulo Administrador.

Figura D.5. Diagrama de classes do módulo GestorQoS.

REFERÊNCIAS

[1] S. Andrade and R. de Araujo Macedo. A component-based real-time architecture for distributed supervision and control applications. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 1, pages 8 pp. –22, 19-22 2005.

[2] S. S. Andrade. Sistemas distribuídos de supervisão e controle baseados em componentes de tempo-real. Masterthesis, Universidade Federal da Bahia, 2005.

[3] S. Bagchi, K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer. The chameleon infrastructure for adaptive, software implemented fault tolerance. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pages 261 – 270, Washington, DC, USA, 1998. IEEE Computer Society.

[4] D. Bakken. Paradigms for distributed fault-tolerance. Notas de aula, School of Electrical Engineering and Computer Science, 2004.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *In Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, pages 187–198, 1992.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. pages 199–216, 1993.

[7] R. Capobianchi, A. Coen-Porisini, D. Mandrioli, and A. Morzenti. A framework architecture for supervision and control systems. *ACM Comput. Surv.*, page 26, 2000.

[8] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34:56–78, 1993.

[9] R. J. de Araújo Macêdo. An integrated group communication infrastructure for hybrid real-time distributed systems. In *IX Workshop on Real-Time Systems*, volume 1, pages 81 – 88, 2007.

[10] B. N. Dept, B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. Doors: Towards high-performance fault tolerant corba. In *in Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA 2000*, pages 39–48, 2000.

[11] J. Fraga, F. Siqueira, and F. Favarim. An adaptive fault-tolerant component model. In *9 th IEEE International Workshop on ObjectOriented Real-Time Dependable Systems, Capri Island*, pages 179–186, 2003.

[12] T. K. George Coulouris, Jean Dollimore. *Distributed Systems - Concepts and Desing*. Addison Wesley, 3a. edição edition, 2000.

[13] T. Goncalves and A. R. Silva. Passive replicator: A design pattern for object replication. Technical report, In The 2 nd European Conference on Pattern Languages of Programming, EuroPLoP '97, 1997.

[14] D. Group. *TAO*, 2008.

[15] F. Jahanian. Consistency and replication (part b). Notas de aula, University of Michingan, 2007.

[16] M. Jazayeri. Distributed systems. Notas de aula, Vienna University of Technology, 2005.

[17] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: A case study using enterprise java beans. *Reliable Distributed Systems, IEEE Symposium on*, 0:89 – 98, 2003.

[18] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[19] C. Larman. *Utilizando UML e Padrões - Uma introdução à anáise e ao projeto orientado a objetos*. Bookman, 1ª edição edition, 2000.

[20] M. C. Little and D. McCue. The replica management system: a scheme for flexible and dynamic replication. In *in The Proceedings of the 2nd International Workshop on Configuration*, pages 46–57, 1994.

[21] M. C. Little and D. L. McCue. The replica management system: a scheme for flexible and dynamic replication. Technical report, 1994.

[22] S. Maffeis. Piranha: A corba tool for high availability. *Computer*, 30(4):59–66, 1997.

[23] V. Marangozova. Component quality configuration: the case of replication. Technical report, In The 8th International Workshop on Component-oriented Programming, 2003.

[24] P. Marti, J. Aguado, F. Rolando, M. Velasco, J. Colomar, and J. Fuertes. A java - based framework for distributed supervision and control of industrial processes. In *7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA99*, volume 1, pages 33 – 8, 1999.

[25] S. MICROSYSTEMS. *JSR 220: Enterprise JavaBeans, Version 3.0*, 2008. Version 3.0.

[26] MYSQL. *MYSQL NDB Cluster*, 2007. Version 5.1.

[27] MYSQL. *MYSQL Replication*, 2007. Version 5.1.

[28] e. a. Nanbor Wang. Configuring real-time aspects in component middleware. In C. Agia Napa, editor, *International Symposium on Distributed Objects and Applications*, number 3291 in Lecture Notes in Computer Science, pages 1520– 1537. Springer Verlag, Oct. 2004.

[29] P. Narasimhan. Fault-tolerant corba: From specification to reality. *Computer*, 40:110–112, 2007.

[30] O. M. G. (OMG). *CORBA Overview*, 1999. Version 2.1.

[31] O. M. G. (OMG). *Fault Tolerant CORBA*, 1999.

[32] O. M. G. (OMG). *Common Object Request Broker - Architecture: Core Specification*, 2002.

[33] N. A. S. P. A. Barret. Towards an Integrated Approach to fault-tolerance in Delta-4. *Distributed Systems Engineering*, 1(2):59 – 66, 1993.

[34] S. M. Pertet. Proactive fault-recovery in distributed systems. Masterthesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2006.

[35] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. S, R. E. Schantz, and M. Seri. Aqua: An adaptive architecture that provides dependable distributed objects. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, 1998.

[36] D. H. Robert Orfali. *Client/ Server Programming with JAVA and CORBA*. Wiley, 2ª edição edition, 1998.

[37] G. T. Santos, L. C. Lung, and C. Montez. Ftweb: A fault tolerant infrastructure for web services. In *Proceedings of the 9th IEEE International Enterprise Computing Conference*, pages 95–105. IEEE Computer Society, 2005.

[38] D. Schimidt. An Architectural Overview of the ACE framework - A case-study of Successful cross-platform Systems Software Reuse. *USENIX Login Magazine*, (Tools Special Issue):257 – 274, 1997.

[39] D. C. Schmidt. Developing distributed object computing application with corba. Notas de aula, Vanderbilt University, 2008.

[40] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21:294–324, 1998.

[41] F. B. Schneider. Replication management using the state-machine approach. pages 169–197, 1993.

[42] O. Security. Securemiddleware - secure, model-driven components platform, June 2010. `http://www.objectsecurity.com/en-products-secmw-tech.html`.

[43] F. V. B. Sheila Regine Vasconcelos. Serviços para Tolerância a Faltas no Ambiente Operacional Seljuk-Amoeba. *Anais do VII Simpósio de Computadores Tolerantes a Falhas*, pages 237 – 251, 1997.

[44] P. Sousa, N. F. Neves, and P. Verissimo. Resilient state machine replication. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, pages 305–309, Washington, DC, USA, 2005. IEEE Computer Society.

[45] M. R. Spiegel. *Estatística*. McGraw - Hill do Brasil Ltda, 1976.

[46] D. H. Vania Marangozova. An Architectural Approach to Replication Configuration. *6th International Conference on Principles of Distributed Systems (OPODIS'2002)*, pages 211 – 222, 2002.

[47] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 26(2):45 – 54, 2003.

[48] T. Wofl. *Replication of Non-deterministic objects*. PhD thesis, Escola Politécnica Federal de Lausanne, 1998.

[49] N. S. Xavier Défago, André Schiper. Semi-passive replication. *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1(17):16 – 28, March 1998.