UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS–GRADUAÇÃO EM COMPUTAÇÃO

Réplicas para Alta Disponibilidade em Arquiteturas Orientadas a Componentes com Suporte de Comunicação de Grupo

por

MARCIA PASIN

Tese de Doutorado submetida à avaliação,
como requisito parcial para a obtenção do grau de Doutor
em Ciência da Computação

Profa. Taisy Silva Weber
Orientadora

Prof. Michel Riveill
Co–orientador

Porto Alegre, julho de 2003.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Pasin, Marcia

Réplicas para Alta Disponibilidade em Arquiteturas Orientadas a Componentes com Suporte de Comunicação de Grupo / por Marcia Pasin. - Porto Alegre : PPGC da UFRGS, 2003.
129p. : il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientadora: Weber, Taisy Silva; Co-orientador: Riveill, Michel.

1. Alta disponibilidade. 2. Replicação. 3. Comunicação de grupo. 4. Orientação a componentes. I. Weber, Taisy Silva. II. Riveill, Michel. III. Título.

Aos meus pais, Lucia e Neuton, e irmãos, Marta e Marcelo, que me ensinaram a gostar da vida acadêmica e a acreditar na universidade pública, com ensino de qualidade e gratuito.

Agradecimentos

Agradeço principalmente e especialmente à Profa. Taisy, que me acompanhou atenciosamente nessa caminhada, desde o princípio, e me cativou com a sua sinceridade, disponibilidade e honestidade.

Agradeço às Profas. Ingrid e Maria Lúcia, aos Profs. Netto, Geyer, Lisboa, Navaux, e ao sempre bem-humorado Prof. Tiaraju, que de alguma forma colaboraram para o meu crescimento pessoal e como profissional na área de Informática.

Aos colegas e amigos do doutorado: Adenaurer, Iara, aos Edsons da sala 245, Eduardo Appel, Hércules, Jorge Barbosa, César Zeferino, Carla Lima e Rodrigo Quites.

Aos amigos dos cafés, almoços, jantares e momentos mais agradáveis vividos no Instituto: Patty (minha tutora afetiva), Ju, Lê, Mozart, Mônica e Bohrer, Emerson, Márcia Cristina (que compartilhou comigo o PC *coçadinha*) e Cadinho.

Agradeço aos colegas Renata de Mattos Galante e Rafael Sagula pela incrível disponibilidade durante a apresentação de trabalhos no CLEI 99.

Aos colegas do *Grupo de Pesquisa de Tolerância a Falhas*, com os quais compartilharam comigo grande parte da minha vida acadêmica: Ceretta, Cechin, Silvia, Pollo, César Ida, Drebes e Mello.

Aos funcionários do II: Silvânia, Sula, Luís Otávio, Ida, Henrique, Bia e Lisiane, pela sempre paciente e constante atenção, e pela agilidade com que os pequenos problemas foram resolvidos.

Ao pessoal da França (ESSI): Prof. Riveill, às Profas. Anne–Marie Pinne e Mireille Blay, à doutoranda Audrey e à secretária Patrícia Lauchame, pelo companheirismo, disponibilidade, e pela atenção durante o meu estágio e após o estágio.

Sumário

Lista de Abreviaturas

ABCAST	Atomic BroadCAST
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	Application Programming Interface
CLR	Common Language Runtime
COM	Component Object Model
COS	CORBA Object Services
COTS	Components Off–The–Shelf
CTS	Common Type System
CORBA	Common Object Request Broker Architecture
CORBA–FT	CORBA–Fault Tolerant
EJB	Enterprise JavaBeans™
FD	Failure Detection
GenIC	Generate Interposition Classes
GMS	Group Membership Service
HSQLDB	Hypersonic Structured Query Language Database
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IP	Internet Protocol
IIOP	Internet Inter–ORB Protocol
IL	Intermediary Language
IOR	Interoperable Object Reference
J2EE	Java™ 2 Enterprise Edition
JDBC	Java™ Database Connectivity
JRE	Java™ 2 Runtime Environment, Standard Edition
J2SE SDK	Java™ 2 SDK, Standard Edition
JIT	Just In Time
JMS	Java™ Message Service
JSP	Java™ Server Pages
JNDI	Java™ Naming and Directory Interface
JOnAS	Java Open Application Server

JRMP	Java™ Remote Method Protocol
JTA	Java™ Transaction API
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MOM	Message Oriented Middleware
MTS	Microsoft Transaction Server
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
PDA	Personal Digital Assistants
RMI	Remote Method Invocation
RMI–IIOP	Remote Method Invocation – Internet Inter–ORB Protocol
ROWA	Read One Write All
RPC	Remote Procedure Call
SQL	Structured Query Language
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TOCAST	Total Order multiCAST
2PL	Two–Phase Locking
UDP	Universal Datagram Protocol
VSCAST	View Synchronous multiCAST
XML	eXtensible Markup Language

Lista de Figuras

Resumo

Alta disponibilidade é uma das propriedades mais desejáveis em sistemas computacionais, principalmente em aplicações comerciais que, tipicamente, envolvem acesso a banco de dados e usam transações. Essas aplicações compreendem sistemas bancários e de comércio eletrônico, onde a indisponibilidade de um serviço pode representar substanciais perdas financeiras. Alta disponibilidade pode ser alcançada através de replicação. Se uma das réplicas não está operacional, outra possibilita que determinado serviço seja oferecido. No entanto, réplicas requerem protocolos que assegurem consistência de estado.

Comunicação de grupo é uma abstração que tem sido aplicada com eficiência a sistemas distribuídos para implementar protocolos de replicação. Sua aplicação a sistemas práticos com transações e com banco de dados não é comum. Tipicamente, sistemas transacionais usam soluções *ad hoc* e sincronizam réplicas com protocolos centralizados, que são bloqueantes e, por isso, não asseguram alta disponibilidade.

A tecnologia baseada em componentes Enterprise JavaBeans (EJB) é um exemplo de sistema prático que integra distribuição, transações e bancos de dados. Em uma aplicação EJB, o desenvolvedor codifica o serviço funcional que é dependente da aplicação, e os serviços não–funcionais são inseridos automaticamente. A especificação EJB descreve serviços não–funcionais de segurança, de transações e de persistência para bancos de dados, mas não descreve serviços que garantam alta disponibilidade.

Neste trabalho, alta disponibilidade é oferecida como uma nova propriedade através da adição de serviços não–funcionais na tecnologia EJB usando abstrações de comunicação de grupo. Os serviços para alta disponibilidade são oferecidos através da arquitetura HA (*highly-available architecture*) que possui múltiplas camadas. Esses serviços incluem replicação, chaveamento de servidor, gerenciamento de membros do grupo e detecção de membros falhos do grupo.

A arquitetura HA baseia-se nos serviços já descritos pela especificação EJB e preserva os serviços EJB existentes. O protocolo de replicação corresponde a uma subcamada, invisível para o usuário final. O serviço EJB é executado por membros em um grupo de réplicas, permitindo a existência de múltiplos bancos de dados idênticos. Conflitos de acesso aos múltiplos bancos de dados são tratados estabelecendo–se uma ordem total para aplicação das atualizações das transações. Esse grupo é modelado como um único componente e gerenciado por um sistema de comunicação de grupo.

A combinação de conceitos de bancos de dados com comunicação de grupo demonstra uma interessante solução para aplicações com requisitos de alta disponibilidade, como as aplicações EJB. Os serviços adicionais da arquitetura HA foram implementados em protótipo. A validação através de um protótipo possibilita que experimentos sejam realizados dentro de um ambiente controlado, usando diferentes cargas de trabalho sintéticas.

O protótipo combina dois sistemas de código aberto. Essa característica permitiu acesso à implementação e não somente à interface dos componentes dos sistemas em questão. Um dos sistemas implementa a especificação EJB e outro implementa o sistema de comunicação de grupos. Os resultados dos testes realizados com o protótipo mostraram a eficiência da solução proposta. A degradação de desempenho pelo uso de réplicas e da comunicação de grupo é mantida em valores adequados.

Palavras–chave: alta disponibilidade, replicação, comunicação de grupo, orientação a componentes.

TITLE: "REPLICAS TO ACHIEVE HIGH AVAILABILITY IN COMPONENT-BASED ARCHITECTURES USING THE GROUP COMMUNICATION ABSTRACTION"

Abstract

High availability is one of the most important properties in computational systems. It is strongly required in commercial applications that, typically, involve database accessing and transactions. These applications include banking systems and electronic commerce, where the unavailability of a service could lead to significant financial losses. High availability can be achieved through replication. If one of the replicas is not operational, another one allows an appropriate service. However, replicas require special protocols to guarantee the consistency of their contents.

The abstraction of group communication has been efficiently applied to implement replication protocols in distributed systems. However, in practice, its use in transactional systems is still rare. Transactional systems often implement *ad hoc* solutions, with centralized approaches and blocking protocols to synchronize the replicas, which harm their high availability.

Enterprise JavaBeans (EJB) component–based technology is an example of a practical system that integrates distribution, transactions and databases. In an EJB application, the developer codifies the functional service that is application–dependent, and the non–functional services are automatically included. The EJB specification describes non–functional services as security, transactions and persistence for databases. It does not describe services that allow high availability.

In this work, high availability is achieved as a new property by the addition of non–functional services in the EJB technology using the group communication abstraction. The services for high availability are provided through a HA (highly–available) architecture which is composed by multiple layers. These services include replication, server switching, group membership and failure detection.

The HA architecture is based on services described by the EJB specification and preserves the existing EJB services. The replication protocol corresponds to a sub–layer that is transparent to the user. The highly–available EJB services are provided by members of a replication group, thus allowing the existence of multiple identical databases. Concurrent accesses to multiple databases are dealt by means of a total-order mechanism for the updates. This group is handled as a single component and is managed by a group communication system. This combination of databases with the group communication abstraction provides an interesting solution for applications with high availability requirements.

We implemented the services of the HA architecture in a prototype. The validation through a prototype allows conducting experiments on a controlled environment, using different synthetic workloads. Our prototype combines two open-source systems. This feature allows accessing both component and interface implementations. One system implements the EJB specification and the other, the group communication system. The results of the tests applied to our prototype shown the efficiency of the proposed solution. The performance degradation due to replication and the group communication overhead is kept in suitable values.

Keywords: high availability, replication, group communication, component–based system.

1 Introdução

Replicação tem sido proposta e usada como uma técnica viável para garantir alta disponibilidade de serviço na presença de falhas em ambientes distribuídos. Projetistas de *software* têm tradicionalmente se preocupado com o aumento de produtividade, o que levou ao desenvolvimento de técnicas de reuso de *software* e à abstração de componentes, e não estão dispostos a implementar serviços não–funcionais, como *protocolos de replicação* que garantem a consistência de réplicas, a partir do zero para cada aplicação. Técnicas e ferramentas que facilitem a implementação de réplicas visando alta disponibilidade são altamente desejáveis.

Muita pesquisa tem sido desenvolvida nesse sentido e a área de replicação não é nova. Falta, entretanto, suficiente experimentação com o assunto e sua aplicação a uma mais vasta gama de sistemas práticos para que os problemas relacionados à replicação, como a degradação de desempenho proporcionada pelos protocolos de replicação, sejam suficientemente compreendidos. Faltam também sistemas práticos onde possam ser efetuadas medidas de desempenho e de disponibilidade.

Tipicamente protocolos de replicação em sistemas de bancos de dados são implementados com conceitos centralizados, usados em sistemas transacionais, e muitas soluções são *ad–hoc* [GAR2003]. Os protocolos centralizados são bloqueantes, i.e., a falha de um processo pode impedir a finalização do protocolo e, por isso, implementam o conceito de *segurança* (*safety*) [LAM77], onde um estado consistente é garantido mesmo em caso de falhas. Entretanto, sistemas altamente disponíveis requerem *sobrevivência* (*liveness*) [LAM77] para garantir que o sistema continue operacional e que o serviço não seja interrompido apesar de falhas.

Protocolos centralizados não tratam *sobrevivência* como parte do protocolo e também não tratam determinismo porque o nodo coordenador do protocolo impõe o determinismo, assegurando que, para diferentes operações executadas na mesma ordem, diferentes réplicas produzirão o mesmo estado. Assegurar determinismo em sistemas de bancos de dados através de protocolos distribuídos é mais complexo, uma vez que não há nodo coordenador para impor o determinismo. Entretanto, protocolos distribuídos garantem alta disponibilidade, pois eles implementam mecanismos que garantem *sobrevivência* apesar de falhas.

A abstração de grupos [BIR87] tem provado ser um mecanismo eficiente para implementar protocolos de consistência de réplicas e baseia–se em protocolos distribuídos. Essa abstração facilita a implementação de determinismo e *sobrevivência* em sistemas distribuídos [GUE97]. Contudo, essa abstração não é muito usada em sistemas práticos que requerem serviços com alta disponibilidade.

A abstração de grupos possibilita que um grupo de réplicas seja modelado como uma única entidade, o que facilita a implementação de protocolos que garantem o *critério de serializabilidade* (ou *one–copy serializability* [BER87, JAL94]). O critério de serializabilidade especifica que quando um objeto é atualizado, suas réplicas também precisam ser atualizadas, para manter um estado distribuído consistente.

Primitivas de comunicação de grupo podem ser usadas para manter o estado distribuído consistente. Essas primitivas permitem que um processo externo ao grupo comunique–se com todos os membros de um grupo enviando apenas uma mensagem a esse grupo.

Propriedades específicas implementadas por essas primitivas asseguram determinismo e sobrevivência apesar de falhas em sistemas distribuídos.

1.1 Objetivo deste trabalho

Neste trabalho, alta disponibilidade é oferecida como uma propriedade implementada através de *serviços não–funcionais adicionais*, que são inseridos automaticamente em uma implementação baseada em componentes. Os *serviços não–funcionais adicionais para alta disponibilidade* serão chamados de serviços HA (ou *highly-available services*), e incluem o *serviço de replicação*, de grupos, de detecção de membros falhos e de *chaveamento*.

Esses serviços podem ser configurados em uma arquitetura baseada em componentes. O *serviço de replicação* implementa um protocolo de replicação. Cada réplica no serviço de replicação é modelada como um membro de um grupo de réplicas. O grupo de réplicas pode ser contatado por clientes externos como se fosse uma única entidade. Os serviços de grupos e de detecção de membros falhos são implementados com ajuda de um sistema de comunicação de grupo [BAN99]. O *serviço de chaveamento* possibilita que um cliente encontre outra réplica no grupo de replicação, se a réplica atual falhar.

Através dos serviços HA, inseridos em uma implementação pré–existente baseada em componentes, para permitir alta disponibilidade, espera–se:

- manter a compatibilidade dos serviços HA com os serviços já oferecidos pela arquitetura baseada em componentes, de forma que o protocolo de replicação seja implementado com invisibilidade para o usuário final, e (*i*)
- manter o desempenho adequado para os serviços oferecidos pela arquitetura baseada em componentes, apesar da degradação de desempenho necessária para a execução do protocolo de replicação e do sistema de comunicação de grupo (*ii*).

O grande desafio no primeiro item (*i*) é considerar o acesso a sistemas de banco de dados usado pela arquitetura baseada em componentes pré–existentes. Em outras palavras, a alta disponibilidade precisa garantir *sobrevivência* enquanto que apenas *segurança* é provida (pelo serviço transacional). No contexto deste trabalho, *sobrevivência* será assegurada através das abstrações de grupo.

A idéia de integrar conceitos de bancos de dados (transações e persistência) e de comunicação de grupo já foi lançada anteriormente [SCH96]. Trabalhos recentes [HOL99, HOL99a, KEM2001, LIT2000, PED98, STA98, WIE2000] comprovaram que esses dois paradigmas podem ser integrados de forma promissora. Entretanto, esses trabalhos não consideram arquiteturas baseadas em componentes. Adicionalmente, soluções apresentadas por esses trabalhos não objetivam sistemas usados pela indústria de *software*.

A integração dos dois paradigmas de forma adequada, leva a uma solução promissora para o segundo item (*ii*): a degradação de desempenho associada com a comunicação de grupo pode ser escondida no custo de execução das transações do banco de dados, e a complexidade de serviços não–funcionais do protocolo de replicação e do serviço de grupos pode ser escondida na arquitetura baseada em componentes.

1.2 Motivação

Na literatura são encontrados diversos serviços que oferecem alta disponibilidade a aplicações distribuídas, executadas sobre arquiteturas baseadas em componentes. Entretanto, há pouca referência descrevendo como esses serviços são implementados ou porque esses serviços são desenvolvidos pela indústria de *software* [BEA2001, SUN2000, SIL2001], ou porque são trabalhos acadêmicos [COS2003, SOU2001]. Os sistemas práticos não representam um avanço do estado da arte, pois são soluções proprietárias. Tipicamente, os trabalhos acadêmicos são mais conceituais do que práticos ou consideram um ambiente restrito, por exemplo, não tratam replicação de dados persistentes ou não descrevem um serviço transacional. Dessa forma, este presente trabalho apresenta um estudo de como a replicação pode ser oferecida por serviços não–funcionais para aplicações distribuídas com requisitos de alta disponibilidade, inclusive considerando dados persistentes, concluindo com alguns experimentos com esses serviços.

Uma implementação aberta [OBJ2001] da popular tecnologia baseada em componentes Enterprise JavaBeans (EJB) [KAS2000, SUN2001] foi usada para possibilitar a experimentação prática. O principal motivo de usar essa implementação aberta foi o contato com o grupo de pesquisa *Rainbow* liderado pelo Prof. Michel Riveill da Universidade de Nice – Sophia Antipolis (França), que vinha desenvolvendo pesquisa nesta área [BEL95, BRU2001, DAN2000, , ISS99].

Os protocolos de replicação já vinham sendo explorados anteriormente no contexto deste trabalho [PAS98, PAS98a, PAS98b, PAS98c, PAS99, PAS99a, PAS99b, PAS2000], através das abstrações de comunicação de grupos, sob orientação da Profa. Taisy S. Weber aqui neste PPGC. Contudo, ainda havia a carência de uma arquitetura adequada, onde o estudo de alta disponibilidade pudesse ser mapeado com eficiência e onde pudesse ser realizada uma experimentação prática. O casamento entre as áreas de computação baseada em componentes (estudada pelo Prof. Riveill) e de replicação para alta disponibilidade (estudada pela Profa. Taisy) possibilitou apresentar uma solução realmente atrativa [PAS2001, PAS2001a, PAS2001b, PAS2002], como será visto no andamento deste texto. Para facilitar o desenvolvimento da parte prática deste trabalho, foi realizado um estágio (doutorado sanduíche), com duração de um ano (2000–2001) no ESSI da *Université de Nice – Sophia Antipolis* (França). O estágio possibilitou maior interação com o Prof. Riveill e seu grupo de pesquisa, e com os desenvolvedores do JOnAS durante as reuniões do *Projeto Arcad* [ARC2000]. O JOnAS é um dos sistemas que foram usados na experimentação prática.

1.3 Metodologia usada neste trabalho

Este trabalho começou propondo um modelo de arquitetura multicamadas para um servidor EJB. A arquitetura multicamadas possibilita a execução de aplicações EJB altamente disponíveis. Essa arquitetura executa sobre uma camada base que implementa o serviço de comunicação de grupo, usado para facilitar a implementação do protocolo de replicação. Uma arquitetura multicamadas neste estilo foi anteriormente proposta por Amir *at al*. [AMI94], mas não considerou o escopo de componentes nem persistência de dados.

A partir dessa arquitetura, foi conduzido um experimento prático através da construção de um protótipo para uma especificação prática. Partindo–se de uma implementação aberta da tecnologia EJB foi implementada uma extensão com serviços não–funcionais

para possibilitar alta disponibilidade de componentes. Os serviços não–funcionais incluem o serviço de replicação, de grupos, de detecção de falhas e de chaveamento de servidor. Esses serviços são inseridos automaticamente com transparência para o desenvolvedor de aplicações EJB e sem alterar drasticamente a estrutura dessa arquitetura baseada em componentes.

Para sincronizar as réplicas do serviço de replicação, foi implementado um protocolo de replicação baseado em uma técnica híbrida, que mescla os protocolos clássicos de réplicas ativas [SCH93] e de cópia primária [BUD93]. Um sistema de comunicação de grupo implementa o serviço de grupos como uma camada base (i.e., bloco básico de construção), e também é materializado por uma implementação aberta.

Na prática, os serviços HA não–funcionais são implementados como um conjunto de classes que integra duas bibliotecas da linguagem de programação Java (a biblioteca EJB e a biblioteca de comunicação de grupo), buscando obedecer às diretivas da especificação EJB com impacto mínimo para os programas de aplicação. O impacto mínimo é almejado em termos de invisibilidade para o desenvolvedor de aplicação e em termos de desempenho esperado.

Para observar o impacto da adição dos serviços HA à tecnologia EJB, foram realizados experimentos práticos com o protótipo. Considerando que o desenvolvimento de um sistema computacional tenha três fases (projeto, protótipo e operação) [IYE96], a prototipação permite fazer uma validação experimental desse sistema. A validação experimental possibilita que um experimento seja realizado dentro de um ambiente controlado, enquanto que, se os experimentos forem conduzidos na fase de operação, e se o sistema não operar conforme o esperado, pode não haver mais chances de reparo.

1.4 Resultados e contribuições

Para verificar a degradação de desempenho que os serviços HA adicionam ao serviço EJB, foram realizados experimentos com o protótipo variando a quantidade de réplicas do grupo de replicação e as propriedades das aplicações. Os experimentos possibilitaram demonstrar que a degradação de desempenho gerada pelo protocolo de replicação e pelo serviço de grupos é realmente pequena e, portanto, viável em um sistema real. A degradação de desempenho não é tão significativa como inicialmente esperado.

A principal contribuição deste trabalho é integrar, de forma experimental e prática em uma arquitetura de componentes (a EJB), as abstrações de comunicação de grupo que geralmente são aplicadas a sistemas distribuídos com as abstrações de sistemas de bancos de dados, que considerem propriedades transacionais. Embora a abstração de grupos seja amplamente divulgada na teoria de sistemas distribuídos, na prática parece ser muito pouco usada na implementação de protocolos de replicação. A indústria de *software* prefere implementar replicação em sistemas de banco de dados com tradicionais mecanismos baseados em transações e protocolos centralizados.

O suporte a abstrações de grupo e à alta disponibilidade não é descrito na especificação EJB. O uso (ou não) de réplicas é uma escolha de implementação dos fornecedores de sistemas EJB ou do desenvolvedor da aplicação. Entretanto, apesar da grande popularidade da especificação EJB, apenas uma pequena parcela dos fornecedores possuem produtos EJB que implementam serviços com alta disponibilidade. Abstrações de comunicação de grupo aparentemente não são usadas nessas implementações. A exceção dessas implementações é a solução usada no servidor JBoss 3.0 [BUR2002]

lançada recentemente, mas que ainda não suporta replicação consistente em sistemas de banco de dados.

Tolerância a falhas é uma propriedade garantida na especificação EJB através da implementação de persistência e do uso de transações. As transações garantem que estados inseguros não ocorram no banco de dados apesar de falhas. A persistência garante que a informação armazenada não será perdida, por exemplo, em caso de colapso de servidor. Entretanto, a tolerância a falhas descrita na especificação EJB assegura *segurança* e não *sobrevivência*, que é requerida por aplicações altamente disponíveis.

O que se pode comprovar, com o desenvolvimento deste trabalho, é que as abstrações de projeto como aquelas providas pela modelagem baseada em componentes nem sempre são benéficas durante o desenvolvimento de um sistema: elas podem representar sérios problemas. Por exemplo, uma abstração que parece conveniente para uma sub–camada da arquitetura multicamadas pode não ser ideal para uma outra camada dessa arquitetura. Como conseqüência, pode ser necessário inserir *ganchos* nas sub–camadas para possibilitar a adição de serviços nas camadas superiores. Como resultado final, a programação não é trivial e pode dificultar a legibilidade e a eficiência do código anteriormente programado. Sendo assim, as grandes vantagens da programação baseada em componentes como proporcionar a construção de sistemas mais robustos e com maior facilidade de reutilização, de manutenção e de evolução, podem ser comprometidas.

1.5 Organização do texto

Este texto é organizado em 6 capítulos. O capítulo 2 principalmente introduz as abstrações de grupo, e conceitua e exemplifica arquiteturas baseadas em componentes. A intenção não é fazer uma descrição completa das arquiteturas nem das abstrações de comunicação de grupo, mas fornecer ao leitor informação suficiente para compreender o restante deste texto. O capítulo 3 enfoca protocolos de replicação com e sem comunicação de grupo, além do mapeamento dessas abstrações para sistemas de banco de dados. O capítulo 4 descreve a arquitetura multicamadas e os algoritmos implementados pelos serviços HA, de replicação e de chaveamento de servidor, para inserir alta disponibilidade. O capítulo 5 descreve os sistemas de suporte usados para implementar os serviços HA, a implementação e a avaliação de um protótipo com os serviços HA, e finaliza com algumas soluções da indústria de *software* para implementar alta disponibilidade em servidores EJB. O capítulo final aponta as principais conclusões e direciona trabalhos futuros.

2 Sistemas distribuídos

Desde que o objetivo deste trabalho visa alta disponibilidade em sistemas distribuídos, este capítulo apresenta conceitos relacionados a esses sistemas. O capítulo descreve um modelo de sistema distribuído que é usado no contexto deste trabalho.

Inicialmente, o capítulo define as estruturas físicas e lógicas de um sistema distribuído, e identifica os seus principais componentes. A estrutura interna desses componentes não é relevante.

Como a alta disponibilidade pode ser alcançada através da existência de um serviço de replicação *on–line*, e desde que réplicas podem estar localizadas em diferentes nodos, a comunicação entre nodos em sistemas distribuídos também é abordada.

A seguir, o capítulo descreve os modelos de desenvolvimento de sistemas de orientação a processo e a objetos e exemplifica o modelo de objetos através de arquiteturas baseadas em componentes. A implementação de serviços adicionais não–funcionais, como o serviço de replicação, para essas arquiteturas pode ser facilitada através das abstrações de *comunicação de grupo* e do mecanismo de *reflexão computacional*.

O desfecho do capítulo descreve como réplicas podem ser implementadas para garantir alta disponibilidade em sistemas distribuídos.

2.1 Modelo do sistema

O modelo do sistema distribuído pode ser dividido em modelo físico e lógico [JAL94]. O modelo lógico define serviços onde a alta disponibilidade é desejada. Esses serviços executam sobre uma rede de computadores, que é descrita pelo modelo físico. Os componentes do modelo físico podem falhar, e comprometer os serviços oferecidos pelo modelo lógico. O objetivo da alta disponibilidade é garantir que os serviços do modelo lógico estão operacionais mesmo que ocorra falha em algum componente do modelo físico.

2.1.1 Modelo físico

O modelo físico de um sistema distribuído é composto por nodos (i.e., computadores ou estações de trabalho) geograficamente separados e conectados por uma rede de comunicação. Nodos são autônomos, comunicam–se por troca de mensagens através da rede de comunicação e podem falhar por colapso.

Nodos em um sistema distribuído não compartilham memória nem relógio global. Cada nodo armazena seus próprios dados e possui seu próprio sistema operacional, que é executado por um processador.

Em uma rede ponto–a–ponto, a rede de comunicação é formada por um conjunto de *links*. Cada *link* conecta dois nodos através de uma interface de rede. A troca de mensagens em uma rede ponto–a–ponto requer protocolos de comunicação. Os protocolos são um conjunto de regras, que podem ser implementados em diversas camadas. Os protocolos tratam as diferenças entre as arquiteturas (i.e., diferenças de *hardware*) e sistema operacionais (i.e., diferenças de *software*) dos nodos do sistema distribuído, além de erros de transmissão de mensagens. Protocolos de comunicação,

como o TCP/IP [SIL2000, TAN92] e o OSI [SIL2000, TAN92], foram propostos para garantir entrega de mensagens confiável (ver item 2.2.5) entre os nodos.

Além da rede ponto–a–ponto, outra topologia de rede muito popular usa um barramento onde os nodos são conectados. A rede Ethernet implementa essa topologia. Na rede Ethernet, um nodo que quer transmitir uma mensagem, primeiro escuta o canal. O canal implementa no modelo lógico o componente que corresponde ao barramento no modelo físico. Se não há alguma mensagem sendo transmitida, o nodo começa a sua transmissão. Se múltiplos nodos começam a transmitir simultaneamente, as mensagens colidem. Então a retransmissão das mensagens é abortada e os nodos reiniciam a transmissão depois de aguardar um período randômico de tempo.

2.1.2 Modelo lógico

Sobre o modelo físico anteriormente descrito existe um sistema distribuído composto por processos concorrentes que cooperam para executar determinado serviço. Processos podem comportar múltiplas *threads* que compartilham o acesso a recursos que o nodo oferece. Múltiplos processos podem compartilhar um mesmo processador em um mesmo nodo ou podem executar em nodos distintos. Dado que processos podem executar em diferentes nodos, não se pode fazer nenhuma afirmação sobre a sua velocidade relativa.

No modelo lógico, o sistema distribuído é formado por um conjunto de processos e canais de comunicação entre esses processos. Os canais representam a conexão lógica entre os processos, tem *buffer* infinito e são livres de erros. Grande parte dos protocolos de comunicação provê esse tipo de abstração.

No modelo lógico, a rede que comporta processos é suposta ser totalmente conectada, i.e., a topologia de rede não é considerada no modelo lógico. Isso significa que, se um nodo está conectado a outros nodos, então uma mensagem pode ser enviada daquele nodo para qualquer outro nodo na rede.

Freqüentemente, para efeitos práticos, afirmações são feitas sobre os limites de tempo em um sistema distribuído. O *sistema síncrono* define limites de tempo para a trans–missão de mensagens e para a resposta de processos.

Se nenhuma restrição sobre o limite de tempo é feita, o sistema é chamado de *assíncrono* [GAR99, JAL94]. Em sistemas síncronos, a falha de um componente é detectada pela ausência de resposta dentro de um determinado limite de tempo. Em um sistema assíncrono não há como prever quanto tempo uma mensagem enviada por um nodo leva para chegar em outro nodo. Assim sendo, nodos de um sistema distribuído assíncrono não podem diferenciar um nodo falho de um nodo atrasado devido à sobrecarga da execução de serviços. Este trabalho considera nodos em um sistema assíncrono por ser um modelo mais genérico.

Assim como os protocolos de comunicação do modelo físico, o modelo lógico também é estratificado em diferentes camadas. Uma estratificação bem popular para nodos é mostrada na figura 2.1, e comporta três camadas executadas sobre o *hardware*: aplicação, *middleware* e sistema operacional.

A aplicação é a interface do sistema computacional com o usuário. É a aplicação que implementa o serviço funcional (ou a lógica do negócio), como um sistema bancário, um sistema de controle de estoque, ou um sistema para compra e reserva de passagens aéreas.

FIGURA 2.1 – Camadas de um sistema computacional

Um *middleware* é a camada de *software* entre o sistema operacional e a aplicação. O *middleware* provê serviços comuns à camada de aplicações, i.e., os serviços não–funcionais como persistência, gerenciamento de transações e segurança.

O sistema operacional executa serviços básicos do sistema computacional, como a localização de processos, gerenciamento de memória, i.e., o sistema operacional provê a infra-estrutura para as camadas superiores.

2.2 Comunicação em sistemas distribuídos

Para que processos em um sistema distribuído cooperem para realizar um serviço, eles precisam trocar informações. A comunicação permite a troca de informações entre processos. Freqüentemente, comunicação entre processos requer sincronização para controlar a diferença de velocidade entre os processos. A sincronização é vista como uma forma de ordenar eventos em um sistema distribuído, já que não há relógio global.

Em um sistema distribuído, a troca de mensagens é usada para sincronização e comunicação. Comunicação é obtida quando um processo emissor envia uma mensagem com uma informação para outro processo receptor. A sincronização é obtida desde que a troca de mensagens implica que um processo recebe uma mensagem depois que ela foi enviada por outro processo.

Uma mensagem é enviada pelo comando *enviar* (*send*) e recebida pelo comando *receber* (*receive*). Os comandos *enviar* e *receber* podem identificar, respectivamente, o processo receptor e o processo emissor. Um caso particular dessa interação é o modelo de *comunicação cliente–servidor*, onde o comando receber não especifica o emissor. O emissor é extraído da mensagem.

2.2.1 Comunicação cliente–servidor

Na comunicação cliente–servidor [JAL94, SIL2000, TAN92], mostrada na figura 2.2, um *processo cliente* envia uma mensagem com uma *requisição* de serviço para um *processo servidor* que realiza o serviço e envia uma mensagem com a *resposta* para o cliente.

Nesse esquema, um cliente envia uma mensagem com uma requisição de serviço para um servidor, e suspende o seu processamento. O servidor recebe a mensagem, executa o serviço e envia a mensagem de resposta ou a confirmação indicando que o serviço foi realizado ou um código de erro. Quando o cliente recebe a resposta, continua seu processamento. O paralelismo entre cliente e servidor não acontece, pois os dois processos não executam ao mesmo tempo: um espera pela mensagem do outro.

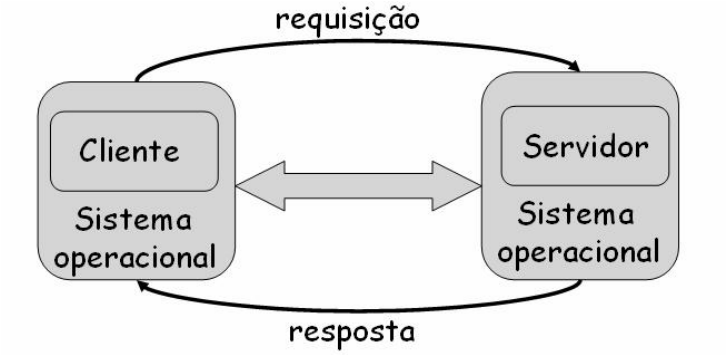

FIGURA 2.2 – Comunicação cliente–servidor

A principal vantagem da comunicação cliente–servidor é a economia, já que não é preciso grande capacidade de processamento ou muito espaço de armazenamento de dados em todos os nodos do sistema. Se o processamento *multi–thread* está disponível, o servidor pode atender a vários clientes simultaneamente, permitindo alto desempenho ao sistema.

2.2.2 Comunicação síncrona e assíncrona

No modelo cliente–servidor, as mensagens que são enviadas de um processo a outro podem ser guardadas em um *buffer*, formando uma fila de mensagens. A cada comando *receber*, a primeira mensagem do *buffer* é entregue ao receptor. Se não há mensagens no *buffer*, o processo bloqueia até que uma mensagem chegue.

O *buffer* é uma estrutura entre o emissor é o receptor. Se o tamanho do *buffer* é infinito, a troca de mensagens é dita *assíncrona* [JAL94]. Na troca de mensagens assíncrona, o emissor nunca bloqueia. Contudo, o receptor bloqueia se o *buffer* está vazio. Se o *buffer* possui tamanho finito, a troca de mensagens é dita *buferizada*. Em contraste, se não há *buffer* entre o emissor e o receptor, a troca de mensagens é dita *síncrona*.

Deve ser observado que as definições de comunicação síncrona e assíncrona não tem relação com o tipo de sistema classificado quanto aos limites de tempo.

2.2.3 Localização de processos

Para dois processos se comunicarem, eles precisam ser localizados. O *serviço de nomes* é quem localiza um processo, a partir de parâmetros como o nome. A operação que localiza um processo é chamada *procedimento de resolução*. Cada processo tem um endereço lógico no sistema operacional e uma localização física na rede.

O procedimento de resolução envolve a resolução de nomes e a resolução de endereços. A resolução de nomes converte nomes para endereços lógicos, que é uma informação estrutural para o sistema operacional manter e localizar processos. Um nome identifica um processo e indica o seu endereço. Usualmente nomes são únicos, mas o mesmo nome pode corresponder a múltiplos endereços, por exemplo, para a replicação. A resolução de endereços converte endereços lógicos para rotas de rede que indicam a localização física de um processo.

Depois que o processo é localizado, através da resolução de nomes e da resolução de endereços, a comunicação pode ser feita. Como há ausência de memória compartilhada, processos em um sistema distribuído comunicam–se através da troca de mensagens

2.2.4 Remote Procedure Call

A RPC (*remote procedure call*) [BIR84] é um esquema de troca de mensagens síncrona que implementa a comunicação cliente–servidor. Um processo cliente, que quer se comunicar com um servidor, constrói uma mensagem no seu espaço de endereçamento e executa uma chamada de sistema para que seu sistema operacional envie esta mensagem para o servidor, que recebe a mensagem através da rede.

A comunicação por RPC pode ser em chamadas locais (quando os processos estão no mesmo nodo) ou chamadas remotas (quando os processos estão em nodos distintos). A comunicação remota é permitida através de protocolos de comunicação implementados em diversas camadas.

O objetivo da RPC é manter a transparência da execução, fazendo com que chamadas remotas se pareçam como chamadas locais. A RPC faz isso usando um *stub* no cliente e um *skeleton* no servidor. O *skeleton* e o *stub* são procedimentos auxiliares, gerados pelo compilador RPC.

A função do *stub* é fazer com que uma chamada remota do cliente, que é executada no nodo servidor, se pareça como uma chamada local, feita a um procedimento dentro do mesmo nodo do cliente. A função do *skeleton* é fazer com que uma requisição de serviço de cliente, que é recebido de outro nodo, se pareça como uma chamada de requisição local, feita a um procedimento dentro do mesmo nodo do servidor.

Para que a comunicação cliente–servidor seja possível, é preciso que o servidor registre–se em um *binder*. O *binder* é um programa que conhece a localização do servidor, e faz a conexão do cliente e do servidor com o *serviço de nomes*. O *stub* do cliente realiza uma chamada ao *binder* solicitando um servidor e o *binder* fornece o endereço do servidor para o cliente.

A RPC possibilita a comunicação entre nodos com diferentes sistemas operacionais ou com diferentes configurações de *hardware*, pois a mensagem transferida é escrita em uma estrutura de dados padronizada. Outra vantagem é que um nodo menos favorecido pode requisitar serviços para outro nodo mais robusto. Este esquema é particularmente útil em sistemas com servidores de arquivos, onde nodos com discos de alta capacidade de armazenamento disponibilizam serviços de arquivos para seus clientes.

2.2.5 Comunicação e difusão confiáveis

A *comunicação confiável* assegura as propriedades de confiabilidade e ordem entre dois nodos, i.e., na comunicação ponto–a–ponto. Segundo Jalote [JAL94], uma comunicação entre dos nodos é dita *confiável* quando uma mensagem enviada por um nodo chega em outro nodo sem ser corrompida e a ordem das mensagens entre os dois nodos é preservada. Note que o conceito de comunicação confiável é diferente do conceito de *difusão confiável*.

Embora a comunicação ponto–a–ponto (ou *unicast*) seja suficiente para muitas aplicações, algumas aplicações requerem que um nodo envie para vários nodos ao mesmo tempo, i.e., ponto–a–multiponto. Um exemplo são os protocolos de replicação, que serão enfocados no próximo capítulo.

Há duas formas para a comunicação ponto–a–multiponto: difusão (ou *broadcast*) e difusão em grupo (ou *multicast*). Difusão é um paradigma de comunicação onde o

emissor envia uma mensagem para todos os nodos de um sistema distribuído. Na difusão em grupo, o emissor envia uma mensagem para um grupo de nodos de um sistema distribuído.

Pode ser observado na literatura [BIR87, CHO2001, GUE97, JAL94, PED98] que há uma pequena discrepância com os termos usados nas definições para difusão e difusão em grupos. Dado que autores como Birman *et al.* [BIR87] e Jalote [JAL94] descrevem propriedades de interesse para primitivas de difusão (*broadcast*), e não para difusão em grupo (*multicast*), inicialmente serão revisados os termos por eles usados.

Segundo Jalote [JAL94], quando uma mensagem é enviada por difusão em um sistema distribuído, há três propriedades de interesse:

- **confiabilidade**: uma mensagem de difusão deve ser *recebida* por todos os nodos operacionais em um sistema distribuído;
- **ordem consistente**: diferentes mensagens difundidas por diferentes nodos são *entregues* na mesma ordem;
- **preservação de causalidade**: a ordem na qual as mensagens são *entregues* nos nodos deve ser consistente com a causalidade entre os eventos que enviaram essas mensagens.

Observe que os conceitos de **entregar** e **receber** são distintos. Basicamente, um nodo primeiro *recebe* uma mensagem e, então, executa alguma coordenação com outros nodos, para garantir as propriedades anteriormente descritas, e então *entrega* a mensagem, i.e., executa a operação relacionada à mensagem. Observe também que o serviço de gerenciamento de membros (ou *membership*) não é abordado porque o enfoque é sobre nodos em um sistema e não sobre nodos membros de um grupo.

Cada uma dessas propriedades define um tipo diferente de primitiva de difusão e tem suas aplicações específicas. A difusão confiável (*reliable broadcast*) apesar de falhas [HAD90, CHA84] requer apenas confiabilidade, i.e., se uma mensagem é enviada com difusão confiável, ela é recebida por todos os nodos operacionais em um sistema distribuído, mesmo que ocorra falha no sistema.

A difusão atômica (*atomic broadcast*) requer as propriedades de ordem e de confiabilidade. A difusão causal (*causal broadcast*) garante que a ordem na qual as mensagens são recebidas é consistente com a ordem causal dessas mensagens. Como essas primitivas são implementadas em sistemas distribuídos não importa no escopo deste trabalho, mas essa informação pode ser encontrada em Jalote [JAL94].

Originalmente, Birman *et al.* [BIR87] definiu três primitivas de difusão: difusão atômica (*atomic broadcast* ou *abcast*), difusão causal (*causal broadcast* ou *cbcast*) e difusão em grupo (como *group broadcast* ou *gbcast*). Segundo Birman, todas essas primitivas são atômicas, i.e., uma mensagem de difusão é recebida mais cedo ou mais tarde por todos os nodos operacionais, mesmo que ocorra alguma falha no sistema.

Observe que a propriedade que Birman definiu como *atomicidade*, Jalote definiu como *confiabilidade*. Sendo assim, pode-se concluir que as primitivas de *difusão atômica* e de *difusão confiável* definidas em Jalote e em Birman *et al.* são equivalentes.

A difusão em grupo usa o conceito de grupos de processos. Um grupo ou um *grupo de processos tolerante a falhas* [BIR87] é uma abstração que pode ser definida como um conjunto de processos que cooperam para realizar um serviço distribuído, e interagem através de um protocolo de comunicação. O grupo possui um nome e um endereço para

determiná–lo. Todos os membros de um grupo sabem quais são os demais membros do grupo. Essa informação é mantida localmente em cada membro de uma estrutura chamada *visão do grupo* [BIR87].

Cada vez que um processo entra no grupo e cada vez que um membro abandona o grupo, uma nova visão é instalada para todos os membros desse grupo. Assim, a seqüência de visões reporta a composição do grupo em determinados instantes. Se uma mensagem é enviada por *gbcast* [BIR87], todos os membros operacionais desse grupo recebem essa mensagem. Os membros não operacionais serão excluídos na próxima visão.

Guerraoui e Schiper [GUE97] definem duas primitivas de comunicação de grupos em sistemas assíncronos: *view synchronous multicast* e *total–order multicast*. A definição da primitiva *view synchronous multicast* ou VSCAST inclui o conceito de *grupo dinâmico* e as sucessivas visões implementadas pelos membros desse grupo. Um grupo dinâmico sofre alterações na quantidade de membros durante a sua vida útil.

Cada visão de um grupo dinâmico contém os membros que não estão em suspeita de falha em um determinado instante de tempo. Cada vez que um membro de um grupo na visão corrente está em suspeita de falha, ou cada vez que um novo processo ingressar no grupo, uma nova visão é instalada, para refletir a nova configuração do grupo.

Quando uma mensagem é enviada com a primitiva VSCAST para algum membro do grupo na visão corrente, a seguinte propriedade é garantida: se um membro na visão corrente entrega a mensagem antes de instalar a nova visão, então nenhum membro instala a nova visão antes de receber essa mensagem. Esse mecanismo garante que o estado de cada membro de um grupo será o mesmo para uma determinada visão.

A primitiva *total–order multicast* ou TOCAST [GUE97] garante as seguintes propriedades:

- **ordem**: se dois nodos em um grupo entregam diferentes mensagens enviadas por diferentes nodos, eles entregam essas mensagens na mesma ordem;
- **atomicidade**: se uma réplica em um grupo entrega uma mensagem, todos os nodos *corretos* (i.e., não falhos) no grupo entregam a mensagem;
- **término**: se um nodo em um grupo está correto e não é *suspeito de falhas* e *envia* uma mensagem, todos os demais nodos corretos desse grupo *entregam*, mais cedo ou mais tarde, essa mesma mensagem.

A propriedade de término é uma condição de *sobrevivência* e garante o progresso do sistema distribuído apesar de falhas. O grupo nunca bloqueia, mesmo que alguma falha ocorra.

Originalmente, a primitiva TOCAST não considera visões seqüenciais de um grupo, i.e., ela não considera grupos dinâmicos porque admite que um nodo falho pode ser recuperado mais adiante. Sendo assim, o nodo deverá receber as mensagens que foram enviadas ao grupo durante o período no qual o nodo esteve falho. Esse procedimento é chamado de **transferência de estado**. Entretanto, uma implementação diferente da primitiva TOCAST pode considerar grupos dinâmicos e sucessivas visões.

Observe que os nodos são suspeitos de falha. Na teoria, comunicação de grupo não pode ser aplicada a sistemas assíncronos, mesmo com a restrição de se permitir apenas um único nodo em falha por colapso (que é o modelo mais restrito de falhas [CRI86]), devido à impossibilidade de se obter consenso em sistemas distribuídos [FIS85]. Na

prática, esse problema pode ser resolvido com o uso de um detector de falhas que indique nodos *suspeitos de falhas* [CHA96, GUE97].

A primitiva VSCAST está para a primitiva de difusão confiável [BIR87, JAL94] assim como a primitiva TOCAST está para a primitiva de difusão atômica [BIR87, JAL94]. Entretanto, a primitiva VSCAST é definida para o contexto de grupos (i.e., um grupo de nodos em um sistema), enquanto que a primitiva de difusão confiável abrange todos os nodos em um sistema. O contexto de grupos implica em estabelecer sucessivas visões para o grupo se algum membro não está funcionando corretamente ou se algum membro foi adicionado ao grupo.

A primitiva TOCAST divide a propriedade de confiabilidade [JAL94] anteriormente definida em duas propriedades, atomicidade e término, mas difere–se da difusão atômica porque, como a primitiva VSCAST, está definida para o contexto de grupos.

As diferenças de nomenclaturas usadas na literatura não param por aqui. Pedone *et al.* [PED98, PED99] define a primitiva de difusão atômica com as mesmas propriedades que Guerraoui e Schiper [GUE97] definiram para a primitiva TOCAST.

Chockler *et al*. [CHO2001] recentemente organizou um estudo sobre especificações usadas em comunicação de grupo, inclusive abordando primitivas de comunicação de grupo, tentando estabelecer um mapeamento entre diversas nomenclaturas usadas na literatura. Entretanto, divergências ainda persistem. No contexto deste trabalho optou–se por usar a nomenclatura descrita por Guerraoui e Schiper [GUE97] por se tratar de uma publicação relativamente recente e por abordar o contexto de grupos em sistemas assíncronos.

2.2.6 Comunicação de grupo

A abstração de grupos permite ao desenvolvedor usar apenas uma primitiva para estabelecer comunicação entre um cliente e o grupo de servidores (fig. 2.3).

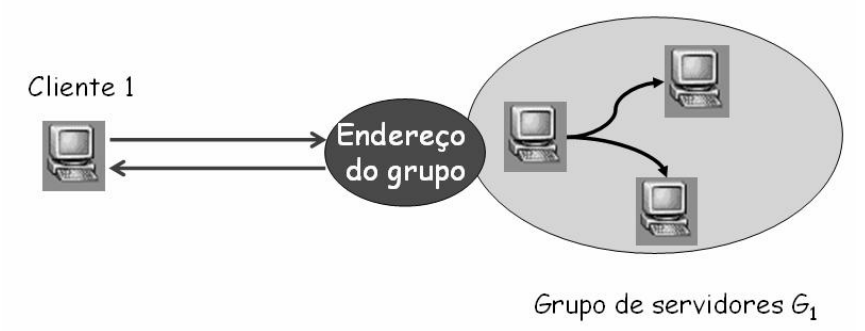

FIGURA 2.3 – Abstração de grupos

Todos os processos que querem se tornar membros são registrados no grupo através do **comando ingressar** (*join*). Um membro abandona o grupo por **suspeita de falha** ou voluntariamente através do **comando abandonar** (*leave*). Processos que participam de um grupo são chamados de **membros** e são identificados unicamente no grupo. O serviço que mantém a visão de todos os membros consistente é chamado de composição do grupo (ou *membership*).

Vários grupos podem existir simultaneamente em um sistema distribuído. A existência de múltiplos membros em um grupo é invisível para o cliente. Um cliente que deseja se comunicar com um grupo de servidores, ao invés de enviar uma mensagem individual

para cada membro, envia apenas uma mensagem para o endereço do grupo. A abstração de grupos evita que um processo externo tenha que conhecer individualmente cada um dos membros de um grupo.

Um sistema de comunicação de grupo é um *framework* que possibilita mecanismos para implementar o serviço de composição do grupo, detecção de falhas e primitivas de comunicação para um conjunto de processos. Em muitos sistemas de comunicação de grupo (como o JavaGroups [BAN99], Ensemble [VAY98] e Horus [REN96]), implementar as primitivas de comunicação consiste na seleção de determinadas propriedades, providas por uma estrutura em camadas compostas por micro–protocolos.

Classificação de grupos

Grupos podem ser classificados segundo suas características. A principal classificação de grupos é quanto à facilidade do grupo refletir ou não as alterações do sistema distribuído [GUE97]. Em grupos estáticos, não ocorre qualquer alteração na quantidade de membros do grupo durante a vida útil do sistema. Não é necessário um serviço para controlar a entrada e saída de membros. Um membro falho continua a fazer parte do grupo.

Grupos dinâmicos sofrem alterações durante a vida útil do sistema, refletindo entradas e saídas de membros no grupo. Em grupos dinâmicos, qualquer alteração no número de membros de um grupo é indicada na visão do grupo.

Os grupos também podem ser classificados quando à forma na qual o *multicast* é implementado [TAN92]. Se todos os membros estão aptos a fazer o *multicast*, sem necessitar de um membro coordenador, o grupo é não–hierárquico ou simétrico.

Quando o grupo precisa de um membro coordenador para realizar o seqüenciamento de mensagens, o grupo é assimétrico ou hierárquico. Neste caso, apenas o coordenador precisa estar apto a fazer o *multicast*. Os demais membros que desejam realizar um *multicast* enviam uma mensagem ponto–a–ponto para o coordenador.

Os grupos também podem ser classificados em fechados e abertos. Um grupo fechado somente recebe mensagens de membros. Um grupo aberto recebe mensagens de qualquer processo.

2.3 Modelo de orientação a processos

O modelo de orientação a processos contempla os conceitos apresentados anteriormente. Nesse modelo, os serviços computacionais prestados são deficitários porque as aplicações são construídas sobre serviços de comunicação limitados, como as conexões TCP. Serviços não–funcionais como replicação, migração, tolerância a falhas e persistência são implementações específicas, de acordo com a aplicação. A maior desvantagem de implementar serviços específicos é o esforço de construir serviços comuns para cada aplicação, além de dificultar ou impossibilitar a interoperabilidade entre aplicações. Soluções recentes usam uma estrutura flexível, baseada em abstrações como a reflexão computacional, para implementar um conjunto de serviços não–funcionais comuns.

2.4 Modelo de orientação a objetos

O modelo de orientação a objetos modela processos em um sistema computacional e outros conceitos definidos anteriormente como objetos e permite maior flexibilidade: objetos implementados em diferentes linguagens de programação estabelecem comunicação através de troca de mensagens.

Contornar a diferença de implementação entre objetos é muito importante, pois permite conectar diferentes tecnologias de *software* e de *hardware*, possibilitando maior interoperabilidade, mas mantém a necessidade de implementação de serviços requeridos em sistemas orientados a processos, como concorrência, tolerância a falhas e alta disponibilidade.

2.4.1 Conceito de objeto

Um objeto [NIE87] em um sistema computacional é qualquer entidade como arquivos, diretórios, processos, serviços, mensagens, réplicas, grupos, etc. Um objeto possui um estado interno encapsulado (ou memória local privativa) que é guardado em variáveis de instância e um **comportamento** bem definido, descrito por um conjunto de métodos. O estado interno pode ser alterado ou consultado através da execução dos métodos ativados por troca de mensagem.

Além do **encapsulamento** [NIE87], objetos possuem como propriedades polimorfismo e herança. A herança permite que diferentes objetos compartilhem código e oferece como benefício a redução do código duplicado. O polimorfismo permite que uma interface possua implementações diferentes do mesmo método, cada uma, por exemplo, com um tempo de execução diferente. Quando comparado a sistemas orientados a processos, sistemas orientados a objetos apresentam facilidades para reusabilidade, manutenção e depuração de código.

Um objeto interage com outros objetos usando interfaces, que escondem a implementação interna do comportamento. Uma **interface** é uma descrição do conjunto de métodos do objeto, i.e., o conjunto das assinaturas dos métodos que especifica os serviços oferecidos por um objeto ou processo.

Objetos são criados e destruídos através de determinados métodos. O resultado de uma criação de objeto aparece para outro objeto como uma referência que descreve o novo objeto.

Objetos são agrupados em classes. Objetos de uma mesma classe têm as mesmas variáveis de instância e métodos, e respondem às mesmas mensagens. Objetos podem ser organizados em subclasses, formando uma hierarquia de classes. Objetos em uma subclasse **herdam** todas as variáveis de instâncias e métodos de uma classe. Herança múltipla também é permitida. A hierarquia de classes é vista como um mecanismo que permite reusabilidade de código, economizando e simplificando a programação. A herança de interface permite que um objeto suporte múltiplas interfaces.

2.4.2 Remote Method Invocation

O RMI (*remote method invocation*) [SUN87] é uma implementação da RPC, usada no modelo de orientação a processos, para o contexto de objetos. O objetivo do RMI, assim como na RPC, é esconder detalhes de comunicação, fazendo com que a invocação de método remoto apareça como uma invocação local.

O RMI é baseado no princípio de que a interface e a implementação de um serviço são conceitos separados e, portanto, podem executar em máquinas virtuais distintas. Esse princípio é adequado para a implementação da comunicação cliente–servidor, onde o cliente preocupa–se em conhecer a interface e o servidor, em oferecer um ou mais serviços.

No RMI, duas classes implementam a mesma interface (fig. 2.4): a primeira classe é a implementação do serviço, e executa no servidor; a segunda classe atua como um *proxy* para o serviço remoto e executa no cliente. Quando uma aplicação cliente faz uma chamada de método usando o *proxy* do objeto, o RMI envia a requisição para a máquina virtual remota, que envia a requisição para a implementação do serviço. Se houver alguma resposta, ela é enviada do serviço remoto para o *proxy* e, então, para a aplicação cliente. O *proxy* é um complemento do *stub* e encapsula os detalhes de comunicação para a aplicação cliente. O *stub* é o componente responsável pela preparação de objetos enviados como parâmetros ou recebidos como resultado em cada invocação. No lado do servidor, o componente *skeleton* é o responsável por esses mesmos serviços.

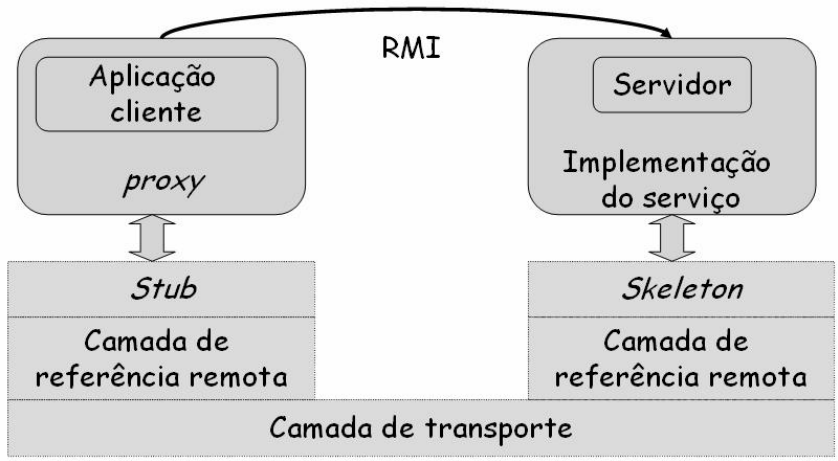

FIGURA 2.4 – Comunicação entre cliente e servidor

A implementação do RMI usa três camadas de abstração:

- **camada do *stub* e do *skeleton***: está abaixo da visão do desenvolvedor. Essa camada intercepta as chamadas de métodos para o cliente e redireciona essas chamadas para o serviço remoto;
- **camada da referência remota**: interpreta e gerencia referências de clientes para o objeto remoto;
- **camada de transporte**: é baseada em conexões TCP/IP entre nodos de uma rede e provê conectividade básica, entre outras funcionalidades.

O uso de uma arquitetura em camadas possibilita que cada camada seja modificada ou substituída sem afetar o resto do sistema.

A camada do *stub* e do *skeleton* usa o *proxy* para passar adiante as chamadas de métodos entre os objetos participantes. O *stub* executa o serviço do *proxy*, e a implementação do serviço remoto executa o serviço realmente oferecido usando informações fornecidas pelo *skeleton*.

A camada de referência remota define e suporta a semântica de invocação da conexão RMI. Essa camada implementa uma referência que representa uma ligação para a implementação do serviço remoto. A primeira implementação do RMI possibilitava apenas conexão ponto–a–ponto (*unicast*) entre clientes e o serviço remoto. Para que um cliente use um serviço remoto, o serviço precisa ser instanciado no servidor e exportado pelo sistema RMI. Se essa requisição é a primeira requisição da aplicação cliente ao serviço, ela precisa ser registrada usando o serviço RMI Registry.

Atualmente, outros tipos de semântica de conexão são possíveis. Por exemplo, com *multicast*, um único *proxy* pode enviar uma requisição para múltiplos objetos simultaneamente e aceitar a primeira resposta, melhorando o tempo de execução de uma aplicação e, possivelmente, permitindo maior disponibilidade.

A camada de transporte conecta máquinas virtuais diferentes. Todas as conexões são baseadas em *stream*[1] e usam TCP/IP, mesmo que duas máquinas virtuais estejam executando no mesmo nodo. Sobre o TCP/IP, existe um protocolo proprietário chamado Java Remote Method Protocol (JRMP). A primeira versão do JRMP requeria um *skeleton* no servidor. Na segunda versão, a classe *skeleton* foi suprimida para possibilitar maior desempenho, uma vez que seus serviços podem ser realizados dinamicamente, sem necessidade de uma classe de apoio. Sun Microsystems e IBM trabalharam juntas em uma versão do RMI, chamada RMI–IIOP. Ao invés de usar o JRMP, o RMI–IIOP usa o IIOP do CORBA para fazer a comunicação entre clientes e servidores.

Contudo, o desenvolvimento do modelo de orientação a objetos não desencadeou uma disseminação generalizada de bibliotecas de classes para o desenvolvimento de sistemas. O **encapsulamento**, a **herança** e o **polimorfismo** dos objetos proporcionam o reuso de funcionalidades. No entanto, apenas essas características não foram suficientes para a obtenção do reuso esperado. O modelo de orientação a objetos apresenta alguns obstáculos para a criação de um mercado de objetos reutilizáveis [CHA96a]:

- distribuição de objetos implica na distribuição de seu código fonte, sendo que a distribuição de seu código binário restringe sua utilização aos mesmos ambientes de desenvolvimento;
- por estar em código fonte, a reutilização entre objetos implementados em diferentes linguagens não é possível;
- quando um objeto é alterado, é preciso recompilar toda a aplicação.

A abstração de componentes não apresenta esses obstáculos e oferece um mecanismo efetivo de reutilização de *software*, através de blocos de *software* binários com interfaces bem definidas. Um componente não é necessariamente um objeto, ou um conjunto de objetos, ou um processo. Porém, tanto o modelo de objetos quanto o modelo de componentes visam reutilização.

2.5 Modelo de orientação a componentes

Um componente de *software*, ou simplesmente componente, é uma unidade de composição com interfaces especificadas por contratos e que possui dependências exclusivamente explícitas por contexto [SZY99]. Cada componente em uma aplicação tem uma interface de serviço, possui um certo tipo, requer recursos, executa uma regra e

[1] um *stream* é um fluxo de caracteres

é uma unidade de instalação [VOE2003]. Um componente de *software* é desenvolvido de forma autônoma e pode ser usado para compor diferentes aplicações, desenvolvidas por diferentes desenvolvedores.

Associar a um componente o conceito de unidade de composição significa que o propósito de um componente é ser composto com outros componentes [VOE2003]. Assim sendo, uma aplicação baseada em componentes (ou orientada a componentes) usa um conjunto de componentes que colaboram entre si, através de uma ou mais interfaces. Essas interfaces formam um **contrato** entre o componente e seu ambiente. A interface define que serviços o componente oferece [VOE2003]. Esses serviços podem ser especificados em termos de operações com parâmetros e tipos em uma **interface de serviço**. Essa interface também pode especificar a semântica das operações através de projeto–por–contrato ou por máquinas de estado. A especificação da semântica por projeto–por–contrato indica as pré e pós-condições para operações enquanto que a especificação da semântica por máquinas de estado, as seqüências de invocações e possíveis restrições de tempo. Contudo, a maioria dos sistemas baseados em componentes não especifica a semântica e apenas descreve as operações e suas assinaturas [VOE2003]. É o caso da especificação EJB.

Tipicamente, um *software* depende de um **contexto** específico, como recursos requeridos pela aplicação, por exemplo, as conexões com bancos de dados. Um contexto particularmente interessante ocorre quando um conjunto de componentes precisa estar disponível para colaborar com um outro conjunto de componentes. Para que tais composições sejam possíveis, i.e, para que componentes sejam compostos por outros componentes ou para que componentes colaborem com outros componentes, as dependências entre os componentes precisam ser explicitamente especificadas [VOE2003].

2.5.1 Tipos de componentes

Devido ao fato de um componente poder ser usado como um bloco de construção, para compor uma aplicação ou um sistema, é necessário que ele especifique os recursos que ele necessita para sua execução. Recursos comuns são conexões com bancos de dados, filas de mensagens e interfaces de serviços de outros componentes. Idealmente, os recursos devem ser especificados para que seja possível que o ambiente detecte onde uma aplicação pode executar, ou quais recursos estão faltando. A especificação EJB, por exemplo, especifica esses recursos como parte do descritor do *bean*, usando linguagem XML.

Em uma aplicação, um componente executa uma regra que pode ser uma **entidade**, um **serviço** ou um **componente de processo**. Se o componente representar uma entidade (por exemplo, uma pessoa), ele tem um estado que pode ser persistente. Se o componente representar um serviço será, tipicamente sem estado (i. e., *stateless*). Um componente que representa um componente de processo possibilita a execução de tarefas como preenchimento de formulários. Tipicamente, um componente de processo possui estado (i.e, é *stateful*) mas este estado não é persistente.

Um componente de *software* pode ser de dois tipos: **técnico** ou **lógico**. Um componente lógico é um pacote de uma funcionalidade relacionada, como um subsistema ou uma aplicação que é parte de um grande sistema. Um componente lógico admite mecanismos para controlar a complexidade de um sistema, organizar o controle de versões ou aspectos de gerenciamento de projeto. Outras características como interface de serviços,

recursos, regras e **meta-infomações** podem ser especificadas informalmente, não necessariamente através do uso de ferramentas.

Basicamente, componentes lógicos podem ser de três tipos: componente de domínio (ou *domain component*), componente de dados (ou *data component*) ou componente de usuário (ou *user component*). Componentes de domínio tratam da lógica funcional (ou *business logic*). Componentes de dados possibilitam acesso a dados e, possivelmente, validação e conversão de dados. Componentes de usuário fazem parte da aplicação do cliente e, tipicamente, oferecem propriedades e acesso para componentes de domínio e de dados. Há também o conceito de componentes de negócio (ou *business components*), uma agregação de dados, domínio e usuário que personifica um sistema completo.

Componentes técnicos são oferecidos como blocos de construção para compor as aplicações. Componentes técnicos são especificados formalmente e podem ser avaliados e entendidos por uma ferramenta chamada *container*.

Há uma relação importante entre componentes lógicos e técnicos: em muitos casos, componentes lógicos são uma agregação de componentes técnicos, i.e., componentes técnicos são usados como blocos de construção para componentes lógicos.

Em uma aplicação baseada em componentes, é possível substituir um componente por outro que ofereça uma interface compatível. Esse mecanismo possibilita facilidade manutenção, operação e desenvolvimento.

Os componentes permitem altos níveis de reutilização, que a orientação a objetos não permite, e a existência de um mercado de compra e venda de componentes, uma vez que são unidades de *software* binárias. A reutilização é um mecanismo que possibilita criar *softwares* melhores. Construir novas aplicações a partir de COTS (componentes de prateleira), que já são testados, e favorece o planejamento e a redução de prazos e de custos de desenvolvimento, permitindo a previsibilidade e possibilitando linhas de produção de *software* concorrentes.

Quando os componentes são desenvolvidos segundo o modelo de orientação a objetos, eles agregam as vantagens do alto potencial de reuso de componentes com a robustez e a facilidade de evolução e manutenção da orientação a objetos. Pode–se afirmar, então, que objetos são inerentemente centralizados e atuam dentro das aplicações, enquanto que componentes são distribuídos e entre aplicações.

2.5.2 *Container*

O *container* oferece um ambiente de execução para componentes. Essa ferramenta possibilita a separação das funcionalidades dos componentes. Cada serviço oferecido ou usado por uma aplicação é mantido em um diferente componente. Um *container* oferece propriedades não-funcionais para componentes de uma aplicação. Essas propriedades não-funcionais dependem do domínio da aplicação, como segurança (ou *security*), gerenciamento transacional, etc. O desenvolvedor não precisa implementar essas funcionalidades manualmente para cada aplicação.

Para que um *container* ofereça propriedades para um componente técnico, o componente precisa oferecer uma interface técnica com a qual o container possa tratar uniformemente todos os componentes mantidos por ele.

Opcionalmente, componentes podem oferecer meta-informações, i.e., ou informações que os descrevam. A especificação de recursos é freqüentemente parte da meta-informação. A meta-informação pode estar disponível em tempo de execução ou em

tempo de construção do componente ou ambos. Meta-informação em tempo de construção é importante para ferramentas de montagem de aplicações. Meta-informação em tempo de execução permite que clientes tirem proveito das propriedades que os componentes oferecem.

Atualmente, existem duas principais formas para componentes técnicos: em aplicações no cliente ou em sistemas *multi-tier*, no servidor. O *container* para componentes do cliente é tipicamente um IDE[2], onde os componentes são configurados em tempo de desenvolvimento. Componentes do cliente podem ser visíveis ou invisíveis. Componentes do servidor tipicamente encapsulam lógica de negócio em sistemas *multi-tier.* Um *container* é tipicamente parte de uma aplicação no servidor.

2.6 Arquiteturas baseadas em componentes

Esse item descreve brevemente as principais arquiteturas baseadas em componentes. O objetivo de descrever arquiteturas baseadas em componentes é situar o leitor no contexto deste trabalho. A descrição procura oferecer uma visão uniforme dessas arquiteturas, onde o serviço de replicação proposto nesta tese será aplicado sobre uma delas, a EJB [KAS2000, SUN2001] que faz parte da plataforma J2EE [SUN2002].

Arquiteturas baseadas em componentes têm sido usadas com sucesso para tratar necessidades específicas de aplicações em sistemas distribuídos, como a heterogeneidade. A abstração de componentes possibilita a construção de sistemas abertos com interfaces padronizadas e públicas. Sistemas abertos podem ser implementados através de *middlewares*, que oferecem facilidades para a programação distribuída.

Sistemas abertos permitem interoperabilidade entre diferentes sistemas. Contrastando com sistemas proprietários, sistemas abertos possibilitam que usuários escolham entre diversos fabricantes de tecnologias, i.e., sistemas de prateleira, para diferentes plataformas, as quais se adequam a suas necessidades. Esses sistemas oferecem, além da interoperabilidade, escalabilidade e portabilidade possibilitando que diferentes sistemas sejam integrados. Adicionalmente, o *software* desenvolvido em uma plataforma pode executar em outra, com adaptações mínimas.

A desvantagem de sistemas abertos é a complexidade no projeto de novas tecnologias e a adesão de especificações abertas. A superação dessas dificuldades e a preservação das vantagens da computação heterogênea ocasionaram o surgimento de organizações como OMG (*Object Management Group*) e do modelo CORBA [OMG95], além de outras plataformas mais recentes como a J2EE e. NET [MIC2002] da Microsoft. Essas plataformas serão abordadas a seguir. O leitor familiarizado poderá ir diretamente ao item 2.6.

2.6.1 CORBA

O principal objetivo do modelo CORBA (*Common Object Request Broker Architecture*) [OMG95] da OMG é permitir interoperabilidade para objetos com independência de implementação, de linguagem de programação e de plataforma de sistema operacional e de *hardware*. Cada fornecedor CORBA possui sua própria implementação, oferecida

[2] *Integrated Development Environment* é um ambiente que facilita a construção de aplicações ou programas

através de uma interface padrão CORBA, tendo em vista o melhor desempenho possível.

CORBA é um *middleware* aberto composto por objetos distribuídos. Em CORBA, um barramento de *software* é usado para conectar componentes. CORBA descreve regras específicas para criação, remoção, cópia e migração de objetos permitindo reusabilidade, portabilidade e interoperabilidade para sistemas distribuídos em ambientes heterogêneos. Um objeto CORBA pode possuir diferentes nomes e oferece diferentes mecanismos para garantir segurança.

Comunicação no CORBA

CORBA segue o modelo de comunicação cliente–servidor usando RPC. Um programa típico em CORBA pode ter a interface codificada em linguagem Orbix ou outra linguagem – que *publica* o serviço em um arquivo IDL (Interface Definition Language)[3] – e os demais códigos (programa cliente e programa servidor) em linguagem C++, ou outra linguagem como Ada95, Java ou Smalltalk. O compilador IDL transforma o arquivo IDL em um arquivo de cabeçalho na linguagem entendida pelo compilador dos programas cliente e servidor.

Uma implementação completa de CORBA permite interoperabilidade de ORBs através de conexões TCP, usando o protocolo IIOP (*Internet Inter–ORB Protocol*). Assim, qualquer servidor CORBA pode ser invocado por qualquer cliente CORBA, mesmo que eles possuam implementações distintas. Há também a possibilidade de interoperabilidade de sistemas CORBA com sistemas não–CORBA.

Arquitetura de gerenciamento de objetos

Os produtos compatíveis com o CORBA interoperam através do protocolo de alto–nível IIOP oferecido pelo ORB (*Object Request Broker*), que é a implementação mínima de uma aplicação CORBA. O ORB fornece os mecanismos básicos para o envio e o recebimento de mensagens combinando invocações de clientes com servidores. O cliente vê a requisição de forma independente de onde o objeto está localizado, qual linguagem de programação ele foi implementado, ou qualquer outro aspecto que não está refletido na interface do objeto.

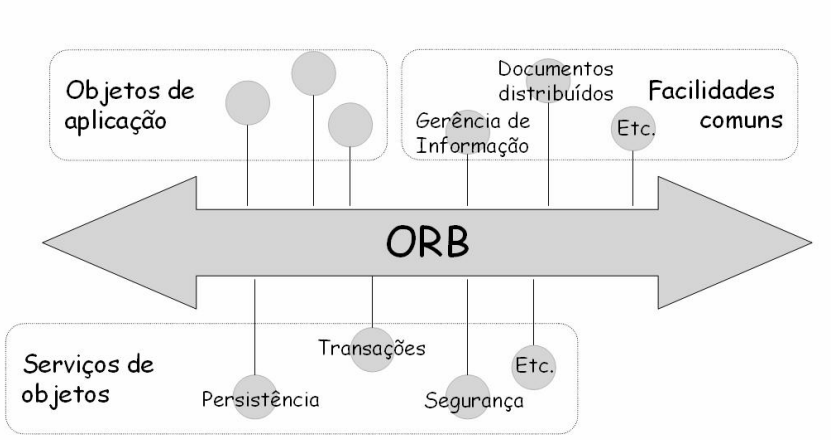

FIGURA 2.5 – Arquitetura de Gerenciamento de Objetos (OMA) [ORF97]

[3] arquivo escrito em Orbix ou outra linguagem de definição de interface

O ORB faz parte da Arquitetura de Gerenciamento de Objetos ou OMA (*Object Management Architecture*), que inclui mais três principais componentes (fig. 2.5):

- serviços de objetos: são uma coleção de serviços (interfaces e objetos) que suportam funcionalidades básicas para usar e implementar objetos;
- facilidades comuns: são uma coleção de serviços que muitas aplicações podem compartilhar, mas que não são tão fundamentais como os serviços de objetos;
- objetos de aplicação: são aplicações de usuários com necessidades específicas.

Há vários mecanismos associados ao ORB para facilitar a programação. O desenvolvedor pode implementar um ORB que escolha automaticamente um outro servidor, se o servidor corrente não estiver operacional. O ORB também pode ser programado para filtrar automaticamente invocações que gerem erros, verificar condições de segurança e fazer escolhas entre vários objetos capazes de realizar o mesmo serviço para permitir, por exemplo, balanceamento de carga.

Serviços CORBA

O ORB não executa todas as tarefas necessárias para os objetos interoperarem, pois fornece apenas os mecanismos básicos. Outros serviços necessários são oferecidos por objetos com interface IDL, que a OMG vem padronizando, como o serviço de nomes (CosNaming) [SUN2003] e o serviço de controle de concorrência.

O serviço de nomes do CORBA permite a comunicação entre objetos através de identificadores ou nomes definidos por contexto[4]. O serviço de controle de concorrência trata acessos concorrentes a recursos compartilhados permitindo resolução de conflitos.

2.6.2 J2EE

A J2EE (Java 2 Platform, Enterprise Edition) [SUN2002] é uma plataforma orientada a componentes desenvolvida pela Sun Microsystems e por um grupo de vendedores da indústria de *software* que possibilita a construção de aplicações distribuídas e reutilizáveis em linguagem de programação Java.

Um componente J2EE [SUN2002] é uma unidade de *software* auto-contida que pode ser combinada com outros componentes para compor uma aplicação J2EE. Componentes J2EE seguem a especificação J2EE. A plataforma J2EE define quatro tipos de componentes: componentes *web*, *applets*, clientes de aplicação e componentes de negócio. Os componentes de negócio são os Enterprise JavaBeans [SUN2001].

A plataforma J2EE é construída sobre a plataforma J2SE, que inclui APIs (Application Programming Interface) básicas para o desenvolvimento de programas e aplicações Java (J2SE SDK ou Java 2 SDK, Standard Edition) e um ambiente de execução (JRE ou Java 2 Runtime Environment, Standard Edition) e, portanto, aproveita muitos conceitos e serviços desenvolvidos para a linguagem Java.

[4] onde a hierarquia de nomes é construída usando árvores de diretórios com nomes ASCII para identificar objetos e atributos armazenados para cada objeto (como tamanho, último acesso, proprietário, etc.)

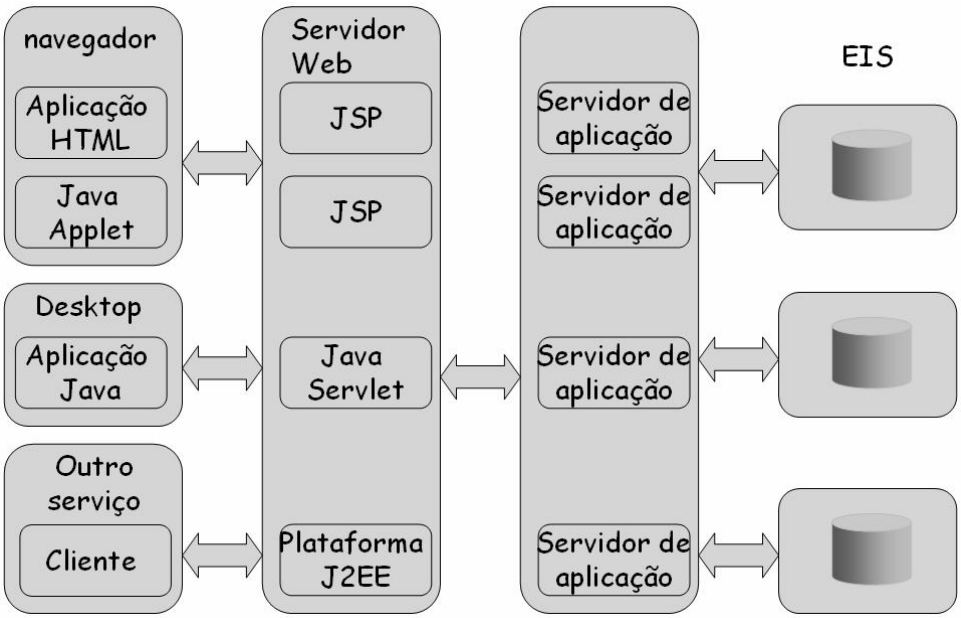

FIGURA 2.6 – Arquitetura da plataforma J2EE com multi–*tiers* [SUN2001]

A plataforma J2EE usa uma estrutura *multi–tier*, onde a lógica da aplicação é dividida em componentes. Os componentes podem estar instalados em diferentes nodos. A figura 2.6 mostra uma possível configuração para uma aplicação J2EE *multi–tier*: o cliente (navegador, *desktop* ou outro serviço), dois servidores J2EE (servidor Web e servidor de aplicação) e o servidor de banco de dados, que possibilita provê acesso ao serviço EIS[5] (ou Enterprise Information System).

Um cliente requisita serviço à aplicação, que é executada por um servidor J2EE, através de um navegador executando um *applet* (i.e., um pequeno programa Java que executa no cliente) ou através de páginas JSP (JavaServer Pages [SUN2003a]) ou *servlets* executando na camada Web, ou até mesmo através de um código Java executado a partir de um console. O servidor J2EE está separado em dois nodos, configurando duas camadas: uma para o servidor Web e outra para o servidor EJB.

O servidor Web executa as tecnologias *servlet* e JSP para gerenciar requisições de clientes externos. Um *servlet* é um programa escrito em linguagem de programação Java que executa no lado servidor e trata requisições HTTP enviadas por um cliente. Um *servlet* pode responder a requisições HTTP pela construção dinâmica de uma resposta, tratar requisições concorrentes e repassar requisições para outros servidores ou *servlets*.

Toda JSP vira um *servlet*. O desenvolvedor escreve um JSP, que na verdade é um arquivo que contém código Java e HTML. Na primeira execução do código JSP, ele é reescrito pelo *container*[6] do servidor Web na forma de um *servlet*. Depois esse *servlet* é compilado e esse é o código que é realmente executado. Quando o servidor HTTP recebe uma requisição que aponta para uma JSP ele passa essa requisição para o *container* que chama o respectivo *servlet* compilado e devolve o HTML para o servidor HTTP.

[5] o EIS é o banco de dados, que armazena informações relativas ao negócio, i.e., uma empresa, uma universidade, etc.
[6] um *container* é uma interface entre um servidor e outro componente, nesse caso, é o código JSP

O *container* EJB é uma interface entre o servidor EJB e os *componentes de negócio* ou componentes EJB, ou ainda *beans*. Os *beans* são os objetos da especificação EJB e executam o serviço funcional (como transferência bancária, compra de produtos pela Web, inscrição para um congresso, etc.) e o servidor EJB executa o serviço não–funcional que é configurado pelo desenvolvedor da aplicação EJB.

De acordo com as necessidades da aplicação, o servidor Web e o servidor EJB (ou Enterprise JavaBeans [SUN2001]) podem estar em nodos distintos, podem ser agrupados no mesmo nodo ou um deles pode até mesmo não existir. O servidor EIS permite, principalmente, o acesso ao banco de dados que pode ser qualquer banco de dados que seja compatível com a interface JDBC (ou Java Database Connectivity [SUN2003b]).

A plataforma J2EE especifica os serviços que as implementações J2EE devem oferecer. Esses serviços possuem APIs que cada produto J2EE deve fornecer às aplicações, como JDBC [SUN2003b], JMS [HAA2002], JNDI [LEE2001], JTA [SUN2002] e EJB [KAS2000, SUN2001] que será enfocada mais detalhadamente a seguir por se tratar da arquitetura-alvo escolhida para fazer a parte experimental deste trabalho.

A interface JDBC permite o acesso a bancos de dados relacionais, com independência de fornecedor. A interface JMS (ou Java Message Service [HAA2002]) é dedicada a um serviço de mensagens. A JMS comunica–se com um MOM (*message oriented middleware* ou *middleware* orientado a mensagens) permitindo a troca de mensagens ponto–a–ponto, e a publicação e assinatura entre sistemas. Essa API faz o mesmo tipo de padronização para mensagens que o JDBC trouxe para os bancos de dados. A JMS oferece independência de fornecedor de MOM em aplicações Java.

A JNDI (ou Java Naming and Directory Interface [LEE2001]) é uma interface unificada que permite acesso a diferentes tipos de serviço de diretório e de nomes. A JNDI é usada para registrar e consultar componentes de negócio e objetos em um serviço J2EE. A JNDI inclui suporte para Lightweight Directory Access Protocol (LDAP), CORBA Object Services (COS) e para o Java RMI Registry.

A JTA ou Java Transaction API [SUN2002] é um conjunto de APIs para gerenciamento de transações. As aplicações podem usar as APIs JTA para começar, executar e abortar transações.

A especificação EJB

A especificação Enterprise JavaBeans (EJB) [KAS2000, SUN2001] descreve componentes para serviços do servidor que proporcionam suporte embutido para aplicações como gerenciamento de transações, segurança (*security*) e conectividade de banco de dados. Ela constitui o núcleo da plataforma J2EE.

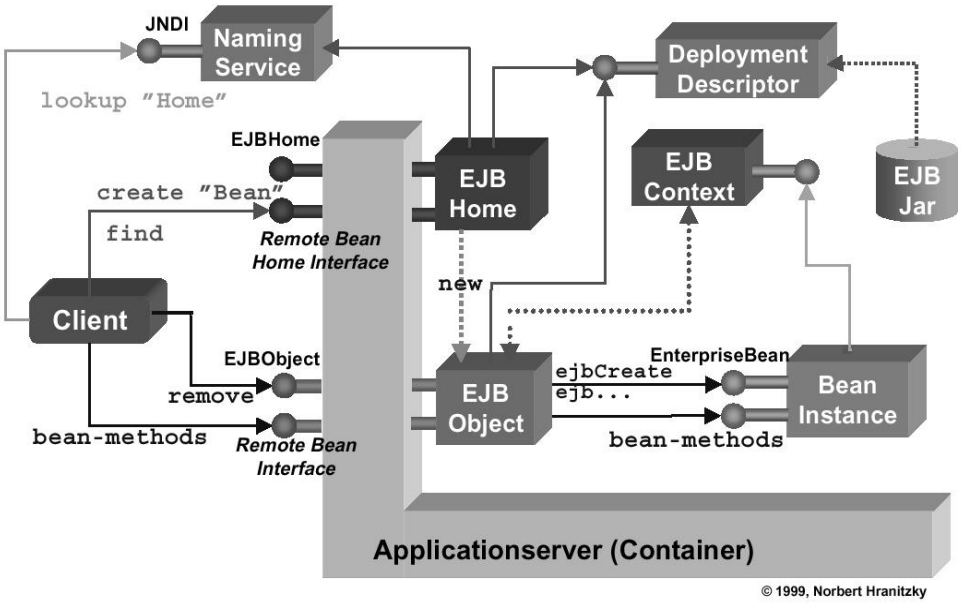

FIGURA 2.7 – Detalhamento da especificação EJB [HRA99]

A especificação EJB define uma arquitetura para um *servidor de aplicação* que, juntamente com o cliente e um banco de dados, compõe uma estrutura *three–tier*[7]. O ambiente de execução EJB é formado pelo servidor de aplicação e pelo *container* e oferece os serviços não–funcionais que são configurados pelo desenvolvedor através da seleção de propriedades[8]. O *container* representa uma interface entre o *bean* e o servidor de aplicação e é gerado por ferramentas associadas ao servidor de aplicação. Os serviços não–funcionais EJB incluem gerenciamento transacional, persistência de dados, segurança, resolução de nomes, troca de mensagens, etc.

Os *beans* representam o serviço funcional da aplicação EJB. As aplicações EJB podem transferir arquivos entre nodos distintos, manipular tabelas de bancos de dados, calcular operações matemáticas, etc. O desenvolvedor codifica as partes funcionais do *bean,* de acordo com a aplicação, e depois usa uma ferramenta (tipicamente implementada por uma interface gráfica) para inserir automaticamente os serviços não–funcionais.

Observe que a especificação EJB descreve um serviço de persistência de dados que é implementada pelo serviço transacional e pelo banco de dados, mas não garante alta disponibilidade às aplicações.

Beans

Uma aplicação EJB é formada por um conjunto de *beans* inter–relacionados. Cada *bean* modela o comportamento de uma entidade. Por exemplo, para uma aplicação que realiza a matrícula dos alunos de uma universidade, um dos *beans* pode representar as disciplinas enquanto que outro *bean* representa os alunos. A aplicação EJB irá tratar a relação entre alunos e disciplinas, além da manutenção, i.e., atualização, remoção,

[7] e difere da estrutura usada na comunicação cliente–servidor que é *two–tier*

[8] as propriedades são configuradas pelo desenvolvedor em tempo de desenvolvimento de componentes. Por exemplo, um desenvolvedor pode ou implementar na aplicação o seu próprio serviço de segurança ou escolher uma opção que insira o serviço de segurança automaticamente. Um serviço inserido automaticamente pode ser configurado através de propriedades. Por exemplo, como será mostrado a seguir, um *bean* pode ter estado ou não ter estado. O desenvolvedor simplesmente escolhe como propriedade de um *bean* ter estado. Ele não precisa se preocupar com a manutenção desse estado.

inserção de registros e de tabelas. Um *bean* é composto pelas seguintes partes (figura 2.7), desenvolvidas pelo desenvolvedor:

- objeto EJB ou classe (ou ainda componente EJB): é um arquivo em linguagem Java que implementa o serviço funcional das aplicações EJB através de métodos construídos pelo desenvolvedor e de métodos padronizados da especificação EJB. Os métodos padronizados possibilitam que o *bean* seja gerenciado pelo *container*;
- interface *home*: é um arquivo em linguagem Java que contém as assinaturas de todos os métodos usados para gerenciar o ciclo de vida de um *bean* e também os métodos para reaver uma instância de um *bean* (encontrar um ou vários *beans*), usados pela aplicação cliente.
- interface remota: é um arquivo em linguagem Java com a assinatura de todos os métodos funcionais de um *bean;*
- descritor (*deployment descriptor*): é um arquivo em formato XML que contém as propriedades dos *beans*. As propriedades dos *beans* são configuradas pelo programador.

Tipos de *beans*

A especificação EJB define três tipos de *beans* (figura 2.8): *beans* de sessão (*session beans*), *beans* de entidade (*entity beans*) e *beans* orientados a mensagens (*message–driven beans*).

A. *Beans* de sessão

Os ***beans* de sessão** são objetos que existem apenas durante a execução de uma sessão cliente–servidor. Possuem um ciclo de vida curto e, por isso, são chamados de *beans* transitórios. Os *beans* de sessão não são compartilhados por clientes concorrentes, pois são objetos associados apenas a um cliente.

Um *bean* de sessão executa operações como cálculos e FTP *downloads*. Os *beans* de sessão podem usar transações, e possuem dois modelos: com informação de *estado conversacional* (*stateful*) no servidor ou sem informação de estado (*stateless*) no servidor. Quando a sessão finaliza, o estado conversacional de um *bean* de sessão é perdido.

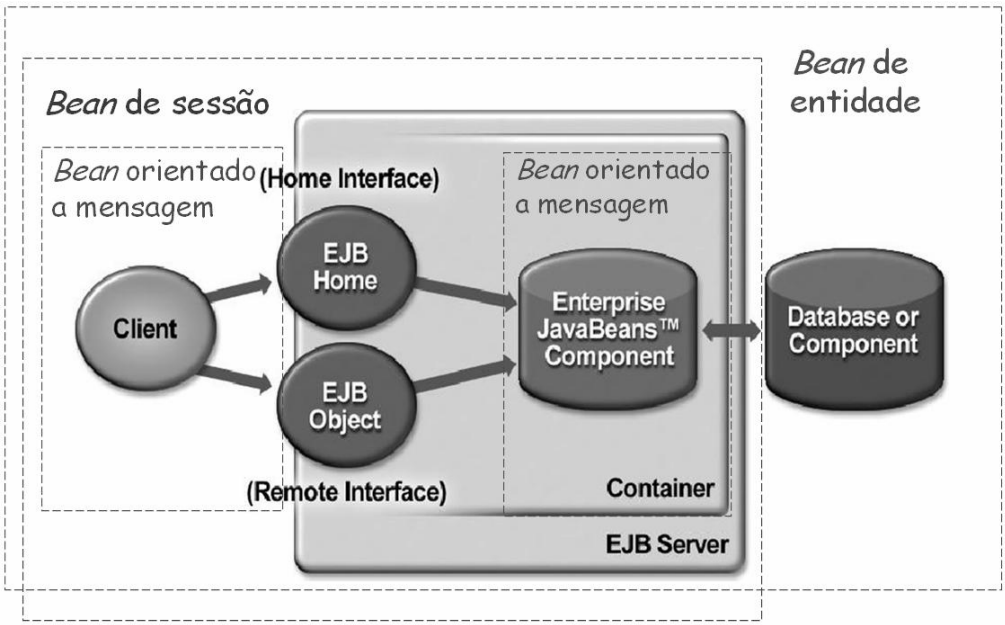

FIGURA 2.8 – Diferentes tipos de *beans* [SUN2001a]

B. *Beans* de entidade

Os ***beans*** **de entidade** são objetos persistentes que representam itens de dados em um banco de dados. Eles podem ser compartilhados por vários clientes e, por isso, são identificados por chaves primárias. A especificação EJB descreve o gerenciamento esses objetos através da implementação de persistência como propriedade não–funcional.

O estado de um *bean* de entidade pode ser armazenado no banco de dados, e ser preservado depois que a execução do *bean* de entidade termina. A persistência pode ser gerenciada de forma implícita (e invisível) pelo *container*, ou explícita pelo próprio desenvolvedor, ao construir o *bean*.

C. *Beans* orientados a mensagens

Os ***bean*** **orientados a mensagens** representam objetos que podem ser considerados como *message listeners*. Eles são executados quando recebem uma mensagem através do serviço de mensagens JMS. Os *beans* orientados a mensagens tipicamente não usam transações e não têm estado.

Serviços EJB

Uma das principais vantagens da especificação EJB é o aspecto declarativo, onde os serviços não–funcionais não são codificados na aplicação, mas são configurados pelo programador através de uma interface gráfica e são armazenados no descritor do *bean*. Esses serviços incluem o gerenciamento transacional, o serviço de gerenciamento de persistência e o serviço de gerenciamento de segurança (*security*).

Por exemplo, o gerenciamento transacional pode ser *codificado* explicitamente pelo desenvolvedor no código do *bean* ou ser demarcado implicitamente de acordo com valores de atributos transacionais. Nesse caso, o desenvolvedor não precisa escrever

código com o gerenciamento transacional, que é gerado automaticamente pelo *container* [KAS2000].

2.6.3 Microsoft .NET

A plataforma .NET [MIC2002] é uma resposta da Microsoft à plataforma J2EE e a toda cultura da linguagem Java. A .NET é uma plataforma para a criação e o uso de aplicativos, de processos e de Web *sites* baseados em linguagem XML como serviços que compartilham e combinam informações e funções, em qualquer sistema operacional ou dispositivo inteligente[9].

A Microsoft .NET procura solucionar problemas como segurança (*security*), interoperabilidade e integração de aplicações. Por estar baseada em XML, remove barreiras do compartilhamento de dados e promove a integração de *software*. A Microsoft .NET usa padrões abertos e seus componentes podem ser escritos em qualquer linguagem de programação. O modelo baseado em componentes, usado pela plataforma .NET, permite criar serviços XML reutilizáveis como componentes que interoperam entre si.

A plataforma Microsoft .NET

A plataforma Microsoft .NET tem quatro principais componentes que são mostrados na figura 2.9: .NET *framework* (que inclui serviços para a *Web* e a tecnologia COM+[10]) e Visual Studio .NET; a infra–estrutura de servidor; os serviços de blocos de construção, e *software* especial para interconectar computadores e outros dispositivos como PDAs, telefones celulares, etc.

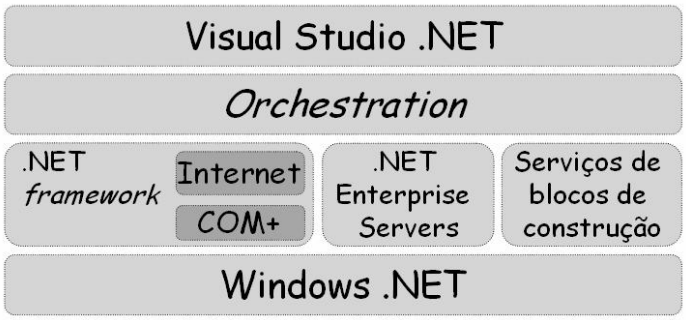

FIGURA 2.9 – Plataforma .NET [DUN2003]

O Visual Studio .NET é um conjunto de ferramentas de programação de serviços XML para a *Web* que suporta várias linguagens de programação.

A infra–estrutura de servidor inclui um ambiente de execução com os servidores Windows .NET e .NET Enterprise Servers. Essa infra–estrutura é um conjunto de aplicativos para construir, desenvolver e operar serviços XML na *Web*. Esses aplicativos permitem a construção de aplicações com escalabilidade e serviços para armazenar e recuperar dados estruturados em XML.

---

[9] Um dispositivo inteligente adapta-se automaticamente às necessidades do usuário e às condições da rede (como largura de banda). Ele reconhece outros dispositivos, as informações que acessa, e os *softwares* e serviços introduzindo a conectividade apropriada para as interações do usuário.

[10] COM+ é a evolução dos produtos COM (Component Object Model) e MTS (Microsoft Transaction Server) da Microsoft, usada para desenvolver aplicações distribuídas para a plataforma Windows da Microsoft

O *Orchestration* é um conjunto de serviços, como gerenciamento transacional e trata–mento de exceções, e possibilita que o desenvolvedor construa aplicações capazes de se recuperar após falha. Essas facilidades são requeridas para serviços XML robustos.

Os serviços de blocos de construção são um conjunto de serviços XML para a *Web* centrados no usuário, permitindo simplicidade e consistência personalizadas. O conjunto inclui serviços de identificação do usuário, de transmissão de mensagens, de armazenamento de arquivos, de gerenciamento das preferências do usuário e de agenda.

Os serviços XML são módulos de *software* independentes para interconectar aplicativos, serviços e dispositivos inteligentes. Esses serviços podem ser agregados para formar um *software* capaz de realizar determinada tarefa. Os serviços XML também permitem que os desenvolvedores escolham entre construir e comprar seus aplicativos, e entre consumir outros serviços XML para completar suas soluções ou expor seus próprios serviços para serem usados por outros aplicativos ou serviços. Os serviços XML independem de dispositivo e não se restringem a uma determinada linguagem de programação, aplicativo ou serviço *on–line*.

O .NET *framework*

O .NET *framework* (fig. 2.10) é o conjunto de interfaces programáveis que constitui o núcleo da plataforma Microsoft .NET. O .NET *framework* é formado principalmente por um ambiente de execução CLR (*Common Language Runtime*) e por um conjunto de bibliotecas de classes. O CRL é o grande diferencial da .NET, se comparado a JRE – *Java Runtime Environment*, pois permite um ambiente de execução para diferentes linguagens de programação. Essas duas camadas permitem ambientes para o desenvolvimento de aplicações Windows (*Windows forms*), e para a Web além de serviços XML através do ASP .NET, que é uma versão do *Active Server Pages* (ou ASP) [ASP2002] para a plataforma .NET.

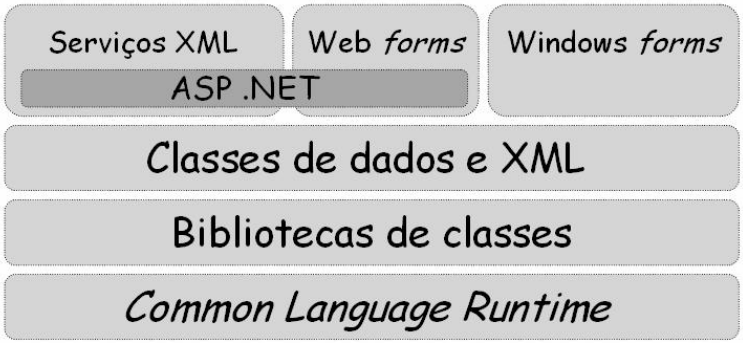

FIGURA 2.10 – .NET *framework* [MIC2002a]

O ambiente de execução CLR possui as seguintes características:

- conversão de uma linguagem de baixo nível estilo *assembler*, chamada *intermediary language* (IL), para código nativo da plataforma;
- gerência de memória, incluindo a coleta de lixo;
- restrições de segurança para o código em execução;
- carga e execução de programas, com controle de versão e outras características.

O CLR usa o CTS (*Common Type System*) para reforçar a segurança da tipagem das classes, assegurando que todas as classes sejam compatíveis entre si e estabelecendo uma abstração padronizada.

Os programas .NET são construídos de *assemblies* que são semelhantes aos *beans* da especificação EJB. Um *assembly* é uma coleção de código compilado[11], e forma uma unidade funcional atômica. Cada *assembly* contém um manifesto com o nome, o número de versão, a localização, a lista de arquivos que forma o *assembly*, as dependências do *assembly* e as características exportadas pelo *assembly*.

Os *assemblies* podem ser simples ou compostos. *Assemblies* simples contêm todas as informações em um único arquivo, enquanto que *assemblies* compostos suportam recursos externos (como *bitmaps*, ícones, sons, etc.), e diferentes arquivos para o código do núcleo e bibliotecas. Os *assemblies* também podem ser compartilhados ou privados. Os *assemblies* privados pertencem a uma aplicação específica. *Assemblies* compartilhados são armazenados em um repositório comum, e podem ser usados por múltiplos desenvolvedores. Eles têm mais restrições de segurança que os *assemblies* simples: requerem controle de versão e de segurança, e um mecanismo para assegurar que o nome do *assembly* seja único.

2.7 Alta disponibilidade através de réplicas em sistemas orientados a componentes

Um sistema pode ser altamente disponível mesmo que apresente períodos freqüentes de inoperabilidade, se estes períodos forem extremamente curtos [JOH96]. Alta disponibilidade pode ser implementada em sistemas computacionais por réplicas de componentes de *software* e *hardware*. O uso de réplicas permite que componentes falhos sejam substituídos por outros componentes operacionais.

Em sistemas distribuídos, onde tipicamente o modelo cliente-servidor é implementado, *failover* [JEW2000] pode ser definido como a habilidade de uma requisição que está sendo processada executar um chaveamento (*switchover*) de um servidor falho para um servidor alternativo, visando alta disponibilidade sem interromper o serviço para os clientes. *Failover* é uma das propriedades mais desejadas e mais difícil de ser assegurada em sistemas distribuídos porque determinar o estado no servidor falho não é uma tarefa simples. A grande dificuldade é que o último estado acessado por um cliente precisa estar disponível no sistema distribuído para que o serviço possa ser reinicializado no servidor alternativo.

Replicação

O uso de réplicas possibilita o *failover*. Entretanto, a replicação introduz seus próprios problemas como a manutenção da consistência das diferentes réplicas. Problemas de inconsistência podem acontecer devido à concorrência de operações de atualização de estado de réplica (quando dois ou mais clientes concorrentes fazem diferentes requisições ao serviço replicado) ou a falhas em nodos. Protocolos de consistência de réplicas visam resolver esses problemas e serão abordados no próximo capítulo. Esses protocolos são facilmente implementados por primitivas de comunicação de grupo.

[11] portanto é um componente

Grupo com suporte de réplicas

O modelo de grupos oferece uma estrutura para gerenciar réplicas em um sistema distribuído, e mantê–las em um estado consistente. Nesse caso, o grupo é chamado *grupo de réplicas* ou *grupo de replicação*. É assumido que todos os membros de um *grupo de replicação* mantêm as mesmas réplicas (de componentes, objetos ou processos).

Um cliente se comunica com esse grupo exclusivamente por um protocolo de comunicação ponto–a–ponto. O cliente não conhece todas as réplicas do grupo, mas apenas o endereço do grupo. Um *serviço de nomes* externo é encarregado de fornecer o endereço do grupo para o cliente.

As primitivas de comunicação podem admitir diferentes propriedades (por exemplo, *ordem causal* e *ordem total* [BIR87, BIR96]) e facilitam a construção de aplicações com réplicas.

2.8 Implementação de réplicas em sistemas distribuídos

A implementação de réplicas para alta disponibilidade depende da estrutura do sistema distribuído disponível. A sua forma mais usada é a replicação por *software* ou por *hardware*. Segundo Guerraoui *et al.* [GUE97], a replicação por *software* é a tendência atual, pois apresenta baixo custo de desenvolvimento em relação ao *hardware,* que requer componentes específicos.

O ideal é que o serviço de sincronização das réplicas seja automático para o usuário e para o desenvolvedor. Alguns sistemas operacionais possuem protocolos de replicação embutidos, mas eles não são muito populares. Um problema destes sistemas é que eles geralmente são associados a um *hardware* especial, que dificulta o crescimento do sistema e encarece o seu preço. O outro problema é que implementar réplicas em sistemas operacionais restringe a portabilidade de aplicações entre sistemas operacionais distintos, já que cada sistema operacional pode implementar um protocolo de replicação diferente (ou então não implementar replicação).

Briot e Guerraoui [BRI97, BRI98, GUE96a] descrevem três diferentes estratégias para que o modelo de objetos seja aplicado a sistemas distribuídos:

- **como bibliotecas ou aplicativa**: aplica os conceitos de orientação a objetos para estruturar sistemas distribuídos como bibliotecas de classes e *frameworks*;
- **por integração:** integra os conceitos de sistemas distribuídos com conceitos de objetos;
- **por interceptação**: integra bibliotecas de classes em uma linguagem de programação orientada a objetos.

Essas diferentes estratégias podem ser usadas para incorporar réplicas de *software* no nível da aplicação, em *middleware* ou de mesmo no nível de sistema operacional, como objetos em um sistema distribuído. O ideal é que essas estratégias sejam aplicadas sobre sistemas distribuídos populares, facilitando o uso do produto final por desenvolvedores de aplicação.

A estratégia **aplicativa** é orientada para desenvolvedores de sistemas e ajuda a identificar as abstrações que representam componentes básicos de sistemas distribuídos,

i.e., consiste em modelar o sistema distribuído usando as abstrações da programação orientada a objetos.

A estratégia **integrativa** é orientada a desenvolvedor de aplicações e usa as facilidades de linguagens de alto–nível (orientadas a objetos) para programar sistemas distribuídos [GUE96a]. Deixar a cargo de desenvolvedor de aplicações a implementação de réplicas possui a desvantagem de que cada aplicação requer um protocolo de replicação específico. Por outro lado, o desenvolvedor de sistema pode implementar réplicas adequadas a um sistema genérico e o desenvolvedor de aplicações pode se preocupar somente com os serviços funcionais.

A terceira estratégia é orientada a ambos, desenvolvedores de sistemas e de aplicação, e provê uma infra–estrutura para habilitar a customização dinâmica com mínimo impacto sobre os programas de aplicação [BRI98, GUE96a]. O sucesso de um sistema com interceptação depende da linguagem de alto–nível e de uma biblioteca com poderosas abstrações. A vantagem de usar uma infra–estrutura é que o desenvolvedor pode contar com uma infinidade de código já programado e verificado, evitando muitos erros de programação.

Segundo Guerraoui *et al.* [GUE96a], sob o ponto de vista de um sistema distribuído, a estratégia aplicativa é a mais promissora, apesar das estratégias *integrativa* e *reflexiva* serem mais usadas na prática. A idéia da estratégia aplicativa é aplicar conceitos como encapsulamento e abstração, e possivelmente os conceitos de classe e herança, como ferramentas estruturais para construir sistemas distribuídos. É exatamente neste sentido que a programação baseada em componentes, usada no contexto deste trabalho, mostra–se eficiente em fase de projeto.

Usando uma arquitetura de componentes já existente, este trabalho acrescenta serviços adicionais através da integração de duas bibliotecas, procurando preservar os códigos e a arquitetura já existentes.

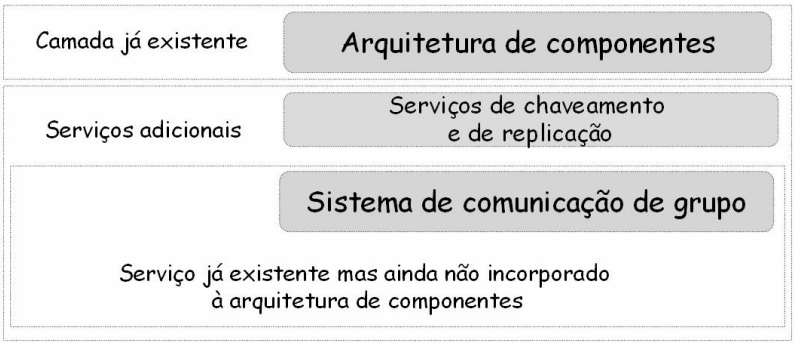

FIGURA 2.11 – Arquitetura com os serviços adicionais

Uma das bibliotecas implementa um sistema de comunicação de grupo e outra implementa uma arquitetura de componentes (figura 2.11). Para integrar as duas bibliotecas foi necessário codificar uma nova biblioteca que implementa a interface entre essas duas bibliotecas. Essa nova biblioteca enxerga componentes como membros em um grupo de replicação, que são gerenciados por um sistema de comunicação de grupo.

2.9 Observações do capítulo

A especificação EJB é o núcleo da plataforma J2EE e descreve serviços não–funcionais que são automaticamente oferecidos por um servidor às aplicações distribuídas. É justamente essa arquitetura que serviu como base para o desenvolvimento do serviço de replicação aqui desenvolvido.

CORBA já possui uma especificação para um serviço de alta disponibilidade, o FT–CORBA [SIE2001]. Essa especificação descreve uma estrutura onde cada desenvolvedor implementa sua extensão para permitir alta disponibilidade de acordo com as necessidades de sua aplicação. É o desenvolvedor que precisa inserir o protocolo de replicação no FT–CORBA, enquanto que no serviço de replicação aqui desenvolvido, o protocolo é implementado automaticamente pelo ambiente de execução.

A especificação EJB é aberta e cada provedor pode fazer a sua implementação, enfatizando as necessidades que julgar importantes desde que um núcleo mínimo de serviços seja oferecido. O código EJB é implementado totalmente nas linguagens Java e XML, evitando a conversão entre diferentes tipos de linguagens.

3 Replicação para alta disponibilidade

A natureza isolada dos nodos em um ambiente distribuído pode ser facilmente aproveitada para implementar alta disponibilidade através da replicação através do conceito de grupos. A replicação permite alta disponibilidade, quando uma réplica localizada em um nodo falho é substituída por outra, e alto desempenho principalmente através de acessos concorrentes para leitura em réplicas.

Contudo, a replicação induz ao problema de consistência de réplicas: quando um objeto é alterado, suas réplicas também precisam ser atualizadas, para manter um estado distribuído consistente. Em outras palavras, o critério de serializabilidade (ou *one–copy serializability* [BER87, JAL94]) precisa ser garantido. Manter o estado distribuído consistente requer protocolos específicos para assegurar a consistência das réplicas ou *protocolos de replicação*.

Para satisfazer o critério de serializabilidade, um protocolo de replicação precisa obedecer às seguintes propriedades [GUE97, p.69]:

- **ordem**: operações de atualização concorrentes, enviadas por diferentes clientes, devem ser executadas na mesma ordem por réplicas distintas;
- **atomicidade**: quando uma réplica realiza uma operação de atualização, todas as demais réplicas também devem realizar esta operação.

Protocolos de replicação podem ser aplicados a sistemas de bancos de dados e a sistemas distribuídos. Originalmente, *aplicações* em sistemas de bancos de dados consideravam grande volume de dados, enquanto que *programas* em sistemas distribuídos contemplam computação sobre um volume reduzido de dados, como cálculos científicos. A observação dessas características das aplicações e de programas é extremamente importante e possibilita fazer escolhas adequadas na implementação dos protocolos de replicação.

3.1 A replicação como um problema abstrato

Wiesmann et al. [WIE2000] apresenta uma interessante comparação entre os conceitos e implementações usados nos protocolos de replicação em duas diferentes comunidades, bancos de dados e sistemas distribuídos. Primeiro a replicação é apresentada como um problema abstrato, descrito em diferentes fases (fig. 3.1):

- requisição do cliente: o cliente submete uma requisição para uma ou mais réplicas;
- coordenação das réplicas: as réplicas coordenam–se para estabelecer a ordem da execução da requisição;
- execução: a requisição é executada nas réplicas;
- concordância entre as réplicas: as réplicas concordam com o resultado obtido pela execução da requisição;
- resposta ao cliente: o resultado da execução da requisição é enviado ao cliente, por uma ou mais réplicas.

Dependendo das características do protocolo de replicação, como será visto mais adiante, algumas dessas fases poderão não existir, otimizando o processo de replicação.

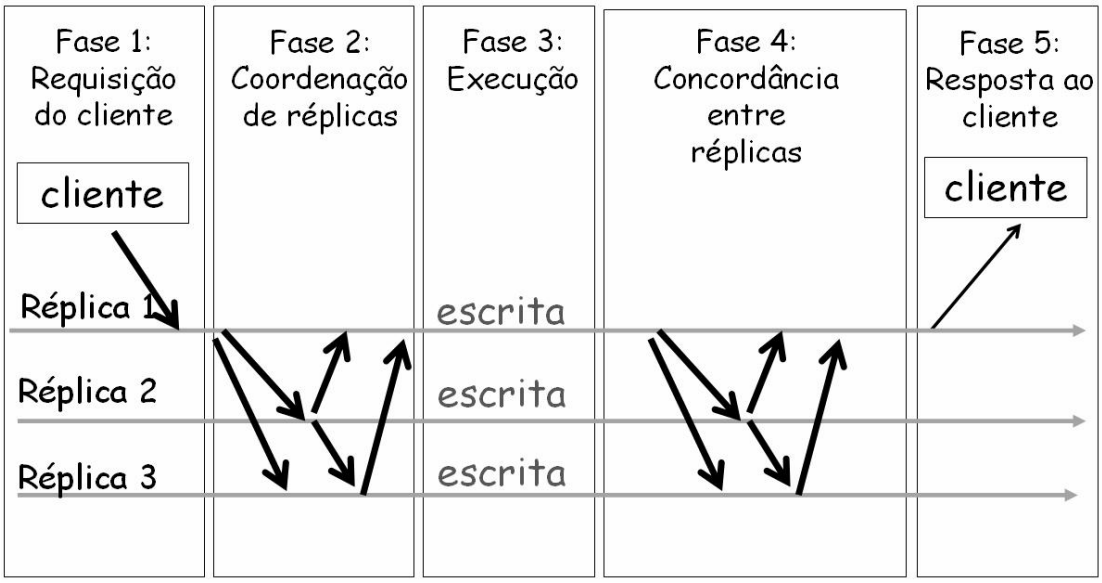

FIGURA 3.1 – A replicação como um problema abstrato

A partir dessa abstração (fig. 3.1), *Wiesmann et al.* [WIE2000] analisa várias técnicas para implementar protocolos de replicação em bancos de dados e em sistemas distribuídos destacando similaridades e diferenças entre essas duas comunidades. Essa análise serviu como uma importante ferramenta para explorar a integração de soluções em bancos de dados e sistemas distribuídos no escopo deste trabalho.

3.2 Protocolos clássicos de replicação

Na literatura, dois protocolos de replicação são amplamente difundidos: cópia primária [BUD93, GUE96, GUE97, WIE2000] e réplicas ativas [SCH93, WIE2000]. Tipicamente, utiliza–se a estratégia *lê somente uma e atualiza todas as réplicas* (*read one write all* ou ROWA). A *atualização de estado* é uma operação distribuída e envolve todas as réplicas de um objeto enquanto que a *leitura de estado* é uma operação local e envolve apenas uma réplica.

3.2.1 Cópia primária

Na cópia primária [ASL76, BUD93], uma das réplicas é escolhida como cópia ou réplica primária ou ainda *servidor primário*. As demais réplicas são *backups* ou réplicas secundárias. Os clientes estabelecem comunicação apenas com o servidor primário. O servidor primário gerencia as demais réplicas e envia a resposta da operação para o cliente.

Nas operações de *leitura de estado*, o servidor primário simplesmente envia uma resposta ao cliente, sem consultar os demais servidores que são apenas repositórios de dados. Nas operações de *atualização de estado*, ao receber uma requisição do cliente (fase 1 na fig. 3.2), o servidor primário (réplica 1) propaga a requisição aos servidores secundários (fase 2) que executam o serviço localmente (fase 3). Depois de verificar que cada secundário também conseguiu atualizar seu estado e realizar a alteração seu estado (fase 4), o servidor primário responde ao cliente (fase 5).

Se o servidor primário detectar que algum secundário não conseguiu atualizar seu estado, este secundário é simplesmente descartado. Falhas no servidor primário são

detectadas pelos clientes, quando não recebem a resposta do servidor depois de determinado tempo de solicitação. Quando o servidor primário falha, um novo servidor primário precisa ser escolhido entre os secundários.

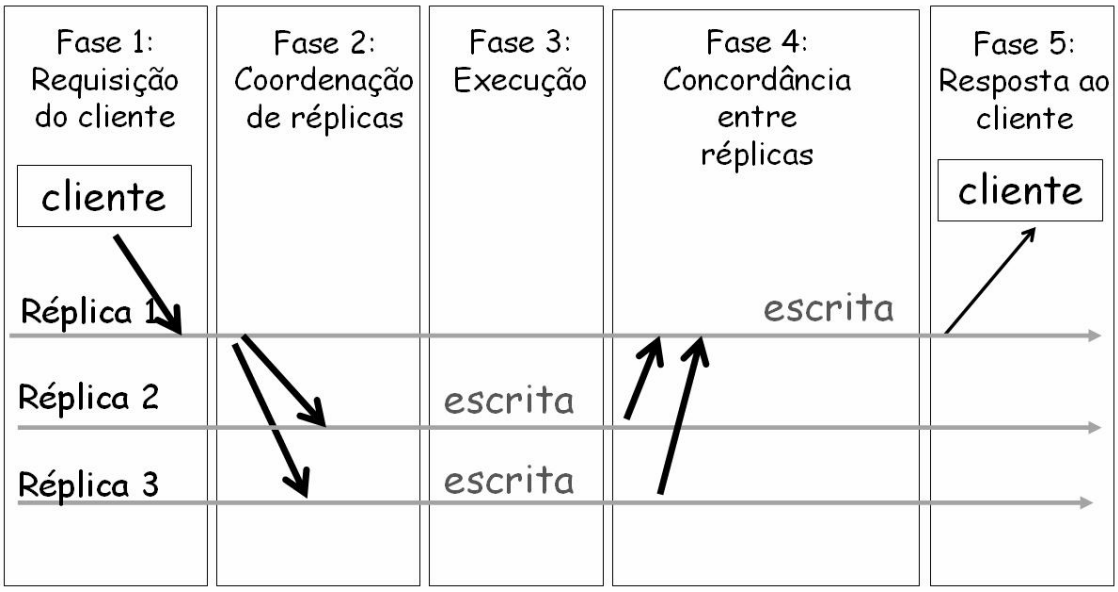

FIGURA 3.2 – Servidor primário somente é atualizado após secundários

Uma variação deste protocolo usa apenas um secundário para cada primário e por isso é chamada primário–*backup*. A desvantagem do protocolo primário–*backup* é que a disponibilidade é reduzida, uma vez que somente uma réplica de segurança é usada para cada objeto.

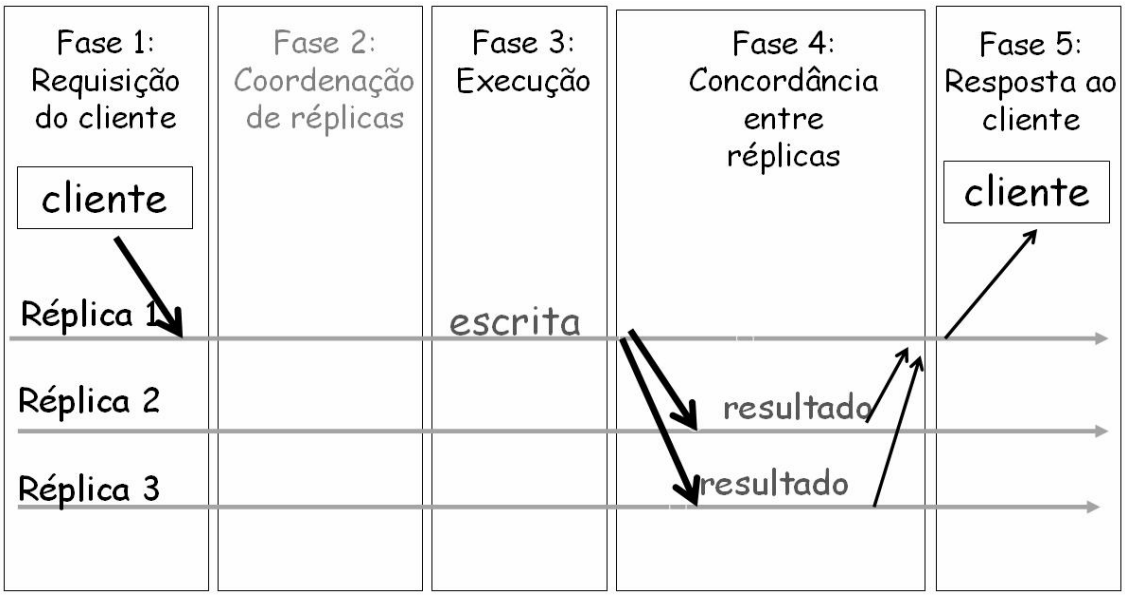

FIGURA 3.3 – Protocolo de cópia primária otimizado

Muitos autores [GUE96, GUE97, WIE2000] descrevem uma otimização do protocolo de cópia primária (fig. 3.3), onde o primário atualiza seu estado (fase 3) e envia o resultado do serviço para os secundários (fase 4), antes de responder ao cliente (fase 5). A otimização ocorre pois não é necessária nenhuma coordenação entre as réplicas (fase 2 não existe) e, principalmente, porque apenas o primário realiza o serviço. As demais réplicas apenas atualizam seu estado com o resultado fornecido pelo primário.

3.2.2 Réplicas ativas

Nas réplicas ativas [SCH93], também chamadas *máquinas de estado*, todas as réplicas executam a mesma seqüência de requisições. Não há uma réplica centralizadora, como no protocolo de cópia primária. Consistência é garantida supondo que todas as réplicas que recebem a mesmas requisições executam as mesmas operações e produzem resultados idênticos, representando um *comportamento determinístico*[12]. As operações executadas sobre as réplicas possuem duas propriedades [SCH93, p.175]:

- acordo (*agreement*): todas as réplicas que não estão em *suspeita de falha* recebem as mesmas requisições;
- ordem: se uma réplica processa a requisição r_1 antes da requisição r_2, então todas as réplicas processam essas mesmas requisições nessa ordem.

As duas propriedades anteriormente descritas asseguram o critério de serializabilidade requerido em protocolos de replicação. Note que a propriedade de *acordo* também é requerida pelo protocolo de cópia primária e que essa propriedade foi definida como *atomicidade* no capítulo anterior.

Note também que a ordem do processamento de requisições pode ser uma propriedade relativa para requisições concorrentes de clientes distintos. Desde que todas as réplicas recebam duas requisições distintas na mesma ordem, o sistema estará consistente. Se clientes concorrentes enviam requisições distintas para um mesmo objeto, e desde que, pela ausência de um relógio global não há como determinar que cliente enviou a requisição antes, qualquer ordem de entrega de mensagens pode ser aceita pelas réplicas, desde que essa ordem seja exatamente a mesma em todas as réplicas.

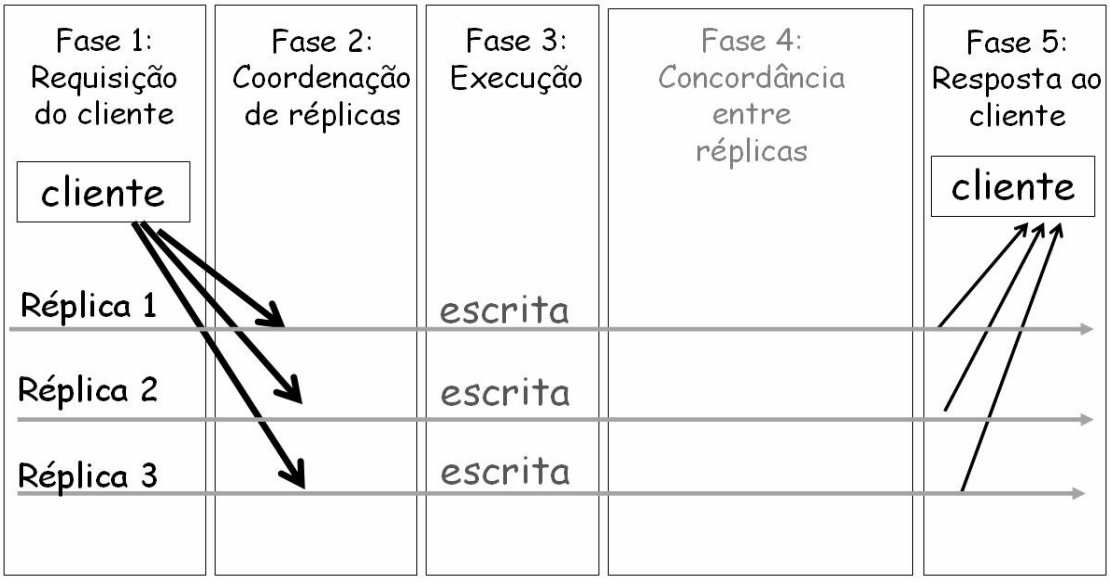

FIGURA 3.4 – Protocolo de réplicas ativas

Nas réplicas ativas, durante uma operação com *atualização de estado*, o cliente difunde uma mensagem para todas as réplicas (fase 1 na fig. 3.4). As réplicas realizam alguma coordenação para impor ordem entre requisições concorrentes (fase 2). Então cada réplica executa a operação (fase 3), alterando seu estado, e retorna o resultado da

[12] o resultado de uma operação depende somente do estado inicial da réplica e da seqüência de operações executadas sobre essa réplica [GUE97]

operação para o cliente (fase 5). Não é necessária nenhuma concordância entre as réplicas antes de responder ao cliente (i.e., a fase 4 não existe). O cliente espera até receber a primeira resposta de qualquer uma das réplicas ou até receber a maioria de respostas idênticas. Nas operações de leitura, o cliente difunde uma mensagem para todas as réplicas e aguarda a primeira resposta ou a maioria das respostas idênticas.

3.2.3 Comparação entre os protocolos clássicos de replicação

Uma vantagem do protocolo de cópia primária [GUE97, WIE2000] é que ele usa menor poder de processamento se comparado ao protocolo de réplicas ativas [WIE2000], quando apenas o servidor primário realiza o processamento do serviço (enquanto que nas réplicas ativas todas as réplicas realizam o processamento). A principal desvantagem do protocolo de cópia primária é a necessidade de escolha de um novo servidor primário. Em caso de falha do servidor primário (ou coordenador), um servidor primário deve ser escolhido entre os demais servidores. A falha (ou suspeita de falha) do servidor primário não é invisível para o cliente. O cliente detecta suspeita de falha no servidor primário tipicamente por *timeout*. Se o servidor primário falha, o cliente que conhece apenas o endereço do servidor primário deve receber um novo endereço para que o serviço não seja interrompido. A ocorrência de falhas em servidores secundários é invisível para o cliente.

Outra desvantagem é que a replicação por cópia primária pode gerar tempos de respostas inaceitáveis, quando o primário torna–se um gargalo, pois ele centraliza todas as operações de todos os clientes. Nas réplicas ativas, o cliente conhece apenas o endereço do grupo de replicação (e não o endereço de cada membro do grupo). A ocorrência de falhas em membros é invisível. A principal desvantagem do protocolo de réplicas ativas é a necessidade de um mecanismo que assegure o determinismo para atualizações de clientes concorrentes.

Outra vantagem é que o protocolo de cópia primária aceita servidores não–determinísticos. A ordem na qual as requisições são processadas pelas réplicas é imposta pelo servidor primário, i.e., apesar dos servidores serem não–determinísticos, o primário impõe o determinismo. Esse mecanismo é adequado para implementar réplicas de servidores *multi–thread*. O servidor primário assegura o controle de concorrência entre as réplicas e o determinismo, uma vez que clientes concorrentes podem solicitar atualização sobre um mesmo item de dado de um (único) servidor não–determinístico.

3.3 Replicação com comunicação de grupo em sistemas distribuídos

A abstração de grupos [BIR87] facilita o desenvolvimento de aplicações distribuídas confiáveis com réplicas. Um processo externo a um grupo de processos enxerga o grupo como uma única unidade. As réplicas de um mesmo objeto podem formar um grupo de réplicas ou *grupo de replicação*. Quando uma réplica no grupo atualizar seu estado, todas as réplicas do grupo atualizam seu estado, mantendo um *estado distribuído* consistente.

O *grupo de replicação* é mantido por um grupo de servidores, onde o mapeamento é de um para um, i.e., uma réplica para cada servidor. Os servidores que mantêm réplicas de objetos em um grupo são chamados de *membros*. Para simplificar, servidores também podem ser chamados simplesmente de *réplicas*.

Um cliente que deseja comunicar uma operação para o grupo, ao invés de enviar uma mensagem individual para cada membro, pode usar uma *primitiva de comunicação de grupo*. Um processo cliente se comunica com um grupo usando um único endereço. O cliente não precisa saber o nome dos membros do grupo de servidores.

No contexto deste trabalho, o grupo de replicação implementa replicação total de objetos, onde todos os membros de um grupo de replicação contêm as mesmas réplicas. Um sistema de comunicação de grupo provê as primitivas necessárias para implementar consistência a réplicas, bem como um serviço de composição do grupo e um serviço de detecção de nodos (ou réplicas) suspeitos de falha. A abstração de grupos pode facilitar a implementação dos protocolos de replicação, como será mostrado a seguir.

3.3.1 Grupo com cópia primária

No protocolo de cópia primária com abstração de grupos [GUE97, WIE2000], uma das réplicas do grupo é o servidor primário e coordena todas as atividades do grupo. As demais réplicas são os servidores secundários ou *backups*. Cada cliente sabe quem é o servidor primário e estabelece comunicação somente com ele. As operações de *leitura de estado* são recebidas e respondidas pelo servidor primário, sem consulta aos secundários.

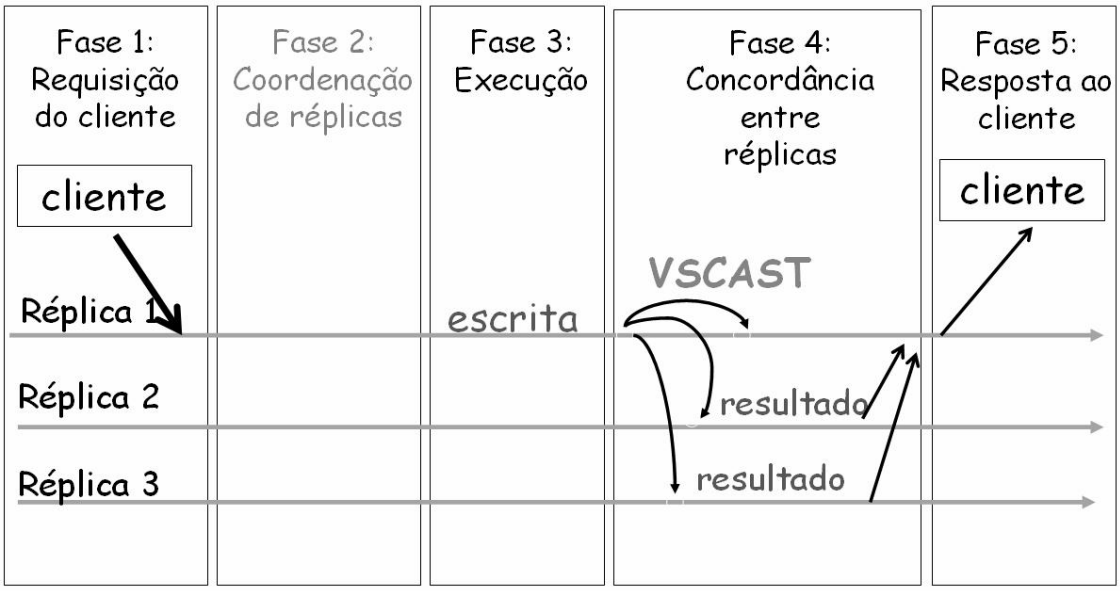

FIGURA 3.5 – Protocolo de cópia primária com grupo [WIE2000]

As operações de *atualização de estado* solicitadas por clientes (fase 1 na figura 3.5) também são recebidas e executadas somente pelo primário (fase 3). Tipicamente, os secundários não processam o serviço, apenas alteram o seu estado de acordo com o processamento realizado no servidor primário (fase 4). A comunicação entre o servidor primário e os secundários garante que atualizações são recebidas e processadas na mesma ordem. A primitiva de comunicação de grupo adequada, nesse caso é a *view synchronous multicast* ou VSCAST [GUE97]. Apenas o servidor primário responde ao cliente (fase 5). Observe que o protocolo de cópia primária não exige qualquer coordenação (fase 2) entre as réplicas do grupo de replicação.

3.3.2 Grupo com réplicas ativas

O protocolo de réplicas ativas é uma estratégia não–centralizada que submete todas as réplicas de um grupo às mesmas regras. Clientes contatam o endereço do grupo e não endereçam servidores individualmente. A consistência das réplicas é garantida pela primitiva TOCAST [GUE97].

Depois que a requisição de um cliente foi enviada a todos os membros do grupo (fase 1 na figura 3.6), alguma coordenação[13] é executada entre as réplicas para garantir a ordenação total de mensagens (fase 2) requerida pela primitiva TOCAST. Então, cada réplica processa a requisição (fase 3) e retorna a resposta ao cliente (fase 5), que normalmente espera até receber a primeira resposta [WIE2000]. Não existe qualquer concordância entre as réplicas (i.e., a fase 4 não existe). O cliente pode aguardar por todas as respostas ou pela maioria das respostas e fazer uma comparação entre o que foi recebido.

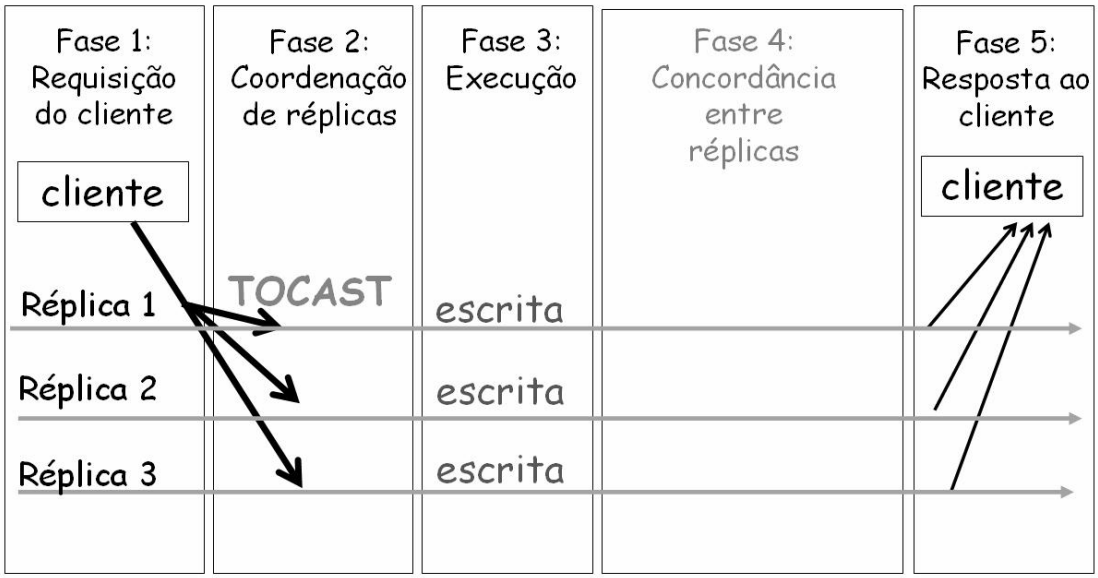

FIGURA 3.6 – Replicação ativa com grupo [WIE2000]

Em operações de *leitura de estado,* o cliente acessa apenas uma réplica, pois é assumido que se uma réplica recebe uma mensagem através da primitiva TOCAST, então ela atualiza o seu estado, caso contrário ela será excluída do grupo de replicação. O conceito de sucessivas visões, anteriormente descrito para o protocolo de cópia primária, também é aplicado às réplicas ativas, uma vez que as réplicas ativas podem falhar durante a execução da primitiva TOCAST e que, mesmo assim, a atomicidade na entrega das mensagens precisa ser garantida.

3.3.3 Observações sobre os protocolos com grupos

Além das vantagens e desvantagens descritas no item 3.2.3, no contexto de grupos pode–se dizer que uma vantagem do protocolo de réplicas ativas é que ele pode ser facilmente mapeado para um grupo de replicação. O protocolo de cópia primária requer que um dos servidores (servidor primário) seja usado como endereço do grupo de replicação. Isso pode comprometer a transparência da abstração de grupos quando um protocolo como o RMI é usado para conectar cliente e o servidor primário. Se o servidor

[13] Essa coordenação depende de como o subsistema implementa a primitiva TOCAST

primário falha, o cliente que conhece apenas o endereço do servidor primário deve receber um novo endereço para que o serviço não seja interrompido.

Nas réplicas ativas, o cliente contata o endereço do grupo de replicação. A ocorrência de falhas em membros é invisível. A desvantagem é que o determinismo precisa ser assegurado para operações de atualização.

Sistemas de comunicação de grupo implementam, na prática, o conceito de *grupos fechados*, o que torna o uso do protocolo de cópia primária mais eficiente que o protocolo de replicação ativa. Um grupo fechado requer que cada processo, cliente ou servidor, ingresse no grupo para requisitar ou colaborar em um serviço, i.e., para executar um *multicast* para o grupo. Se muitos processos são mantidos em um grupo, o gerenciamento do grupo pode ser uma tarefa complexa.

3.4 Replicação em sistemas de bancos de dados e em sistemas distribuídos

A replicação em sistemas de banco de dados pode ser assíncrona ou síncrona. Técnicas assíncronas realizam cópias dos objetos em intervalos de tempo pré–estabelecidos, de preferência quando o serviço para os clientes é reduzido ou inexistente. Por exemplo, ao final de cada dia operacional.

Técnicas síncronas requerem servidores secundários ativos (*on–line backups*), que são atualizados à medida que as operações são processadas no banco de dados do primário. Sistemas de bancos de dados replicados com requisitos rígidos de alta disponibilidade usam técnicas síncronas. Interromper o serviço dos clientes ou aguardar uma oportunidade para fazer uma cópia de segurança em momento adequado é uma solução considerada ultrapassada.

Tipicamente, a implementação de réplicas em sistemas de banco de dados é baseada nas propriedades ACID[14]. Entretanto, a rigidez das propriedades ACID nem sempre é necessária. Por exemplo, a propriedade de *atomicidade* (i.e., tudo ou nada) é muito rígida: protocolos de replicação podem aceitar que apenas alguns nodos executem uma operação de escrita porque os demais nodos falharam. A operação não precisa ser abortada porque alguns nodos não realizaram o serviço. Os nodos falhos podem ser excluídos do *grupo de replicação*. Observe que o conceito de *atomicidade* em banco de dados difere do conceito de *atomicidade* anteriormente definido no contexto de grupos e de sistemas distribuídos (ver item 2.2.5).

Sistemas de banco de dados síncronos tradicionalmente implementam alta disponibilidade usando técnicas de replicação centralizadas para propagar as atualizações de uma transação. A implementação dessas técnicas é freqüentemente baseada no protocolo de *commit* de duas fases [GRA78]. Neste protocolo, um nodo é encarregado de coordenar (fase 1) e confirmar (fase 2) a difusão de novas versões para as réplicas. Contudo, esse protocolo adiciona um retardo substancial para realizar a comunicação entre processos que confirmam cada transação.

Além desse retardo, o protocolo de *commit* de duas fases não garante serviço altamente disponível em caso de falha no nodo coordenador. Se o nodo coordenador falha, os demais processos podem ser incapazes de confirmar a última transação. O sistema distribuído pode bloquear e permanecer indisponível até que cada processo seja reinicializado.

[14] quatro propriedades de uma transação: atomicidade, consistência, isolamento e durabilidade

O protocolo de *commit* de três fases [SKE81] permite maior disponibilidade que o protocolo de *commit* de duas fases, mas tem a desvantagem de adicionar *rounds* de comunicação (i.e., requer mais processamento). Outra desvantagem é que esse protocolo usa um modelo de falhas restritivo [SCH83], onde os processos falham por *crash*. Adicionalmente, a detecção de falhas é suposta precisa e imediata.

Protocolos implementados com conceitos de comunicação de grupo possuem vantagens [VAY98, p.67] quando comparados aos protocolos de *commit*. Comunicação de grupo permite tratar um modelo menos restrito de falhas e implementa protocolos não–bloqueantes.

A abstração de grupos é normalmente usada para tratar replicação em sistemas distribuídos, onde as operações são solicitadas por troca de mensagens. Entretanto, essa abstração pode ser estendida para comportar servidores de bancos de dados e conceitos transacionais [HOL99, PED99]. A vantagem de se usar abstração de grupos em sistemas de banco de dados é que essa abstração permite um modelo de falhas mais abrangente, incluindo falhas em múltiplos membros [VAY98]. Membros suspeitos de falha são automaticamente detectados e excluídos na nova visão do grupo.

Uma diferença de replicação em sistemas distribuídos e em bancos de dados é que sistemas de banco de dados aceitam protocolos bloqueantes, onde a falha de um processo pode impedir a finalização do protocolo. Sistemas distribuídos tradicionalmente usam protocolos não–bloqueantes, o que possibilita estabelecer uma diferença fundamental entre replicação em sistemas distribuídos e em sistemas de banco de dados. A especificação da replicação pode ser decomposta em duas propriedades [LAM77]:

- sobrevivência (*liveness*): nem sempre tudo está errado,
- segurança (*safety*): alguma coisa boa mais cedo ou mais tarde acontece.

As propriedades transacionais ACID asseguram *segurança* a sistemas de banco de dados com réplicas: a falha de uma réplica resulta em um estado seguro através da propriedade de atomicidade – tudo ou nada, i.e., não há inconsistências. Protocolos de sistemas de banco de dados não tratam *sobrevivência* como parte do protocolo. *Sobrevivência* não é assegurada porque sistemas de banco de dados aceitam protocolos bloqueantes. Protocolos bloqueantes podem admitir, em alguns casos, a intervenção do usuário. Por exemplo, quando um servidor falhar, o usuário pode intervir para que o sistema bloqueado prossiga. Esse procedimento não é usado em sistemas distribuídos, onde a substituição de uma réplica por outra pode ser integrada ao protocolo evitando o bloqueio do sistema.

Outra diferença fundamental de replicação em sistemas distribuídos e em bancos de dados é que sistemas distribuídos distinguem determinismo de não–determinismo [WIE2000]. Determinismo em replicação assume que para as mesmas operações na mesma ordem, diferentes réplicas produzirão o mesmo estado. O determinismo é difícil de ser aplicado a sistemas de banco de dados replicados. Bancos de dados admitem que uma transação que atualiza um objeto deve considerar também transações concorrentes executadas nas réplicas desse objeto.

Uma outra diferença entre replicação em sistemas distribuídos e replicação em banco de dados pode ser descrita no que corresponde ao estado do grupo de replicação. Sistemas de bancos de dados normalmente consideram réplicas com estado persistente (armazenados em disco). Réplicas em sistemas distribuídos são tradicionalmente modeladas por estados voláteis e, quando são descritas por estado persistentes, são

modeladas por arquivos ou por objetos onde o estado é relativamente pequeno, se comparado a um banco de dados. Sendo assim, a manutenção da consistência das réplicas é normalmente mais simples de ser implementada em sistemas distribuídos que em sistemas de banco de dados.

3.5 Transações e dependências entre transações

Uma transação é uma seqüência de operações sobre itens de dados que satisfaz as propriedades ACID (atomicidade, consistência, isolamento e durabilidade) [GRA93] (fig. 3.7).

FIGURA 3.7 – Estrutura de uma transação

As operações nessa seqüência podem ser classificadas em dois tipos: leituras (*read*) e escritas (*write*), e são limitadas por uma operação que começa a transação (*begin*) e outra que finaliza ou confirma (*commit*) essa transação.

Se a transação não pode executar na sua totalidade, i.e., se alguma das operações contidas em uma transação não pode ser realizada, a transação é abortada. *Query* ou *read–only transaction* ou ainda *transação de leitura* é uma transação que não contém nenhuma operação de atualização. Uma transação de atualização é uma transação que contém pelo menos uma operação de atualização.

Transações de leitura são resolvidas localmente. Todas as *transações de atualização* sobre réplicas possivelmente resultarão em uma *transação distribuída*. Uma *transação distribuída* pode ser gerada por qualquer um dos *n* servidores e atualiza atomicamente todas as réplicas de um item de dado, i.e., ela atualiza o estado distribuído.

O modelo de sistema, no contexto deste trabalho, é um *grupo de replicação* formado por *n* servidores de banco de dados que acessam *n* bases de dados locais. Cada instância de servidor armazena e atualiza dados em sua base de dados local, idêntica as demais bases do grupo. Um membro do *grupo de replicação* recebe uma transação de um cliente e executa essa transação localmente. Se a transação é de **atualização**, esse membro, chamado de *primário*, difunde as atualizações da transação ao grupo de replicação quando a transação é confirmada.

Estados de transações

Uma transação em execução sobre um sistema em um *grupo de replicação* assume diferentes estados [SIL93] mostrados na fig. 3.8, extraída de Pedone *et al*. [PED99]. A transação começa no estado *executando*. Nesse estado, as operações (de leitura ou de escrita) da transação são executadas localmente (no nodo que funciona como *primário* para o cliente que requisitou a transação).

Quando o cliente requer o *commit* de uma transação, a transação passa do estado *executando* para o estado *confirmando.* Nesse estado, se a transação envolve operações de escrita, ela é propagada aos *secundários* em forma de uma *transação distribuída*.

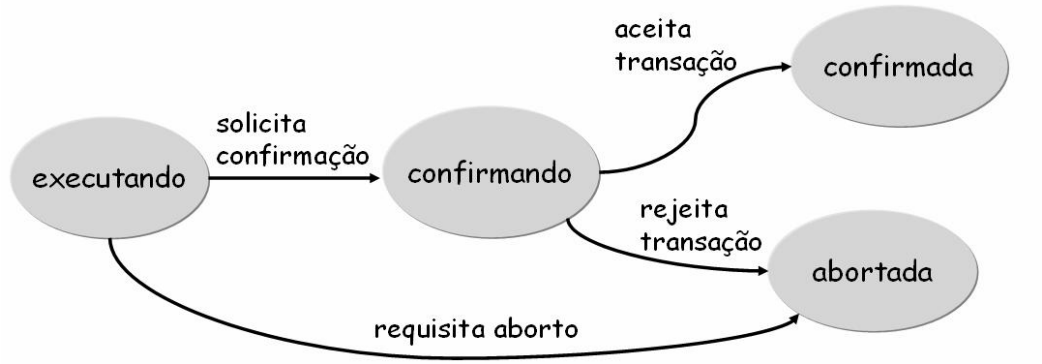

FIGURA 3.8 – Estados de transações [PED99]

Uma transação que é recebida por um *secundário* também está no estado *confirmando*. Se a transação é de *leitura*, então ela não é propagada aos demais nodos. Uma transação no estado *confirmando* permanece nesse estado até que seja confirmada ou abortada (respectivos estados *confirmada* e *abortada*).

Dependências entre transações em bancos de dados

As dependências entre transações representam um importante problema em sistemas de bancos de dados. Tratar dependências entre transações requer *detectar e impedir situações de execução de transações que gerem conflitos* (ou estado inconsistente) no estado das réplicas. Duas ou mais operações executadas por diferentes transações estão em conflito se ambas acessam concorrentemente o mesmo objeto e pelo menos uma delas executa uma operação de atualização.

Tratamento de conflitos de transações é um aspecto importante a ser considerado no projeto de um banco de dados replicado. Isso porque um cliente c_1 pode acessar um dado no banco de dados do nodo s_i, enquanto que outro cliente c_2 pode estar, simultaneamente, acessando o mesmo dado (replicado) no banco de dados do nodo s_j. Se pelo menos uma das duas operações de clientes for uma atualização, conflitos poderão ocorrer no sistema de banco de dados replicado.

A execução de transações requisitadas por dois clientes concorrentes pode causar conflitos no estado do banco de dados por compartilharem o mesmo dado (ainda que o banco de dados seja centralizado). Basta que uma das transações contenha uma operação de atualização. O sistema de banco de dados precisa sincronizar as transações localmente, usando um mecanismo como o *two–phase locking* (2PL) [ESW76]. O 2PL garante que clientes concorrentes não acessam nem alteram os mesmos dados.

Definição de transação em conflito

Para que um servidor s_i confirme localmente uma *transação distribuída* t_a (i.e., para que uma *transação distribuída* passe do estado *confirmando* para *confirmada*), o servidor s_i precisa verificar se a *transação distribuída* t_a não está em conflito com as demais transações executadas sobre as demais réplicas.

Segundo Pedone [PED99], uma transação t_b está em conflito com uma transação t_a, se t_a e t_b têm operações em conflito e se t_b não precede t_a. Duas operações estão em conflito se elas atuam sobre o mesmo item de dado e pelo menos uma delas atualiza esse item. Observe que t_a representa *uma* transação distribuída a ser confirmada no banco de dados

e t_b representa *qualquer* transação (local ou distribuída) que possa estar sendo executada no grupo de replicação.

A noção de conflito é definida pela relação *precede* ($\rightarrow$) entre as transações e as operações emitidas pelas transações. Se ambas transações estão no mesmo nodo, então $t_b \rightarrow t_a$ se t_b entra no estado *confirmando* antes de t_a. Se t_a e t_b estão em nodos distintos, então $t_b \rightarrow t_a$, se t_b confirma antes que t_a entra no estado *confirmando*.

A relação T_b não $\rightarrow t_a$ significa que T_b foi confirmada depois que t_a começou sua execução. Baseado nessas definições, diz–se que a transação T_b conflita com t_a se T_b não $\rightarrow t_a$ e se t_a e T_b têm operações conflitantes. Note que a notação T_i identifica uma transação *i* que já foi confirmada no grupo de replicação, enquanto que t_i identifica uma transação *i* que ainda não foi confirmada.

3.6 Taxonomia de replicação em sistemas de bancos de dados e sistemas distribuídos

Replicação em banco de dados é amplamente tratada na literatura [BER87, SIL93]. Gray *et al.* [GRA96] organizou um estudo sobre replicação em sistemas banco de dados usando dois parâmetros: *consistência do banco de dados* e *execução do serviço para os clientes*. No contexto deste trabalho, esse estudo foi ampliado para comportar outros parâmetros que podem ser aplicados em sistemas de banco de dados e em sistemas distribuídos. Por exemplo, quanto à *atualização das réplicas* [BER87].

Consistência de estado

O parâmetro *consistência de estado* é tipicamente usado por protocolos de replicação de banco de dados. Quanto à consistência de estado do grupo de replicação, duas técnicas são possíveis, de acordo com o instante no qual a resposta é enviada ao cliente [GRA96]:

- **replicação preguiçosa** (ou *lazy replication*): o servidor responde imediatamente ao cliente, sem garantir que a atualização foi completada com sucesso em todas as réplicas. Essa técnica também é conhecida como consistência fraca;
- **replicação ávida** (ou *eager replication*): o servidor responde ao cliente somente depois de saber se o estado de todas as réplicas está consistente. Essa técnica também é conhecida como consistência forte.

A *replicação preguiçosa* visa desempenho enquanto que a *replicação ávida* visa consistência. Na replicação ávida, o servidor espera bloqueado até receber uma resposta de todos os membros do grupo de replicação, que não estão em suspeita de falha, antes de enviar a resposta final ao cliente.

Execução do serviço do cliente

Quanto à execução do serviço do cliente [GRA96], duas técnicas são possíveis: *cópia primária* (estratégia centralizada) e *qualquer réplica* (estratégia distribuída). Essas técnicas já foram anteriormente enfocadas como protocolos de replicação por cópia primária e réplicas ativas, respectivamente.

Na cópia primária, apenas um único servidor primário executa o serviço para um determinado cliente, que desconhece a existência dos demais servidores. No protocolo de réplicas ativas, todos os servidores (*update everywhere*) executam o serviço para um determinado cliente, embora o preço para a coordenação desses servidores seja significativo.

Técnica de atualização

O parâmetro *técnica de atualização* é tipicamente usado por protocolos de replicação de banco de dados. Quanto à atualização das réplicas (quando propagar a escrita para as demais réplicas?) [BER87, p.270–271], duas técnicas são possíveis:

- **atualização imediata** (*immediate update*): propaga as escritas de uma transação a cada atualização no servidor primário. Cada operação de escrita em uma *transação de atualização* gera uma *transação distribuída*;
- **atualização adiada** (*deferred update*): propaga as escritas de uma transação a cada operação de *commit* no servidor primário. Com essa técnica, cada *transação de atualização* gera uma *transação distribuída*.

A principal desvantagem da **atualização imediata** é que ela corrompe a propriedade de *isolamento* ou *independência* de uma transação. A propriedade de *isolamento* prevê que os resultados obtidos por uma transação somente serão utilizados, por esta mesma transação, até o encerramento e confirmação destes resultados, e serão disponibilizados para uso externo à transação em questão apenas no caso de confirmação desses resultados. Sendo assim, com a técnica de **atualização imediata**, o estado do grupo de replicação pode até ser consistente, mas a propriedade de *isolamento* não será assegurada. O problema principal acontece quando por algum motivo uma transação distribuída precisa ser *abortada* e os dados alterados por essa transação precisam retornar ao estado inicial.

Em caso de *aborto de transação,* quando uma transação distribuída precisa ser desfeita e restabelecer o estado anterior aos dados alterados, todos os membros do grupo de replicação serão obrigados a recuperar o estado anterior dos dados manipulados pela transação distribuída, comprometendo o desempenho do serviço computacional.

Uma outra desvantagem da técnica da atualização imediata é que a ocorrência de conflitos pode acontecer em casos extremos. Considere uma transação local t_c que lê o valor de um item de dado no banco de dados de um dos membros do grupo de replicação. Esse valor é válido. Entretanto, imediatamente após a leitura do valor por t_c, uma transação distribuída t_d atualiza o valor desse item de dado. A transação t_c, então, faz qualquer computação com o dado anteriormente lido e atualiza o dado no banco de dados, em uma transação distribuída. O resultado será que o valor atualizado por t_d será sobreposto pelo valor gravado por t_c. Moral da estória: as duas transações estavam em conflito e não poderiam ter sido executadas entrelaçadas. A conclusão é que a atualização imediata também requer tratar conflito entre transações.

Difusão de escritas

A difusão de escritas consiste em um procedimento que difunde um estado ou uma operação de atualização de uma réplica primária para as demais réplicas. O parâmetro

difusão de escritas é tipicamente usado por protocolos de replicação de bancos de dados.

A difusão de escritas pode ser de dois tipos: adiada (*deferred propagation*) e imediata (*immediate propagation*). A difusão adiada atualiza localmente uma das réplicas e, então, executa uma operação distribuída para atualizar o estado das demais réplicas. Na **difusão imediata**, primeiro é executada uma operação distribuída para atualizar o estado das réplicas secundárias e, então, o serviço é confirmado localmente.

A. Difusão adiada

A **difusão adiada** confirma cada transação localmente (no nodo origem ou servidor primário) e depois propaga a atualização gerada pela transação aos demais membros do grupo de replicação. Uma primitiva que garanta *multicast* confiável pode ser usada para difundir as escritas aos demais membros de um grupo de replicação.

Esse cenário é possível tanto com o servidor primário quanto para qualquer réplica executando o serviço. A partir de uma requisição do cliente (1 na figura 3.9), o primário verifica se a transação altera o estado do seu banco de dados local (2). Se isso ocorre, a requisição é difundida pelo emissor (i.e., pelo primário ou por qualquer réplica) através de *multicast* confiável (3) para o grupo de replicação. Cada nodo no grupo executa o comando SQL localmente ou algum outro procedimento para garantir consistência do estado distribuído (4).

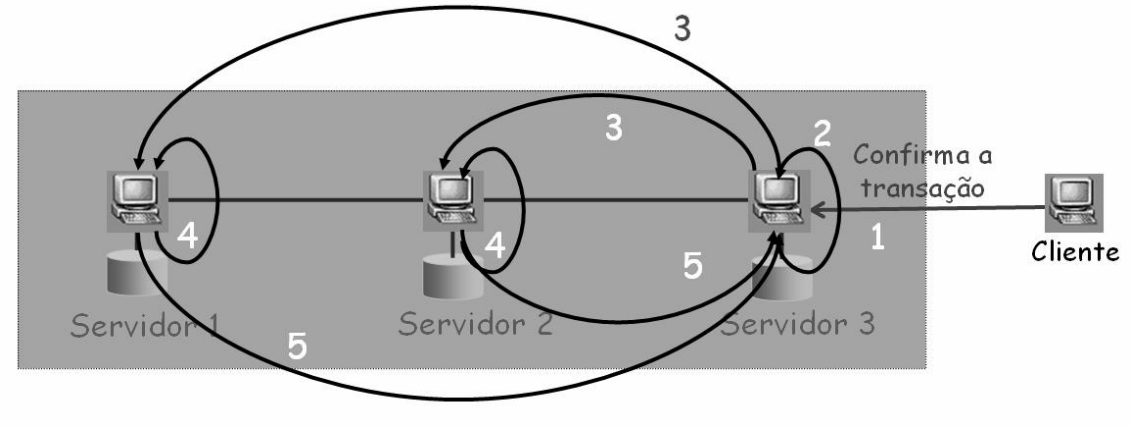

FIGURA 3.9 – Fluxo de mensagens para a difusão adiada

Cada servidor do grupo pode, então, responder ao servidor emissor se a transação foi executada com sucesso (5) ou se a transação foi abortada. Se for usada a replicação ávida, o emissor aguarda até receber todas as confirmações para a visão atual do grupo (figura 3.9) ou a maioria das confirmações. Como o servidor emissor considera a visão atual do grupo, ele descarta as confirmações dos servidores secundários suspeitos de falha. Outra opção é usar a replicação preguiçosa onde o servidor emissor assume que ao enviar uma mensagem através de uma primitiva de *multicast* confiável, cada *membro correto* do grupo de replicação irá processar corretamente a mensagem e gerará um estado distribuído consistente. Essa estratégia é usada por Pedone *et al*. [PED98] e será vista mais adiante.

As principais vantagens da difusão adiada são a eficiência, pois o emissor (i.e., o servidor primário) nunca espera[15], e a simplicidade de implementação. A principal desvantagem é que a difusão adiada requer um mecanismo para tratar inconsistências que podem ocorrer quando duas ou mais réplicas atualizam o mesmo objeto. O mecanismo deve detectar o conflito de acesso ao mesmo objeto e escolher qual transação será abortada. Soluções para tratamento do conflito de transações serão tratadas no item 3.7.

B. Difusão imediata

A **difusão imediata** propaga escritas de uma transação para um grupo antes que a transação seja localmente *confirmada*. Inicialmente, o cliente contata um servidor primário (1 na figura 3.10). Ao receber a mensagem com a requisição de replicação, o primário envia essa requisição ao grupo de replicação (2). Ao receber essa mensagem, cada réplica executa a transação (localmente) no seu banco de dados (3). Então, cada réplica pode responder ao primário (4), que finalmente atualiza o seu banco de dados local (5). Como na difusão adiada, o servidor primário pode usar ou a replicação ávida ou a replicação preguiçosa.

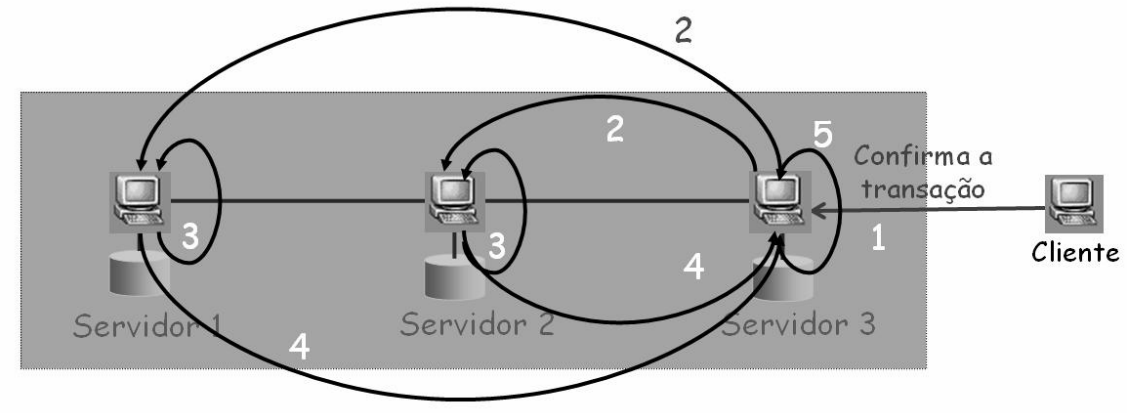

FIGURA 3.10 – Fluxo de mensagens para a difusão imediata

A desvantagem da difusão imediata é que o emissor de uma mensagem bloqueia até que ele receba uma cópia da mensagem por ele mesmo difundida. Segundo Bela Ban [BAN2002], na prática isso não representa um problema, pois um *loop* local na camada IP deverá resultar em quase imediata recepção de sua própria mensagem, que pode ter sido difundida por *multicast*. Usar primitivas de comunicação de grupo para controlar a ordem da execução das transações é uma estratégia interessante, e pode evitar inconsistência no grupo de replicação quando acesso concorrente é permitido.

A principal desvantagem da difusão imediata é o baixo desempenho em sistemas onde as atualizações são freqüentes porque, na prática, um dos membros do grupo de replicação precisa ordenar todas as transações.

[15] se for usada a técnica *replicação preguiçosa*, como é o caso deste exemplo. Uma outra opção é fazer o emissor esperar por todas as confirmações de cada um dos membros do grupo de replicação – *replicação ávida.*

Resposta ao cliente

Quanto à resposta ao cliente (quem responde ao cliente), duas técnicas podem ser usadas: *apenas um servidor responde ao cliente* e *todos os servidores respondem ao cliente*.

Quando apenas um servidor responde ao cliente, mas há suspeita de falha[16] nesse servidor, o cliente refaz a última requisição a um servidor alternativo ou um serviço intermediário é encarregado de refazer a requisição do cliente.

Quando todos os servidores do grupo respondem ao cliente, o cliente aceita apenas a primeira resposta recebida e descarta as demais. A vantagem é que a ocorrência de falhas em servidor é invisível para o cliente e que o cliente nunca refaz uma requisição. A desvantagem é o aumento de tráfego de mensagens na rede.

Sumário da taxonomia de replicação

A taxonomia da replicação é sumarizada na figura 3.11. Observe que os parâmetros *protocolo de replicação, execução do serviço do cliente* e *resposta ao cliente* são típicos de sistemas distribuídos, enquanto os parâmetros *consistência de banco de dados, técnicas de atualização* e *difusão de escritas* são típicos de sistemas de banco de dados. Na prática, um protocolo de replicação pode implementar uma combinação desses parâmetros, como mostra o capítulo 5 (ver quadro comparativo da figura 5.6).

Protocolo de replicação	Replicação ativa	Cópia primária
Consistência em banco de dados	Replicação ávida	Replicação preguiçosa
Execução do serviço do cliente	Cópia primária	Qualquer réplica
Técnica de atualização	Atualização adiada a cada *commit*	Atualização imediata a cada *write*
Difusão de escritas	Difusão adiada	Difusão imediata
Resposta ao cliente	Apenas um nodo responde ao cliente	Todos os nodos respondem ao cliente

FIGURA 3.11 – Sumário da taxonomia de replicação em bancos de dados e sistemas distribuídos

3.7 Tratando dependências entre transações em bancos de dados com réplicas usando comunicação de grupo

Na literatura podem ser encontradas diferentes estratégias para tratar dependências entre transações. Esse assunto é tipicamente abordado em sistemas de bancos de dados convencionais, uma vez que *situações de execução de transações que gerem conflitos*

[16] modelo do sistema assíncrono

podem ocorrer tanto em sistemas de banco de dados centralizados quanto em sistemas de banco de dados distribuídos (e inclusive em sistemas replicados).

Há diferentes estratégias para detectar e impedir situações de execução de *transações que gerem conflitos em sistemas replicados*. Algumas estratégias são mais simples de implementar, como usar o *primário* como ponto centralizador do serviço dos clientes, enquanto que outras são mais complexas, porém mais eficientes como a estratégia proposta por Pedone *et al.* [PED98]. Diferentes estratégias serão descritas a seguir.

3.7.1 Atualizações apenas no *primário*

Nesta estratégia, *transações distribuídas* somente podem ser coordenadas pelo *servidor primário* [ALS76, BUD93, GUE97, JAL94]. *Transações de leitura* são permitidas em qualquer réplica, embora admita–se que o estado da réplica possa estar inconsistente. Os objetos são replicados apenas no sentido do *primário* para os *secundários*. Pode acontecer que um objeto já foi atualizado no *primário*, mas ainda não foi replicado (nos *secundários*). Desde que atualizações são feitas somente no *primário*, não haverá conflito causado na replicação. Os *secundários* servem como um repositório *on–line* (ou *warm standby*) e podem assumir o serviço quando o *servidor primário* falhar.

Apesar de ser simples, essa estratégia é ineficiente na medida que o *servidor primário* funciona como um gargalo para as *transações distribuídas*. Também não há garantia de que um cliente que acesse um *secundário* obtenha a informação mais recente do banco de dados, como na replicação assíncrona.

Se essa estratégia for combinada com a *difusão adiada*, pode ser útil agrupar várias transações já confirmadas pelo primário, em uma única transação, e enviá–la ao grupo de replicação em intervalos fixos de tempo [ALS76, JAL94 p.263]. Por exemplo, pode ser possível propagar as atualizações do *primário* para os *secundários* em um intervalo fixo de tempo Δt. Neste caso, quando o *primário* falhar, pode ocorrer n segundos (onde n é o valor de Δt), no pior caso, de dados perdidos. Entretanto, se tal perda não pode ser admitida, outra solução deve ser usada.

Um exemplo prático é o caso de uma pequena empresa onde atualizações do banco de dados não são freqüentes e podem ser reconstruídas a partir do papel (notas fiscais, por exemplo). Se a empresa possui filiais espalhadas no país, então o banco de dados central pode ser replicado em cada uma das filiais usando conexões *dial–up*. Os dados podem ser replicados a cada final de cada dia operacional. Todas as atualizações podem ser mantidas localmente em um *log* e, então, enviadas às filiais através de difusão adiada.

Um exemplo de uma implementação onde transações distribuídas são coordenadas por um servidor primário é o Pronto [PED2000]. O Pronto é um protocolo que usa a difusão adiada para implementar *failover* em bancos de dados que são COTS. O não–determinismo dos bancos de dados replicados é tratado com o protocolo de cópia primária onde um servidor primário executa as transações, e envia aos secundários as informações que permitam que eles tomem as mesmas decisões não–determinísticas que o primário. Enviar as transações ao invés das atualizações ou *logs* das transações permite utilizar diferentes bancos de dados no grupo de replicação.

O Pronto é baseado em *jobs*, i.e., lógica específica com um conjunto de comandos SQL, e não em operações de leitura e escrita. Usar *jobs* permite transparência e mascaramento de falhas para o cliente. Se um comando SQL é executado, e ocorre uma falha, o comando talvez não possa ser reexecutado. Por outro lado, a lógica específica pode

selecionar outros comandos SQL baseado em resultados não-determinísticos retornados por um comando anterior.

3.7.2 Atualizações de dados sem conflito

Esta estratégia considera que qualquer servidor (*primário* ou *secundário*) pode receber e executar requisições de clientes [BAN2002]. Não haverá conflito causado por acesso simultâneo aos dados replicados porque o acesso aos dados é particionado nos servidores para que os dados não sejam sobrescritos.

Como mostra o exemplo da figura 3.12, o servidor s_1 armazena os objetos A e C, enquanto que o servidor s_2 armazena os objetos B e D, sendo que os conjuntos {A, C} e {B, D} são disjuntos. Todo acesso aos objetos A e C é endereçado a s_1, que replica os objetos A e C em s_2. Da mesma forma, todos os acessos aos objetos B e D são endereçados a s_2, que replica B e D em s_1.

Se s_1 falhar, os objetos A e C podem ser acessados através de s_2, que passa a servir como repositório principal para o conjunto {A, B, C, D}. Quando s_1 recuperar, ele pode operar como *secundário* para o novo conjunto de replicação {A, B, C, D}, que é mantido em s_2 (ou *novo primário*). Entretanto, s_1 precisa realizar a *transferência de estado* do conjunto {A, B, C, D}. A transferência de estado é um procedimento que atualiza o estado das réplicas de um nodo.

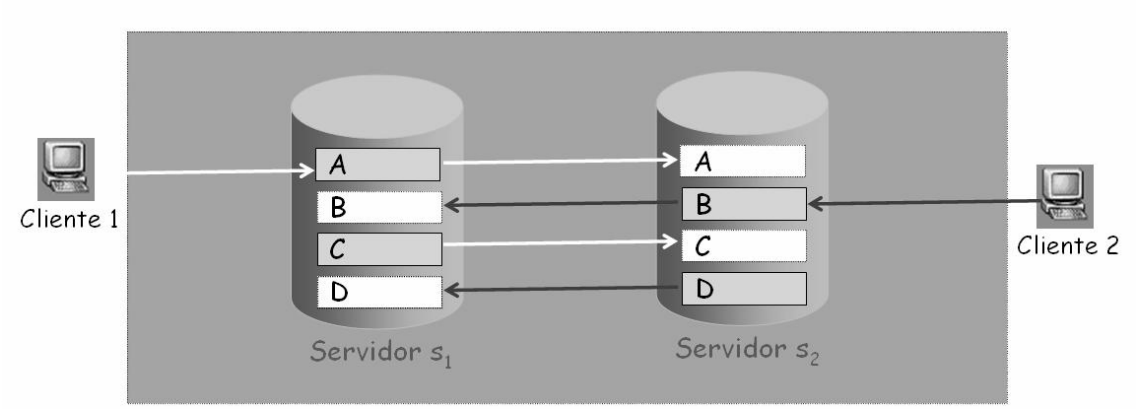

FIGURA 3.12 – Clientes acessam diferentes dados em diferentes servidores [BAN2002]

A desvantagem de usar servidores com conjunto de dados disjuntos é que algum mecanismo precisa garantir que a requisição do cliente alcance sempre o dado no servidor adequado. Um serviço de resolução de nomes deve oferecer esse mecanismo.

3.7.3 Atualizações de dados com conflito

Esta estratégia permite acesso concorrente a um dado replicado em diferentes nodos, mas não garante que conflitos não ocorram. De alguma forma, essa estratégia é otimista: conflitos podem ocorrer, mas supõe-se que os conflitos não prejudicam o serviço do sistema distribuído. A vantagem é a simplicidade de implementação e a rapidez do serviço, uma vez que mecanismos como o bloqueio distribuído (que será descrito no item 3.7.5) não precisam ser implementados.

3.7.4 Atualizações de dados com ordenação total

Esta estratégia [BAN2002, PED98] baseia–se na idéia de que ao estabelecer uma ordem total para que as atualizações sejam executadas em todos os membros de um grupo, pode–se garantir que cada membro processará a mesma seqüência de transações na mesma ordem. A ordem para a execução das atualizações pode ser garantida através de uma primitiva que garanta ordem e atomicidade, como a TOCAST.

A desvantagem desta estratégia é que a primitiva TOCAST (implementada através de um sistema de comunicação de grupo) é bem onerosa, se comparado com uma primitiva que não precisa de ordem total na entrega das mensagens.

Por exemplo [BAN2002], considere uma ocorrência na tabela *fornecedor* com chave primária ID=32 que é adicionada ao banco de dados de um servidor s_2 (fig. 3.13) e quase simultaneamente ao banco de dados de um servidor s_3. A transação em s_2 será chamada de t_a e a transação em s_3, de t_b. Considere também que cada dado associado com o fornecedor não é exatamente o mesmo: t_a tem valor para o telefone enquanto que t_b não tem telefone.

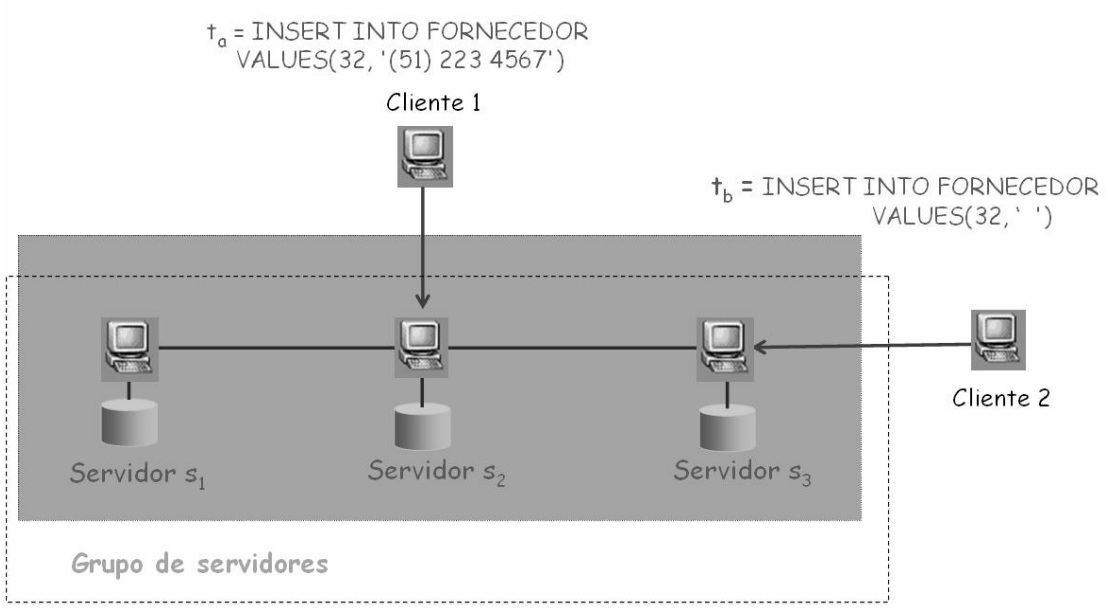

FIGURA 3.13 – Cenário inicial para inserção de novos valores na tabela *fornecedor*

A. Difusão adiada

Se for usada a difusão adiada, s_2 executa t_a (que contém o número de telefone) e replica a atualização para s_3. Então, s_3 executa t_b (a transação sem número de telefone), e então dispara a replicação no nodo s_2. Quando s_3 recebe t_a replicado de s_2, ocorrerá um conflito pois o *fornecedor* com chave ID=32 já está no banco de dados do nodo de s_2. A mensagem replicada de t_a será descartada em s_3. Igualmente, s_2 descarta a mensagem replicada t_b, porque a chave primária é duplicada. O cenário final apresenta ambas réplicas com a identificação do fornecedor com chave ID=32, mas com dados ligeiramente diferentes associados a elas. Dependendo de como a réplica é acessada, podem ser percebidos os diferentes dados no sistema distribuído (figura 3.14). O nodo s_1 pode executar primeiro t_a ou t_b. Na figura 5.6, o nodo s_1 executou primeiro t_a.

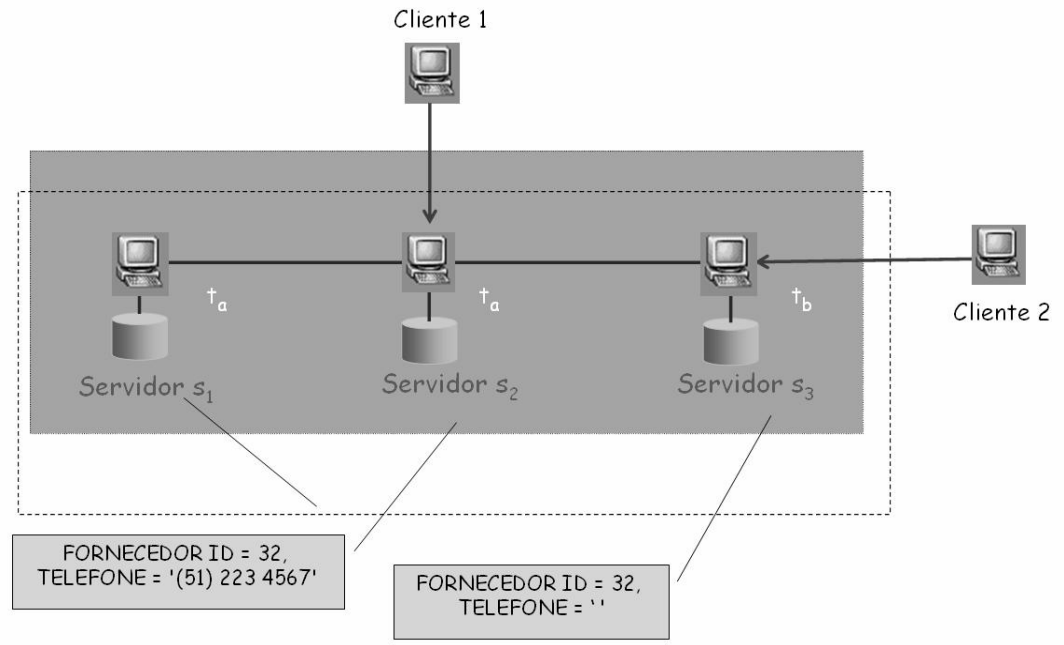

FIGURA 3.14 – Cenário possível com difusão adiada e conflito de dados

Protocolo de término

Baseado na estratégia de difusão adiada, Pedone *et al.* [PED98] propõem o *protocolo de término com ordem total.* Um protocolo similar é descrito em Holliday *et al.* [HOL99, HOL99a], como protocolo A4. O *protocolo de término com ordem total* considera transações executadas sobre nodos que implementam a abstração de máquinas de estado (ou réplicas ativas) em uma estrutura *three–tier* contendo cliente, servidor de aplicação e banco de dados.

Inicialmente, as transações são sincronizadas localmente no banco de dados de um servidor, usando o protocolo de bloqueio em duas fases ou *two–phase locking* (2PL) [ESW76]. Quando o cliente requer a confirmação de uma transação, as alterações geradas por essa transação e algumas estruturas de controle são propagadas ao grupo de replicação, onde a transação será *certificada* (ver figura 3.8) e, possivelmente, *confirmada*. Esse procedimento, que inicia na operação de *commit* de uma transação, é chamado de *protocolo de término*. O protocolo de término propaga as transações de um servidor s_i ao grupo de replicação e certifica–as, através do *teste de certificação*.

O *teste de certificação* verifica se uma transação local em s_i está em conflito com as demais transações no sistema distribuído. O teste de certificação garante o *critério de serializabilidade*: a transação é abortada se a sua confirmação gera estado inconsistente no grupo de replicação. Ou seja, o teste de certificação detecta possíveis conflitos entre transações concorrentes em membros do grupo de replicação. Se a transação passa no teste de certificação, então ela é confirmada no grupo de replicação.

Uma das vantagens da *difusão adiada,* que é usada neste protocolo, é que ela não requer qualquer mecanismo de bloqueio distribuído (ver item 3.7.5) para sincronizar transações durante sua execução. As vantagens de não usar bloqueio distribuído incluem a economia de mensagens, aliviando o tráfego pela rede, e a impossibilidade de gerar *deadlock* distribuído. Contudo, isso não evita que transações possam ser abortadas devido a acesso conflitante. O número de transações abortadas pode ser reduzido

através da *reordenação das transações* que foi posteriormente proposto por Pedone *et al.* [PED99].

O *procedimento de reordenação de transações* é baseado na observação de que a ordem na qual as transações são *confirmadas* não precisa ser a mesma ordem na qual as transações são entregues aos membros de um grupo para serem *certificadas*. A idéia é obter uma ordem que produza menos abortos. Para obter a ordem que produza menos abortos, as transações são mantidas em uma lista. A desvantagem do *procedimento de reordenação de transações* é justamente a contenção de dados que ele acrescenta ao manter os itens bloqueados na lista.

B. Difusão imediata

Se for usada a estratégia de difusão imediata mas sem a restrição de ordenação total na entrega das mensagens, ambas transações t_a e t_b são difundidas aos secundários antes de serem atualizadas no nodo origem. Desde que a ordenação total não é envolvida, s_2 pode receber t_a e depois t_b, e s_3 pode receber t_b e depois t_a. Neste caso, podem ocorrer diferenças entre os dados associados com a mesma chave primária. O cenário mostrado na figura 3.13, para a tabela *fornecedor*, também pode ser obtido nessa situação, ainda que outro cenário final seja possível.

Se a ordenação total for introduzida através da primitiva TOCAST, como no protocolo A2 de Holliday *et al*. [HOL99, HOL99a], ambos servidores s_2 e s_3 poderão receber ou t_a seguida por t_b, ou t_b seguida por t_a, mas eles não receberão t_a seguida por t_b em um servidor e t_b seguida por t_a em outro servidor (ou vice–versa). Se a operação for adicionar uma nova ocorrência de fornecedor, a transação distribuída que acontece antes adiciona uma linha na tabela com chave ID=32, enquanto que a segunda transação será descartada (fig. 3.15). Nesse caso, o cliente cuja transação foi descartada executa uma exceção quando tentar confirmar a transação. A idéia é que a ordem total garanta que o dado associado com a mesma chave primária é exatamente o mesmo.

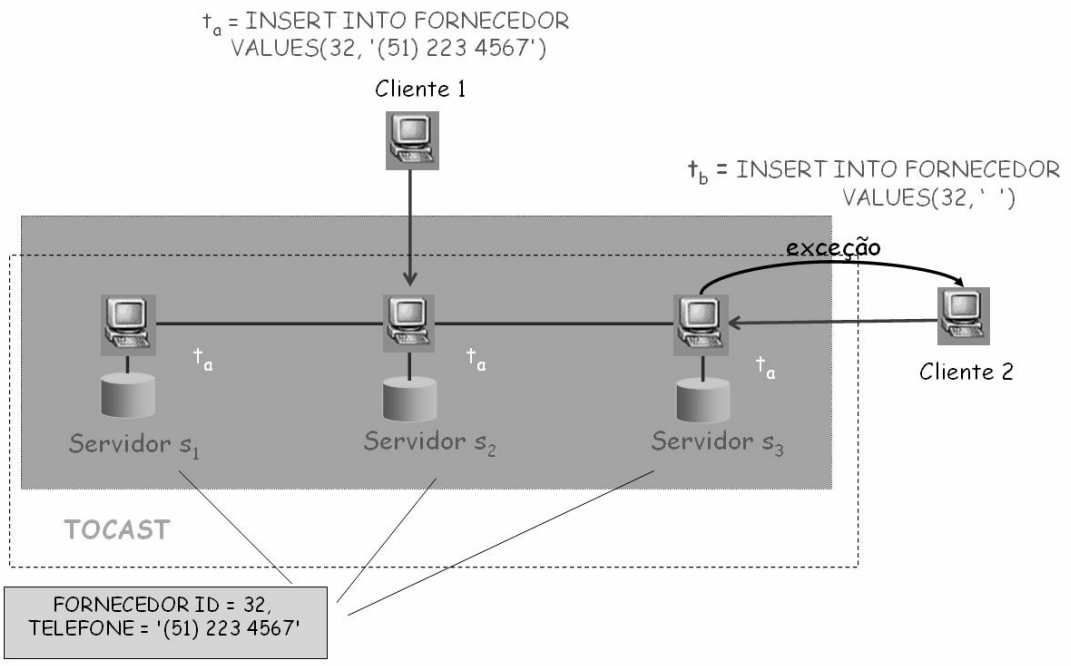

FIGURA 3.15 – Cenário possível para difusão imediata com TOCAST

Lembre–se que a principal desvantagem da difusão imediata é o baixo desempenho em sistemas onde as atualizações são freqüentes. Cada atualização em um objeto local resulta em processamento executado por todos os nodos do grupo de replicação. Neste sentido, a difusão adiada é mais eficiente e foi a estratégia usada no contexto deste trabalho, como será mostrado no próximo capítulo.

3.7.5 Atualizações com bloqueio distribuído

Esta estratégia [BAN2002] possui menor custo que a estratégia anterior pois pode ser implementada com uma primitiva de *multicast* confiável sem a restrição de ordenação total na entrega das mensagens. A estratégia aqui apresentada é similar ao protocolo A3 de Holliday *et al*. [HOL99, HOL99a].

O procedimento usado para garantir que atualizações são consistentes quando aplicadas aos membros do grupo consiste em bloquear operações de atualização nos membros do grupo de replicação antes de propagar uma mensagem, e desbloqueá–las depois que a mensagem foi processada. Entretanto, essa estratégia é ineficiente quando as operações de atualização são muito freqüentes devido ao alto tráfego de mensagens na rede para bloquear e desbloquear o grupo de replicação.

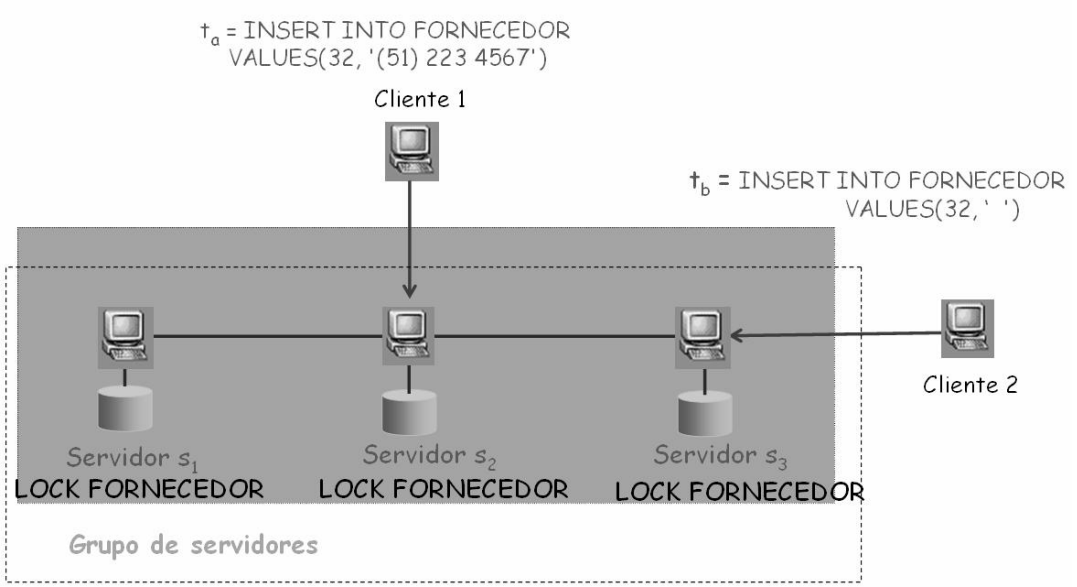

FIGURA 3.16 – Bloqueio para a tabela fornecedor pelo servidor s_2

Considere o exemplo do item anterior que insere uma nova linha na tabela *fornecedor* com chave primária ID=32, e suponha que os servidores s_2 e s_3 querem criar essa mesma linha. Esses servidores difundem simultaneamente uma mensagem para bloquear a tabela *fornecedor* no grupo de replicação. Entretanto, suponha que s_2 é ligeiramente mais rápido e consegue bloquear a tabela *fornecedor* (figura 3.16) antes que s_3.

Ao obter o bloqueio, s_2 pode prosseguir e difundir suas mensagens com operações de atualização. Todas essas operações são executadas sobre a tabela *fornecedor*. Quando s_3 tentar bloquear a tabela fornecedor, no grupo de replicação, não irá conseguir porque o bloqueio já foi processado. O servidor s_3 deverá aguardar. Quando s_2 liberar o bloqueio, s_3 poderá prosseguir sua execução, e obter o bloqueio do grupo de replicação.

Contudo, como já existe a linha com a chave primária ID=32 no grupo de replicação, as atualizações de s_3 irão falhar. Neste caso, s_3 notifica o cliente da transação que uma

falha ocorreu (desviando para uma exceção) e libera o bloqueio (figura 3.17). Note que a mensagem com a operação de atualização e a mensagem com a operação de bloqueio podem ser combinadas em apenas uma mensagem.

Esta estratégia é semelhante ao protocolo de *commit* de duas fases [GRA78], onde um coordenador comanda os demais servidores enviando a mensagem para bloquear determinado recurso. Falha no coordenador pode levar a bloqueio do sistema. Por isso, o bloqueio pode estar associado a um *timeout*, que expira automaticamente após determinado tempo (tipicamente medido em segundos). O uso de *timeouts* previne que um único servidor monopolize o bloqueio por muito tempo e evita a contenção de bloqueios por réplicas, que pode levar a ocorrência de *deadlocks*. Quando um servidor está falho (ou suspeito de falha), todos os bloqueios mantidos por esse servidor são removidos. A grande dificuldade do uso de *timeouts* é em sistemas assíncronos, onde é a escolha de um limite de tempo adequado deve evitar que nodos lentos sejam erroneamente supostos como falhos.

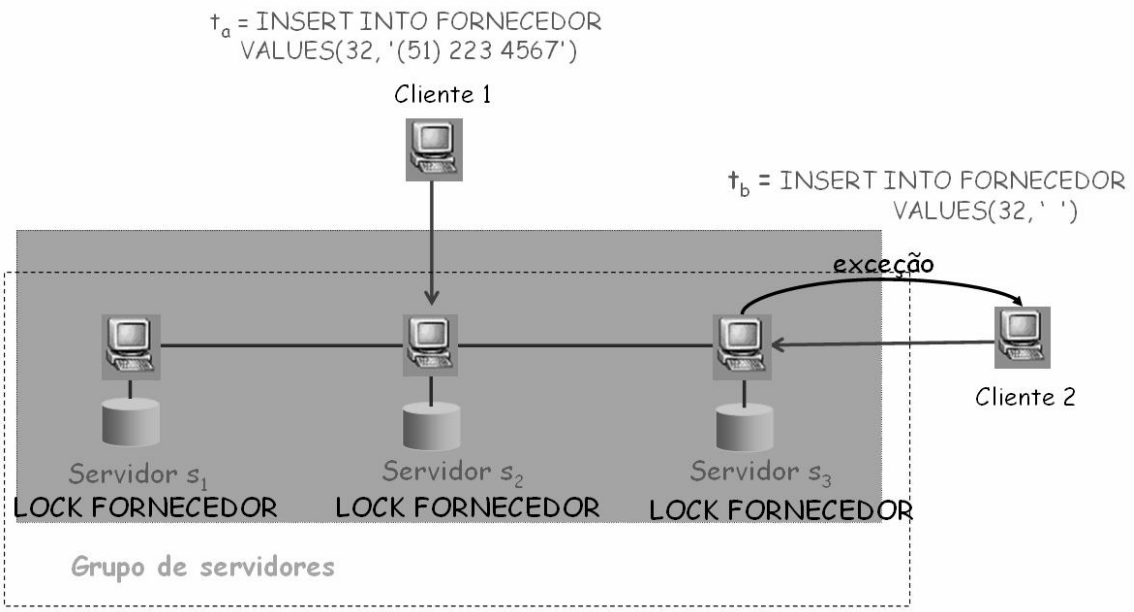

FIGURA 3.17 – Bloqueio para a tabela fornecedor pelo servidor s_3

Considere o caso onde o servidor s_2 gerencia o bloqueio na tabela *fornecedor*, e aguarda até que s_3 libere o bloqueio da tabela *produtos*. Enquanto bloqueia *produtos*, s_3 aguarda que s_2 libere seu bloqueio na tabela *fornecedor*. O resultado desse cenário pode ser *deadlock*, ao menos que um ou ambos bloqueios sejam liberados pela expiração do limite de tempo do *timeout*. Depois de um intervalo de tempo escolhido randomicamente, para reduzir a chance dos servidores tentarem requisitar o bloqueio no mesmo instante, s_2 e s_3 tentam novamente conseguir o bloqueio.

Os bloqueios podem estar associados a vários recursos. Por exemplo, a criação de uma nova tabela requer bloquear todo o grupo de replicação, enquanto a inserção ou a atualização de uma linha em uma tabela requer bloquear somente uma tabela específica. Note que, se a linha a ser bloqueada pode ser determinada, apenas essa linha da tabela pode ser bloqueada, ao invés da tabela inteira. Essa solução de menor granularidade é muito mais atraente, pois aumenta o poder de concorrência entre transações.

3.8 Observações do capítulo

Este capítulo explorou técnicas e protocolos de replicação para alta disponibilidade em sistemas de bancos de dados através de abstrações de grupos, tipicamente usadas em sistemas distribuídos. A solução mais atrativa, entre as referências consultadas, foi o *protocolo de término com ordem total* de Pedone *et al*. [PED98, PED99], por se tratar de uma solução distribuída. A distribuição é uma propriedade importante para alta disponibilidade pois garante *sobrevivência* apesar de falhas. É justamente esta solução que será abordada no próximo capítulo.

Outras soluções que exploram replicação para alta disponibilidade através de abstrações de grupos podem ser encontradas na literatura. Babaoglu e Toueg [BAB93] também usam comunicação de grupo para assegurar consistência sobre réplicas em sistemas de banco de dados. Porém, a solução apresentada é baseada em um protocolo centralizado (semelhante ao item 3.7.5), que não assegura *sobrevivência* apesar de falhas.

4 Replicação de objetos com comunicação de grupo

Esse capítulo descreve os serviços requeridos para que o *grupo de replicação* garanta a alta disponibilidade. Os membros do grupo de replicação executam um protocolo híbrido, que combina as características dos protocolos de réplicas ativas e de cópia primária anteriormente descritos. O protocolo híbrido usa como suporte um sistema de comunicação de grupo.

4.1 Modelo do sistema para arquitetura com alta disponibilidade

O modelo do sistema usando nesse capítulo, é definido a partir do modelo descrito no capítulo 2, no item 2.1. O modelo do sistema é composto por um conjunto de nodos que executam sobre um sistema distribuído, onde há a ausência de um relógio global. Os nodos são autônomos, sem memória compartilhada e se comunicam por troca de mensagens em uma rede de comunicação.

Os processos executam em um *sistema assíncrono*, sobre canais de *comunicação confiável*. Os processos podem ser clientes ou servidores. Os servidores formam um *grupo de replicação*. Cada servidor possui réplicas dos mesmos objetos e está associado a um nodo diferente. Os servidores do grupo de replicação são sincronizados através de um protocolo híbrido (que será descrito no item 4.3).

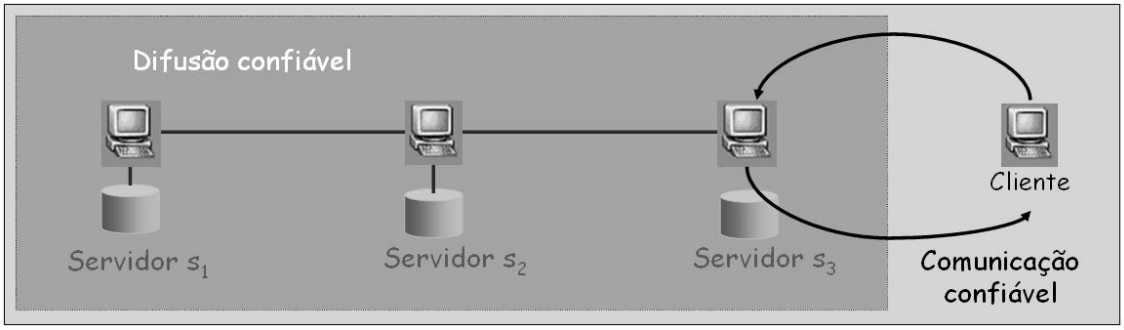

FIGURA 4.1 – Comunicação confiável entre os servidores

A comunicação cliente–servidor é síncrona[17] e ponto–a–ponto. Cada cliente pode solicitar um objeto para um dos servidores do grupo de replicação e chama esse servidor de *servidor primário*. A comunicação entre os servidores do grupo de replicação pode ser ponto–a–ponto ou ponto–a–multiponto. A comunicação ponto–a–multiponto possui as propriedades de ordem e atomicidade [GUE97] anteriormente definidas. Um sistema de comunicação de grupo [BAN99] provê suporte para difusão confiável em grupos, detecção de nodos suspeitos de falhas e o serviço de composição do grupo.

Os nodos podem falhar por *crash*. Os nodos que executam processos servidores mais cedo ou mais tarde recuperam–se após uma falha e são reinseridos no grupo de replicação. Possíveis cenários de falhas em nodos e com perda de mensagens entre cliente e servidor são tratados no item 4.4. Não são tratadas falhas de particionamento de rede.

[17] o protocolo pode ser RPC, RMI ou RMI–IIOP

Cada processo servidor representa um servidor de aplicação e um servidor de banco de dados com um banco de dados a ele conectado. Um *container* provê uma interface entre os serviços não–funcionais oferecidos pelo servidor e os serviços funcionais oferecidos pelos objetos. Os serviços não–funcionais incluem gerenciamento de memória e gerenciamento transacional para o banco de dados.

O servidor de aplicação gerencia a sessão cliente–servidor, e pode manter o estado da sessão em memória volátil. Quando a sessão termina, esse estado é perdido. O servidor de banco de dados trata da memória persistente, que armazena a base de dados. Quando a sessão cliente–servidor termina, os dados armazenados em memória persistente permanecem.

Transações são submetidas pelos clientes e são executadas pelos servidores do grupo de replicação. Entretanto, operações de leitura e de atualização de objetos, sem o uso de transações, também podem ser solicitadas pelos clientes ao grupo de replicação. Nesse caso, as operações também serão atendidas pelo servidor de aplicação, com a restrição de que elas não executam sobre um item de dado, mas sobre um objeto armazenado na memória principal.

Os objetos são modelados como componentes, e possuem interfaces bem definidas, para que possam ser independentes e manipulados facilmente pelos servidores e podem ser dos seguintes tipos:

- ***objetos sem estado***: tipicamente são usados sem transações. Eles são adequados para aplicações como calculadoras simplificadas (sem memória), onde um usuário executa cálculos e obtém um resultado que não precisa ser persistido nem compartilhado com outros usuários.
- ***objetos com estado volátil***: mantêm o estado no servidor primário apenas durante a execução do serviço, i.e, durante a sessão cliente–servidor. Clientes que usam *objetos com estado volátil* podem ou não usar o serviço transacional. Esses objetos são adequados para modelar aplicações como calculadoras com memória, mas que não são compartilhadas entre usuários.
- ***objetos com estado persistente***: são usados para manipular itens de dados em bancos de dados, por isso estão sempre associados ao uso de transações. Desde que objetos com estado persistente podem ser compartilhados por múltiplos clientes, chaves primárias são requeridas para que a consistência seja assegurada. Objetos com estado persistente são ideais para modelar aplicações como sistema de controle de estoque ou sistema de reserva e compra de passagens aéreas. Desde que múltiplos guichês podem vender bilhetes das mesmas companhias e dos mesmos vôos, enquanto uma compra é realizada em um guichê, é necessário algum mecanismo garanta que mais ninguém está lendo ou atualizando os referidos dados e tabelas em outros guichês.

Embora sejam possíveis combinações entre os tipos de objetos, elas estão fora do escopo deste trabalho. Operações aninhadas, que envolvem múltiplos objetos, não serão tratadas.

4.2 A arquitetura em múltiplas camadas

A arquitetura de alta disponibilidade ou arquitetura HA (ou *highly–available architecture*) é composta por quatro camadas (fig. 4.2): camada do usuário (ou do

cliente), camada de aplicação, camada de replicação e camada de comunicação de grupo. Cada camada implementa o serviço correspondente.

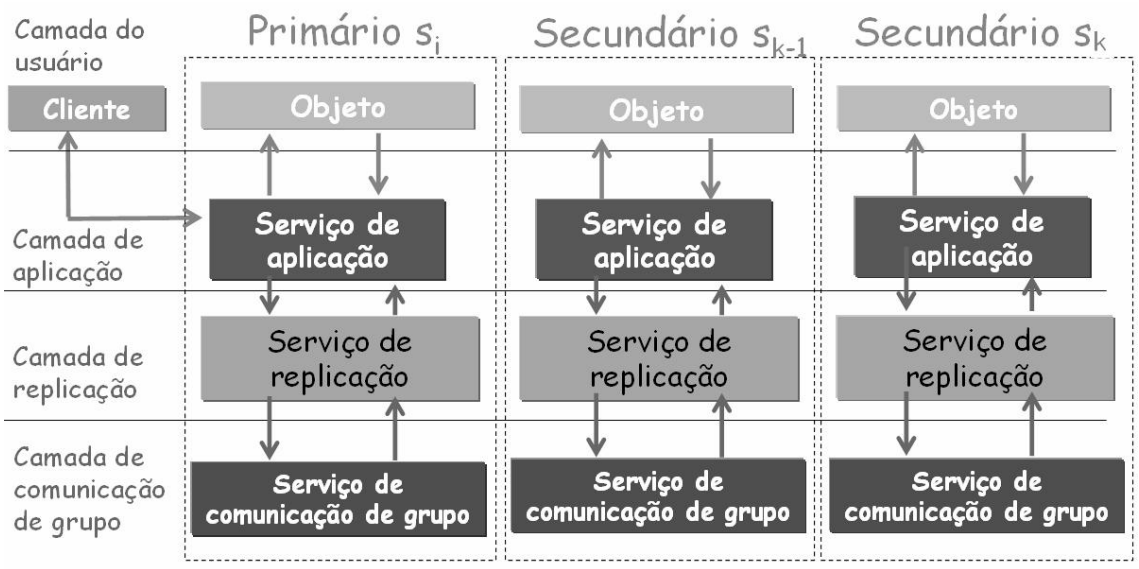

FIGURA 4.2 – Camadas para o serviço replicado

A camada de aplicação é implementada por um servidor de código aberto [OBJ2001]. O servidor de aplicação permite acesso a um banco de dados. O servidor de aplicação e o banco de dados são implementados por COTS. Sendo assim, os múltiplos bancos de dados mantidos pelo grupo de replicação podem ser construídos por diferentes fornecedores.

A camada de replicação é a camada desenvolvida neste trabalho e conecta a camada de comunicação de grupo com a camada de aplicação. A camada de replicação implementa o *serviço de replicação* através de um grupo de servidores ou *grupo de replicação*. As funções da camada de replicação consistem em estabelecer o *estado distribuído* e manter a consistência dos múltiplos bancos de dados. O estado distribuído é o conjunto de *estados locais*, que são armazenados na memória principal de cada servidor do grupo de replicação.

A camada de comunicação de grupo é implementada por um sistema de comunicação de grupo [BAN99]. O sistema de comunicação de grupo, assim como servidor de aplicação e o banco de dados, também é implementado por um COTS, e oferece o serviço de grupos, i.e, o serviço de composição de grupos e o serviço de detecção de falhas. O serviço de grupos, juntamente com o *serviço de replicação* e com o *serviço de chaveamento de servidor,* desenvolvidos neste trabalho, formam os *serviços* HA.

4.3 Protocolo híbrido

O *serviço de replicação* sincroniza réplicas do grupo de replicação de acordo com um *protocolo híbrido*. Esse protocolo combina as características dos protocolos de réplicas ativas e cópia primária. No protocolo híbrido, como no protocolo de réplicas ativas, qualquer réplica pode executar o serviço para o cliente, basta que ela receba uma requisição de um cliente. Como no protocolo de cópia primária, apenas um servidor (o primário) se comunica com o cliente.

O protocolo híbrido suporta múltiplos servidores primários, evitando o gargalo que aparece quando apenas um servidor primário é usado no protocolo de cópia primária. Os

clientes invocam suas requisições para um dos servidores primários do grupo de replicação usando comunicação ponto–a–ponto.

O servidor primário para um dado cliente é o servidor que o cliente contata através de um *serviço de nomes externo*. Os demais servidores do grupo de replicação funcionam como secundários. Uma vez que a sessão é criada em um servidor *primário*, o mesmo servidor é usado durante toda a sessão, ou até que o servidor falhe. Nesse caso, um outro servidor do grupo de replicação pode criar uma nova sessão para aquele cliente e continuar a execução do serviço usando dados replicados.

Embora o *serviço de nomes* esteja fora do escopo deste trabalho, consiste em um interessante ponto para a pesquisa para a arquitetura aqui descrita, e deve ser incluído em trabalhos futuros. É tarefa do serviço de nomes escolher qual é o servidor de aplicação mais adequado para cada cliente, com o objetivo de ofercer um serviço com desempenho eficiente.

O servidor primário atual analisa a requisição do cliente e verifica se a operação é local (operação de leitura) ou distribuída, i.e., difusão de escritas. A difusão de escritas gera um *estado distribuído* globalmente consistente para o grupo de replicação através de uma primitiva *multicast* que considere grupos dinâmicos. A manutenção do estado distribuído consistente no grupo de replicação possibilita que uma réplica falha (ou em suspeita de falha) seja substituída por outra réplica operacional.

A figura 4.3 apresenta uma possível configuração para o grupo de replicação. Para o cliente c_1, o servidor s_2 funciona como *primário*, enquanto que os servidores s_1 e s_3 funcionam como *secundários*. Analogamente, para os clientes c_2 e c_3, o servidor s_3 funciona como *primário*, enquanto que os servidores s_1 e s_2 funcionam como *secundários*. Os servidores do grupo de replicação são *multi–thread*. Para cada cliente, uma *thread* é criada na camada de aplicação quando uma sessão cliente–servidor inicia. Essa *thread* é destruída quando a sessão finaliza.

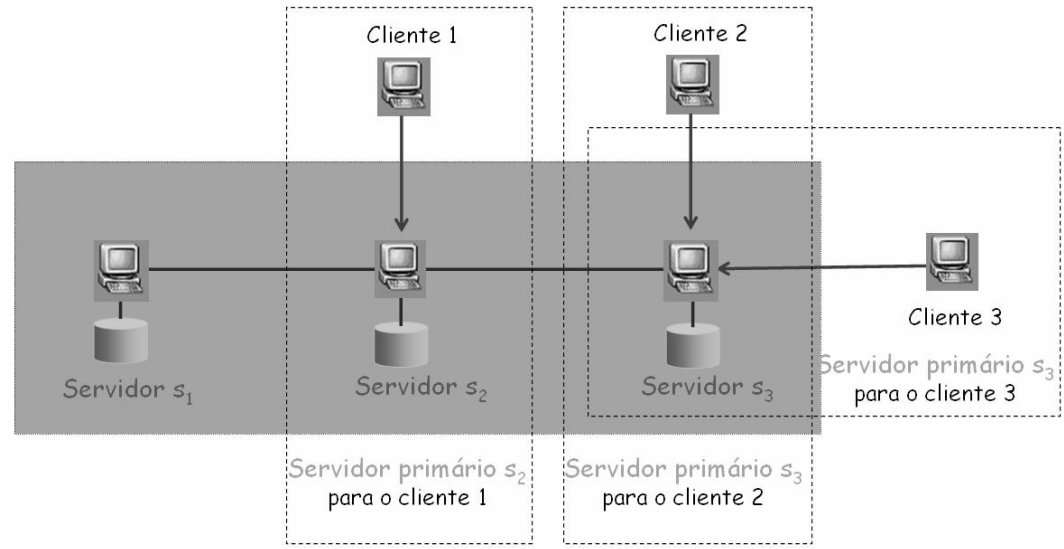

FIGURA 4.3 – Serviço com múltiplas réplicas de banco de dados

O cliente faz uma requisição para um servidor *primário* s_i e aguarda bloqueado até que ele receba uma resposta. Isso significa que mesmo que, um cliente contate apenas um servidor *primário* para solicitar um serviço, o sistema distribuído, como um todo, funciona como no protocolo de réplicas ativas.

O protocolo híbrido suporta duas variações para permitir alta disponibilidade de aplicações distribuídas: **execução com serviço distribuído** e **execução com serviço local**. A execução de serviço distribuído é usada quando as aplicações usam objetos persistentes. A execução local é adequada quando as aplicações usam objetos com memória volátil. Essas duas variações serão descritas a seguir.

4.3.1 Protocolo híbrido com serviço distribuído

A figura 4.4 mostra o protocolo híbrido com serviço distribuído. As operações de atualização de estado solicitadas por um cliente (fase 1) são recebidas pelo servidor primário s_i, e são enviadas com a primitiva TOCAST (fase 2) para a execução em todos os servidores (fase 3) do grupo de replicação. A primitiva TOCAST garante as propriedades de ordem total e de atomicidade na entrega das mensagens para todos os servidores do grupo de replicação. Apenas o servidor primário s_i responde ao cliente (fase 5).

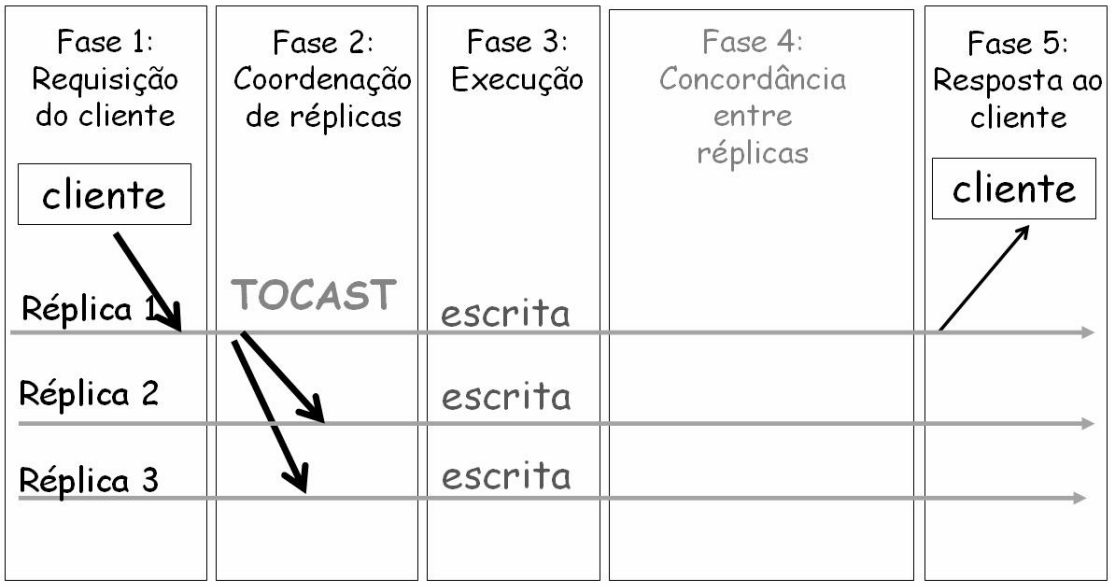

FIGURA 4.4 – Protocolo híbrido com serviço distribuído

Note que a primitiva TOCAST precisa comportar grupos dinâmicos pois o servidor primário s_i para um dado cliente pode falhar e ser excluído na próxima visão.

4.3.2 Protocolo híbrido com serviço local

A figura 4.5 mostra o protocolo híbrido com a execução do serviço local. As operações de atualização de estado solicitadas por um cliente (fase 1) são recebidas e executadas somente pelo servidor primário s_i (fase 3). Os secundários daquele cliente não processam o serviço, apenas alteram o seu estado de acordo com o resultado do serviço realizado no servidor primário s_i (fase 4). A primitiva VSCAST garante que atualizações são recebidas e processadas na mesma ordem em um grupo de replicação. Apenas o servidor primário s_i responde ao cliente (fase 5).

Note que o protocolo híbrido com serviço local é mais otimizado que o protocolo híbrido com serviço distribuído, pois o serviço local é executado somente uma vez pelo grupo de replicação (no servidor primário s_i) e a propriedade de ordem total não é

requerida, desde que esta variação do protocolo híbrido é usada por objetos com estado volátil e objetos voláteis são compartilhados entre clientes.

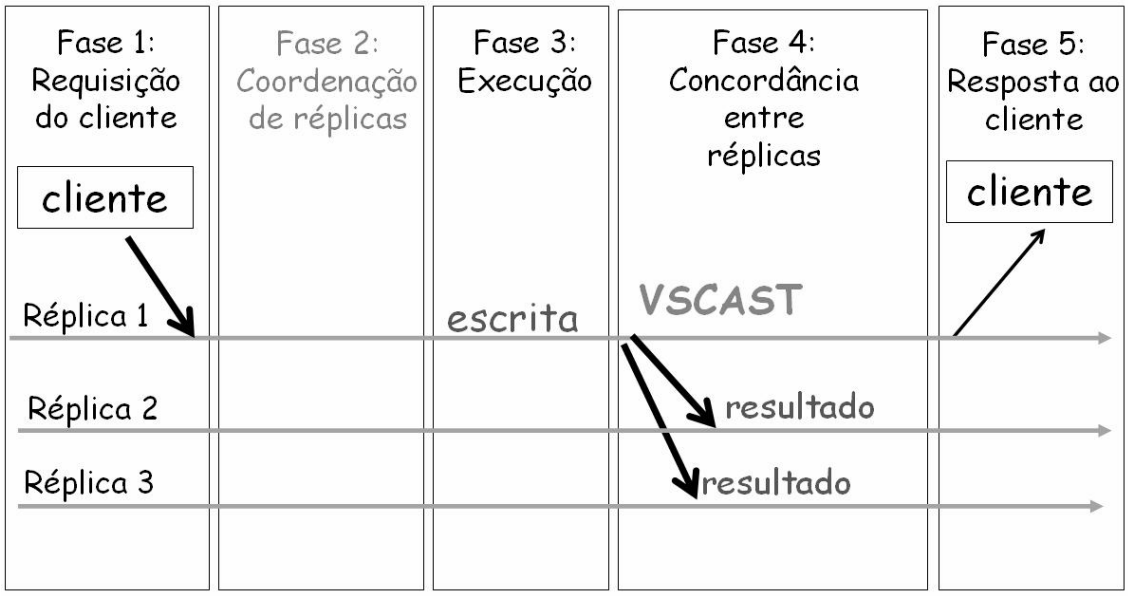

FIGURA 4.5 – Protocolo híbrido com serviço local

4.4 Alta disponibilidade em presença de falhas

Manter a transparência no serviço de replicação apesar de falhas requer tratar problemas de comunicação entre um cliente e um servidor primário s_i. Como a visão que o cliente tem do grupo de replicação é o servidor primário s_i, falhas em servidor primário são aparentes para os clientes e são mais difíceis de serem tratadas que falhas em servidores secundários.

Dois tipos de falhas são tratadas no escopo deste trabalho: *falhas de comunicação* e *falhas de crash em nodos*. Para o cliente esses dois tipos de falha são indistinguíveis. O cliente não sabe se o nodo que executa o servidor primário s_i falhou ou se o canal de comunicação entre ele e o servidor falhou. O cliente apenas percebe que o serviço não foi executado através da expiração do seu *timeout*.

Uma falha de comunicação ocorre quando um nodo operacional não consegue se comunicar com outro nodo. Uma falha de comunicação pode ocorrer porque a mensagem de requisição do cliente para um servidor primário s_i foi perdida ou porque a resposta do servidor primário s_i para o cliente foi perdida. Falhas em nodos podem ocorrer em nodos pertencentes ao grupo de replicação ou em nodos que executam clientes. Falhas em clientes não serão tratadas. Falhas em secundários são invisíveis aos clientes. Nodos em suspeita de falha são automaticamente detectados e excluídos da nova visão do grupo de replicação através do sistema de comunicação de grupo.

Quando o cliente suspeitar, por *timeout*, que o servidor está em falha, o *chaveamento de servidor* deve ocorrer. O cliente contata novamente o *servidor de nomes* para obter o endereço de um servidor alternativo. Então, o cliente refaz a última requisição para o servidor alternativo, que pertence ao grupo de replicação. Note que, se o cliente está executando uma requisição dentro de uma transação, o cliente abortada a transação e reinicializa a transação no servidor alternativo. A falha de um servidor no grupo de

replicação sempre será invisível para o usuário final desde que o grupo de replicação mantenha, no mínimo, um membro operacional.

Tipicamente, as aplicações distribuídas que não oferecem um serviço de alta disponibilidade implementam a semântica RPC [BIR84] *no máximo uma*, i.e., quando o servidor ou o banco de dados falha, o usuário recebe imediatamente uma mensagem reportando uma situação de erro. Outra opção é que o cliente espere que o servidor reinicialize e repita a última invocação. A idéia é que o usuário ou, automaticamente, o cliente tente outra vez até que a resposta seja obtida. Essa semântica é chamada no *mínimo uma* e garante que a invocação do cliente será executada no servidor *no mínimo uma*, mais possivelmente mais de uma vez. Essas não são adequadas para um serviço altamente disponível. Nenhuma dessas semânticas é atrativa para um serviço altamente disponível. Um serviço altamente disponível deve aplicar a semântica de execução *exatamente uma*.

O uso de transações em sistemas de banco de dados facilita a implementação da semântica de execução *exatamente uma* no serviço de replicação [FRØ99]. Se cliente invoca um método dentro de uma transação e, a não ser que ele falhe, a invocação é executada *exatamente uma* vez, e o resultado é entregue ao cliente imediatamente ou mais tarde. Se a requisição não pode ser executada porque o servidor falhou, a requisição é abortada e os bloqueios sobre o banco de dados são liberados.

Na prática, a semântica *exatamente uma* é difícil de ser aplicada em casos de falha de servidor. A manutenção da semântica *exatamente uma* está relacionada com *idempotência* de operações executadas no servidor e grande parte das operações são inerentemente não–idempotente [TAN92].

O tratamento de falhas no servidor primário s_i do protocolo híbrido é baseado nas estratégias descritas em Guerraoui *et al.* [GUE97] para o protocolo de cópia primária. O primário pode falhar antes, durante ou depois de propagar uma atualização ao grupo de replicação. O cliente detecta por *timeout* a falha no servidor primário, executa o chaveamento do serviço para um servidor alternativo e repete a invocação para esse servidor. Como o cliente sempre repete a última requisição, o servidor alternativo deve ser capaz de implementar a semântica de execução *no máximo uma*.

Ao receber a requisição, um servidor s_k sempre verifica se a requisição é original ou se é retransmissão. As requisições contêm, além da identificação do cliente, um número seqüencial para que uma mensagem retransmitida seja distinguida de uma requisição original. Se a requisição é uma retransmissão, o servidor sabe que ocorreu um chaveamento e verifica no *estado distribuído* se há informação, ou não, para o respectivo cliente com a finalidade de assegurar a semântica *exatamente uma*. Se o primário s_i falhou antes de propagar uma requisição ao grupo de replicação, o servidor alternativo executa o serviço e envia o resultado ao cliente. Se o primário s_i falhou depois de propagar o estado distribuído mas antes de responder ao cliente, o servidor alternativo responde com a informação do estado distribuído.

Se o primário s_i falhar enquanto propaga um estado distribuído ao grupo de replicação, o sistema de comunicação de grupo garante a *atomicidade* na entrega das mensagens ao grupo de replicação. É assumindo que, se todos os servidores do grupo de replicação entregaram uma mensagem através da primitiva de comunicação de grupo, então o estado das réplicas do grupo de replicação é idêntico.

4.5 Algoritmos para os serviços de alta disponibilidade

A implementação dos serviços para permitir alta disponibilidade depende do tipo de objeto e se aplicação usa ou não o serviço transacional. Em caso de falha do servidor primário corrente que mantenha *objetos sem estado,* manter o serviço de alta disponibilidade requer apenas o chaveamento para um servidor alternativo no grupo de replicação (item 4.5.1).

Se um cliente acessa um *objeto com estado volátil*, o serviço é executado localmente, no servidor primário s_i escolhido pelo serviço de nomes. Os secundários não participam dessa operação. Quando um cliente requer uma operação de atualização no estado de um *objeto com estado volátil,* o primário s_i propaga essa atualização a todos os secundários para manter um estado distribuído consistente no grupo de replicação. Para garantir alta disponibilidade apesar de falhas, além do serviço de chaveamento de servidor (item 4.5.1), é necessário um serviço que implemente o estado distribuído a cada atualização no estado (item 4.5.2), ou a cada confirmação de transação (item 4.5.3), se a aplicação usa transações.

Implementar alta disponibilidade para *objetos com estado persistente* é mais complexo que implementar alta disponibilidade em *objetos com estado volátil*. *Objetos com estado volátil,* no escopo deste trabalho, são implicitamente determinísticos: um novo estado volátil depende apenas da operação (ou da transação) emitida pelo cliente, pois *objetos com estado volátil* não são compartilhados por diferentes clientes. *Objetos com estado persistente* são não–determinísticos: um novo estado de um *objeto com estado persistente* depende das operações do banco de dados local emitidas por um cliente para o objeto e também das operações no banco de dados dos demais membros do grupo de replicação. Permitir um serviço de alta disponibilidade para *objetos com estado persistente* requer o serviço de chaveamento de servidor (item 4.5.1) e um serviço (item 4.5.4) que considere transações locais e distribuídas e em execução no grupo de replicação.

Tipo de componente	Aplicação sem transação	Aplicação com transação
Componente sem estado	chaveamento de servidor (item 4.5.1)	chaveamento de servidor (item 4.5.1)
Componente com estado volátil	chaveamento de servidor (item 4.5.1) + protocolo híbrido com serviço local (item 4.5.2)	chaveamento de servidor (item 4.5.1) + protocolo híbrido com serviço local com o uso de transações (4.5.3)
Componente com estado persistente		chaveamento de servidor (item 4.5.1) + protocolo híbrido com serviço distribuído (item 4.5.4)

FIGURA 4.6 – Serviços para permitir alta disponibilidade a servidores

Os serviços requeridos para assegurar alta disponibilidade a objetos em aplicações distribuídas são esquematizados no quadro comparativo da figura 4.6 e serão descritos

separadamente nos próximos itens. A restrição desses algoritmos é que todas as operações são supostas idempotentes.

4.5.1 Chaveamento de servidor

O serviço de chaveamento de servidor (ou *switchover*) é realizado pelo cliente e é necessário para implementar alta disponibilidade em todos os tipos de objetos. Quando o primário falhar durante a execução de um serviço, o cliente detecta a falha por *timeout* e realizará o chaveamento para um servidor secundário. O secundário, que passa a ser o novo primário, gera uma nova instância do objeto e continua a execução do serviço.

O serviço de chaveamento é implementado de forma invisível para o usuário através da modificação do cliente para detectar a expiração do *timeout*. O cliente aguarda até que o *timeout* expire, contata novamente o serviço de nomes para receber o nome do servidor secundário alternativo e refaz a última invocação para o servidor alternativo, se o cliente não usar o serviço transacional.

Contudo, se o cliente usar o serviço transacional, a transação deve ser abortada no cliente e reinicializada usando o servidor alternativo. Nesse caso, o servidor alternativo, que passa a ser o novo primário, gera uma nova instância do objeto e continua a execução do serviço desde o começo da última transação. O endereço do novo primário é automaticamente alterado no cliente durante a próxima execução de método no componente.

Ao contrário dos demais serviços para permitir alta disponibilidade (descritos nos itens 4.5.2 a 4.5.4), que são implementados no servidor, o chaveamento de servidor é implementado no cliente. Embora esse serviço possa ser implementado através de serviços intermediários como *proxies* inteligentes, no escopo deste trabalho, o chaveamento é implementado no cliente. A idéia é que o cliente gerencie uma lista de servidores, que é construída por um administrador de sistemas. Se um desses servidores está falho, um outro servidor da lista de servidores pode ser posto a prova.

Descrição do algoritmo do serviço de chaveamento de servidor

O algoritmo da figura 4.7 mostra o serviço do cliente sem chaveamento de servidor. No serviço sem chaveamento, o cliente solicita ao *servidor de nomes* um servidor (linha 3). Então, o cliente cria uma referência para esse objeto (linha 4) que é instanciado no servidor, e depois invoca um ou mais métodos para esse objeto (linha 6). Se houver algum erro, ocorre uma exceção. Observe que o cliente pode invocar uma seqüência de métodos dentro de uma transação (linhas 5 e 10).

```
1.  cliente {
2.          objeto obj;
3.          servidor := lookup (nome_servidor);
4.          criar (obj);
5.          [ começa a transação ]
            para cada método_no_cliente faça {
6.                  executar_método (obj);
                    se houver exceção {
7.                          imprime mensagem de erro
8.                    }
9.          }
10.         [ termina a transação ]
11. }
```

FIGURA 4.7 – Algoritmo executado pelo cliente sem chaveamento

O serviço do cliente com chaveamento de servidor é mostrado no algoritmo da figura 4.8. Observe esse serviço é muito semelhante ao serviço anterior. A diferença é que, se a invocação do método gerou alguma exceção, então ocorre o chaveamento do servidor (linha 8). Observe também que, como os métodos podem estar dentro de uma transação, a transação terá que ser abortada e reinicializada no cliente em caso de falha no servidor.

```
1.  cliente {
2.          objeto obj;
3.          servidor := lookup (nome_servidor);
4.          int candidato := obter_número (nome_servidor);
5.          criar (obj);
6.          [ começa a transação ]
            para cada método_no_cliente faça {
7.                  executar_método (obj);
                    se houver exceção {
8.                          chaveamento (candidato +1);
9.                          executar_método (obj);
10.                   }
11.         }
12.         [ termina a transação ]
13. }
```

FIGURA 4.8 – Algoritmo executado pelo cliente com chaveamento

O chaveamento de servidor é mostrado no algoritmo da figura 4.9. O chaveamento ordena seqüencialmente os servidores de uma lista que descreve nodos que possivelmente executam membros do grupo de replicação. Essa lista é chamada de *lista de servidores candidatos*. Se o servidor s_i da lista não está disponível, ocorre uma exceção e o servidor s_{i+1} (i.e., o próximo servidor candidato) da lista é posto à prova. O algoritmo é executando recursivamente até que algum servidor operacional seja alcançado ou até que todos os servidores candidatos sejam testados. Neste caso, a mensagem "*nenhum servidor disponível*" é impressa no console do cliente (linha 9).

```
1.  chaveamento  (inteiro candidato) {
        inteiro total = total_de_servidores_candidatos();
2.      se candidato <= total {
                faça  {
3.                      nome_servidor = obter_servidor (candidato);
4.                      objeto obj;
5.                      servidor := lookup (nome_servidor);
6.                      criar (obj);
                } se houver exceção {
                        candidato := candidato + 1;
7.                      chaveamento (candidato);
8.              }
        } senão {
9.              imprime "nenhum servidor disponível"
        }
10. }
```

FIGURA 4.9 – Algoritmo de chaveamento de servidor executado pelo cliente

Os itens a seguir mostram os serviços necessários para o servidor possibilitar o *failover*, para as variações do protocolo híbrido anteriormente descritas, anteriormente descritas.

4.5.2 Algoritmo para o protocolo híbrido com serviço local

O protocolo híbrido com execução de serviço local pode ser usado com eficiência para replicar o estado volátil de objetos que não usam transações. Nesse caso, as atualizações de estado de um objeto instanciado no primário são aplicadas aos secundários depois que cada método é invocado no objeto do primário. Essa técnica pode ser aplicada até mesmo quando o cliente usa transações, que nesse caso serão desconsideradas.

Descrição do algoritmo do protocolo híbrido com serviço local

O algoritmo executado pelo servidor primário é mostrado na fig. 4.10. Cada vez que um cliente executa uma operação de atualização em um objeto, o primário atualiza seu próprio estado e requisita uma operação na camada de replicação para estabelecer um novo estado distribuído com os demais servidores. A camada de replicação gera uma mensagem contendo essa atualização e transmite a mensagem para o sistema de comunicação de grupo, que difunde essa mensagem para o grupo de replicação usando a primitiva VSCAST (linha 5). Cada servidor s_k do grupo que recebe uma mensagem do servidor primário s_i entrega essa mensagem para a camada de replicação. A camada de replicação usa essa mensagem para implementar o *estado distribuído*. Observe que, como objetos com estado volátil não são compartilhados entre os clientes, a ordem total, implementada pela primitiva TOCAST não é requerida.

```
1.  servidor_primário {
2.      operação op;
        enquanto (sempre) {
3.        aguardar até (receber (op)) de cliente;
4.        resultado := executar (op);
5.        VSCAST (resultado, grupo_de_replicação);
        }
6.  }
```

FIGURA 4.10 – Algoritmo do protocolo híbrido com serviço local executado pelo servidor primário sem transações

O algoritmo da figura 4.11 descreve o comportamento dos servidores secundários. Os secundários tratam três tipos de operações realizadas sobre o estado distribuído: criação, remoção e atualização.

```
1.  servidor_secundário {
2.     mensagem msg;
       enquanto (sempre) {
            recebe (msg);
            se msg = nova_visão
                  instalar_nova_visão();
3.          se msg = op  {
4.                // separar dados da operação de msg
5.                se op = write {
6.                     // criar ou atualizar estado }
7.                senão se op = remove {
8.                     // remove estado }
9.          }
10.    }
11. }
```

FIGURA 4.11 – Algoritmo do protocolo híbrido com serviço local executado pelos servidores secundários sem transações

O estado distribuído é criado no grupo quando ocorre uma atualização sobre o objeto e ainda não há nenhum estado distribuído armazenado para esse objeto. O estado distribuído é atualizado quando ocorre uma atualização sobre o objeto e já existe um estado distribuído armazenado para esse objeto (linha 5). O estado distribuído é removido quando o objeto é destruído no cliente (linha 7), i.e., quando a sessão cliente–servidor termina.

4.5.3 Algoritmo para o protocolo híbrido com serviço local com o uso de transações

Esse serviço possibilita alta disponibilidade apesar de falhas em aplicações que usam transações para acessar *objetos com estado volátil*. O protocolo híbrido com execução de serviço local também pode ser usado com eficiência para replicar o estado volátil de objetos que usam transações. Nesse caso, as atualizações de estado de um objeto instanciado no primário são aplicadas aos secundários depois que cada transação é confirmada no primário, i.e., a cada execução da operação *commit*.

Descrição do algoritmo para o protocolo híbrido com serviço local com o uso de transações

O algoritmo do servidor primário para o protocolo híbrido com execução de serviço local e transações é análogo ao algoritmo mostrado na figura 4.10. A diferença é que a primitiva VSCAST é executada a cada *commit* e não a cada atualização. Quando o cliente solicita a confirmação de uma transação t_m, ou cria ou remove um objeto volátil, o servidor primário s_i executa uma operação na camada de replicação para tratar o estado distribuído. O serviço de replicação gera uma mensagem contendo a última atualização de estado e transmite a mensagem para o serviço de comunicação de grupo. O serviço de comunicação de grupo difunde a mensagem para os demais servidores s_k usando a primitiva VSCAST. Observe mais uma vez que, como objetos com estado

volátil não são compartilhados entre os clientes, a ordem total, implementada pela primitiva TOCAST não é requerida.

O serviço dos servidores secundários com atualização adiada de estado distribuído volátil é mostrado no algoritmo da figura 4.12. Cada servidor s_k do grupo de replicação que recebe (linha 3) uma mensagem do servidor primário s_i entrega essa mensagem para o serviço de replicação. Um servidor s_k do grupo pode receber dois tipos de mensagens: mensagem para instalar uma nova visão (linha 4), ou mensagem para gerenciar o *estado distribuído* (a partir da linha 5).

```
1.   servidor_secundário {
2.      mensagem msg;
3.      enquanto (sempre) {
            receber (msg);
4.          se msg = nova_visão
                    instalar_nova_visão();
5.          se msg = op  {
6.                  // separar dados da operação de msg
7.                  se op = create {
8.                      // criar estado }
9.                  senão se op = commit {
10.                     // atualizar estado }
11.                 senão se op = remove {
12.                     // remover estado }
13.         }
14.     }
15.  }
```

FIGURA 4.12 – Algoritmo do protocolo híbrido com serviço local executado pelos servidores secundários com transações

No protocolo híbrido com serviço local e transações, três operações são realizadas sobre os estados locais (que compõem o estado distribuído) do grupo de replicação: criação, atualização e remoção. O estado local é criado quando o objeto é criado no cliente (linha 7). O estado local é atualizado cada vez que ocorre a confirmação de uma transação t_m (linha 9). O estado local é removido quando o objeto é destruído no cliente (linha 11).

4.5.4 Algoritmo para o protocolo híbrido com serviço distribuído

O algoritmo do protocolo híbrido com serviço distribuído é baseado em Pedone *el at*. [PED98]. O protocolo de término é inicializado pelo servidor primário e consiste em enviar o estado distribuído para que o grupo de replicação confirme uma transação. Esse protocolo foi anteriormente descrito no capítulo 3, e compreende duas partes: o teste de certificação e a confirmação da transação.

Descrição do algoritmo do protocolo híbrido com serviço distribuído

O algoritmo da figura 4.13 descreve o comportamento do servidor primário, com a inicialização do protocolo de término. Antes de confirmar as atualizações de uma transação t_m sobre um *objeto com estado persistente*, o servidor primário s_i envia uma mensagem para o grupo de replicação *certificar* e *confirmar* a transação t_m (linha 6). O servidor s_i envia essa mensagem para todo o grupo, inclusive para ele mesmo, executando uma primitiva TOCAST. A mensagem contém o *writeset*, o *readset* e o

novo estado para o objeto atualizado pela transação t_m. O *writeset* é o conjunto de dados escritos por uma transação, enquanto que o *readset* é o conjunto de dados lidos por uma transação.

```
1.  servidor_primário {
2.     mensagem msg;
3.     operação op;
4.     enquanto (sempre) {
5.          aguardar até (receber (op, estado)) de cliente;
6.          TOCAST ((op, writeset, readset, estado),
    grupo_de_replicação);
       }
7.  }
```

FIGURA 4.13 – Algoritmo do protocolo híbrido com serviço distribuído executado pelo servidor primário

Entregando a mesma seqüência de mensagens para o grupo de replicação através da primitiva TOCAST, o protocolo de término assume que cada servidor no grupo irá aplicar o mesmo conjunto de atualizações sobre o seu estado local para compor o *estado distribuído*.

O algoritmo da figura 4.14 descreve o comportamento dos servidores secundários. Note que esse algoritmo também é executado pelo servidor primário s_i, já que o primário envia um TOCAST para o grupo de replicação, incluindo ele mesmo.

```
1.    servidor_secundário {
2.       mensagem msg;
         enquanto (sempre) {
             receber (msg);
             se msg = nova_visão
                   instalar_nova_visão;
3.           se msg = op  {
4.                 se op = begin
5.                       adicionar estado_local em estado_distribuído ();
6.                  senão se op = commit {
7.                       booleano resultado := teste_de_certificação (transação);
                         se resultado = verdadeiro {
8.                            commit (transação);
9.                            // confirmação
10.                        } senão abortar (transação);
                      }
11.           remover_se_possível_estado_local ();
12.           }
13.      }
14.  }
```

FIGURA 4.14 – Algoritmo do protocolo híbrido com serviço distribuído executado pelos servidores secundários

Para que um servidor do grupo de replicação certifique uma transação t_m, ele precisa saber quais transações estão em conflito com t_m. Para detectar conflitos, ao receber uma mensagem que não é troca de visão, cada servidor s_k do grupo de replicação executa o *teste de certificação* (linha 7).

Se o teste de certificação retornou *verdadeiro*, a transação t_m é *confirmada* no servidor s_k (linha 8) do grupo de replicação. Caso contrário, a transação t_m é abortada (linha 10).

Os códigos das linhas 5 e 11 do algoritmo dos servidores secundários não fazem parte do protocolo de término [PED98], mas foram aqui adicionados para que o grupo de replicação faça o controle das transações que estão sendo executadas. Informações dessas transações (como a identificação do servidor primário, a identificação do cliente, o *writeset* e o *readset*) são armazenadas no estado distribuído.

O algoritmo que *remove os estados locais* verifica, periodicamente, se um estado local de uma transação t_m dentro do estado distribuído pode ser removido. Um estado local pode ser removido nas seguintes condições:

- quando uma transação t_m vai executar o *commit* e é detectado conflito e a transação t_m é abortada (*i*).
- quando a transação t_m foi confirmada e não está em conflito com nenhuma transação do estado distribuído (*ii*);
- quando a transação t_m está confirmada previamente no estado distribuído, desde que não exista nenhuma transação não confirmada antes da transação t_m (*iii*).

Por exemplo, a figura 4.15 mostra duas transações, t_a e t_b com os valores iniciais para três variáveis.

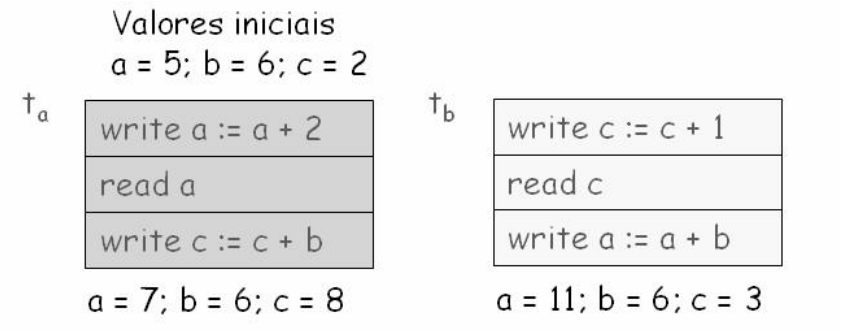

FIGURA 4.15 – Duas transações exemplo

Se a execução das transações no grupo de replicação for serial (i.e., uma após a outra), os valores finais, para as três variáveis, serão, respectivamente, $a = 13$, $b = 6$ e $c = 9$. Se as transações forem intercaladas no grupo de replicação, a manutenção do estado distribuído pelo grupo de replicação garante que aplicação dos valores das variáveis no grupo de replicação serão os mesmos valores da execução serial.

A figura 4.16 mostra duas execuções possíveis (situação *a* e situação *b*) para as transações t_a e t_b. Os quadrinhos marcados com <*begin* t_i> e <*commit* t_i> indicam que um estado local será armazenado no estado distribuído, enquanto que os demais quadrinhos indicam que operações estão sendo executadas localmente em algum servidor primário s_i.

Na situação *a*, observe que o estado distribuído <*commit* t_b> não pode ser removido quando t_b é confirmada, pois t_a ainda não foi confirmada. Nesse caso, t_a não confirma e será abortada, e o seu estado distribuído <*begin* t_a> será removido (pela condição *i*). A seguir, os estados <*begin* t_b> e <*commit* t_b> podem ser removidos (pela condição *iii*).

Na situação *b*, observe que o estado distribuído <*commit* t_b> não pode ser removido quando t_b é confirmada, pois t_a ainda não foi confirmada, apesar de t_a ter começado antes de t_b. Nesse caso, t_a não confirma e será abortada, e o seu estado distribuído <*begin* t_a> será removido (pela condição *i*). Como na situação *a*, os estados <*begin* t_b> e <*commit* t_b> podem ser removidos (pela condição *iii*).

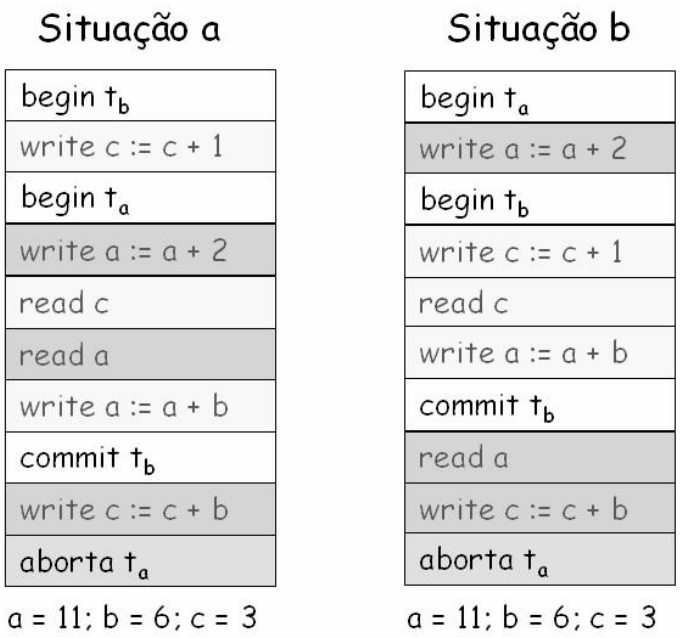

FIGURA 4.16 – Duas intercalações possíveis para as transações *a* e *b*

O algoritmo do *teste de certificação* é mostrado na figura 4.17. O teste de certificação verifica conflitos entre transações. Cada servidor s_k aborta t_m se há alguma transação T_j confirmada que conflita com t_m. O teste de certificação retorna *falso* (linha 8) se há conflito de alguma transação confirmada T_j com a transação t_m, e *verdadeiro* se não há em conflito entre transações (linha 2).

```
1.  booleano teste_de_certificação (transação) {
2.        booleano retorno = verdadeiro
3.        // verificar transações em conflito
4.        para cada estado_local em estado_distribuído {
5.              se transação.readset = estado_local.writeset ou
6.                   se transação.writeset = estado_local.readset ou
7.                     se transação.writeset = estado_local.writeset
8.                              retorno := falso;       // transações em conflito
9.        }
10.       retornar retorno;
11. }
```

FIGURA 4.17 – Algoritmo para o teste de certificação

Depois do teste de certificação, ocorre a *confirmação* da transação. A *confirmação* aborta transações locais em execução, que estão em conflito com a transação T_m. Se existir alguma transação t_n em execução no servidor s_k, cujo *writeset* ou *readset* conflitam com o *writeset* de T_m, t_n será abortada no servidor s_k.

4.6 Observações do capítulo

Embora as quatro camadas da arquitetura sejam teoricamente bem distintas, na implementação do protótipo (que será descrita no capítulo 5) elas aparecem de forma integrada. O *servidor dorothy* (i.e., o servidor do protótipo) é implementado por um

processo UNIX que representa o servidor de aplicação com os serviços de persistência do banco de dados, de comunicação de grupo e de replicação.

Amir *et al*. [AMI94] também apresenta uma arquitetura em camadas e um protocolo baseado na primitiva de *muticast* para manter a consistência das réplicas de um banco de dados. Entretanto, o protocolo de Amir *et al*. considera que clientes executam métodos em banco de dados, ao invés de transações.

Com respeito à semântica de execução *exatamente uma,* Frølund *et al.* [FRØ99] trata da manutenção dessa semântica em um sistema *three–tier*: navegador no cliente, servidor Web e de banco de dados. Na solução proposta, as requisições dos clientes são ordenadas, e cada cliente invoca somente um servidor de aplicação onde o serviço é realizado sem manutenção de estado. Frølund *et al.* usa o conceito de *transações testáveis*, onde a resolução (*commit* ou *abort*) e o resultado de uma transação, i.e., a execução do comando SQL, podem ser determinados com a garantia de um serviço com tolerância a falhas e de alta disponibilidade.

5 Implementação e avaliação dos serviços de replicação e de chaveamento para servidores EJB

Para materializar e validar os serviços HA propostos neste trabalho, foi implementado um protótipo para servidores de aplicação EJB. O protótipo, chamado (servidor) *dorothy*, implementa a arquitetura multicamadas anteriormente descrita e usa como suporte os sistemas JOnAS [OBJ2001] e JavaGroups [BAN99]. O protótipo inclui no JOnAS o serviço de replicação, que materializa a interface entre os sistemas JOnAS e JavaGroups, o serviço de chaveamento de servidor, e os serviços oferecidos pelo JavaGroups.

A construção de um protótipo possibilita descobrir muitos problemas que serão encontrados no sistema real. Assim, soluções de antemão podem ser propostas. O protótipo também possibilita um *framework* onde pode ser feita a validação experimental do sistema através da variação de características do protocolo de replicação, como o número de réplicas, e onde podem ser estimadas para o usuário final medidas de desempenho e de alta disponibilidade.

5.1 Sistemas de suporte

Dois sistemas de suporte foram usados para implementar o protótipo: o servidor EJB JOnAS (Java Open Application Server) [OBJ2001] e o sistema de comunicação de grupo JavaGroups [BAN99]. Esses dois sistemas têm código aberto, i.e., permitem que o seu código seja alterado e distribuído com as alterações, e distribuição gratuita, que facilita seu uso principalmente no ambiente acadêmico.

5.1.1 *Java Open Application Server*

O sistema JOnAS (*Java Open Application Server*) [DAN2000, OBJ2001] é uma implementação aberta da especificação 1.1 EJB. O JOnAS é implementado totalmente em linguagem Java, e é parte do ObjectWeb Open Source, uma iniciativa onde colaboram vários parceiros incluindo a France Telecom's R&D e o INRIA/França.

Os principais serviços do JOnAS incluem (figura 5.1) os EJB *containers,* uma ferramenta (GenIC ou *Generate Interposition Classes*) para gerar as classes intermediárias, o serviço transacional (que provê suporte JTA e coordenação de transações distribuídas), um serviço de banco de dados que provê suporte à interface JDBC, e um conjunto de ferramentas para desenvolver e gerenciar *beans*.

Os serviços do JOnAS podem ser acessados por clientes RMI ou por clientes HTTP através de um servidor Web. O protótipo considera apenas clientes RMI.

A versão atual do JOnAS é a 3.0 de 4 de março de 2003. Entretanto, a versão usada no protótipo é a 2.4. O JOnAS 2.4 possui cerca de 4.302 arquivos dispostos em 393 diretórios que totalizam aproximadamente 20,7 MB.

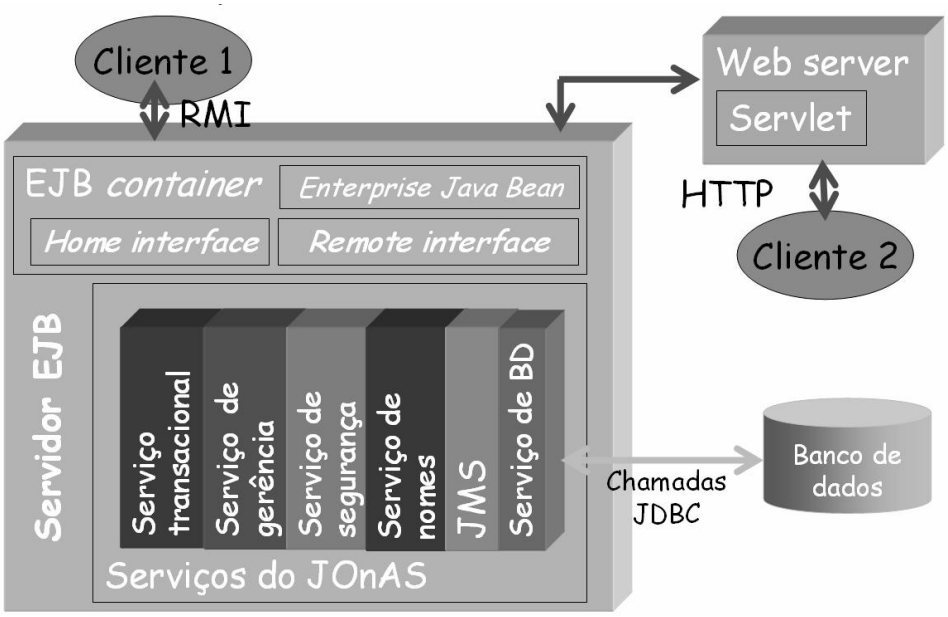

FIGURA 5.1 – Arquitetura do JOnAS

Inicialmente, a implementação do protótipo acompanhou os inúmeros *upgrades* do JOnAS. Contudo, a cada *upgrade* alguma alteração significativa era necessária no serviço de replicação, ocasionando sensível atraso na finalização da implementação. Portanto, optou–se por estacionar o *upgrade* na versão 2.4 do JOnAS, que parece ser bem estável e, posteriormente, poderá ser realizado o *upgrade* para uma versão mais recente. Para facilitar o *upgrade*, o serviço de replicação foi implementado principalmente no diretório `org.objectweb.dorothy` e pode ser configurado como um serviço adicional oferecido pelo servidor JOnAS.

5.1.2 *JavaGroups*

O sistema de comunicação de grupos JavaGroups [BAN99] provê serviços básicos como entrega confiável de mensagens para um grupo e composição de grupos à camada de replicação, através da escolha adequada de micro–protocolos pelo desenvolvedor.

O JavaGroups implementa entrega confiável de mensagens para um grupo usado IP *multicast* em *hardware*. Segundo Amir *et al*. [AMI94], a principal vantagem de sistemas baseados em IP *multicast* é o desempenho. O IP *multicast* não necessita *software* especial para realizar a difusão de mensagens sobre uma rede de comunicação ponto–a–ponto.

Originalmente, o JavaGroups foi desenvolvido na Cornell University. Atualmente o sistema encontra–se disponível em `http://www.javagroups.com` ou em `http://javagroups.sourceforge.net`.

Para enviar e receber mensagens usando o serviço de comunicação do JavaGroups, um processo precisa ser conectado a um canal lógico confiável., i.e., ele tem que ser *membro* de um grupo. Um canal é gerado quando o primeiro processo se conecta a esse canal. Os demais processos conectam–se a esse mesmo canal para formar o grupo. Todos os processos conectados a um mesmo canal recebem todas as mensagens enviadas para o grupo. Assim sendo, o canal conecta diferentes processos que são membros de um mesmo grupo de replicação.

Quando um canal é criado, as propriedades do canal são especificadas para todos os membros. Essas propriedades são implementadas individualmente por protocolos que compõem uma *pilha de protocolos*, semelhante à arquitetura de camadas usadas nos sistemas Horus [REN96] e Ensemble [HAY98]. Cada protocolo implementa uma propriedade específica, relacionada com entrega confiável de mensagens ou com o serviço de composição de grupos. Todas as mensagens enviadas para um grupo, através de um canal confiável, atravessam essa pilha de micro–protocolos e incorporam essas propriedades.

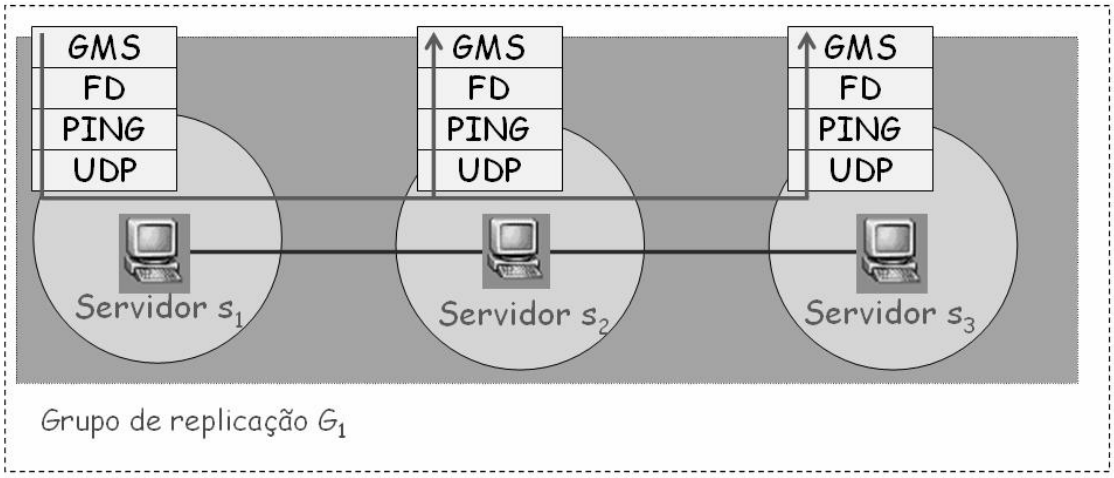

FIGURA 5.2 – Comunicação confiável através da
pilha de micro–protocolos do JavaGroups

A figura 5.2 mostra 3 pilhas de micro–protocolos para três membros em um grupo de replicação G_1. O servidor s_1 envia uma mensagem para os demais membros. Essa mensagem atravessa todos os micro–protocolos da pilha com a finalidade de obter um serviço confiável. A camada GMS (*group membership*) implementa o serviço de composição de grupos. A camada FD (*failure detection*) é um serviço que detecta membros em suspeita de falha. O protocolo da camada PING conecta um processo com um grupo já existente. A camada UDP representa o final dessa pilha, enviando mensagens entre os membros usando IP e UDP não confiável. Essa lista de micro–protocolos é flexível: outros protocolos podem ser acrescidos ou retirados, e não é exaustiva: esses são apenas os principais protocolos e há outros protocolos em desenvolvimento. O micro–protocolo TOTAL, por exemplo, provê a propriedade de ordem para a recepção de mensagens entre os membros de um grupo. Esse protocolo é fundamental para mapear a primitiva TOCAST para o JavaGroups. Se nenhum protocolo é informado, a pilha de protocolos *default* do JavaGroups é usado.

A versão atual do JavaGroups é a 2.0.6 de 25 de janeiro de 2003. Essa versão foi usada na implementação do serviço de replicação para o protótipo. Observe que, no contexto do serviço de replicação, fazer o *upgrade* de versão do JavaGroups é mais simples que fazer o *upgrade* do JOnAS uma vez que o código do JavaGroups não é alterado na implementação do serviço de replicação, enquanto que o código do JOnAS precisou ser alterado, ainda que as alterações sejam mínimas.

Extensões do JavaGroups relacionadas ao contexto deste trabalho

Duas extensões do JavaGroups são consideradas importantes no contexto deste trabalho: HSQLDB/R [BAN2002] que é uma extensão do bando de dados HSQLDB

que comporta réplicas, e um serviço de replicação para a plataforma J2EE que combina o servidor Tomcat e o JavaGroups.

A. Replicação usando o HSQLDB/R

O HSQLDB/R [BAN2002] sincroniza réplicas do banco de dados HSQLDB usando abstrações de comunicação de grupo. O HSQLDB/R é um *software* livre experimental. A estratégia de *difusão de escritas* usada é a *adiada*. A implementação atual do HSQLDB/R garante a consistência dos membros do grupo de replicação através do protocolo de cópia primária ou do esquema de replicação de dados disjuntos. Cada comando SQL, que atualiza o banco de dados em um dos membros, será automaticamente replicado nos outros membros.

O primeiro membro de um grupo no HSQLDB/R é inicializado através dos arquivos `*.data` e `*.script`, e funciona como coordenador do grupo. Os demais membros recebem o estado do coordenador durante a transferência de estado.

O HSQLDB/R é construído a partir de classes do banco de dados HSQLDB que foram modificadas para comportar a *difusão de escritas* através do JavaGroups. As principais alterações afetam as classes `Log` e `Database` do HSQLDB.

A classe `org.hsqldb.Log` do HSQLDB/R possui métodos para guardar o estado do banco de dados em uma estrutura e para receber do coordenador do grupo uma estrutura com informação para inicializar um novo membro do grupo.

A classe `org.hsqldb.Database` do HSQLDB/R possui métodos para que o nodo verifique qual comando modificou o banco de dados e difunda (por *multicast*) atualizações executadas e confirmadas no seu banco de dados aos demais servidores; e métodos para que o nodo busque o estado do banco de dados do coordenador, e métodos para atualizar o estado do banco de dados do nodo com a informação recebida do coordenador.

Um *listener* aguarda mensagens *multicast* com solicitação de replicação de dados. Se a mensagem foi enviada pelo próprio nodo, será descartada. A classe `org.hsqldb.replication.ReplicationData` contém o comando SQL que será enviado através de *multicast*.

No HSQLDB/R, a replicação é configurada como uma propriedade do sistema de banco de dados definida em um arquivo XML que descreve se a replicação está habilitada ou não, o nome do grupo de replicação e a pilha de protocolos do JavaGroups. A pilha de protocolos também pode ser um endereço URL indicando um arquivo XML.

O HSQLDB/R usa um recurso de linguagens de programação orientadas a objeto chamado padrão fábrica (ou *factory*). Este padrão permite construir uma classe auxiliar que decide sobre a construção de objetos em função dos parâmetros fornecidos. O programa fica independente da forma como os objetos são criados, o que permite maior flexibilidade. O padrão fábrica possibilita integrar replicação no HSQLDB/R de forma não–intrusiva, modificando o mínimo possível o código já codificado. Quando a replicação não está habilitada, a aplicação não perde desempenho porque as classes que são relacionadas à replicação não são usadas.

Embora não tenha sido projetado para a plataforma J2EE e, conseqüentemente, para a especificação EJB, o HSQLDB pode permitir alta disponibilidade para *beans*, se ele for compatível com a interface JDBC. As principais desvantagens de usar essa solução

incluem a restrição ao uso de um banco de dados específico para plataforma J2EE e o fato de não apresentar um serviço de chaveamento de servidor.

B. Replicação usando Tomcat + JavaGroups

O servidor Tomcat foi estendido [BUR2002, HAN2002] para habilitar o chaveamento da execução de serviço entre duas ou mais instâncias do servidor mantidas em um grupo de replicação. Esse sistema implementa um novo conjunto de classes para o servidor Tomcat 4, que é um servidor Web. As novas classes permitem replicação de sessão HTTP usando JavaGroups. O sistema possibilita o desenvolvimento de serviços de balanceamento de carga e gerência de *cluster* para o Tomcat.

Para usar essa implementação, o arquivo `server.xml` do servidor Tomcat precisa descrever as configurações do serviço do JavaGroups. Cada vez que uma sessão HTTP é criada em uma instância do grupo de replicação, essa instância é sincronizada com os demais servidores do grupo usando informação armazenada *in–memory*. O coordenador do grupo envia essa informação para a nova instância.

A referência consultada [BUR2002, HAN2002] não descreve informações sobre a degradação de desempenho obtida com a execução do sistema Tomcat com o JavaGroups.

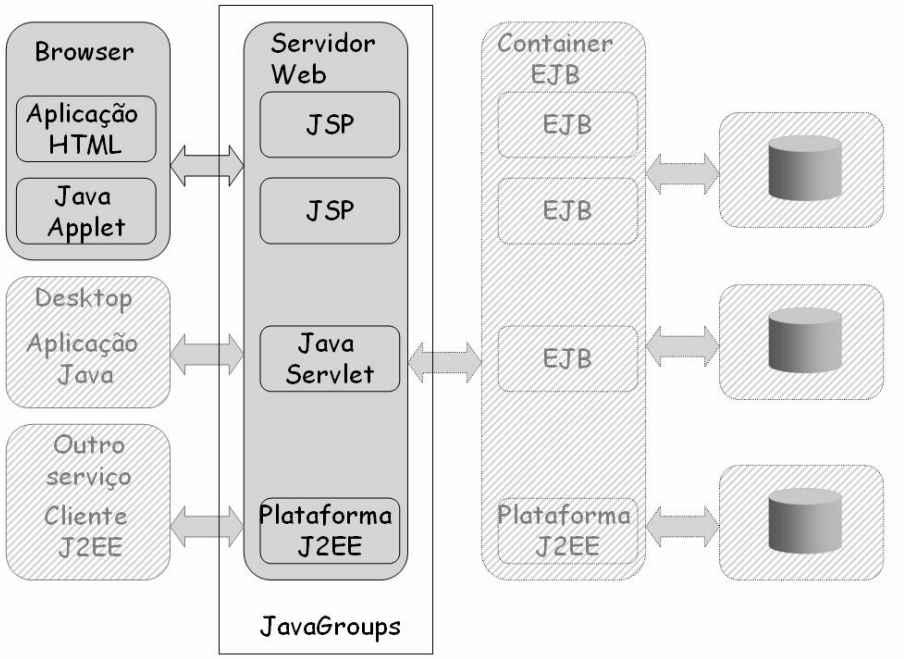

FIGURA 5.3 – Plataforma J2EE considerada para a implementação do Tomcat+JavaGroups

Observando a plataforma J2EE na figura 5.3, a integração do Tomcat (representado como Servidor Web) com o JavaGroups oferece alta disponibilidade somente para os serviços que incluem os servidores Web, que não mantêm estado para os clientes (i.e., os servidores são *stateless*). Não há um serviço para os servidores EJB, incluindo o serviço transacional. A parte com hachuras na figura 5.3 não é considerada nessa implementação.

5.2 Integrando o JOnAS e o JavaGroups para implementar o protótipo

A implementação do protótipo procurou aproveitar de forma adequada a estrutura baseada em componentes oferecida pelo sistema JOnAS. O protótipo busca simplicidade e eficiência para implementar os serviços HA. O desafio do desenvolvimento do protótipo consistiu em modificar a estrutura provida pelo JOnAS sem comprometer em demasiado o desempenho esperado para o serviço EJB e mantendo a compatibilidade com a especificação EJB. O protótipo é chamado de servidor *dorothy* e integra as classes oferecidas pelo JavaGroups com as classes existentes no JOnAS através da adição de *ganchos* (*hooks*) nas classes do JOnAS e da adição de novas classes. Um *gancho* é um comando, ou um conjunto de comandos, adicionado em um código. Um *parser* pode adicionar *ganchos* a um código e assim possibilitar o acréscimo de novas funcionalidades. Esse mecanismo é tipicamente usado pelas ferramentas do sistema JOnAS.

O JOnAS 2.4, assim como suas outras versões, possui um ambiente complexo para a implementação de novas extensões. A documentação da implementação está dispersa no código e algumas classes básicas, como a `EJBObject`, não são implementadas pelo JOnAS, mas aproveitadas de outra implementação. A dificuldade de não dispor do código de classes como a `EJBObject` é que a implementação de novas extensões dessas classes fica dificultada. Por exemplo, acrescentar um método para recuperar o estado de um *bean* em `EJBObject` atualmente não é possível.

5.2.1 Modelo e ambiente do sistema usado no protótipo

O modelo do sistema usado na implementação do protótipo *dorothy* segue o modelo da arquitetura HA descrito no capítulo anterior, onde as operações sobre os objetos são idempotentes. Assim sendo, o protótipo implementa a semântica de execução no *mínimo uma*.

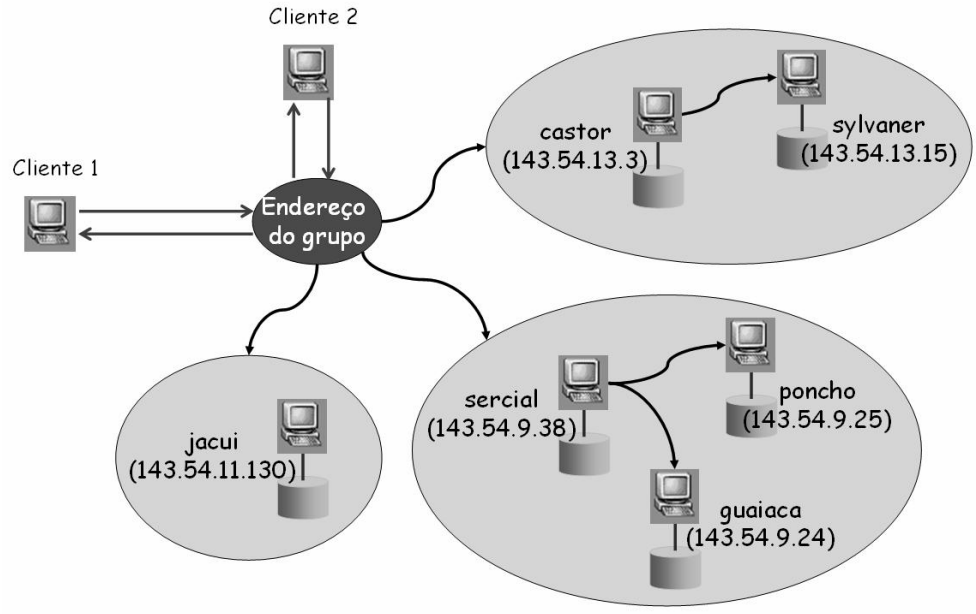

FIGURA 5.4 – Ambiente de execução do protótipo

O ambiente possui nodos heterogêneos, que são estações de trabalho ULTRA 1, ULTRA 5, ULTRA 10, SparcStation 20 e ULTRA 1 Enterprise 150 conectadas em sub–

redes Ethernet de 10M*bps*, conforme ilustra a figura 5.4. Os clientes (representado pelos clientes 1 e 2) e os servidores (representados pelos nodos *castor, sylvaner, jacui, sercial, poncho* e *guaiaca*) executam em nodos destas sub–redes. Um cliente conhece apenas o endereço de um dos servidores, que é o endereço IP do servidor primário s_i e que representa o **endereço do grupo** de replicação para o cliente. Observe que o endereço do grupo pode representar o endereço IP de qualquer um dos servidores anteriormente listados.

Um cliente se comunica com o servidor primário s_i, usando o endereço do grupo, através do protocolo RMI. Um servidor JNDI é usado em cada nodo que executa um servidor para exportar o serviço que esse servidor oferece. Dessa forma, membros do grupo de replicação localizados em sub–redes distintas podem se encontrar.

Embora os servidores EJB possam estar localizados no mesmo nodo, é desejável que eles sejam instanciados em diferentes nodos para oferecer um serviço de alta disponibilidade. Um cliente pode estar localizado no mesmo nodo que o seu servidor primário ou em nodo diferente.

5.2.2 Alterando o comportamento do servidor JOnAS

O protótipo foi desenvolvido através da modificação de classes do JOnAS e da adição de novas classes. Essas classes são mostradas na figura 5.5.

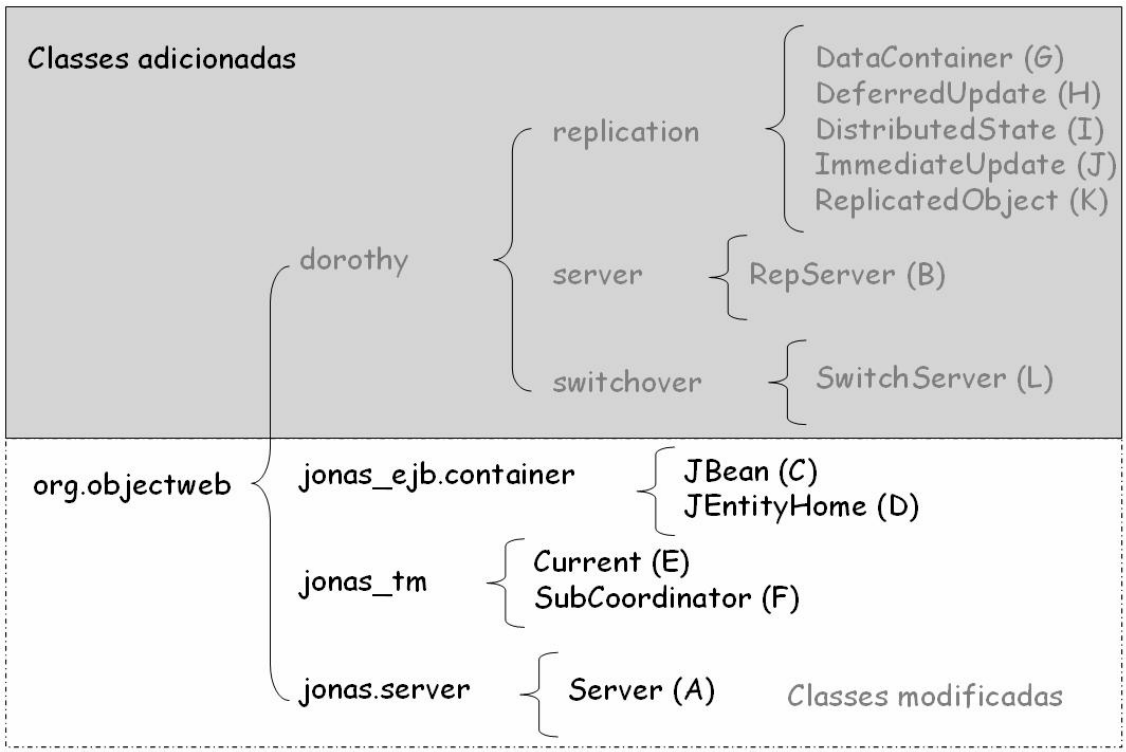

FIGURA 5.5 – Principais pacotes e classes que foram adicionadas ou modificadas

O protótipo implementa alta disponibilidade preservando os serviços do JOnAS e mantém a compatibilidade com a especificação EJB. Por exemplo, o JOnAS continua usando o serviço de comunicação RMI entre o cliente e o servidor. O serviço de comunicação de grupo é usado somente entre os servidores do grupo de replicação.

A implementação do protótipo faz o mapeamento das variações do protocolo híbrido para os diferentes tipos de objetos da especificação EJB com a finalidade de permitir

failover e garantir alta disponibilidade. Os *beans* de sessão podem usar ou não o serviço transacional e podem ser sem estado ou com estado volátil. As aplicações com *beans* de sessão sem estado requerem apenas o chaveamento de servidor para oferecer alta disponibilidade, mesmo que elas usem o serviço transacional. O chaveamento de servidor também é adequado para possibilitar alta disponibilidade a aplicações com *beans* orientados a mensagens. As aplicações que usam transações sobre *beans* de sessão com estado executam o chaveamento de servidor e o protocolo híbrido com execução de serviço local com transações para permitir alta disponibilidade. As aplicações que usam transações sobre *beans* de entidade executam o chaveamento de servidor e o protocolo híbrido com execução de serviço distribuída para permitir alta disponibilidade. A figura 5.6 mostra um sumário com os serviços necessários para possibilitar *failover* para cada tipo de *bean*. Observe que cada tipo de *bean* da figura 5.6 está relacionado com os tipos de objetos apresentados anteriormente no quadro comparativo da figura 4.6.

Tipo de *bean*	Aplicação sem transação	Aplicação com transação
Beans de sessão sem estado e *beans* orientados a mensagens	chaveamento de servidor	chaveamento de servidor
Beans de sessão com estado	chaveamento de servidor + protocolo híbrido com serviço local (apenas uma réplica executa o serviço); replicação preguiçosa; atualização imediata a cada atualização; difusão adiada	chaveamento de servidor + protocolo híbrido com serviço local (apenas uma réplica executa o serviço); replicação preguiçosa; atualização adiada a cada *commit*, difusão adiada
Beans de entidade		chaveamento de servidor + protocolo híbrido com serviço distribuído (todas as réplicas executam o serviço); replicação ávida, atualização adiada a cada *commit*; difusão imediata

FIGURA 5.6 – Serviços para permitir *failover* em aplicações EJB

A seguir serão listadas as classes descritas na figura 5.5, e que implementam os serviços descritos na figura 5.6. Para facilitar a organização do texto, cada classe da figura 5.5 foi associada a uma letra.

A. org.objectweb.jonas.server.Server

O serviço de replicação aqui proposto é oferecido por um grupo de servidores idênticos, instanciados a partir da classe `org.objectweb.jonas_ejb.Server`, que é implementada pelo JOnAS. Essa classe foi modificada para inicializar uma instância do servidor do grupo de replicação (fig. 5.7) que é implementada pela classe `org.objectweb.dorothy.server.RepServer`.

Para que o grupo de replicação mantenha o estado distribuído consistente, é preciso coletar informação em diferentes classes, dispersas no sistema JOnAS. A dificuldade consiste em sincronizar as diferentes classes, uma vez que a informação sobre o estado dos *beans* deve ser obtida nessas diferentes classes e não na classe `org.objectweb.dorothy.server.RepServer`, que é a classe que conecta todas essas informações e envia o estado distribuído para os demais servidores. Por exemplo, para que a classe `RepServer` verifique quando uma transação é confirmada em um *bean* de sessão ou de entidade, a informação deve ser coletada na classe `org.objectweb.jonas_ejb.container.JBean`.

```
1.  package org.objectweb.jonas.server;
2.  ...
3.  try {
4.      dorothy =  new RepServer();
5.  } catch (Exception e) {
6.      throw new Exception("Cannot create the replicated
    server ("+e+")");
7.  }
8.  ...
```

FIGURA 5.7 – Fragmento da classe `org.objectweb.jonas_ejb.Server` com inicialização do serviço de replicação

B. org.objectweb.dorothy.server.RepServer

A classe `RepServer` implementa as variações do protocolo híbrido. A escolha de qual variação será executada é feita em tempo de inicialização do serviço, mas futuramente poderá ser configurada em um arquivo XML. Observe que o micro–protocolo que implementa a propriedade de *ordem total* apenas é requerido para o protocolo híbrido com a execução do serviço distribuído, onde a atualização dos membros do grupo de replicação é *adiada*.

```
1.   RepServer  {
2.      string grupo_de_replicação = "dorothy";
3.      canal ch;
4.      se técnica_de_replicação = atualização_adiada e
    propagação_imediata
5.           string propriedades = "UDP:PING:FD:STABLE:NAKACK:
             UNICAST:FRAG:FLUSH:GMS:TOTAL:VIEW_ENFORCER:
             STATE_TRANSFER:QUEUE";
6.      senão
7.           string propriedades = "UDP:PING:FD:STABLE:NAKACK:
             UNICAST:FRAG:FLUSH:GMS:VIEW_ENFORCER:
             STATE_TRANSFER:QUEUE";
8.      ch := gerar_canal (propriedades);
9.      atualizar (ch);
10.     conectar (ch, grupo_de_replicação);  // servidor se conecta ao
                                             //grupo
11.     inicializar thread_primária();
12.     inicializar thread_secundária();
13.  }
```

FIGURA 5.8 – Algoritmo de inicialização do serviço de replicação

A classe `RepServer` inicializa o serviço de grupos (linha 2 até linha 10 na fig. 5.8) e duas *threads* concorrentes: a *thread* primária executa o serviço do servidor primário (linha 11) e a *thread* secundária para executar o serviço do servidor secundário (linha 12), já que um mesmo servidor pode funcionar como primário para um cliente e como secundário para outro cliente. Antes de ingressar no grupo (linha 10), cada servidor recebe o estado do grupo de replicação (linha 9) para manter o estado distribuído consistente.

A *thread* primária aguarda em um *loop* infinito informações sobre os *beans* que estão em execução no servidor primário e, depois, envia aos secundários as informações para que o estado distribuído seja gerenciado. A *thread* primária executa um dos algoritmos descritos no capítulo anterior, dependendo tipo de objeto EJB e do uso ou não do serviço transacional.

A *thread* secundária aguarda em um *loop* infinito mensagens do grupo de replicação para instalar novas visões e para gerenciar o estado distribuído. Dependendo do tipo da mensagem (troca de visão ou operação), a decisão de execução será tomada. Por exemplo, se a mensagem for do tipo *troca de visão*, uma nova visão será instalada para o secundário, que removerá ou adicionará membro(s) na sua visão local do grupo de replicação. Se a mensagem for do tipo *operação*, o secundário separa os parâmetros da mensagem e descobre que ação dever ser tomada sobre o estado distribuído, de acordo com a técnica de atualização usada pelo grupo de replicação (atualização imediata ou de atualização adiada). A *thread* secundária executa um dos algoritmos descritos no capítulo anterior, dependendo do algoritmo executado pela *thread* primária e tipo de objeto EJB.

O serviço de replicação deverá priorizar mensagens da *thread* secundária, com relação às mensagens da *thread* primária pois a maior prioridade das mensagens do sistema de comunicação de grupo garante um estado distribuído consistente para o grupo de replicação.

A informação difundida no protocolo de término, juntamente com a informação armazenada no estado distribuído permite que todos os servidores do grupo de replicação tomem as mesmas decisões e armazenem o mesmo estado persistente. O protocolo de término possui duas fases: o teste de certificação e a confirmação.

No *teste de certificação,* cada membro do grupo de replicação verifica se a transação t_a em execução não está em conflito com alguma transação distribuída já confirmada. Se a transação t_a está em conflito com uma transação distribuída T_b, o nodo aborta t_a. Caso contrário, o nodo confirma t_a.

Note que os *beans* de sessão com estado não compartilham seu estado com múltiplos clientes, sendo assim, o teste de certificação não é necessário para aplicações EJB com esse tipo de *bean,* mesmo que o protocolo de replicação use a difusão adiada.

A *confirmação* aborta transações locais em conflito com uma transação distribuída recentemente confirmada no grupo de replicação. A *confirmação* somente é executada se a transação T_a foi confirmada (ou finalizada) em todos os nodos. Esse protocolo verifica se alguma transação local t_c em algum membro do grupo de replicação está em conflito com a transação distribuída T_a, recentemente confirmada. Se há alguma transação local t_c em conflito T_a, a transação t_c é abortada.

C. org.objectweb.jonas_ejb.container.JBean

A classe `JBean` existente no JOnAS foi modificada para detectar o momento no qual uma transação é confirmada ou abortada na técnica de atualização adiada.

Quando uma transação de atualização t_a está prestes a ser confirmada, essa classe envia, através de variável compartilhada, um aviso para que a *thread* primária execute o *protocolo de término.* O protocolo de término consiste na difusão de uma mensagem com a primitiva TOCAST ao grupo de replicação. A classe `JBean` somente prossegue sua execução quando a mensagem enviada pela primitiva TOCAST é entregue na *thread* primária.

D. org.objectweb.jonas_ejb.container.JEntityHome

A classe `JEntityHome` existente no JOnAS foi modificada para informar à *thread* primária que a transação t_m foi confirmada no grupo de replicação durante o protocolo de término.

E. org.objectweb.jonas_tm.Current

A classe `Current` existente no JOnAS foi modificada para informar à *thread* primária que uma mensagem deve ser difundida para que a transação t_m seja confirmada no grupo de replicação durante o protocolo de término. Essa classe foi modificada para servir ao protocolo híbrido com execução distribuída e com execução local.

F. org.objectweb.jonas_tm.SubCoordinator

A classe `SubCoordinator` do JOnAS implementa, juntamente com outras classes, o protocolo de *commit* de duas fases. Essa classe foi modificada para informar à *thread* primária quando uma transação t_m é abortada. Essa classe foi modificada para servir ao protocolo híbrido com execução distribuída.

G. org.objectweb.dorothy.replication.DataContainer

A classe `DataContainer` foi adicionada ao código do JOnAS. Essa classe encapsula informações que são enviadas como mensagem através da classe `DistributedState` juntamente com a operação que codifica a ação que será realizada pela *thread* secundária sobre o estado distribuído e possivelmente sobre os múltiplos bancos de dados do grupo de replicação.

H. org.objectweb. dorothy.replication.DeferredUdpate

A classe `DeferredUdpate` foi adicionada ao código do JOnAS. Essa classe coleta informações de uma transação t_m que serão enviadas a cada *commit* pela *thread* primária ao grupo de replicação no protocolo híbrido com execução distribuída, ou no protocolo híbrido com execução local para aplicações EJB que usam transações.

Para o protocolo híbrido com execução local para aplicações EJB que usam transações, essas informações incluem a identificação do cliente, a identificação do servidor primário e o novo estado de uma transação t_m. Para o protocolo híbrido com execução

distribuída, essas informações incluem a identificação do cliente, a identificação do servidor primário, o novo estado, o *writeset* e o *readset* de uma transação t_m.

I. org.objectweb.dorothy.replication.DistributedState

A classe `DistributedState` foi adicionada ao código do JOnAS. Essa classe armazena o estado distribuído a ser enviado pela *thread* primária para as *threads* secundárias e é mantida localmente por cada membro do grupo de replicação em uma lista com todos os estados distribuídos.

Parte do conteúdo da classe `DistributedState` depende do tipo de *bean* e do uso ou não do serviço transacional pela aplicação. Em comum, a classe `DistributedState` sempre contém a identificação do servidor primário e a identificação do cliente (nome do cliente e identificação do *bean* no cliente).

Se a aplicação atualizar um *bean* de sessão com estado, a classe `DistributedState` armazena informações sobre um ou mais objetos do tipo `ReplicatedObject`, que representam o estado do *bean* de sessão. Se a aplicação atualizar um *bean* de entidade, a classe `DistributedState` guarda informações sobre o *writeset*, o *readset*, a identificação da transação em execução no servidor primário e, possivelmente, informações sobre um ou mais `ReplicatedObject`.

Observe que o `ReplicatedObject`, o *writeset* e o *readset* podem ser uma classe ou várias classes, contemplando tanto classes de tipos primitivos quanto classes definidas pelo usuário.

A classe `DistributedState`, e todo o seu conteúdo, precisa ser serializada na *thread* primária para que possa ser enviada para o grupo de replicação. Como a serialização implica na transformação de um objeto em um *stream*, informações importantes, como a classe do estado do *bean,* podem ser perdidas. Sendo assim, as informações sobre a descrição da classe do estado do *bean,* do *writeset* ou do *readset* também precisam ser guardadas na classe `DistributedState`. Por exemplo, se o `ReplicatedObject` for um inteiro e for serializado, quando esse inteiro for deserializado na *thread* secundária terá que ser recuperada a informação que descreve o valor desse objeto, o nome desse objeto e a classe desse objeto (que neste caso é *int*).

Outra opção de implementação do estado distribuído é usar a memória secundária, ao invés da memória principal, através dos métodos `ejbActivate()` e `ejbPassivate()` que são métodos padronizados da especificação EJB. Durante sua execução, uma instância de um *bean* obedece a um ciclo de vida: é gerada através do método `ejbCreate()`, usada através dos métodos funcionais definidos pelo desenvolvedor no *bean* e destruída através do método `ejbRemove()`.

Os *beans* também podem ser ativados e passivados pelo *container,* que controla o ciclo de vida dos *beans* de acordo com o seu uso e com o espaço disponível em memória. Esses mecanismos são implementados através dos métodos `ejbActivate()` e `ejbPassivate()`. Na ativação, o estado do *bean* é copiado do disco para a memória, e na passivação, o estado do *bean* é copiado da memória para o disco.

No servidor EJB iPlanet [SUN2000], a replicação do estado de um *beans* de sessão com estado ocorre em duas etapas: primeiro o *container* invoca o método `ejbPassivate()` para que o *bean* seja serializado e armazenado no disco, depois o *container* invoca o método `ejbActivate()` para que o estado do *bean* seja copiado do disco para a memória. Note que armazenar o estado do *bean* usando os métodos

`ejbPassivate()`e `ejbActivate()` é uma operação cara devido ao acesso ao disco, isso torna a estratégia usada neste trabalho (i.e., replicação em memória e não em disco) mais atraente.

J. org.objectweb.dorothy.replication.ImmediateUdpate

A classe `ImmediateUdpate` foi adicionada ao código do JOnAS. Essa classe coleta informações como a identificação do cliente, a identificação do servidor primário e o novo estado de um objeto que serão imediatamente enviadas pela *thread* primária ao grupo de replicação no protocolo híbrido com execução local.

K. org.objectweb.dorothy.replication.ReplicatedObject

A classe `ReplicatedObject` foi adicionada ao código do JOnAS. Essa classe armazena informação sobre os objetos que estão sendo usados pelos *beans* de sessão com estado ou pelos *beans* de entidade. A informação inclui o nome, a classe e o valor da variável de instância do objeto. A classe `ReplicatedObject` é serializada para que possa ser enviada da *thread* primária para as *threads* secundárias.

L. org.objectweb.dorothy.switchover.SwitchServer

A classe `SwitchServer` é usada para fazer o chaveamento do servidor falho para um servidor alternativo. O usuário da aplicação EJB somente percebe algum problema se todos os servidores do grupo de replicação estão inativos. A classe `SwitchServer` é a única classe adicionada como serviço do cliente. As demais classes adicionadas ou modificadas são implementadas como serviços do servidor.

A classe `SwitchServer` possibilita que o cliente automaticamente retransmita a requisição falha para um servidor alternativo que é especificado no arquivo `jonas.servers`.

Se o servidor alternativo também não está disponível, outro servidor candidato, descrito no arquivo `jonas.servers` é posto à prova, até que todos os servidores candidatos sejam testados. O arquivo `jonas.servers` descreve uma lista de nodos que podem estar executando servidores EJB candidatos a novo servidor primário. O arquivo `jonas.servers` é construído pelo administrador de sistema quando o serviço de replicação é instalado. Os nodos candidatos são nodos que podem conter instâncias de servidores EJB que compõem o grupo de replicação. Se o cliente realiza o chaveamento do servidor para um nodo descrito em arquivo `jonas.servers` que não está executando uma instância de servidor EJB, a operação que cria o *bean* neste nodo falha e um novo nodo candidato é posto à prova. Novos nodos candidatos podem ser testados sucessivamente, até que se obtenha sucesso, ou até que toda a lista seja percorrida. Se nenhum nodo descrito no arquivo `jonas.servers` estiver executando nenhuma instância de servidor EJB, então será impressa a mensagem '*serviço indisponível'* na console do cliente.

O serviço de chaveamento de servidor substitui o endereço IP do nodo que executa o servidor primário usado pelo cliente para acessar o grupo de replicação, por outro endereço IP, obtido a partir do servidor de nomes que é acessado através do servidor

JNDI. O servidor JNDI representa a interface padrão da plataforma Java para o serviço de nomes.

Para permitir o chaveamento de servidor, o administrador do sistema identifica, no arquivo chamado `jonas.servers`, os nomes de servidores candidatos a formarem o grupo de replicação. O arquivo `jonas.servers` possui o seguinte formato:

`rmi://<endereço_IP>:<número_da_porta>`

onde `endereço_IP` representa o endereço IP do servidor primário ou a palavra–chave *localhost* para indicar o próprio nodo que executa o servidor JNDI armazena o arquivo `jndi.properties`, e onde `número_da_porta` indica o número da porta que identifica o serviço. O valor padrão para a porta é 1099.

Para que o cliente execute o chaveamento de servidor, foram implementados dois métodos:

- `numberOfAvailableServers()`: retorna o número de nodos listados no arquivo `jonas.servers`;
- `getAlternativeServer(int i)`: recebe como parâmetro o número do próximo nodo candidato, retorna o IP desse nodo que está descrito na linha *i* do arquivo `jonas.servers`.

Uma pergunta que o leitor pode fazer é "porque não usar a informação da visão do grupo para evitar que o cliente tente acessar um nodo que está descrito no arquivo `jonas.servers` mas não faz parte do grupo?" Como a visão do grupo é uma informação privada do grupo de replicação, infelizmente não pode ser usada pelo cliente para selecionar um servidor alternativo em caso de falha no servidor atual.

5.2.3 Dificuldades de implementação

Muitas dificuldades de implementação foram encontradas ao longo deste trabalho. A principal delas foi a falta de documentação sobre a implementação do JOnAS apesar da ampla documentação sobre EJB.

Reportar as dificuldades de implementação possibilita que o leitor visualize os obstáculos enfrentados no desenvolvimento e na programação de uma solução, e também facilita a tomada de decisões para o desenvolvimento de trabalhos futuros.

Ausência de documentação do JOnAS

Atualmente, o JOnAS não possui documentação sobre o seu projeto. A documentação é apresentada como comentários incluídos no código fonte. Isso deve–se ao fato de que a implementação do JOnAS é mais voltada para usuários do que para desenvolvedores de serviços adicionais.

Compilação do código a cada modificação

Cada vez que o código fonte do JOnAS é alterado, durante a programação de um novo serviço, ele precisa ser novamente recompilado. Como o código fonte do JOnAS é extenso (aproximadamente 19M*bytes* na versão 2.4), o JOnAS usa o utilitário `gmake`, que facilita o processo de compilação. Entretanto, apesar de somente recompilar as classes modificadas, o `gmake` precisa percorrer todas as classes do JOnAS, a fim de

verificar as possíveis dependências entre as classes. Isso torna o processo de compilação bastante lento, onerando a programação de novas extensões, como a implementação dos serviços HA.

Tempo gasto para inicializar um servidor com os serviços HA

Cada vez que uma modificação é feita no código do JOnAS, o sistema precisa ser recompilado e o servidor JOnAS e o servidor JNDI precisam ser reinicializados para que as modificações sejam testadas.

O tempo gasto para inicializar uma instância do servidor JOnAS, ainda que o servidor não ofereça os serviços HA, é relativamente alto (figura 5.9). Reinicializar um servidor *dorothy* é mais oneroso que o serviço JOnAS convencional. Esse tempo justifica–se porque um servidor *dorothy* precisa realizar a troca de visão, além de inicializar todos os serviços EJB convencionais (segurança, gerenciamento transacional, etc.).

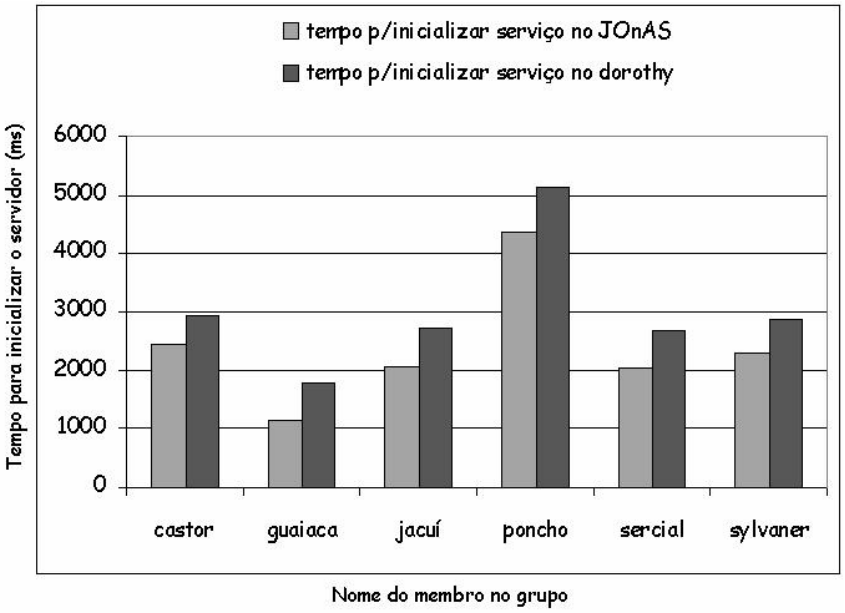

FIGURA 5.9 – Tempos para a inicialização do servidor EJB

Através de testes executados no servidor *dorothy*, concluiu–se que se gasta, em média, 28% de tempo a mais para se inicializar o servidor *dorothy*, do que para inicializar um servidor JOnAS convencional. Para a fase de desenvolvimento, esse acréscimo no tempo de inicialização é significativo, já que o servidor precisa ser reinicializado várias vezes até que a programação de um serviço adicional torne–se estável. Entretanto, quando o sistema está operacional, esse acréscimo no tempo de inicialização não precisa ser computado já que o serviço será inicializado uma vez e reinicializado somente em caso de parada do nodo devido à falha ou manutenção.

5.3 Avaliação experimental

A avaliação experimental, no contexto deste trabalho, busca analisar a degradação de desempenho gerada pelo protocolo de replicação e pelo sistema de comunicação de grupo do servidor *dorothy*.

Um protótipo é uma forma eficiente de estimar as restrições de disponibilidade e de desempenho de um sistema. O protótipo possibilita considerar todas as camadas de

sustentação de uma aplicação e permite alcançar os resultados próximos às condições operacionais reais de um sistema. A medida e a análise de resultados é uma forma de compreender as restrições de disponibilidade e de desempenho de um sistema computacional complexo. O protótipo funciona com uma carga de execução sintética e apropriada, simula a condição real de um sistema, e torna possível considerar gargalos do desempenho e da disponibilidade antes que o sistema real seja efetivamente construído, de modo que as restrições possam ser compreendidas e ajustadas nas fases mais iniciais de projeto de um sistema computacional.

Os experimentos visam mostrar que a replicação conduz à disponibilidade desejada, mas não perturba consideravelmente o desempenho das aplicações EJB, quando comparado com o tempo obtido com a execução da mesma aplicação em servidores de aplicação convencional. A avaliação experimental mostrada neste trabalho usa como base principalmente os experimentos executados nos trabalhos de Pedone *et al*. [PED99] e Mishra *et al.* [MIS99].

5.3.1 Descrição das aplicações

Desde que a implementação de um serviço com alta disponibilidade para aplicações EJB deve considerar diferentes tipos de *beans*, este trabalho apresenta um estudo comparativo usando diferentes aplicações EJB como carga de trabalho sintética. Essas aplicações contêm *beans* de sessão *com informações de estado* e *sem informações de estado*, e *beans* de entidade.

Grande parte das aplicações usadas nos testes é fornecida junto com o pacote do JOnAS. A aplicação que usa *beans* de entidade consiste em um sistema bancário simplificado. Uma tabela armazena dados como números de conta–corrente, nome do correntista e saldo de sua respectiva conta–corrente. Transações são necessárias para garantir que serviços funcionais, como a transferência de saldo entre contas–correntes, sejam executados com sucesso, mesmo na presença de acesso concorrente de clientes diferentes.

As aplicações que usam *beans* de sessão consideram os dois modelos desse tipo de *bean*: *com informação de estado* e *sem informação de estado*. A aplicação que mantém um *bean* com informação de estado usa o serviço transacional. A outra aplicação considera objetos atualizados sem transações.

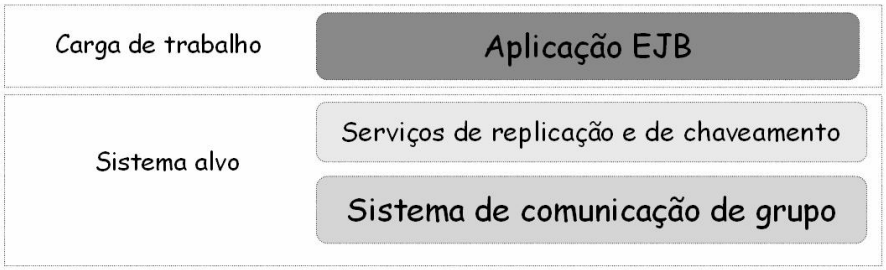

FIGURA 5.10 – Estrutura para a experimentação

Note que as aplicações aqui descritas são usadas como carga de trabalho (*workload*) (fig. 5.10). Não são as aplicações que estão sendo avaliadas, mas os serviços HA, onde o principal serviço é o serviço de replicação dos *beans*.

5.3.2 Análise de desempenho

Para medir a degradação de desempenho pela adição dos serviços HA em uma implementação da especificação EJB, foram considerados dois índices de desempenho: o *throughput* e o tempo de execução, calculados sobre as aplicações anteriormente descritas.

O *throughput* de um servidor pode ser calculado pelo número de aplicações que são executadas por unidade de tempo (hora) [SIL2000]. No contexto deste trabalho, o *throughput* de um grupo de servidores pode ser calculado através da média do número de aplicações que são executadas por unidade de tempo (hora) em cada servidor do grupo de replicação.

O tempo de execução para uma aplicação distribuída é o tempo computado no cliente desde o começo da execução de uma aplicação até o término da execução dessa aplicação.

Observe que as aplicações não são normalizadas, i.e., as medidas obtidas para os índices (*throughput* e tempo de execução) servem somente para fins de comparação. Em outras palavras, as medidas não são absolutas, mas relativas. No contexto deste trabalho, para uma comparação, tipicamente foi executada a mesma aplicação em um servidor sem réplicas e depois em um grupo de replicação com diferentes quantidades de membros.

Throughput

O *throughput* foi medido considerando grupos com diferentes quantidades de servidores. A figura 5.11 mostra o *throughput* médio para duas aplicações com diferentes tipos de *beans*. O estado dos *beans* é implementado através do *protocolo de término* [PED98]. Para uma comparação, foi medido o tempo de execução para o servidor EJB convencional (*castor* na figura 5.11), sem replicação e sem comunicação de grupo.

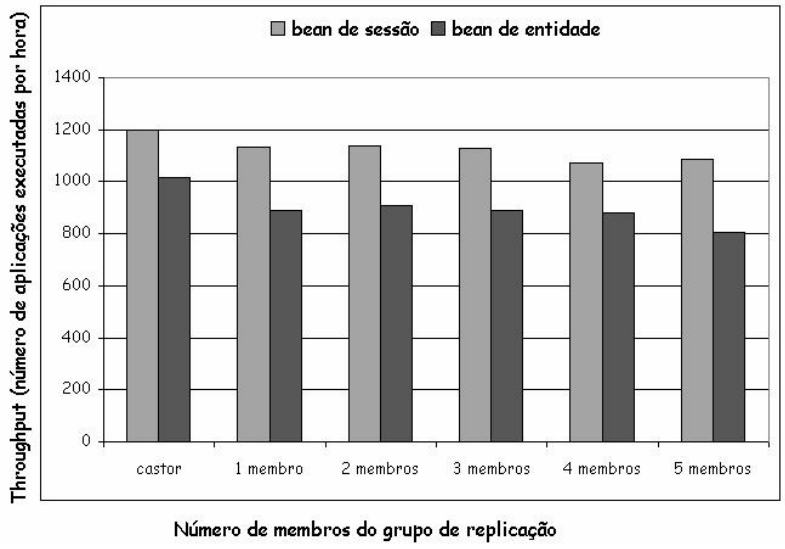

FIGURA 5.11 – *Throughput* para uma aplicação com objeto *com informação de estado* e com objeto persistente

Cada valor mostrado na figura 5.11 foi obtido pela média de execução de 20 vezes com um cliente que emite pedidos de leitura e de escrita em um sistema transacional hipotético usando uma carga de trabalho sintética.

No gráfico, pode–se observar que a degradação de desempenho para a replicação com objeto *com informação de estado* foi, no pior caso (com 4 e 5 membros no grupo), em média, aproximadamente 10% em relação ao tempo de execução da mesma aplicação em um servidor JOnAS convencional, enquanto que a degradação de desempenho para a replicação como objeto persistente foi, no pior caso (com 5 membros no grupo), em média, aproximadamente 20% em relação ao tempo de execução da mesma aplicação em um servidor JOnAS convencional.

No gráfico da fig. 5.11, os valores obtidos para o *throughput* são aproximadamente iguais para uma determinada aplicação (que, neste caso, implementa o *protocolo híbrido com serviço distribuído*). Embora o *throughput* máximo ocorra para o serviço EJB sem alta disponibilidade, o impacto da adição dos serviços HA do servidor *dorothy* não é superior a 20%.

Tempo de execução

Os valores para o tempo de execução são obtidos usando o método `getTime()` da linguagem Java no lado do cliente. O cliente está realmente computando o tempo que uma determinada sessão precisa para ser finalizada.

A figura 5.12 mostra o tempo de execução em *ms* (em *milisegundos*) para uma aplicação com atualização imediata usando as técnicas ávida e preguiçosa para um nodo e para diferentes tamanhos de grupo (i.e., *protocolo híbrido com serviço local*). A aplicação usada como *carga de trabalho* atualiza um *bean* de sessão *com informação de estado*. Observe que a propriedade de ordem não é requerida para este tipo de objeto. A medida obtida com um nodo, sem replicação foi obtida na estação *castor*.

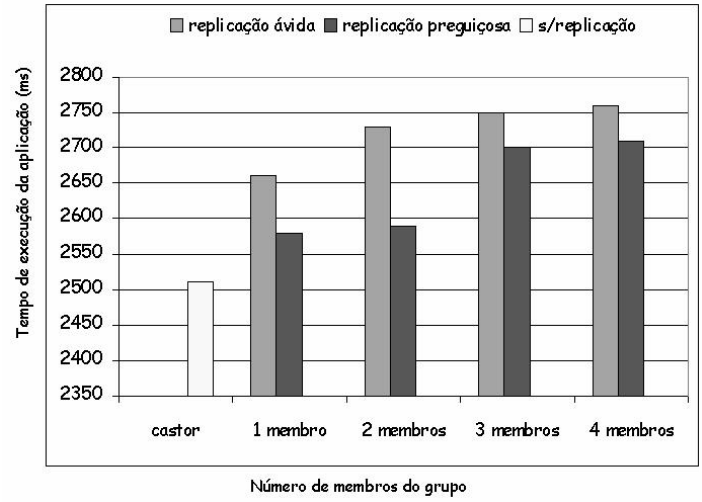

FIGURA 5.12 – Tempo de execução para uma aplicação
com objeto *com informação de estado*

A figura 5.13 mostra o tempo de execução (em *ms*) para uma aplicação que usa transações para manipular um objeto persistente (i.e., *protocolo híbrido com serviço distribuído*), que é usada como carga padrão. Observe que a propriedade de ordem total é requerida para evitar os possíveis conflitos devido à concorrência de acesso aos

múltiplos bancos de dados por clientes distintos. A medida obtida com um nodo foi salientada na figura e foi usada como comparação.

Cada valor mostrado nas figuras 5.12 e 5.13, foi obtido pela média de execução de 12 vezes[18] com um cliente que emite pedidos de leitura e escrita em um sistema transacional hipotético usado uma carga de trabalho sintética. Em ambos experimentos a primeira medida, nomeada *um nodo*, significa sem replicação e sem comunicação de grupo, i.e., o serviço EJB convencional. Esta medida, em cada figura, é usada como base de comparação, e é uma medida essencial sem a qual as demais medidas ficam sem sentido. As outras medidas são obtidas variando o número de membros em um grupo de replicação.

A medida nomeada como *um membro* em ambas as figuras mostra somente a degradação do desempenho devido ao serviço de grupos. Neste caso, de fato, não há comunicação de grupo, como no caso da primeira medida. A degradação de desempenho ocorre porque o primário precisa enviar mensagens do grupo de replicação para ele mesmo.

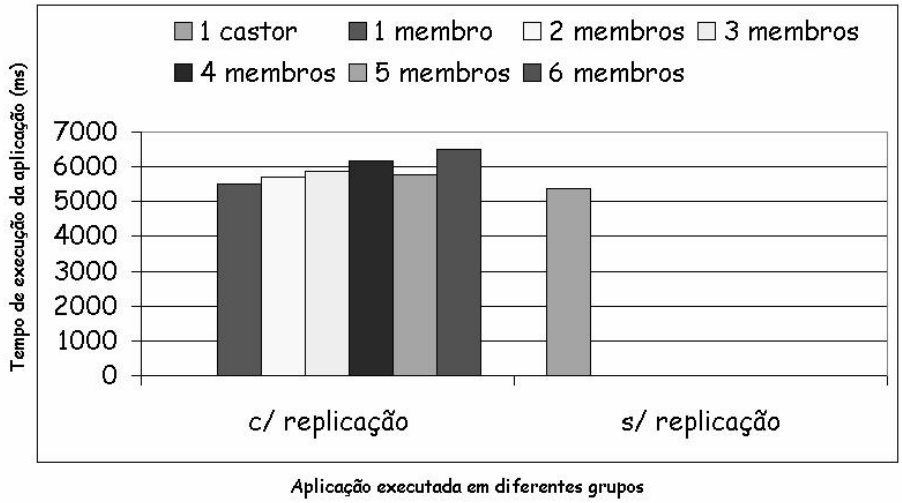

FIGURA 5.13 – Tempo de execução para uma aplicação com objeto persistente

Os serviços de replicação e de grupos, aparecem, efetivamente, com dois ou mais membros, cada um mantendo uma réplica. As medidas mostram uma curva ascendente em ambas figuras (fig. 5.12 e 5.13), mas que tende rapidamente a saturar para as aplicações com objetos com informação de estado (fig. 5.12).

Como pode ser notado, a degradação de desempenho para assegurar alta disponibilidade para objetos com informação de estado é realmente pequena, variando de 3% a 10%, em média, no tempo de execução (figura 5.12). Para objetos persistentes, a penalidade é mais elevada, de 10% a 25% (figura 5.13). Mas mesmo no pior caso analisado, i.e., um grupo de replicação com 6 réplicas, a degradação de desempenho é aceitável para aplicações altamente disponíveis.

[18] os valores foram computados em *ms* (milisegundos) e depois foram convertidos para *s* (segundos), que é uma medida mais apropriada para o tempo de execução de uma aplicação. Cada experimento é executado apenas 12 vezes pois o tempo da execução obtido nas reexecuções das aplicações mostrou-se muito linear, quando medido em *s*.

5.3.3 Análise da disponibilidade

Johnson [JOH96] define a disponibilidade como a probabilidade que um sistema esteja operando corretamente e esteja disponível para executar seu serviço no instante de tempo *t*. Um sistema pode ser altamente disponível mesmo que apresente períodos freqüentes de inoperabilidade se estes períodos forem extremamente curtos.

Ainda, de acordo com Johnson [JOH96], a disponibilidade de um sistema depende *não somente da freqüência com que ele se torna inoperante*, mas da *rapidez do seu reparo.* Técnicas que permitam o reparo eficiente e automático são indispensáveis em sistemas altamente disponíveis.

A disponibilidade (*availability*) de um sistema computacional em um instante de tempo A (*t*), pode ser determinada em termos dos índices MTTF (*mean time to failure*) e MTTR (*mean time to repair*) [JOH96, TRI82]:

$$A(t) = \frac{MTTF}{MTTF + MTTR}$$

Apesar da degradação de desempenho, a arquitetura HA–EJB permite diminuir o MTTR (*mean time to repair*) das aplicações EJB em caso de falha. Observando–se a fórmula para calcular A (*t*), é claro que se o MTTR tende a zero, a disponibilidade A (*t*) tende a 1 (ou 100%).

Assim sendo, os serviços HA aqui propostos buscam reduzir ou evitar o longo tempo de recuperação, mantendo uma réplica do objeto *on–line* para prestar serviços aos clientes mesmo quando o grupo de replicação perde um servidor devido à falha.

A especificação EJB suporta serviços transacionais sobre uma infra–estrutura de comunicação não–confiável, e grande parte das implementações dessa especificação implementa a semântica de execução do melhor esforço (ou *best–effort*). Na semântica do melhor esforço, o cliente emite a mensagem, e o cliente e a infra–estrutura não tentam retransmissões em caso de falha. Essa semântica não é suficiente para serviços altamente disponíveis. Serviços altamente disponíveis requerem uma aproximação mais sofisticada na presença de falhas de servidor.

No experimento mostrado na figura 5.14, existem três clientes, *castor*, *poncho* e *sercial*, cada um executando em uma estação diferente com arquiteturas de *hardware* diferentes. Os clientes compartilham o mesmo grupo de replicação. As medidas (em *ms*) foram computadas enquanto os clientes executavam concorrentemente a mesma aplicação. Neste experimento, os objetos mantidos no grupo de replicação são *sem informações de estado*.

O experimento considera 8 servidores candidatos listados no arquivo `jonas.servers`. Os clientes executam o chaveamento sobre os servidores listados neste arquivo até encontrarem um servidor que possa responder a sua requisição. A falha, de um a quatro servidores, introduzida manualmente, ocorre na primeira requisição de cada cliente e requer de um até quatro chaveamentos.

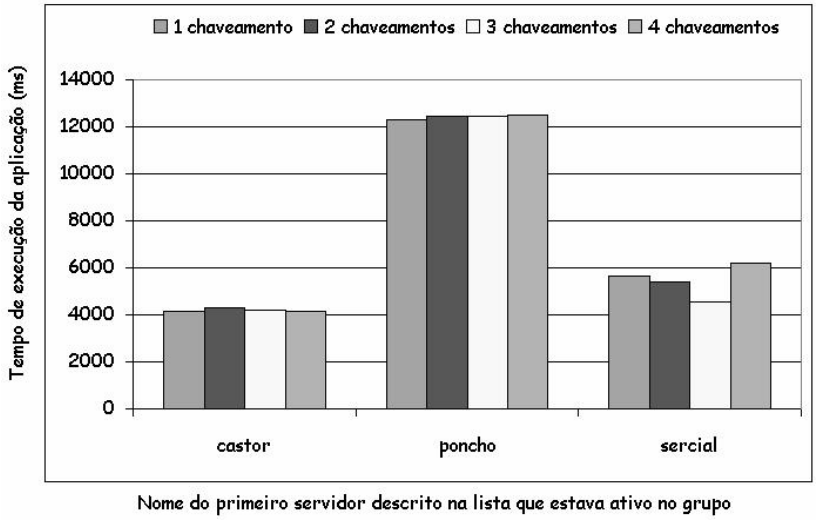

FIGURA 5.14 – Tempo de execução com o serviço de chaveamento

O tempo de execução foi medido na primeira requisição do serviço em cada cliente até que o primeiro servidor do grupo responda e termine ao serviço. Observe que o tempo de execução é praticamente constante para um dado cliente. O caso onde o chaveamento não é fornecido, não é mostrado na figura 5.14. Cada experimento foi repetido 12 vezes para cada caso e, então, a média foi computada.

Como as figuras 5.9 e 5.14 mostram, clientes de diferentes arquiteturas possuem tempos de execução diferentes. O número de servidores falhos não afeta consideravelmente os valores obtidos.

Na fig. 5.14, o tempo de execução, medido no lado do cliente, incluindo o chaveamento do servidor, está na escala de 4*s* a 12.5*s*. Esta medida possui aproximadamente a mesma ordem do tempo de execução medido nas figuras 5.12 (2 a 3*s*) e 5.13 (5 a 7*s*).

Na arquitetura aqui proposta, alta disponibilidade é alcançada através do serviço de chaveamento usando servidores que mantêm réplicas de objetos. O tempo de chaveamento, que pode ser visto no lado do cliente como um atraso no tempo de execução, equivale ao MTTR, se o objeto for sem estado. Caso contrário, o MTTR precisa considerar o tempo de inicializar o objeto com o estado previamente armazenado do servidor alternativo.

Quando o chaveamento não for fornecido e um servidor falhar prestando serviços a um cliente, o usuário percebe o serviço como momentaneamente indisponível. O cliente deve esperar, freqüentemente um intervalo de tempo na ordem de minutos ou mais, até que o servidor se recupere. Esse valor corresponde a um MTTR inviável em um sistema altamente disponível.

Entretanto, se o chaveamento é oferecido, o cliente não percebe o serviço como indisponível, embora o tempo de execução seja afetado. Nesse caso, o cliente não necessita esperar minutos, apenas alguns segundos, em média.

O tempo do *failover* pode ser considerado como o tempo de indisponibilidade do serviço. O tempo do *failover* é o tempo usado para recuperar o serviço, uma aproximação do tempo médio de reparo, MTTR. É assumido que no mínimo um servidor permanece operacional, enquanto servidores falhos estão sendo reparados. O tempo do chaveamento também pode ser considerado como uma outra penalidade ao desempenho. Mas ao contrário das medidas mostradas nas figuras 5.11 e 5.12, esta

penalidade aparece somente quando uma falha por colapso ocorre. A freqüência de falha é geralmente pequena e afeta somente poucas requisições ao longo de um período de tempo significativo. Assim, essa penalidade do desempenho não afeta efetivamente o desempenho total.

Os experimentos mostram que os serviços HA implementados pelo protótipo *dorothy* conduzem à alta disponibilidade, mantendo o MTTR na ordem dos segundos. Entretanto, esses experimentos não são suficientes para validar a *dependabilidade* total da solução apresentada neste trabalho. Para alcançar uma validação mais completa, experimentos detalhados com injeção de falhas são necessários.

5.4 Aspectos de implementação

Descrever aspectos de implementação verificados ao longo do desenvolvimento e implementação dos serviços adicionais para uma arquitetura possibilita apontar características fundamentais atreladas a essa arquitetura. Observe que os aspectos de implementação aqui descritos não são restritos aos serviços desenvolvidos neste trabalho, e podem ser adaptados a qualquer novo serviço para a especificação EJB.

5.4.1 Compatibilidade com o modelo (baseado em componentes) da arquitetura original

Um sistema implementado com a abstração de componentes permite que um serviço não–funcional seja inserido de forma transparente para a aplicação. Os serviços de replicação e de chaveamento de servidor aqui desenvolvidos mantêm a compatibilidade com os serviços descritos na especificação EJB. O desenvolvedor de aplicações EJB continuará a construir suas aplicações de forma convencional. Os serviços adicionais são inseridos nas aplicações EJB com total transparência para o desenvolvedor.

5.4.2 Serviço explícito e implícito

O controle transacional de uma aplicação EJB pode ser gerenciado pelo *componente* ou pelo *container* [KAS2000]. Na arquitetura HA, a escolha de quais métodos serão replicados também poderá ser feita explicitamente pelo desenvolvedor (*component–managed replication*) ou implementada automaticamente pelo *container* (*container–managed replication*).

O protótipo ainda não implementa uma semântica para que a replicação seja explicitada pelo desenvolvedor. Apenas a replicação automática é considerada.

Uma sugestão de implementação da replicação explícita, é que o desenvolvedor da aplicação distribuída explicite, em tempo de desenvolvimento, quais métodos devem ser replicados no descritor do *bean*. Se no descritor do *bean* (figura 5.15) for escolhida a opção *container–managed replication* (linha 1) com replicação em todos os métodos, basta usar o valor "*" (linha 4). Observe que, neste caso, o serviço de replicação escolhe automaticamente quais métodos serão replicados. Somente métodos de atualização de estado de objetos EJB serão replicados.

```
1.  <container-replication>
2.      <method>
3.        <ejb-name>Op</ejb-name>
4.        <method-name>*</method-name>
5.      </method>
6.      <replic-attribute>DeferredUpdate</replic-attribute>
7.   </container-replication >
```

FIGURA 5.15 – Descritor de um *bean* com replicação

Entretanto, métodos também podem ser listados um a um no descritor do *bean.* Isso significa que métodos que não forem definidos pelo desenvolvedor no descritor como replicáveis, não terão sua execução replicada. O *default* é que a replicação não seja habilitada.

5.4.3 Codificação automática de replicação nos *beans*

Atualmente, no servidor *dorothy*, a classe do *bean* precisa ser editada para que o estado do *bean,* que deve ser propagado, seja assinalado pelo desenvolvedor. Entretanto, uma ferramenta para permitir a realização automática dessa tarefa será futuramente implementada.

O JOnAS transforma os *beans* codificados por um desenvolvedor em componentes interpretáveis através de uma ferramenta chamada GenIC (fig. 5.16). O GenIC é formado a partir da combinação de dois compiladores: o `javac` e o `rmic`, para gerar, respectivamente, o código Java e código distribuído (com invocação remota de métodos). No processo de geração de código com o GenIC, primeiramente, é o compilador `javac` que realiza a verificação da classe do *bean* (fig. 5.16, 1). A seguir, o código do *bean* é alterado para inserir algum código de controle para o serviço não–funcional (2). Então, o GenIC gera *stubs* e *skeletons* para os objetos remotos usando o `rmic` (3). Para finalizar, o GenIC compila essas classes usando o `javac` (4).

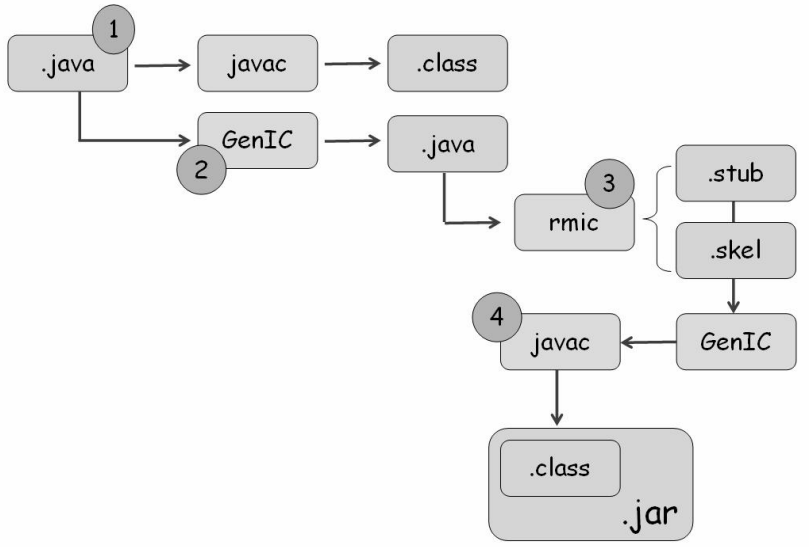

FIGURA 5.16 – Geração de um *bean* no JOnAS

O arquivo de entrada é ou um arquivo `ejb-jar` (arquivo que representa várias classes Java) ou o nome do descritor do *bean*. Se o arquivo de entrada é um arquivo `ejb-jar`, o GenIC adiciona as classes geradas neste arquivo `ejb-jar`.

Para inserir o código dos serviços HA em um *bean*, a sugestão aqui apresentada é que seja aproveitada essa estrutura de geração de código já pronta no GenIC. A diferença fundamental para o processo de inserção de código para a replicação e do código do serviço não–funcional, implementado pelo GenIC é que o GenIC manipula principalmente as interfaces (remota e *home*) de um *bean*, enquanto que a ferramenta para inserir o código da replicação deve atuar principalmente sobre a classe do *bean*, que detém informação sobre o estado do *bean*. Um mecanismo análogo também pode ser usado para inserir o código no cliente para permitir o chaveamento de servidor.

5.5 Servidores EJB com alta disponibilidade da indústria de *software*

Diversas implementações de servidores EJB estão disponíveis reportando a grande popularidade da especificação EJB na indústria de *software*. Essas implementações diferem em muitos aspectos como custo, tipo de suporte, dificuldade de manutenção, etc. Muitas dessas implementações são sistemas proprietários e o seu código fonte não é disponível, o que limita novas extensões e um estudo acadêmico mais aprofundado.

Entretanto, algumas dessas implementações disponibilizam seu código fonte, como o JOnAS e o JBoss [BUR2002]. Outras implementações enfatizam serviços que asseguram alta disponibilidade, como o HP Bluestone Total–e–Server 7.2.1, o Sybase Enterprise Application Server 3.6, o SilverStream Application Server 3.7 e o BEA WebLogic Server 6.0. Kang [KAN2001] apresenta uma comparação entre esses servidores, que têm em comum o mapeamento de um grupo de servidores para um *cluster*, onde um serviço mais robusto que a especificação descreve é oferecido. No escopo deste trabalho, uma rápida revisão desses servidores parece ser oportuna. Os servidores iPlanet (da Sun Microsystems) [SUN2000] e JBoss 3.0 foram adicionados a esta revisão.

HP Bluestone Total–e–Server 7.2.1

O principal diferencial do HP Bluestone Total–e–Server 7.2.1 [HPB2001][19] é ter um servidor JNDI independente em cada nodo e fazer com que cada requisição de cliente passe por um *proxy* LBB (*load balance broker*). O *proxy* garante que cada requisição é executada por uma instância ativa do grupo de replicação. Esse esquema adiciona latência extra a cada requisição, mas permite *failover* automático.

Sybase Enterprise Application Server 3.6 e SilverStream Application Server 3.7

Sybase Enterprise Application Server 3.6 [SYB2001][20] e o SilverStream Application Server 3.7 [SIL2001][21] implementam *failover* através de um banco de dados centralizado, onde todos os servidores do grupo de replicação que executam em *cluster* realizam leituras e escritas. O banco de dados centralizado facilita a implementação do tratamento de concorrência entre clientes, mas pode ser um gargalo se o número de

[19] a versão mais recente do HP Bluestone Total–e–Server é a 7.3, de 5 de março de 2001
[20] a versão mais recente do Sybase Enterprise Application Server é a 4.1.2
[21] SilverStream Application Server atualmente é um produto da Novell, e chama–se Novell exteNd Application Server 4.0

requisições de clientes for grande ou se o grupo de replicação comportar muitos servidores.

BEA WebLogic Server 6.0

O BEA WebLogic Server 6.0 [BEA2000][22] provê alta disponibilidade através do protocolo de cópia primária–*backup* em um *cluster* de servidores. Quando o servidor primário falhar ocorre o *failover*, o *stub* do cliente automaticamente redireciona as requisições para um servidor secundário. O secundário, que passa a ser o novo primário, gera uma nova instância do *bean* usando o estado anteriormente replicado. Depois do *failover*, o novo servidor primário escolhe um servidor no *cluster* para trabalhar como secundário. Os endereços do novo primário e do novo servidor secundário são automaticamente alterados no *stub* do cliente durante a próxima execução de método no *bean*.

O *failover* para *beans* de sessão com estado, no BEA WebLogic Server, é implementado através do mecanismo de replicação *in–memory*. Esse mecanismo copia o estado de uma instância (da memória volátil) de um servidor primário para outra instância em um servidor secundário.

No BEA WebLogic Server 6.0, as atualizações de estado de *beans* no primário são aplicadas ao secundário através da *atualização adiada*, ou depois que cada método é invocado, quando o *bean* não usa transações. Quando o primário falhar antes de propagar atualizações, o cliente realizará o *failover* para o último estado armazenado no secundário.

No BEA WebLogic Server 6.0 não assegura a semântica de execução *exatamente uma*. Embora incomum, o último estado confirmado de um *bean* pode ser perdido em caso de falha no servidor primário antes de propagar uma atualização de estado para o secundário, mas depois de realizar a confirmação de uma transação.

iPlanet

No servidor iPlanet [SUN2000] da Sun Microsystems, o *failover* em *beans* de sessão com estado combina *stubs* inteligentes e uma solução proprietária chamada *distributed store* (DSync), não descrita adequadamente para fins acadêmicos na bibliografia consultada. Entretanto, o DSync parece ser semelhante ao mecanismo de replicação *in–memory*. O compilador dos *beans* do iPlanet gera *stubs* especiais que realizam o *failover* automático. A cada invocação do *bean*, o *stub* do cliente pode detectar falhas no servidor corrente e realizar o *failover*, se necessário. Depois do *failover*, o *stub* invoca novamente o método da requisição falha. Segundo a descrição desse serviço no iPlanet, o *container* garante a semântica de execução *no máximo uma* para métodos reexecutados em caso de falha, embora a bibliografia consultada não descreva informação detalhada de como essa semântica é implementada. A *difusão de escritas* no servidor iPlanet é feita em intervalos regulares.

[22] a versão mais recente do BEA WebLogic Server é a 8.1

JBoss 3.0

O JBoss 3.0 [BUR2002] é uma implementação extremamente interessante no escopo deste trabalho porque também usa os serviços do JavaGroups para permitir alta disponibilidade. Entretanto, apesar de usar a mesma idéia deste presente trabalho, o desenvolvimento do serviço de alta disponibilidade para o JBoss 3.0 começou bem depois das primeiras publicações do serviço do servidor *dorothy*. Uma diferença principal entre as duas implementações é que a solução do JBoss 3.0 é voltada para a indústria de *software* e é desenvolvida por um grupo de pesquisadores, enquanto que a solução aqui apresentada é voltada para o ambiente acadêmico. A descrição desse serviço do JBoss pode ser obtida através da compra da documentação e, portanto não representa um avanço no estado da arte. A solução proposta para o servidor *dorothy* é pública. A implementação do serviço de alta disponibilidade do JBoss 3.0 usando o JavaGroups contribuiu para demonstrar a factibilidade da proposta desta tese. Contudo, o JBoss 3.0 ainda não implementa um serviço de alta disponibilidade para *beans* de entidade.

No JBoss 3.0, o serviço que mantém réplicas para os *beans* de sessão é chamado de *distributed state service*. A idéia da implementação desse serviço é usar o JavaGroups para distribuir o novo estado do *bean* aos demais servidores de um *cluster*.

Proxies inteligentes provêm balanceamento de carga e *failover* automático para o JBoss 3.0. Os *proxies* gerenciam uma lista de conexões RMI para os servidores EJB. Se algum cliente está acessando um servidor que falha durante a execução do serviço, uma nova visão é instalada para excluir o membro falho do grupo de servidores, mantido pelo JavaGroups.

5.6 Observações do capítulo

Este capítulo documenta a implementação de um protótipo chamado servidor *dorothy* que implementa os serviços HA. Os serviços de replicação e de chaveamento, que fazem parte dos serviços HA, são descritos através de protocolos no capítulo 4.

A próxima etapa de desenvolvimento consiste em deixar o servidor *dorothy* mais completo e robusto, codificando serviços que ainda são feitos manualmente como a inserção de *ganchos* no arquivo com a classe de um determinado *bean*, e o suporte a uma maior gama de tipos de estado de objetos replicáveis. Também falta propor e implementar um mecanismo que gere o *readset* e o *writeset* de forma automática para cada transação.

Entretanto, apesar de suas atuais carências de implementação, o protótipo conduziu a resultados realmente satisfatórios em termos de degradação de desempenho e de MTTR, que permitem concluir que a inclusão de serviços, que possibilitam alta disponibilidade suportados por uma camada de comunicação de grupo, é realmente viável.

6 Conclusões finais e trabalhos futuros

Muita pesquisa foi feita integrando comunicação de grupo e sistemas transacionais [AMI94, HOL99, HOL99a, LIT2000, PED99, e SCH96]. Alguns trabalhos [HOL99, HOL99a e PED99] apresentam experimentação com protocolos de replicação, enquanto outros trabalhos [AMI94, LIT2000, PAT2001 e SCH96] permanecem dentro dos modelos conceituais. Contudo, a pesquisa e a experimentação de protocolos de replicação aplicadas a sistemas práticos não é oferecida em nenhum desses trabalhos de modo que os problemas da replicação, como o baixo desempenho supostamente adicionado pelo protocolo de replicação, sejam melhor compreendidos.

O presente trabalho contribui usando pesquisa e experimentação para investigar como uma arquitetura prática e transacional pode ser beneficiada por abstrações de comunicação de grupo através da inclusão de *serviços adicionais para alta disponibilidade* (ou serviços HA). Para concluir este trabalho, serão destacadas as principais vantagens do uso das abstrações de comunicação de grupo para permitir alta disponibilidade em sistemas transacionais, e os resultados obtidos através de experimentação da solução aqui proposta. A experimentação possibilitou verificar, na prática, a viabilidade da solução apresentada. Trabalhos futuros incluem tratar serviços ainda não discutidos e/ou implementados para a arquitetura, proposta neste trabalho, como um mecanismo que garanta a implementação semântica de execução *exatamente uma* apesar de falha.

Neste trabalho, a especificação EJB, baseada em componentes, serviu para materializar os serviços HA. Porém, os serviços HA podem ser aplicados a outros sistemas de desenvolvimento de aplicações distribuídas. A experimentação mostra que a degradação do desempenho em sistemas que usam replicação sobre uma camada de comunicação de grupo não são demasiadamente elevadas. Comunicação de grupo é uma abstração adequada para desenvolver um sistema altamente disponível mesmo para arquiteturas transacionais. A solução apresentada mantém a compatibilidade com a arquitetura baseada em componentes original, e a invisibilidade das réplicas e de falhas para o usuário.

6.1 Conclusões finais

A solução apresentada neste trabalho proporciona alta disponibilidade para as aplicações distribuídas suportadas por uma arquitetura prática baseada em componentes, mantendo compatibilidade com a arquitetura original. A solução integra réplicas na especificação EJB através de abstrações de comunicação de grupo. A replicação é uma técnica que permite alta disponibilidade e pode ser implementada com eficiência através de abstrações de grupo. Apesar de fornecer os serviços de gerência transacional e de persistência de objetos, a especificação EJB não descreve nenhum serviço para alta disponibilidade.

O mapeamento das abstrações de comunicação de grupo para arquiteturas baseadas em componentes já construídas não é trivial. Estratégias para minimizar ainda mais os custos de comunicação e que permitam para adicionar com maior facilidade e robustez serviços adicionais são ainda requeridas, evitando complexidade e identificando gargalos do desempenho. Entretanto, o maior desafio do mapeamento, neste contexto, é preservar a estrutura já existente considerando a resistência das abstrações providas pela especificação EJB (que não descreve replicação).

Considerações de projeto

Algumas medidas para minimizar custos de comunicação foram consideradas no projeto da arquitetura HA, como excluir os clientes do grupo da replicação para evitar a mudança freqüente da visão do grupo. Propagar somente atualizações aos servidores secundários também minimiza custos de comunicação pois reduz a troca de mensagem entre os servidores.

Permitir que um cliente contate qualquer membro do grupo de replicação usando múltiplos servidores primários concorrentemente evita gargalos, um inconveniente típico do protocolo de cópia primária. Por outro lado, requer uma primitiva de comunicação de grupo mais onerosa em termos de desempenho do que se fosse usado simplesmente o protocolo de cópia primária.

Para sincronizar réplicas, este trabalho propõe um protocolo híbrido, que combina os protocolos de *cópia primária* e de *réplicas ativas*. Este protocolo possui duas variações: *protocolo híbrido com serviço local* e *protocolo híbrido com serviço distribuído*. O protocolo híbrido com serviço distribuído (todas as réplicas processam o serviço) requer mais processamento que o protocolo híbrido com serviço local (apenas uma réplica processa o serviço), mas permite sincronizar réplicas em bancos de dados, que aqui são implementados como COTS.

Considerações sobre a implementação

Para que os problemas da integração de réplicas na especificação EJB usando comunicação de grupo fossem compreendidos, foi implementado um protótipo chamado *dorothy*. O protótipo provê os serviços HA, que incluem o *serviço de replicação*, o *serviço de chaveamento de servidor*, o *serviço de grupos* e o *serviço de detecção de falhas*. Os dois últimos serviços são implementados por um sistema de comunicação de grupo [BAN99].

A implementação dos serviços de replicação e de chaveamento consiste principalmente em 7 novas classes agrupadas em um pacote e mais 5 classes do JOnAS que foram modificadas para coletar informação necessária à replicação. Atualmente, o protótipo executa sobre uma rede Ethernet com estações SPARC, executando o sistema operacional Solaris, mas que, de acordo com as características da linguagem Java, pode rodar sobre outras plataformas.

Embora a implementação dos serviços HA tenha sido feita sobre uma arquitetura prática, o protótipo *dorothy* não é um *software* de qualidade industrial. O protótipo é uma implementação onde a viabilidade da integração de transações e comunicação de grupo pode ser certificada em um sistema prático. Observe que o protótipo *dorothy* é uma das implementações possíveis para a arquitetura HA aqui descrita. Outros desenvolvedores são encorajados a oferecer outras implementações.

No sentido de tornar o protótipo mais robusto e próximo a um *software* de qualidade industrial, a implementação de serviços HA está sendo continuada por Marcos Didonet Del Fabro no Grupo de Pesquisa Rainbow no ESSI (Sophia Antipolis, França) sob orientação do Prof. Michel Riveill. Esse trabalho baseia-se fortemente nos resultados desta tese e na implementação do protótipo aqui mostrado.

Compatibilidade com a arquitetura original

A compatibilidade entre os serviços HA e os serviços já implementados pela arquitetura-alvo é mantida principalmente através do uso das abstrações de componentes, da configuração de serviços através de arquivos XML e da preservação das características da interface cliente–servidor da arquitetura original. A configuração de serviços através de arquivos XML permite que o desenvolvedor escolha propriedades que a aplicação deve assegurar. Por exemplo, apenas o serviço de chaveamento pode ser necessário para uma determinada aplicação.

A preservação das características da interface cliente–servidor possibilita que as réplicas sejam implementadas com total transparência, sem comprometer os demais serviços EJB. Por exemplo, o serviço de segurança, que atua na autenticação de mensagens cliente–servidor é preservado. A arquitetura HA assume que se uma mensagem é autenticada entre um cliente e um servidor primário s_i, então essa mensagem é autenticada por todos os secundários do grupo de replicação, já que o grupo é modelado como uma entidade única (i.e, um único componente).

Comunicação de grupo

Embora o serviço transacional garanta estados seguros para objetos apesar de falhas, a especificação EJB não oferece um serviço com alta disponibilidade. Tal serviço pode ser adicionado usando–se réplicas.

Comunicação de grupo representa uma abstração conveniente para sincronizar réplicas e possibilita tratar um modelo de falhas menos restritivo que o modelo usado por sistemas transacionais. Segundo Vaysburd [VAY98, p.66–67], a abstração de grupos é baseada em protocolos não–bloqueantes, mais resistentes a falhas que os bloqueantes, usados tipicamente no serviço transacional. Entretanto, a abstração de grupos ainda representa mais um modelo teórico que um modelo prático, apesar de ser amplamente difundida. Contudo, alguns esforços têm sido feitos para popularizar essa abstração e mostrar sua eficiência, como a implementação do serviço de *failover* de sessão para o servidor JBoss [BUR2002], a integração do TomCat com o JavaGroups [BUR2002, HAN2002], e o desenvolvimento deste trabalho.

Desempenho dos serviços HA

Para avaliar o impacto dos serviços de grupos, de replicação e de chaveamento no JOnAS, um protótipo (*dorothy*) foi implementado e foi medido o desempenho de aplicações EJB em rede local de estações SPARC. Os resultados obtidos demonstram que essa solução é promissora, embora apresente entraves de desenvolvimento, que não são visíveis na fase de projeto pois são ocultados pelas abstrações de orientação a componentes.

Consistência de réplicas

É mais difícil preservar o estado de um objeto em todos os cenários de falha possíveis quando esse objeto tem estado persistente do que quando esse objeto tem estado volátil ou não tem estado. Preservar o estado de objetos com estado persistente exige operações de acesso a tabelas de bancos de dados, armazenadas em memória estável, enquanto que objetos com estado volátil usam operações que manipulam somente a memória principal, tipicamente menos onerosas.

É necessário um mecanismo robusto para assegurar consistência a objetos com estado persistente. Sistemas de comunicação de grupo foram projetados para serem usados em sistemas distribuídos, e não em sistemas de banco de dados. A idéia original dos sistemas de comunicação de grupo é que o grupo mantenha consistência para estados voláteis, sem considerar os serviços transacionais e estados persistentes.

Tipicamente, o mecanismo de *transferência de estado* implementado por sistemas de comunicação de grupo considera apenas estados armazenados em memória principal. A implementação de um serviço de consistência de réplicas com comunicação de grupo mais abrangente requer o desenvolvimento de protocolos de replicação para objetos persistentes que considere serviços transacionais e requisições concorrentes, executadas em múltiplos bancos de dados. O tempo de duração da transferência de estado pode mudar de *ms*, duração típica para a transferência de estados voláteis, para alguns minutos ou até mesmo horas, duração típica para a transferência de estado em sistemas de bancos de dados.

Detecção de falhas

Na arquitetura HA, as falhas são detectadas por dois diferentes esquemas: um para o cliente e outro para o servidor, já que assume-se que o banco de dados e o servidor de aplicação estão localizados no mesmo nodo e que o modelo de falhas tratado é colapso em nodo (portanto, a falha de um resulta na falha de outro).

Falhas em nodos com membros do grupo de replicação são detectadas pelo serviço de detecção de falhas do JavaGroups ([BAN99], p.66). Nesse serviço, as falhas em membros de um grupo são detectadas pela consulta aos membros vizinhos, que é realizada periodicamente. A informação de quais são os vizinhos de um determinado membro é mantida pelo JavaGroups através de um anel virtual.

Falhas na conexão cliente–servidor ou no nodo que executa um servidor primário para um dado cliente são detectadas pelo serviço EJB do JOnAS, usando o mecanismo de *timeout* durante a execução das requisições. Na arquitetura HA, as requisições não–servidas são refeitas para um servidor alternativo, enquanto que nas implementações tradicionais da especificação EJB, o servidor de aplicação gera uma exceção.

Experimento prático

Neste trabalho, um protótipo (*dorothy*) com os serviços HA foi desenvolvido para demonstrar a viabilidade da integração de abstrações de comunicação de grupos em uma arquitetura transacional. Um estudo comparativo foi realizado para avaliar o desempenho do protótipo usando diferentes configurações de números de réplicas e diferentes tipos de objetos.

Os experimentos com o protótipo mostram que o conjunto de serviços HA conduz à alta disponibilidade, mantendo o MTTR na ordem de segundos. Entretanto, os experimentos aqui apresentados não são suficientes para validar a dependabilidade total da solução apresentada. Para alcançar uma validação mais completa, experimentos detalhados com injeção de falhas são necessários. Contudo, os resultados obtidos indicam que a degradação de desempenho não é demasiada, como inicialmente esperado, e que o uso das abstrações de comunicação de grupos facilitam a implementação de protocolos de replicação à medida que o código do serviço de grupos e de detecção de nodos falhos já estava desenvolvido (i.e., foi usado reuso de código).

6.2 Trabalhos futuros

O desenvolvimento dos serviços HA aqui propostos e a implementação do protótipo com o servidor *dorothy* abriram a possibilidade da pesquisa e do desenvolvimento de outros trabalhos futuros, com a finalidade de oferecer maior robustez e confiabilidade aos serviços HA.

Tratamento de exceções no sistema

Há ainda questões de projeto pendentes na arquitetura HA, como que decisão tomar quando um servidor for excluído do grupo de replicação. A característica da linguagem Java, que possibilita a detecção de exceções, é eficaz quanto à facilidade de desvio para um código que possibilite tratamento de exceções. Contudo, note que, no contexto de alta disponibilidade, a detecção de exceções não significa continuidade de serviço, i.e., *sobrevivência*, mas *segurança*. A necessidade de que o sistema execute algum procedimento quando um membro do grupo de replicação falhar ainda precisa ser pesquisada, para que a arquitetura HA permita um serviço realmente robusto.

Reintegração de servidor e transferência de estado

Um servidor que falha por colapso pode ser reintegrado através de um *serviço de reintegração* que permita que o sistema não pare durante a reintegração, i.e., a reintegração deve ser feita com o sistema operando.

O s*erviço de reintegração* de um servidor executa, principalmente, a *transferência de estado*. A principal dificuldade do mecanismo da transferência de estado é atualizar um estado persistente, já que a transferência de estado volátil é provida por mecanismos oferecidos pelo JavaGroups. A reintegração de um servidor com estado persistente em um sistema *on–line* é descrita em Kemme *et al*. [KEM2001]. Esse serviço pode ser adaptado à arquitetura HA proposta neste trabalho.

Tratamento de falhas no cliente

O modelo de sistema considerado no contexto deste trabalho não trata falhas em nodos que executam clientes. Entretanto, em um sistema prático, o nodo que executa o servidor primário de um cliente falho precisa tomar algumas medidas para evitar inconsistências nos seus objetos replicados. Falhas em clientes potencialmente podem gerar ocupação de memória desnecessária e perda de desempenho do sistema distribuído.

Semântica de execução *exatamente uma* na presença de falhas

A especificação EJB não descreve suporte à alta disponibilidade. A maioria das implementações práticas disponíveis da especificação EJB usa a semântica do *melhor esforço*, onde a entrega de uma mensagem ao destino não é garantida. Opcionalmente, o desenvolvedor pode implementar *failover* diretamente na aplicação, ou usar algum mecanismo da infra–estrutura do sistema distribuído para garantir alta disponibilidade. Essas duas opções não são automáticas e, portanto, podem aumentar a complexidade do sistema para o desenvolvedor além de promover a ocorrência de erros de programação, se o desenvolvedor implementar *failover* diretamente na aplicação. Soluções automáticas, como a solução proposta neste trabalho, são mais desejáveis.

Um serviço altamente disponível requer um mecanismo que garanta a implementação da semântica de execução *exatamente uma* em caso de falha de servidor. Essa semântica é difícil de ser assegurada por serviços intermediários.

Apesar dos serviços de replicação e chaveamento aqui desenvolvidos requererem essa semântica, atualmente, o protótipo implementa a semântica de execução *no mínimo uma*. A semântica atual do protótipo precisa ser modificada para atender às necessidades de um sistema real altamente disponível.

Monitoração dos serviços HA

O JOnAS possui uma ferramenta de monitoração para *um único* servidor, mas não para um grupo de replicação. Entretanto, para a monitoração dos serviços HA, uma ferramenta mais robusta é requerida, já que precisam ser monitorados todos os nodos de um grupo de replicação.

A tarefa de monitoração dos serviços HA pode ser facilitada, se uma ferramenta gráfica for desenvolvida. Uma sugestão é implementar alguma ferramenta gráfica para monitorar e gerenciar sistemas distribuídos como o sistema Piranha [MAF97]. Mesmo que o sistema distribuído seja capaz de tomar suas próprias decisões em decorrência de falha, às vezes é desejável a interferência de um operador. A ferramenta deverá mostrar os nodos de uma rede ou *cluster* com indicação de quais nodos estão operacionais e quais nodos estão inoperantes. Para os nodos operacionais, a ferramenta pode mostrar quais nodos estão executando servidores EJB e bem como algumas medidas de desempenho desses nodos. Para que essas informações sejam possíveis, a ferramenta deve se comunicar com o serviço de detecção de falhas do sistema de comunicação de grupos.

Ferramenta para a instalação dos serviços HA

Atualmente, a instalação dos serviços HA no JOnAS é feita manualmente. Uma ferramenta deve ser proposta para facilitar a instalação dos serviços HA.

Configurações de propriedades através de arquivos

A especificação EJB é baseada em *declarações.* As propriedades das aplicações são *declaradas* em arquivos descritores e são inseridas automaticamente nas aplicações. O nome do grupo de replicação e outras propriedades do grupo, como a escolha entre a replicação ávida ou a replicação preguiçosa, podem ser fornecidos como um parâmetro *declarado* no descritor de um *bean.*

Atualmente o protótipo não implementa configuração automática através de arquivos descritores. Trabalhos futuros incluem integrar essa facilidade para facilitar o desenvolvimento de *beans* altamente disponíveis.

Aplicações EJB com múltiplos *beans*

Uma aplicação EJB pode agregar diversos tipos de *beans*. Entretanto, o escopo deste trabalho foi limitado a aplicações que são constituídas por somente um único *bean*. Trabalhos futuros devem incluir um modelo de sistema que suporte replicação em aplicações menos restritivas, tratando aplicações com mais de um *bean* e as interações entre *beans* diferentes.

Serviço de nomes

O serviço de nomes deste trabalho usa um servidor JNDI para cada no nodo com servidor EJB. Esse tipo de configuração é chamado de servidor JNDI independente pois cada nodo tem sua instância, que não se comunica com as demais. Assume-se que se o servidor EJB falha, o servidor JNDI correspondente falha também (i.e., o nodo é *fail-stop*).

Esse tipo de configuração, o servidor JNDI independente, é de fácil implementação. Entretanto, uma implementação mais robusta deve comportar servidores JNDI conectados, que troquem informações para prover balanceamento de carga ao sistema computacional. O servidor JNDI com balanceamento de carga fornecerá ao cliente, o servidor EJB mais adequado em termos de desempenho.

Injeção de falhas

Os experimentos conduzidos neste trabalho mostram que os serviços implementados conduzem à alta disponibilidade, mantendo o MTTR na ordem dos segundos. Entretanto, para alcançar uma validação mais completa, experimentos com injeção de falhas são necessários para validar a *dependabilidade* total da solução apresentada neste trabalho.

Apontamentos finais

O trabalho descrito por este texto foi realizado nos laboratórios do PPGC (Programa de Pós–Graduação em Computação) do Instituto de Informática da UFRGS, entre agosto de 1998 e maio de 2003. Durante esse período, realizei um estágio de doutorado sanduíche entre outubro de 2000 e agosto de 2001, na ESSI (*Ecole Supérieure em Sciences Informatiques*) da *Université de Nice – Sophia Antipolis*, França. Minha permanência na *Université de Nice – Sophia Antipolis* somente foi possível pelo financiamento integral do CNPq/Brasil. O CNPq também apoiou meus estudos no Brasil, onde fui contemplada com uma bolsa de estudo (modalidade doutorado) durante três anos.

Em Sophia Antipolis, o Prof. Michel Riveill me aceitou e me acolheu como orientanda e nova integrante do *Grupo de Pesquisa Rainbow* [RAI2001], do qual o Prof. Riveill participa juntamente com as Profas. Anne–Marie Pinne e Mireille Blay. O contato inicial com o Prof. Riveill foi proporcionado pelo Prof. Navaux, diretor deste Instituto de Informática.

Aqui no PPGC/UFRGS, meu trabalho de pesquisa começou em 1998 quando ingressei no Curso de Mestrado sob orientação da Profa. Taisy Silva Weber. Desde então, a Profa. Taisy tem me acompanhado com sua atenciosa e paciente orientação. Na UFRGS, minha pesquisa é vinculada aos projetos do *Grupo de Pesquisa de Tolerância a Falhas*, encabeçado pelos Profs. Weber, Ingrid, Maria Lucia e Taisy.

Referências

[AMI94] AMIR, Y.; DOLEV, D.; MELLIAR–SMITH, P. M.; MOSER, L. E. **Robust and efficient replication using group communication**. Jerusalem: Institute of Computer Science, the Hebrew University of Jerusalem, 1994. (Report CS94–20).

[ALS76] ALSBERG, P. A.; DAY, J. D. A principle for resilient sharing of distributed resources. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2., 1976. **Proceedings…** New York: IEEE Computer Society, 1976. p.562–570.

[ARC2000] PROJET RNTL ARCAD: Architecture Répartie extensible pour Composants Adaptables. 2000. Disponível em: <http://www.essi.fr/~riveill/recherche/00–02–rntl–arcad/>. Acesso em: 20 jun. 2001.

[ASP2002] MICROSOFT CORPORATION. **Microsoft ASP.NET**. 2002. Disponível em: <http://www.asp.net/>. Acesso em: 22 jan. 2003.

[BAB93] BABAOGLU, O.; TOUEG, S. Non–blocking atomic commitment In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed., New York: ACM Press, 1993. p.147–168.

[BAN99] BAN, B. **JavaGroups user's guide**. 1999. Ithaca: Department of Computer Science, Cornell University. 73p. Disponível em: <http://JavaGroups. sourceforge.net/>. Acesso em: 6 set. 2001.

[BAN2002] BAN, B. **Overview of HSQLDB replication (HSQLDB/R).** 2002. Disponível em: <http://www.javagroups.com/javagroupsnew/docs/hsqldbr/design.htm>. Acesso em: 6 set. 2002.

[BEA2000] BEA SYSTEMS, INC. **Understanding Object Clustering – WebLogic Server 6.0**. 2000. Disponível em: <http://edocs.bea.com/wls/docs60/cluster/object. html#1006807,2000>. Acesso em: 20 jun. 2001.

[BEL95] BELLISSARD, L.; ATALLAH, S. B.; KERBRAT, A.; RIVEILL, M. Component–Based Programming and Application Management with Olan. In: OBPDC 1995. **Proceedings…** Berlin: Springer-Verlag, 1995. p.290–309.

[BER87] BERNSTEIN, P. A.; HADZILACOS, V.; GOODMAN, N. **Concurrency control and recovery in database systems**. Massachusetts: Addison Wesley, 1987. Disponível em: <http://research.microsoft.com/pubs/ccontrol/>. Acesso em: 6 out. 2002.

[BIR84] BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. **ACM Transactions on Computer Systems**, New York, v.1, n.2, p.39–59, 1984.

[BIR87] BIRMAN, K.; JOSEPH, T. A. Reliable communication in presence of failures. **ACM Transactions on Computer Systems,** New York, v.5, n.1, p.47–76, Feb. 1987.

[BIR96] BIRMAN, K. **Building Secure and Reliable Network Applications**. Greenwich: Manning Publications Company, 1996.

[BRI97] BRIOT, J–P.; GUERRAOUI, R. A classification of various approaches to paralell and distributed programming. Survey and classification: tutorial notes. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 11., 1997. **Proceedings…** [S.l.: Aito], 1997.

[BRI98] BRIOT, J.; GUERRAOUI, R.; LOHR, K. Concurrency and distribution in object–oriented programming. **ACM Computing Survey,** New York, v.30, n.3, Sept. 1998.

[BRU2001] BRUNETON, E.; RIVEILL, M. An architecture for extensible middleware platforms. **Software – Practice and Experience**, Sussex, v. 31, n.13, p.1237–1264, 2001.

[BUD93] BUDHIRAJA, N. et al. The primary–backup approach. In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed. New York: ACM Press, 1993. p.199–216.

[BUR2002] BURKE, B.; LABOUREY, S. Clustering with JBoss 3.0. **The O´Reilly Network.** 2002. Disponível em: <http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html>. Acesso em: 24 abr. 2003.

[CHA96] CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM,** New York, v.34, n.2, p.225–267, Mar. 1996.

[CHA96a] CHAPPEL, D. **Understanding ActiveX and OLE**. EUA: Microsoft Press, 1996.

[CHO2001] CHOCKLER, G.; KEIDAR, I.; VITENBERG, R. Group communication specifications: a comprehensive study. **ACM Computing Surveys,** New York, v.33, n.4, p.427–469, 2001.

[COS2003] COSTA, A. A.; BRASILEIRO, F. V. *JBossFT:* Tolerância a Falhas para Aplicações J2EE. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, 2003. **Anais...** Porto Alegre: SBC, 2003. v.1.

[CRI86] CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock synchronization in the presence of omission and performance faults, and processor joins. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, 16., 1986. **Proceedings…** New York: IEEE Computer Society, 1986.

[DAN2000] DANES, A.; DECHAMBOUX, P.; RIVEILL, M. ; VANDOME, G. Technologie à base de composants EJB: expérience et perspectives avec JOnAS. In : OBJECTS, COMPOSANTS ET MODELES – PASSE, PRESENT ET FUTUR, OCM, 2000. **Proceedings…** [S.l] : Université de Nantes, École des Mines de Nantes, 2000.

[DUN2003] DUNPAL, B. **.NET platform:** the start of a new generation. 2003. Disponível em: <http://www.microsoft.com/Seminar/Includes/Seminar.asp?Url=/Seminar/1033/20001024NETPLATBD1/Portal.xml>. Acesso em: 22 jan. 2003.

[ESW76] ESWARAN, K. P.; GRAY, J. N.; LORIE, R. A.; TRAIGER, I. L. The notion of consistency and predicate locks in a database system. **Communications of the ACM,** New York, v.19, n.11, p. 624-633, Nov. 1976.

[FIS85] FISHER, M.; LYNCH, N.; PATERSON, M. Impossibility of distributed consensus with one faulty process. **Journal of the ACM,** New York, v.32, n.2, p.374–382, Apr. 1985.

[FRØ99] FRØLUND, S.; GUERRAOUI, R. **Exactly–once transactions.** 1999. (Report SSC–2000–011). Disponível em: <http://lsewww.epfl.ch/~rachid/papers/publis00.html>. Acesso em: 22 jun. 2001.

[GAR99] GARTNET, F. C. Fundamentals of fault–tolerant distributed computing in asynchronous environments. **ACM Computing Surveys**, New York, v.31, n.1, Mar. 1999.

[GAR2003] GARCIA, S. Alta disponibilidade. **Revista do Linux,** 2003. Disponível em: <http://www.revistadolinux.com.br/ed/035/assinantes/alta_ disponibilidade.php3>. Acesso em: 15 maio 2003.

[CHA84] CHANG, J.; MAXEMCHUK, N. F. Reliable broadcast protocols. **ACM Transactions on Computer Systems,** New York, v.2, n.3, p.251-273, Aug. 1984.

[GRA78] GRAY, J. Notes on database operation systems. In : FLYNN, M. J. *et al* (Ed.). **Operating Systems:** an Advanced Course. Berlin: Springer–Verlag, 1978. (Lecture Notes in Computer Science, v. 60).

[GRA93] GRAY, J.; REUTER, A. **Transaction processing:** concepts and techniques. San Mateo (CA), USA: Morgan Kaufmann Publishers, Inc., 1993. 1070p.

[GRA96] GRAY, J.; HELLAND, P.; O'NEIL, P.; SHASHA, D. The dangers of replication and a solution. In: ACM SIGMOD CONFERENCE, 1., 1996. **Proceedings…** New York: ACM Press, 1996. p.173–182.

[GUE96] GUERRAOUI, R.; SCHIPER, A. Fault–tolerance by replication in distributed systems. In: CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 1996. **Proceedings…** Berlin: Springer–Verlag, 1996. p.38–57. (Lecture Notes in Computer Science, v. 1088).

[GUE96a] GUERRAOUI, R. Distributed programming abstractions. **ACM Computing Surveys,** New York, v. 28 A, n. 4, Dec. 1996.

[GUE97] GUERRAOUI, R.; SCHIPER, A. Software–based replication for fault tolerance. **Computer,** New York, p.68–74, Apr. 1997.

[HAA2002] HAASE, K. **Java Message Service Tutorial**. [S.l.]: Sun Microsystems, 2002. Disponível em: <http://java.sun.com/products/jms/>. Acesso em: 15 maio 2002.

[HAD90] HADZILACOS, V. On the relationship between the atomic commitment and consensus problems. In: SIMONS, B.; SPECTOR, A. Z. (Ed.). **Fault Tolerant Distributed Computing**. Berlin: Springer-Verlag, 1990. p.201-208 (Lecture Notes in Computer Science, v.448).

[HAN2002] HANIK, F. **Clustering technologies – in memory session replication in Tomcat 4.** 2002. Disponível em: <http://www.filip.net/tomcat/ tomcat-javagroups-article.html>. Acesso em: 28 out. 2002.

[HAY98] HAYDEN, M. G. **The Ensemble System.** 1998**.** Ph. D. dissertation - Faculty of the Graduate School of Cornell University, Ithaca.

[HOL99] HOLLIDAY, J.; AGRAWAL, D.; ABBADI., A. E. The performance of database replication with group multicast. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING, FTCS29, 1999. **Proceedings…** New York: IEEE Computer Society, 1999. p.158–165.

[HOL99a] HOLLIDAY, J.; AGRAWAL, D.; ABBADI, A. E. **The Performance of Replicated Databases using Atomic Broadcast Group Communication.** 1999. (TRCS99-11). Disponível em: <http://citeseer.nj.nec.com/ article/holliday99performance.html>. Acesso em: 29 abr. 2003.

[HPB2001] HEWLETT–PACKARD COMPANY. **HP Bluestone Total–e–Server 7.2.1**. 2001. Disponível em: <http://www.bluestone.com/products/total–e–server/ default.htm> . Acesso em: 25 out. 2001.

[HRA99] HRANITZKY, N. **Home Page**. 1999. Disponível em: <http://www. norbert.hranitzky.com/ejblinks.html>. Acesso em: 27 jan. 2003.

[ISS99] ISSARNY, V.; BELLISSARD, L.; RIVEILL, R.; ZARRAS, A. Component–Based Programming of Distributed Applications. In: KRAKOWIAK, S.; SHRIVASTAVA, S. K. (Ed.). **Advances in Distributed Systems**. Berlin: Springer–Verlag, 1999. p.327–353. (Lecture Notes in Computer Science, v. 1752).

[IYE96] IYER, R. K.; TANG, D. Experimental analysis of computer system dependability. In: PRADHAN, D. F. (Ed.). **Fault–tolerant computer system design.** Upper Saddle River: Prentice Hall, 1996. p.282–392.

[JAL94] JALOTE, P. **Fault Tolerance in Distributed Systems**. Upper Saddle River: Prentice Hall, 1994. 432p.

[JEW2000] JEWELL, T. EJB clustering with application servers. **O'Reilly Network,** 2000. Disponível em: <http://www.onjava.com/pub/a/onjava/2000/12/ 15/ejb_clustering.html>. Acesso em: 23 jan. 2003.

[JOH96] JOHNSON, B. W. An introduction to the design and analysis of fault–tolerant systems. In: PRADHAN, D. F. (Ed.). **Fault–tolerant computer system design.** Upper Saddle River: Prentice Hall, 1996. p.1–87.

[KAN2001] KANG, A. J2EE clustering, Part 1. Clustering technology is crucial to good Website design; do you know the basics? **JavaWorld,** 2001. Disponível em: <http://www.javaworld.com/javaworld/jw–02–2001/jw–0223–extremescale.html>.

[KAS2000] KASSEM, N. AND ENTERPRISE TEAM. **Designing enterprise applications with the Java™ 2 platform, Enterprise Edition.** Version 1.0.1 – final release. [S.l], Oct. 2000. 362p.

[KEM2001] KEMME, B.; BARTOLI, A.; BABAOGLU, O. On–line reconfiguration in replicated databases based on group communication. In: THE

INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2001. **Proceedings…** New York: IEEE Computer Society, 2001. p.117–126.

[LAM77] LAMPORT, L. Providing the correctness of multi-process programs. **IEEE Trans. Soft. Eng**., New York, v. 3, n. 2, p.125–143, Mar. 1977.

[LEE2001] LEE, R.; SELIGMAN, S. **JNDI API tutorial and reference: building directory–enable java applications**. 2001. Disponível em: <http://www.sun.com>. Acesso em: 20 jun. 2001.

[LIT2000] LITTLE, M. C.; SHRIVASTAVA, S. K. Integrating group communication with transactions for implementing persistent replicated objects. In: KRAKOWIAK, S.; SHRIVASTAVA, S.K. (Ed.) **Advances in Distributed Systems**. Berlin: Springer–Verlag, 2000. p.238–253. (Lecture Notes in Computer Science, v.1752).

[MAF97] MAFFEIS, S. Piranha: a CORBA tool for high availability. **Computer**, New York, p.59–66, Apr. 1997.

[MIC2002] MICROSOFT CORPORATION. **Saiba mais sobre a plataforma .NET**, 2002. Disponível em: <http://www.microsoft.com/brasil/net/default.asp>. Acesso em: 13 dez. 2002.

[MIC2002a] MICROSOFT CORPORATION. .**NET framework: product overview**, 2002. Disponível em: <http://msdn.microsoft.com/ netframework/ productinfo/ overview/default.asp>. Acesso em: 22 jan. 2002.

[NIE87] NIERSTRASZ, O. M. What is an 'object' in object oriented programming?. In: TSICHRITZIS, D. (Ed.). **Objects and Things**. Suisse: Université de Genève, Centre Universitaire D'Informatique, 1987. p.1–13.

[OBJ2001] OBJECTWEB. **JOnAS: Java™ Open Application Server**, 2001. Disponível em: <http://www.objectweb.org/ jonas>. Acesso em: 20 jun. 2001.

[OMG95] OBJECT MANAGEMENT GROUP. **The Common Object Request Broker: Architecture and Specification.** 1995. Revision 2.0. Disponível em: <http://www.omg.com>. Acesso em: 13 dez. 2002.

[ORF97] ORFALI, R.; HARKEY, D.; EDWARDS, J. Distributed object systems. **Software Development,** [S.l.], p.30–36, April 1997. Disponível em: <http://sdmagazine.com>. Acesso em: 13 dez. 2000.

[PAS98] PASIN, M.; LEITE, F. O.; AMARAL, J. B.; JANSCH–PORTO, I. E. S.; WEBER, T. S. Um sistema de arquivos replicado e distribuído para um ambiente com comunicação de grupo confiável. In: SIMPÓSIO NACIONAL DE INFORMÁTICA, SNI, 3., 1998. **Anais...** Santa Maria: Centro Universitário Franciscano, 1998. p.145–149.

[PAS98a] PASIN, M.; WEBER, T. S. Reintegração de servidores em um sistema de replicação de arquivos. In: WORKSHOP DE TOLERÂNCIA A FALHAS, WTF, 1., 1998. **Anais...** Porto Alegre: SBC, UFRGS, SBC–RS, 1998. p. 13–17.

[PAS98b] PASIN, M.; WEBER, T. S. Reintegração de servidores em um grupo de replicação de arquivos NFS. In: WORKSHOP DE SISTEMAS

DISTRIBUÍDOS, WOSID, 2., 1998. **Anais...** Curitiba: Pontifícia Universidade Católica do Paraná, 1998. p. 97–104.

[PAS98c] PASIN, M.; WEBER, T. S. Reintegração de servidores em um sistema distribuído com replicação de arquivos. In: SIMPÓSIO EN REDES E SISTEMAS DISTRIBUÍDOS – REDES; CONGRESSO LATINO–IBEROAMERICANO DE INVESTIGACION OPERATIVA, 9., 1998. **Proceedings…** Buenos Aires: UBA, 1998. p.79–84.

[PAS99] PASIN, M.; JANSCH–PORTO, I. E. S.; LEITE, F.O.; AMARAL, J. B. Implementando réplicas de arquivos através de uma ferramenta de comunicação de grupo Confiável. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, SCTF, 1999. **Anais...** Campinas: UNICAMP, 1999. p. 84-98.

[PAS99a] PASIN, M.; WEBER, T. S. Protocols for server reintegration in a distributed system providing file replicas. In: INTERNATIONAL SYMPOSIUM ON COMPUTER AND INFORMATION SCIENCES, ISCIS, 14., 1999. **Proceedings…** Bornova: Ege University, 1999. p. 1037-1039.

[PAS99b] PASIN, M. **Tolerância a falhas através de réplicas de objetos em sistemas distribuídos.** 1999. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[PAS2000] PASIN, M.; WEBER, T. S. Server reintegration in a replicated UNIX file system. **Investigacion Operativa,** Rio de Janeiro, v.9, n.1/3, p.71–82, jan./jul. 2000.

[PAS2001] PASIN, M.; RIVEILL, M.; WEBER, T. S. Highly–available Enterprise JavaBeans using group communication system support. In: EUROPEAN RESEARCH SEMINAR ON ADVANCES IN DISTRIBUTED SYSTEMS, 4., 2001. **Proceedings…** [S.l.: s.n.], 2001. p.161–166.

[PAS2001a] PASIN, M. Adding continuous availability to Enterprise JavaBeans application servers using group communication. In: SUPLEMENT OF PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2001. **Proceedings…** New York: IEEE Computer Society, 2001.

[PAS2001b] PASIN, M.; RIVEILL, M.; WEBER, T. Using group communication to establish distributed checkpoint in replicated EJB application servers. In: ARGENTINE SYMPOSIUM ON SOFTWARE ENGINEERING, 2., 2002. **Proceedings…** Buenos Aires : SADIO, 2001. p.241–244.

[PAS2002] PASIN, M.; RIVEILL, M.; WEBER, T. A multi–layer architecture to achieve highly-available Enterprise JavaBeans. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 3.; SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 20., 2002. **Anais...** Porto Alegre: SBC, 2002. p. 53–60.

[PAT2001] PATIÑO–MARTÍNEZ, M.; JIMENEZ-PERIS, R.; AREVALO, S. Group Transactions: An integrated approach to transactions and group communication. In: WORKSHOP ON CONCURRENCY IN DEPENDABLE COMPUTING, 2001. **Proceedings…** [S.l.]: University

of Newcastle upon Tyne, UK, 2001. p.5–15. Disponível em: <http://citeseer.nj.nec.com/article/patino-martinez02group.html>. Acesso em: 29 abr. 2003.

[PED98] PEDONE, F.; GUERRAOUI, R.; SCHIPER, A. Exploiting atomic broadcast in replicated databases. In: EURO-PAR 1998. **Proceedings…** Berlin: Springer–Verlag ,1998.

[PED99] PEDONE, F.; GUERRAOUI, R.; SCHIPER, A. **The database state machine approach**. Switzerland: École Polytechnique Fédérale de Lausanne, Mar. 1999. (Report SSC/ 1999/008).

[PED2000] PEDONE, F.; FRØLUND, S. Pronto: a fast failover protocol for off-the-shelf commercial databases. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 19., 2000. **Proceedings…** New York: IEEE Computer Society, 2000.

[RAI2001] RAINBOW – REFLEXIVE ARCHITECTURE FOR INTERACTION NETWORK BETWEEN OBJECTS WORKING TOGETHER. 2001. Disponível em: <http://www.essi.fr/~rain–bow/>. Acesso em: 13 nov. 2001.

[REN96] RENESSE, R. van; BIRMAN, K; MAFFEIS, S. Horus: A Flexible Group Communications System. **Communications of the ACM,** New York, v.4, n.39, p.76–83, Apr. 1996.

[SCH83] SCHLICHTING, R. D.; SCHNEIDER, F. B. Fail–stop processors: an approach to designing fault–tolerant computing systems. **ACM Transactions on Computer Systems,** New York, v.1, n.3, p.222–238, Aug. 1983.

[SCH93] SCHNEIDER, F. B. Replication management using the state machine approach. In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed. New York: ACM Press, 1993. p.169–198.

[SCH96] SCHIPER, A.; RAYNAL, M. From group communication to transactions in distributed systems. **Communications of the ACM,** New York, v.39, n.4. Apr. 1996.

[MIS99] MISHRA, S.; LEI, L.; XING, G. Design, Implementation and Performance Evaluation of a CORBA Group Communication Service. In: FTCS, 1999. **Proceedings…** New York: IEEE Computer Society, 1999. p.166-173.

[SIE2001] SIEGEL, J. **OMG's new fault tolerant CORBA specification.** 2001. Disponível em: <http://www.omg.org/news/publications/OMG%20 MOTION% 2040 12.dpf>. p.4–5, 2001. Acesso em: 25 out. 2001.

[SIL93] SILBERSCHATZ, A.; KORTH, H. F. **Sistemas de Banco de Dados.** 2. ed. São Paulo: Makron Books, 1993. 748p.

[SIL2000] SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Applied operating system concepts**. New York: John Wiley & Sons, 2000. 840p.

[SIL2001] SILVERSTREAM SOFTWARE, INC. **SilverStream Application Server 3.7.** 2001. Disponível em: <http://www. sil–verstream.com/website/silverstream/pages/products_appserver_tf.html>. Acesso em: 25 out. 2001.

[SKE81] SKEEN, D. Nonblocking commit protocols. In: ACM SIGMOD, 1981. **Proceedings...** New York: ACM, 1981. p. 133–142.

[SOU2001] SOUZA, C. V. P. B. de. **Uso da reflexão computacional em aplicações J2EE.** 2001. 163p.Dissertação de Mestrado - PUC/PR, Curitiba – PR.

[STA98] STANOI, I.; AGRAWAL, D.; ABBADI A. E. Using Broadcast Primitives in Replicated Databases. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 18., 1998. **Proceedings…** New York: IEEE Computer Society, 1998. p.148–155.

[SUN97] SUN MICROSYSTEMS, INC. **Java Remote Method Invocation.** 1997. Disponível em: <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/ rmiTOC.doc.html> Acesso em: 27 set. 2001.

[SUN2000] SUN MICROSYSTEMS, INC. **iPlanet Application Server Programmer's Guide (Java).** 2000. Disponível em: <http:// docs.sun.com/source/816-5713-10/index.html>. Acesso em: 20 nov. 2002.

[SUN2001] SUN MICROSYSTEMS, INC. **EJB Specification 2.0**. 2001. Disponível em: <http://java.sun.com/products/ejb/docs.html>. Acesso em: 27 set. 2001.

[SUN2001a] SUN MICROSYSTEMS, INC. **Sun Developer Connection. EvangCentral.** 2001. Disponível em: <http:/www.sun.com/ developers/ evangcentral>. Aceso em: 20 out. 2001.

[SUN2002] SUN MICROSYSTEMS, INC. **The J2EE Tutorial**. 2002. Disponível em: <http://java.sun.com/ j2ee/tutorial/>. Aceso em: 23 jan. 2003.

[SUN2003] SUN MICROSYSTEMS, INC**. CosNaming Service do CORBA**. 1993. Disponível em: <http://java.sun.com/products/jdk/1.2/docs/api/org/omg/CosNaming/package–summary.html>.

[SUN2003a] SUN MICROSYSTEMS, INC. **JavaServer Pages™: dynamically generates web content**. 2003 Disponível em: <http://java.sun.com/ products/jsp/>.

[SUN2003b] SUN MICROSYSTEMS, INC. **JDBC Data Acess API**. 2003. Disponível em: <http://java.sun.com/products/jdbc/>. Acesso em: 23 jan. 2003.

[SYB2001] SYBASE, INC. **Sybase Enterprise Application Server 3.6**. 2001. Disponível em: <http://www.sybase.com/detail/1,3693,1011065,00.html>. Acesso em: 25 out. 2001.

[SZY99] SZYPERSKI, C. **Component Software – Beyond Object-Oriented Programming**. [S.l]: Addison-Wesley, 1999.

[TAN92] TANENBAUM, A. S. **Modern Operating Systems**. New Jersey: Prentice Hall, 1992. p. 395–462.

[TRI82] TRIVEDI, K. S. **Probability and statistics with reliability, queuing, and computer science applications.** Englewood Cliffs: Prentice Hall, 1982. 624p.

[VAY98] VAYSBURD, A. **Building reliable interoperable distributed objects with the Maestro Tools.** 1998. Ph. D. Dissertation - Cornell University, Ithaca.

[VOE2003] VOELTER, M. A taxonomy of components. **Journal of Object Technology,** [S.l.], v.2, n.4, p.119-125, July-Aug. 2003.

[WIE2000] WIESMANN, M.; PEDONE, F.; SCHIPER, A.; KEMME, B.; ALONSO, G. Understanding replication in databases and distributed systems. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 20., 2000. **Proceedings…** New York: IEEE Computer Society, 2000. p.264–274.