Resumo

Com a popularização da Internet e o desenvolvimento de redes de computadores de alta velocidade, sistemas computacionais passaram a ser usados em diversas novas áreas de aplicação. Varias dessas aplicações demandam um alto grau de confiabilidade, de modo a garantir que: (*i*) o sistema se comporta de acordo com sua especificação; (*ii*) falhas não resultam em eventos catastróficos como perdas de vidas humanas ou grandes perdas financeiras. Tolerância a falhas é uma das abordagens empregadas para garantir que sistemas distribuídos atinjam o grau esperado de confiabilidade. É uma técnica que visa garantir que o sistema se comporta de acordo com sua especificação mesmo na manifestação de falhas. Neste trabalho, apresentamos um estudo sobre tolerância a falhas e sobre os mecanismo de detecção e recuperação de defeitos, que são técnicas importantes para que um sistema mascare as falhas de seus componentes. A detecção de defeitos serve para descobrir a ocorrência de defeitos no sistema. Após a detecção dos defeitos, são usadas técnicas de recuperação de defeitos para guiar o sistema a um estado consistente. Em nosso trabalho selecionamos um conjunto de infra-estruturas de *middleware* composto por infra-estruturas desenvolvidas tanto no âmbito comercial quanto no âmbito acadêmico, sendo elas JBoss, Horus, Sprint, ICE, Tao. Este trabalho visa fornecer uma visão geral dessas infra-estruturas de *middleware*, com ênfase nos mecanismos de tolerância a falhas disponiveis em cada uma delas. Tambem analisamos as técnicas de detecção e recuperação de defeitos implementadas por cada uma e fazemos uma comparação para dar suporte aos desenvolvedores na escolha da infra-estrutura ideal para o desenvolvimento de um sistema de software tolerante a falhas.

Abstract

With the popularization of the Internet and the development of high speed computer networks , computational systems had started to be used in several new areas of application. You vary of these applications demand one high degree of reliability, in order to guarantee that: (i) the system if holds its specification in accordance with; (II) faults do not result in catastrophic events as losses of lives human beings or great financial losses. Fault Tolerance is one of the used approaches to guarantee that distributed systems reach the waited degree of reliability. It is one technique that it aims at to guarantee that the system if in accordance with holds its same specification in the manifestation of faults. In this work, we present a study on fault tolerance and the mechanism of failure detection and failure recovery, that are important techniques so that a system masks the ifailuress of its components. The failures detection serves to discover the occurrence of failure in the system. After the failure detection, techniques of failure recovery are used to guide the system to a consistent state. In our work we in such a way select a set of infrastructures of middleware composed for infrastructures developed in the commercial scope how much in the academic scope, being they JBoss, Horus, Sprint, TAO and ICE, So. This work aims at to supply a general vision of these infrastructures of middleware, with emphasis in the tolerance mechanisms the available imperfections in each one of them. Also we analyze the techniques of failure detection and failure recovery implemented for each one and make a comparison to give has supported to the developers in the choice of the ideal infrastructure for the development of a system of fault tolerant software.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

(Dispostos por ordem de aparição no texto)

TAO – *The ACE ORB*
ICE – *Internet Communications Engine*
NORAD – *North American Aerospace Defense Command*
NASA – *National Aeronautics and Space Administration*
ACID – *Atomicity, Consistency, Isolation, Durability*
FTMP – *Fault tolerant multiprocessor*
SIFT – *Software Implemente Fault Tolerance*
ESS – *Electronic switching system*
IMDB – *in-memory Database*
ES – *Edge Server*
DS – *Data Server*
XS – *Durability Server*
SQL – *Structured Query Language*
ASCII – *American Standard Code for Information Interchange*
IP – *Internet Protocol*
UDP – *User Datagram Protocol*
ATM – *Asynchronous Transfer Protocol*
HCPI – *Horus Common Protocol Interface*
ACE – *ADAPTIVE Communication Environment*
ORB – *Object Request Broker*
CORBA – *Common Object Request Broker Architecture*
QoS – *Quality of Service*
IDL –*Interface Definition Language*
GIOP – *General inter-ORB Protocol*
IIOP – *Internet inter-ORB Protocol*
TCP – *Transmission Control Protocol*
DII – *Dynamic Invocation Interface*
DSI – *Dynamic Skeleton Interface*
BOA – *Basic Object Adapter*
POA – *Portable Object Adapter*
CPU – *Central Processing Unit*
I/O – *In/Out*
CDR – *Common Data Representation*
APIC – *ATM Port Interconnect Controller*
FT CORBA – *Fault-Tolerant CORBA*
IOGR – *Interoperable Object Group Reference*
DOORS – *Distributed Object-Oriented Reliable System*
IOR – *Interoperable Object Reference*
ISIS – *Institute for Software Integrated Systems*
RTCORBA – *Real-Time CORBA*

API – *Application Programming Interface*
SSL – *Secure Sockets Layer*
JMX – *Java Management Extensions*
SAR – *Service Archives*
AOP – *Aspect-Oriented Programming*
JBossTS – *JBoss Transaction Service*
UID – *Unique Identifier*
JTA – *Java Transactions API*
JTS – *Java Transactions Service*
URL – *Uniform Resource Locator*
JNDI – *Java Naming and Directory Interface*

Agradecimentos

Agradeço a meus pais, que com muito sacrifício, me deram a oportunidade de me tornar o que sou hoje. Pelo apoio nesse momento importante da minha vida e pelo esforço empregado para que este trabalho fosse concluído.

Agradeço a todos da minha família, meus irmãos, primos, tios, tias, avós que desde o começo sempre torceram pelo meu sucesso. Principalmente ao meu avô Pedro Coelho e minha tia Célia Frazão, que mesmo não estando mais entre nós, sei que sempre torceram por mim.

Agradeço a todos os meus amigos de faculdade, principalemente, nosso grupo denominado Máfia Poli, que sempre enfretamos as dificuldades com muito bom humor e companheirismo.

Agradeço a todos os meus amigos, que entenderam a minha falta de tempo e me apoiaram para terminar este trabalho. Meus amigos do Senac, onde faço o curso de inglês. Meus velhos, e eternos, amigos com quem eu aprendi muitas coisas nesta vida e que a distancia não conseguiu nos separar.

Agradeço ao Professor Fernando Castor, por ter acreditado na minha capacidade em concluir este trabalho, e pelo tempo disponibilizado por ele para que esse trabalho saísse da melhor forma possível.

Capítulo 1

Introdução

Com a popularização da Internet e o desenvolvimento de redes de computadores de alta velocidade, sistemas computacionais passaram a ser usados em diversas novas áreas de aplicação. Varias dessas aplicações demandam um alto grau de confiabilidade, de modo a garantir que: (*i*) o sistema se comporta de acordo com sua especificação; (*ii*) falhas não resultam em eventos catastróficos como perdas de vidas humanas ou grandes perdas financeiras. Confiabilidade e disponibilidade são cada vez mais desejáveis e necessários em sistemas de computação, pois a cada dia aumenta a dependência da sociedade de sistemas automatizados e informatizados. Por exemplo, uma transação bancaria é uma operação que necessita alta confiabilidade, pois se ocorre uma falha enquanto é feita uma atualização no banco de dados após uma transferência, haverá uma inconsistência nos dados após essa atualização, com potenciais perdas financeiras para as partes interessadas.

Tolerância a falhas é uma das abordagens empregadas para garantir que sistemas distribuídos atingem o grau esperado de confiabilidade. Para desenvolvedores de software ou projetistas de hardware, o domínio das técnicas de tolerância a falhas torna-se essencial na seleção de tecnologias e especificação de sistemas. Com a disseminação de computadores e o aumento da complexidade dos sistemas computacionais na atualidade, a possibilidade de ocorrência de falhas é cada vez maior. Preservar a integridade e a disponibilidade implica em medidas extras de segurança na forma de verificação de consistência e redundância. Redundância é a chave para a tolerância a falhas. Todas as técnicas de tolerância a falhas envolvem alguma forma de redundância. Um sistema implementa redundância se inclui dados, módulos de software e/ou hardware, unidades de processamento ou realiza ações que não seriam necessários para ele prover sua funcionalidade, conforme definida por sua especificação, na ausência de falhas.

Duas funcionalidades de um sistema fundamentais para prover tolerância a falhas são a detecção e a recuperação de defeitos. A detecção de defeitos visa identificar os elementos do sistema que falharam enquanto a recuperação de defeitos leva o sistema para um estado válido. Recuperação de defeitos ocorre depois da detecção do defeito, e seu objetivo é fazer uma troca do estado atual incorreto por um novo estado livre de erros.

Há vários mecanismos que podem ser utilizados na detecção e na recuperação de defeitos. Esses mecanismos podem ser usados nos sistemas para garantir a confiabilidade da operação, mas isso causaria um alto custo computacional, de *hardware* ou de *software*. As infra-estruturas de *middleware* existentes implementam diversos mecanismos de detecção de defeitos e recuperação de defeitos. Por isso, fica difícil para os desenvolvedores estabelecer qual a infra-estrutura mais apropriada para o desenvolvimento de uma aplicação distribuída tolerante a falhas. O sucesso de

um sistema tolerante a falhas depende da eficiência da detecção e recuperação de defeitos na ocorrência de falhas. É importante o estudo da detecção de defeitos por ela ser a fase inicial e de extrema importância para que um sistema tolere falhas e a recuperação de defeitos para que os defeitos sejam recuperados e o sistema não pare de funcionar, o que pode causar diversos danos a sistemas criticos.

1.1.1 Motivação

Este trabalho visa estudar os serviços oferecidos (ou não) por um conjunto de infra-estruturas de *middleware* existentes. Sua principal motivação é fornecer para os desenvolvedores uma base de informações para auxiliar na escolha de uma infra-estrutura que seja mais compatível com os requisitos de cada aplicação. Para a construção desta base de informações selecionamos um sub-conjunto que consideramos representativo das infra-estruturas de *middleware* existentes, composto por infra-estruturas comerciais e infra-estruturas desenvolvidas no âmbito acadêmico. Sendo elas, Sprint, Horus, TAO, ICE e JBoss. Estas informações são organizadas em termos dos mecanismos de tolerância a falhas providos por cada infra-estrutura. Tais mecanismos são divididos em detecção e recuperação de defeitos, pois essas duas funcionalidades são essenciais à construção de sistemas distribuídos tolerantes a falhas. A organização dos resultados de acordo com os quesitos detecção e recuperação de defeitos facilitará o entendimento da comparação e permitirá que desenvolvedores avaliem esses critérios de forma separada.

1.1.2 Organização do Trabalho

Esta monografia está organizada de seguinte maneira. O capítulo 2 apresenta uma descrição de tolerância a falhas, dos tipos de falhas existentes, do que acontece com os sistemas na ocorrência destas falhas e suas conseqüências. Apresetamos uma breve apresentação dos mecanismos de tolerância a falhas e uma melhor explicação dos mecanismos que servirão de base para a análise proposta por este trabalho e por fim alguns exemplos de aplicações. O capítulo 3 descreve as infra-estruturas de *middleware* escolhidas para análise focalizando os mecanismos de tolerância a falhas providos por cada uma delas, além de uma comparação entre o que cada infra-estrutura fornece e quais as diferenças ou variações. Assim forneceremos uma conclusão das utilizações das infra-estruturas estudas de acordo com os mecanismos de detecção e recuperação de defeitos.

Capítulo 2

Visão Geral de Tolerância a Falhas

Neste capítulo são descritos os conceitos principais de tolerância a falhas necessário para o entendimento deste trabalho. Apesar da existencia de técnicas diferentes de tolerância a falhas, este trabalho tem como foco principal o estudo dos mecanismos de detecção e recuperação de defeitos.

2.1 Sistemas Distribuídos

De acordo com Tanembaum e Steen, um sistema distribuído é "*um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente*" [1].
Um sistema distribuído oculta dos usuários a comunicação entre os vários computadores que o compõem. Um sistema distribuído deve estar continuamente disponível mesmo que alguma parte esteja indisponível temporariamente, pois os usuários podem perceber quando uma parte esta avariada ou esta sendo adicionada.
Os sistemas distribuídos costumam ser organizados por meio de uma camada de software, que é situada logicamente entre uma camada de nível mais alto, composta de usuários e aplicações, e uma camada subjacente, que consiste em sistemas operacionais e facilidades básicas de comunicação. Devido a sua localização, essa camada de software é chamada de *middleware*[1] (Figura 1). O *middleware* é utilizado para esconder a heterogeneidade e para mascarar a complexidade dos mecanismos de redes e dos computadores em um sistema distribuído.

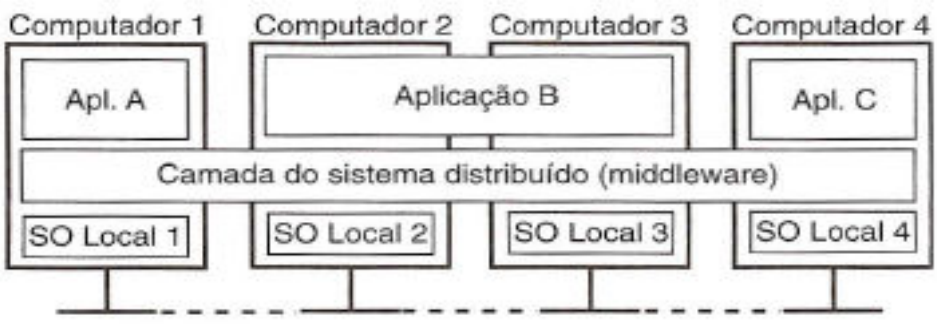

Figura 1. Sistema Distribuído Organizado com Middleware[1].

Um atributo importante no desenvolvimento de um sistema distribuído é a transparência. Um sistema transparente deve ocultar dos usuários e programas o fato de seus processos e recursos estarem distribuídos por vários computadores[1]. Alguns tipos de transparência são descritos na Tabela 1. A transparência de acesso trata de ocultar diferenças em representação de dados e o modo como os recursos podem ser acessados por um usuário, com isso podemos chegar a um acordo de como os dados devem ser tratados por máquinas e sistemas operacionais diferentes. A transparência de migração diz-se dos sistemas distribuídos nos quais os recursos podem ser movimentados sem afetar o modo como podem ser acessados, assim os recursos podem ser relocados enquanto estão sendo usados.

Tabela 1 Diferentes Formas de Transparencia em um Sistema Distribuido

Transparência	*Descrição*
Acesso	Oculta diferenças na representação de dados e no modo de acesso a um recurso
Localização	Oculta o lugar em que um recurso esta localizado
Migração	Oculta que um recurso pode ser movido para outra localização
Relocação	Oculta que um recurso pode ser movido para outra localização enquanto em uso
Replicação	Oculta que um recurso é replicado
Concorrência	Oculta que um recurso pode ser compartilhado por diversos usuários concorrentes
Falha	Oculta a falha e a recuperação de um recurso

2.1.1 Propriedades de sistemas distribuídos

As propriedades de um sistema distribuído são estabelecidas pela sua execução. Lamport define que há duas classes importantes de propriedades para se descrever problemas em sistemas distribuídos: *safety* e *liveness*[2]. As Propriedades de *safety* especificam que o sistema nunca alcançará alguns estados indesejáveis. Por exemplo, uma atualização em um banco de dados réplicado sempre terminará com todas as réplicas consistentes. As propriedades de *liveness*, por sua vez, especificam que certos estados válidos considerados desejáveis ocorrerão em algum momento da execução do sistema. Por exemplo, quando se envia uma mensagem entre duas aplicações, adiante o destinatário receberá a mensagem correta do remetente. Idealmente um sistema distribuído tolerante a falhas deve satisfazer todas as suas propriedades de *safety* e *liveness*. Existem quatro combinações possíveis entre estas propriedades que também podem favorecer tolerância a falhas.
Sistemas que fornecem *safety* e *liveness* são denominados de mascarados(Do inglês: *masking*) e são mais caros, mais rigorosos e mais desejáveis para implementação, pois fornecem maior transparência a falhas. Sistemas que garantem *safety*, mas não *liveness* são chamados *fail*-safe. Tais sistemas garantem a consistência das operações, embora não forneçam garantias de progresso. No exemplo do carro esperando num semáforo, se o semáforo nunca saísse do vermelho, o sistema seria *fail-safe*, já que, apesar de não fazer progresso, ele evitaria a ocorrência de acidentes.

2.2 Falha, Erro e Defeito

Na terminologia de tolerância a falhas, os termos “falha”, “erro” e “defeito” têm significados distinto e bem definidos[3]. Um **defeito** (*failure*) ocorre quando o comportamento externamente observável do sistema se desvia de suas especificações iniciais. Um estado errôneo de um sistema é um estado interno que pode levá-lo a um defeito a partir de uma seqüência de transições validas. O **erro** (*error*) é parte de um estado errôneo que constitui uma diferença de um estado

valido. **Falhas** (*fault*) são causas, físicas ou algorítmicas, de um erro. Conforme a Figura 2, um erro é uma manifestação de uma falha num sistema e um defeito é a manifestação do erro num sistema. No entanto, uma falha não necessariamente provocará um erro, pois a falha pode estar presente no sistema, mas o erro não se manifesta porque o componente da falha não foi utilizado. Analogamente, um erro não necessariamente conduz a um defeito.

Figura 2. Relação entre falha, erro e defeito

A Figura 3 mostra uma esquematização para os conceitos de falha, erro e defeito propostos por Johnson[4] denominada Modelo dos Três Universos. De acordo com este modelo, as falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário. Por exemplo: um chip de memória que apresenta uma falha do tipo preso-em-zero (*stuck-at-zero*) em um de seus bits (falha no universo físico) pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro no universo da informação) e como resultado o sistema exibe um valor incorreto para o usuário num determinado dia(erro no universo do usuário). Vale ressaltar que falhas também podem estar no universo da informação, por exemplo, um *bug* em um programa normalmente é visto com uma falha de projeto[3].

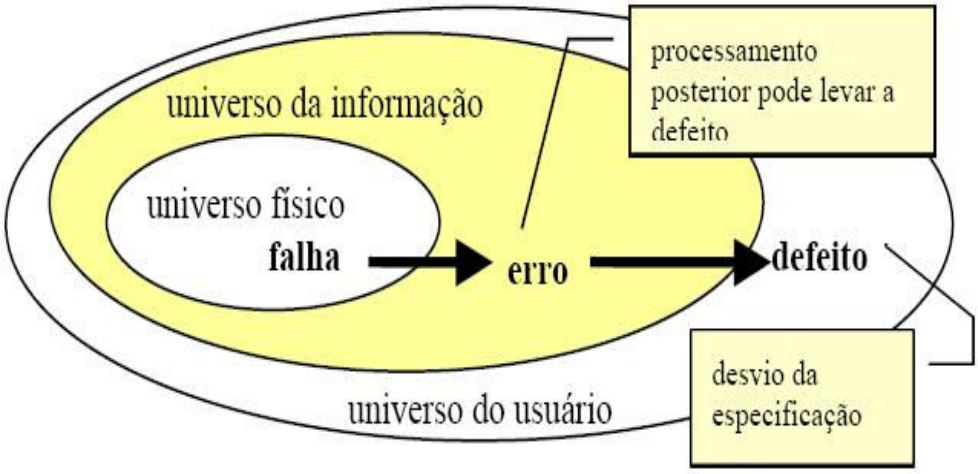

Figura 3. Modelo de 3 Universos: falha, erro e defeito[5].

Existem duas abordagens, como mostrado na Figura 4, que são usadas para garantir a confiabilidade de um sistema: Prevenção de Falhas(*Fault Prevention*), que visa assegurar que todas as possibilidades de ocorrência de falha foram removidas do sistema durante o desenvolvimento[3], e Tolerância de Falhas (*Fault Tolerance*) que supõe que a implementação de um sistema não é perfeita e visa tornar o sistema capaz de lidar com falhas em tempo de execução, a fim para garantir um certo grau de confiabilidade[3]. A prevenção de falhas ainda é dividida em dois aspectos: Previsão de Falhas (*Fault Avoidance*), onde são selecionadas técnicas e tecnologias para evitar a introdução de falhas durante a construção do sistema, e Remoção de Falhas (*Fault Removal*), que se preocupa em checar a implementação do sistema e remover a falhas latentes no sistema. Após a remoção das falhas, o sistema pode ser posto em operação[3].
Na fase de prevenção de falhas, os defeitos são corrigidos durante o desenvolvimento do sistemas. A previsão de falhas utiliza técnicas para previnir que falhas não seram introduzidas no sistema durante a fase de implementação, mas como é difícil detectar todos os defeitos de forma precisa, depois da implementação utiliza-se a remoção de falhas para que defeitos encontrados

depois da fase de implementação sejam removidas. Depois de executada essas duas técnicas de prevenção o sistema pode ser utilizado. Na fase de tolerância de falhas são usadas técnicas para que os defeitos existentes, durante a execução do sistema, sejam detectados e recuperados sem que seja afetada sua execução.

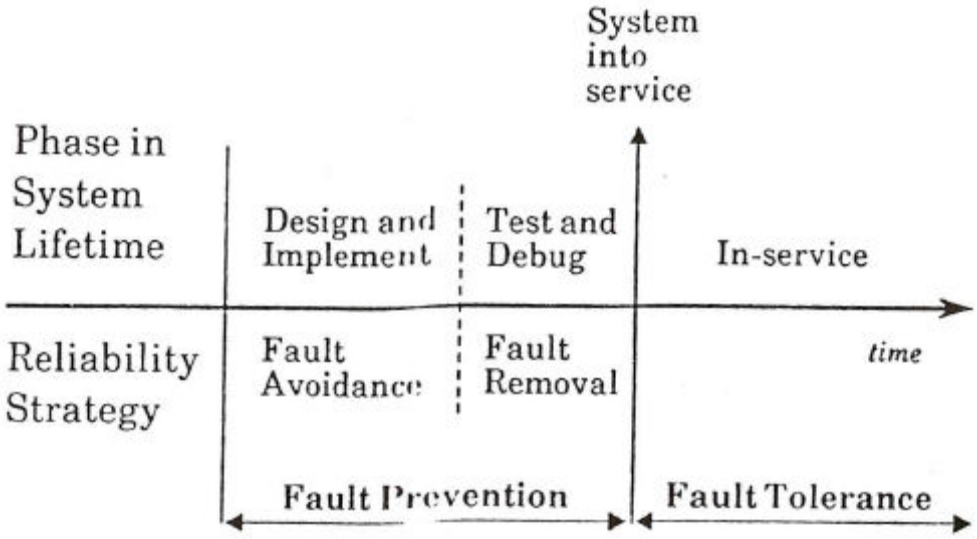

Figura 4. Abordagens de confiabilidade[3].

2.3 Tolerância a Falhas

A tolerância a falhas é uma técnica que visa garantir que o sistema se comporta de acordo com sua especificação mesmo quando falhas se manifestam, através de técnicas de prevenção e remoção de erros.

Um sistema distribuído fornece o serviço de tolerância a falhas quando, no evento de um problema, se recupera de forma automática, mascarando a falha e evitando que o usuário perceba a sua ocorrência. Certas falhas ocorridas no sistema são mascaradas de forma que o usuário não perceba que um recurso deixou de funcionar bem e que o próprio sistema já se recuperou da falha.

O objetivo de tolerância a falhas é alcançar dependabilidade (*dependability*), que indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido[5]. A tolerância a falhas está intimamente ligada à confiabilidade (*reliability*), capacidade de atender a especificação, dentro de condições definidas, durante certo período de funcionamento e condicionado a estar operacional no inicio do período[5] e o controle de falhas. Em um sistema distribuído, confiabilidade envolve alguns atributos adicionais:

- **Disponibilidade** (*availability*): probabilidade de um sistema estar operacional em um dado intervalo de tempo[5];
- **Segurança contra acidentes** (*safety*): probabilidade do sistema ou estar operacional e executar sua função corretamente ou descontinuar suas funções de forma a não provocar dano a outros sistemas ou pessoas que dependam dele[5];
- **Facilidade de manutenção:** facilidade com que um sistema que falhou pode ser consertado [1].

2.3.1 Modelo de falhas

Segundo Schneider[6] um modelo de falhas é uma coleção de atributos e um conjunto de regras que governam a interação entre componentes que falham. Como os sistemas distribuídos são, geralmente, desenvolvidos baseados na comunicação Cliente/Servidor, se essa comunicação não está fornecendo o serviço corretamente, isso significa que o servidor, o canal de comunicação, ou ambos não estão executando da maneira esperada. Porém, nem sempre as falhas ocorrem pelo mau funcionamento do servidor. Se tal servidor depender de outros servidores para prestar seus serviços adequadamente, pode ser que a causa do problema tenha de ser procurada em algum outro lugar[1]. Por este motivo foram criadas algumas formas de classificação das falhas. O modelo clássico de falhas em sistemas distribuídos (Figura 5) é baseado no esquema de Cristian[7]:

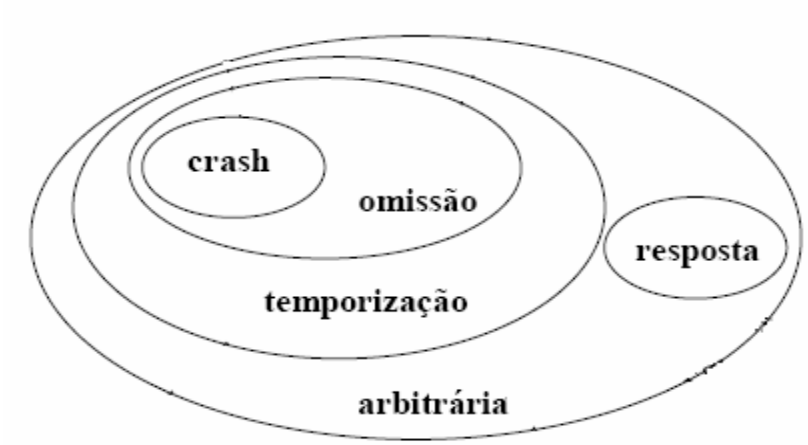

Figura 5. Modelo de falhas em sistemas distribuídos[5].

As diversas classes de falhas nesse modelo indicam o comportamento dos componentes falhos do sistema, quando as falhas ocorrem.

- **Falha por Queda**(*crash fault*): o processo pára de funcionar, mas estava funcionando corretamente até parar. É impossível saber se o processo falhou ou se simplesmente está muito lento[8]
- **Falha por Omissão**(*omission fault*): o processo não consegue responder a requisições que chegam.
- **Falha de Temporização**(*timing fault*): a resposta do processo se encontra fora do intervalo de tempo(adiantada ou atrasada).
- **Falha de Resposta**: a resposta do servidor do processo está incorreta. Podendo a resposta ser um valor errado do servidor ao que lhe foi requisitado, ou um desvio do estado correto devido a uma requisição que não pôde ser reconhecida pelo servidor[1].
- **Falha Arbitrária ou Bizantina**(*byzantine fault*): um processo pode produzir respostas arbitrárias em momentos arbitrários. Um servidor produz saídas que nunca deveria ter produzido, mas que não podem ser detectadas como incorretas. Porém, o servidor pode estar intencionalmente trabalhando maliciosamente com outros servidores para produzir respostas erradas.

Schneider desenvolveu uma extensão do modelo de Cristian, onde ele adiciona o modelo *fail-stop* e desmembra a falha por omissão em omissão de envio e omissão de recebimento.

- ***Fail-stop***: o processo pode parar de funcionar, mas isso pode facilmente ser percebido pelos processos vizinho[8].
- **Omissão de Envio**: o processo não consegue enviar mensagens.
- **Omissão de Recebimento**: o processo não consegue receber mensagens que chegam.

Os dois modelo refletem falhas que afetam as trocas de mensagens entre os nós de comunicação.

2.3.2 Escopo de falhas

As falhas ainda podem ser classificadas de acordo com o seu escopo. Considera-se que há três escopos possíveis para uma falha: falhas físicas(*physical faults*), falhas de interação(*interaction faults*) e falhas de projeto(*design faults*)[9,10]. Podem ainda ser classificadas em termos de Extensão, Valor e Duração. A extensão(*Extent*) de uma falha é dita local se os erros gerados por uma falha afetam somente componentes de uma única variável lógica em um módulo. Falhas são ditas distribuídas, se afetam duas ou mais variáveis lógicas ou subsistemas. O valor(*Value*) de uma falha transforma as variáveis em um valor determinado ou indeterminado. Do ponto de vista de duração(*Duration*), uma falha é dita transiente se ocorre somente em um período de tempo(menor que um limiar determinado), depois desaparece e não ocorre novamente se a operação for repetida. Se a falha continua a existir após o limiar até que o componente faltoso seja substituído, ela é dita permanente[10].

- **Falhas Físicas** estão relacionadas aos componentes físicos do sistema, devido a mal-funcionamento de hardware, fadiga de componentes físicos, perturbações externas(temperatura, radiação,...).
- **Falhas de Projeto** acontecem devido a erros cometidos na fase de projeto. São causados principalmente por especificações erradas, ambíguas ou incompletas e podem ocorrer tanto em nível de hardware, que são difíceis de ser eliminadas, quanto em nível de software, que podem ser corrigidas. Por exemplo, em 1971 na França, durante um experimento metereológico, de 141 balões atmosféricos, 72 explodiram devido a uma falha no software que controlava o experimento[9].
- **Falhas de Interação** são causadas por usos indevidos do operador durante a operação e manutenção do sistema. Podem ser classificadas em Não-Maliciosas, quando o operador viola um procedimento da operação sem ter consciência das possíveis conseqüências de seu ato, e Maliciosas, quando pessoas não-autorizadas e mal-intencionadas levam o sistema a falhar, por exemplo, devido ao efeito de Cavalos de Tróia (*Trojan-Horse*), Vírus ou *Worms*[9].

A combinação entre falhas de interação e falhas de projeto resulta nas chamadas Falhas Humanas[5].

2.3.3 Conseqüência de falhas

Em geral, a manifestação de falhas tem como conseqüência problemas econômicos. Em alguns casos, a manifestação de falhas pode provocar eventos catastróficos, até mesmo com perda de vidas humanas. A Tabela 2 apresenta exemplos de defeitos desastrosos nas décadas de 80 e 90[10]. Os exemplos estão relacionados às causas das falhas e defeitos e à dependabilidade do sistema.

Tabela 2 Exemplos de defeitos desastrosos

	Escopo			Extensão		Disponibilidade (*availability*)/ Confiabilidade (*reliable*)	Segurança contra acidentes (*safety*)
	Físico	Projeto	Interação	Local	Distribuído		
Junho 1980: Falsos alertas da Defesa Aérea Norte Americana (NORAD)	X			X		X	
Abril 1981: Adiamento do primeiro lançamento da espaçonave *Shuttle*		X		X		X	
Junho 1985 – Janeiro 1987: Doses excessivas de Radioterapia(Therac-25)		X		X			X
Novembro 1988: Internet *Worm*		X	X			X	
15 de Janeiro de 1990: Telefonemas de longa distancia fora do ar por 9 horas nos Estados Unidos.		X			X	X	
Fevereiro 1991: Mísseis Scud(Guerra do Golfo)		X	X	X		X	X
Novembro 1992: Colapso no sistema de comunicação do serviço de ambulâncias em Londres		X	X		X	X	X
26 e 27 de Junho 1993: Autorizações não permitidas nas operações de cartão de credito na França	X	X			X	X	

Para exemplicar alguns do acontecimentos citados na Tabela 2 temos que, na Guerra do Golfo em fevereiro de 1991 foram noticiados vários relatos de falhas em mísseis. Em junho de 1993, durante dois dias, não foi autorizada nenhuma operação de cartão de créditos em toda a França. Varias missões da NASA e Marte terminaram em fracasso total ou parcial.

2.3.4 Fases da Tolerância a Falhas

A classificação de técnicas de tolerância a falhas mais comum é composta por 4 fases: detecção de erros, confinamento e avaliação de danos, recuperação de erros e tratamento de falhas. Não é necessário o uso de todas as técnicas ao mesmo tempo. A combinação das técnicas depende do uso no serviço fornecido pelo sistema, pois em alguns casos o sistema pode se tornar caro demais se todas as fases forem empregadas.

- **Detecção de erros**(*Error Detection*): para o sucesso de um sistema tolerante a falhas, esta técnica deve ser a primeira a ser executada, pois a manifestação de uma falha em um sistema pode gerar um erro e um erro, diferentemente de uma falha pode ser detectado por um mecanismo de detecção(ex: duplicação e comparação)[3]. Na literatura de sistemas distribuídos, normalmente usa-se o termo detecção de defeitos(e recuperação de defeitos), já que a detecção de um problema envolve um ou mais componentes do sistema manifestando um defeito que pode ser observado pelos outros componentes(ex: o componente faltoso está rodando num computador que pára de funcionar.
- **Confinamento e avaliação de Danos**(*Damage Confinement and Assessment*): depois da detecção de um erro, devido ao intervalo entre a manifestação e a detecção, alguns estados podem propagar informações erradas dentro do sistema, conduzindo os próximos estados

ao erro[3]. Em consequencia disso, um sistema tolerante a falhas deve ser capaz de impedir essa propagação.

- **Recuperação de Erros**(*Error Recovery*): a recuperação de erro visa transformar o estado errôneo atual em um estado livre de erros para que a operação normal do sistema continue. A recuperação pode ser de duas formas: por retrocesso(*backward error recovery*) e por avanço (*forward error recovery*).
- **Tratamento de Falhas**(*Fault Treatment*): o tratamento de falhas consiste em 2 etapas. Primeiro é localizado a origem da falha, em seguida a falha é reparada ou o restante do sistema é recuperado para evitar a transiência da falha[3].

2.3.5 Redundância

Redundância é a chave para a tolerância a falhas. Todas as técnicas de tolerância a falhas envolvem alguma forma de redundância. Um significado usualmente sugerido de redundância é que o sistema inclui componentes que, durante sua operação normal, não são utilizados porque existe outra parte de sistema que faz o mesmo serviço[8]. Se o sistema nunca falhar, seus componentes nunca serão usados. Redundância pode aparecer de 3 formas diferentes:

- **Redundância de Informação**: bits ou sinais extras são armazenados ou transmitidos junto ao dado, sem que contenham qualquer informação útil[5]. Estes bits servem para detectar erros e mascarar falhas. Exemplos incluem *checksums*(adiciona informações extras a um bloco de informação para possibilitar detecção de erros), paridade(adiciona bit(s) para manter nos bits armazenados uma quantidade par, ou ímpar dependendo da paridade implementada, de bits com o valor 1).
- **Redundância de Tempo**: re-executa computações com as mesmas entradas. Utilizada para indicar se a falha é transiente ou permanente. Usada em sistemas onde o tempo não é critico, ou que possuem processadores parcialmente ociosos[5].
- **Redundância Física**: são adicionados equipamentos(redundância de hardware) ou componentes de *software* extras(redundância de software) para possibilitar que o sistema tolere a perda ou o mau funcionamento de alguns componentes[1]. Na redundância de hardware são replicados componentes, unidades de memória, fontes de alimentação, dentre outros, com a finalidade de detecção de erros ou reparo do sistema transferindo as tarefas de um componente falho para outro redundante. Na redundância de software, ocorre a utilização de versões distintas do mesmo software, desenvolvidas a partir da mesma especificação, porém implementadas utilizando abordagens e equipes de programação distintos.

Apesar de redundância ser um recurso indispensável para tolerância a falhas, o seu uso, em qualquer projeto, deve ser bem ponderado para não haver o aumento de falhas no sistema e desviar de sua dependabilidade. Além disso, redundância pode implicar em um aumento significativo no custo do desenvolvimento do software.

2.3.6 Replicação

Replicação é um caso particular da redundância, onde apenas uma porção do sistema em execução será distríbuido entre as cópias do sistema. A forma mais comum de se fornecer tolerância a falhas em sistemas é por meio de réplicação das funcionalidades. A replicação consiste em manter cópias de um mesmo objeto em dispositivos diferentes para utilizar na recuperação do sistema em caso de falha de algum dispositivo durante a execução do sistema. A

motivação para o uso de réplicação estão no melhoramento de serviços de desempenho, elevar a disponobilidade, ou criar tolerância a falha.
Quando um dado é réplicado uma exigência comum é a transparência da réplicação, ou seja, o cliente normalmente pode não estar ciente de que existem múltiplas cópias *físicas* do dado.os dados são organizados como objetos lógicos individuais e os clientes identificam apenas um item em cada caso quando uma requisitam uma operação para ser desempenhada. As operações podem ser executadas em mais de uma cópia física, porém o cliente recebera apenas um único conjunto de valores.
A réplicação pode ser dividida principalmente de duas formas: réplicação ativa e réplicação passiva.

- **Réplicação ativa**: todas as réplicas têm o mesmo papel, sem existir uma réplica centralizadora.
- **Réplicação Passiva**: usa um servidor primário. Os outros servem apenas como uma cópia de *backup*, e não interagem com o cliente.

Existem cinco tipos de replicação: *stateless, cold passive, warm passive, active* e *active with voting*.

- ***Stateless***(sem estado): não utiliza nenhum mecanismo de tolerância a falhas adicional, pois é usada apenas para leitura de dados.
- ***Cold passive***(passiva fria): apenas os objetos primários recebem as requisições do cliente, enquanto os outros funcionam como um *backup*. Um deles será nomeado como primário na ocorrência de alguma falha. A cada intervalo de *checkpoint* o objeto registra seu estado no *log* que é disseminado para os demais objetos. Desse modo, defeitos do objeto primário podem ser recuperados através da eleição de um novo objeto primário.
- ***Warm passive***(passiva quente): o recebimento das mensagens e o funcionamento dos mecanismos de *checkpoint* e logging são similares ao da passiva fria, porém os objetos *backup* atualizam o estado do objeto imediatamente após o *checkpoint* pelo seu mecanismo de recuperação, independente da ocorrência de falhas.
- ***Active***(ativa): tipo mais seguro e caro de réplicação. Não utiliza mecanismos de *logging* e recuperação porque todos os objetos recebem as requisições do cliente e as processam imediatamente.
- ***Active with voting***(ativação com votação): funciona de forma similar à ativa, porém, as respostas passam por uma votação e a que é considerada mais correta é enviada ao cliente.

2.3.7 Transações

O objetivo das transações é de garantir que todos os objetos gerenciados pelo servidor permaneçam num estado consistente quando são acessadas por múltiplas transações e na presença de servidores falhos. Uma transação distribuída envolve um ou mais hospedeiros da rede. Transações devem garantir a integridade dos dados e a consistência dos objetos, pois transações distribuídas devem alterar os estados de vários processos em máquinas diferentes. Transações distribuídas, como qualquer outra transação, deve dar suporte às propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Todas as mudanças resultante de uma transação são aplicados por inteiro ou não são aplicados(atomicidade); transações bem-sucessidas não levam um banco de dados de um estado correto para um estado incorreto(consistência); execuções concorrentes são equivalentes a execuções seriais das mesmas transações usando uma única cópia do banco de dados(isolamento ou serialização de uma cópia); os efeitos das transações sobrevivem a defeitos de banco de dados(durabilidade).

Uma transação termina com sucesso ou aborta em uma das duas formas ou o cliente aborta ou o servidor aborta. Se um processo do servidor falha inesperadamente, ele será substituido por um novo. Este novo servidor aborta qualquer transação que não tenha obtido sucesso e utiliza procedimentos de recuperação para restaurar os valores dos objetos pelos valores produzidos pela mais recente transação de sucesso.para lidar com falhas dos clientes durante a transação, servidores podem fornecer ao cliente um limite de tempo para que transações que não foram completadas sejam abortadas. Se um servidor falha enquanto uma transação está em progresso, o cliente ficará ciente disto quando uma das operações retornar uma exceção após o termino do limite de tempo. Se um servidor falha mas se recupera durante o progresso da transação, esta transação não será válida por muito tempo então o cliente será informado por uma exceção pela próxima operação.

2.3.8 Modelos computacionais

Sistemas distribuídos não possuem memória compartilhada nem relógio global. Toda interação entre processos deve ser realizada por troca de mensagens. Assim, os sistemas distribuídos são classificados como síncrono ou assíncrono. Num sistema síncrono existe um limite de tempo para transmissão de mensagens e respostas dos processos[8]. Se essas características não são satisfeitas, o sistema é denominado de assíncrono. O modelo assíncrono é mais fraco já que não é imposto nenhum limite de tempo arbitrário para a entrega das mensagens, os procesos podem ter atrasos diferentes, e a comunicação é o único mecanismo para sincronização de processos no sistema, por isso qualquer algoritmo que funcione no modelo assíncrono também funciona em outros modelos. Entretanto, sistemas síncronos são mais propensos a comportamentos incorretos caso sua implementação viole a condição de tempo. Sendo assim, os sistemas assíncronos são maioria em aplicações distribuídas. Adicionalmente, o modelo assíncrono representa de forma mais fidedigna o comportamento de redes de computadores de grande escala, como a Internet.

2.3.9 Detecção de defeitos(*failure detection*)

A detecção é a primeira atividade que deve ser realizada para se prover tolerância a falhas, pois ela identifica a parte do sistema onde a falha se manifestou. Em um sistema distribuídos, a ocorrência de um defeito pode afetar a seqüência da execução do sistemas, o que pode causar efeitos catastróficos. Por isso, defeitos devem ser detectados pelo sistema tão cedo quanto possivel. Em principio, quanto mais técnicas de detecção de defeitos forem usadas, maior será a confiabilidade do sistema. A principal limitação na escolha de diferentes técnicas de detecção de defeitos é o seu custo, tanto do ponto de vista computacional quanto em termos de recursos de hardware e software redundantes[3]. Existem diversas técnicas de detecção de defeitos, mas aqui apresentaremos apenas as técnicas estudadas para nossa comparação:

- **Consenso**: considerando um conjunto de processos onde cada um possui um valor inicial, eles decidem sobre um dos valores iniciais propostos por um subconjunto deles. Consenso é uma forma de acordo. O objetivo geral de algoritmos de consenso é que processos sem falhas cheguem a um acordo sobre uma questão, por exemplo, eleição de líder ou se um processo falhou ou não. O acordo na presença de falhas arbitrarias em sistemas síncronos é conhecido como **Acordo Bizantino**. O acordo sobre um vetor de valores é chamado de **Consistência Interativa**.
- **Broadcast Atômico**: permite aos processos o envio de mensagens confiáveis em *broadcast*. Todos os processos envolvidos devem concordar sobre as mensagens enviadas e a ordem em que foram enviadas. O broadcast atômico deve garantir uma semântica

"tudo ou nada", ou todos as mensagens sao entregues e na mesma ordem ou nenhuma é entregue.

- **Detecção de defeitos não-confiável**: é proposto um módulo de programa que atua como um oráculo de estados funcionais dos processos vizinho. Um módulo detector de defeitos local é executado por cada processo. Esse módulo irá monitorar uma parte dos processos do sistema e manterá uma lista dos processos suspeitos de falhas. Os módulos podem cometer erros na adição de processos corretos na lista de suspeitos. Assim, cada módulo pode repetidamente adicionar e remover processos da lista de suspeitos. Duas propriedades para este tipo de detector de defeitos são *completeness* quando um detector de defeitos suspeita que um processo falhou, se algum módulo detector de defeitos local suspeitar que o processo está falho, e *accuracy* quando o detector de defeitos não suspeitará que um processo correto tenha falhado. Basicamente nos detectores de defeitos não-confiavéis todos os processos enviam mensagens "estou vivo" uns aos outros. Se um processo ultrapassar o *time-out* ele será adicionado à lista de suspeitos, caso seja recebida a mensagem "estou vivo" deste processo, ele será removido da lista de suspeitos e aumenta o *time-out* do processo para que evitar nova suspeita errônea. Mensagens de *heartbeat*, membros do grupo de gerentes mandam periodicamente mensagens de *heartbeat* entre eles e agentes enviam periodicamente *heartbeat*s para o gerente do grupo, *heartbeat*s tem por conseqüência uma minimização de alarmes falsos e de overhead. Funciona com o envio periódico de mensagens de *heartbeat* para todos os demais módulos do sistema. Para cada mensagem de *heartbeat* é calculado um *timeout*, tempo de atraso para a transferência de mensagens. Caso o tempo decorrido desde o envio da mensagem de *heartbeat* exceda o *timeout*, o processo é classificado como suspeito. Caso o tempo decorrido esteja dentro do *timeout*, o processo é classificado como ativo. Esta classificação é efetuada e atualizada por cada módulo detector de defeitos.

2.3.10 Recuperação de erros(*error recovery*)

Recuperação transforma o estado do sistema que contém um ou mais erros em um estado sem erros detectáveis e falhas que podem ser ativadas novamente[10]. A recuperação pode ser de duas formas: recuperação por retorno(*backward error recovery*) e recuperação por avanço(*forward error recovery*)

- **Recuperação por Retorno**: o sistema retorna ao estado correto anterior à detecção do erro e se reconfigura realocando processos(Figura 6-A) e escolhendo caminhos alternativos de comunicação entre os processos. Este estado salvo é chamado de ponto de recuperação(*checkpoint*)[10] e corresponde a um estado global consistente. As técnicas de recuperação por retorno não utilizam tanta redundância, os processos salvam seus estados independentemente. A técnica de recuperação por retorno não é utilizada em uma vasta gama de sistema de tempo real usado em controle de processos contínuos devido à impossibilidade de retornar(*rollback*) a um estado seguro armazenado pelo sistema. A recuperação por retorno pode causar no sistema um efeito dominó. Ao desfazer a computação, um processo pode deixar mensagens órfãs(perdidas) na rede. Para que isso não aconteça é necessário que os processos, ao desfazerem a computação, dissemínem as informações para que os outros processos também desfaçam suas computações. Esse efeito pode no pior caso, fazer o sistema retornar ao início, por isso são necessárias restrições à comunicação entre os processos.

- **Recuperação por Avanço**: o sistema avança para um novo estado ainda não ocorrido desde a última manifestação de erro[5] (Figura 6-B). O sistema desfaz toda a computação realizada após a falha, para refazê-la sem repetir a falha(com mesmo software ou por uma outra versão do software escrito com a mesma especificação).

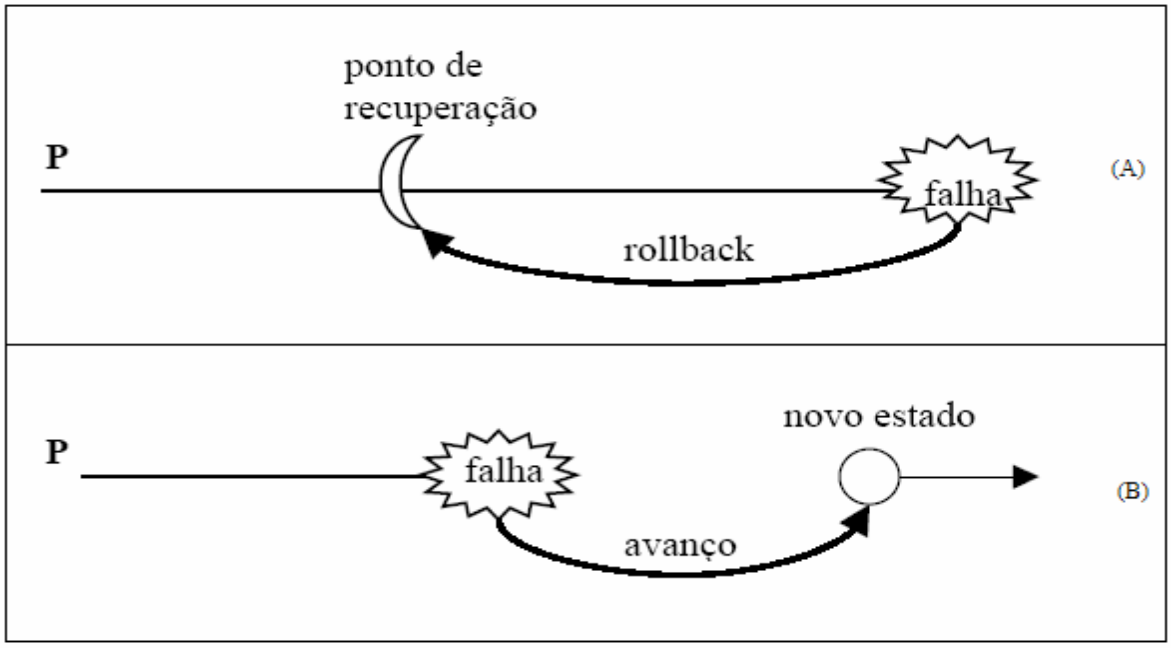

Figura 6. Recuperação por retorno e por avanço[5].

2.4 Aplicações de Sistemas Tolerantes a Falhas

Como apresentado nas seções anteriores, existem várias técnicas para a implementação de sistemas tolerantes a falha. A utilização de todas elas na construção de um sistema, embora desejável, é inviável pois pode elevar o custo do projeto de forma excessiva. Por isso, a escolha da especificação do projeto de acordo com a sua aplicação e sua exigência de dependabilidade devem ser bem exploradas. Nesta seção mostramos algumas aplicações que tradicionalmente exigem a implementação de um ou mais mecanismos de tolerância a falhas.
As áreas tradicionais onde são empregadas sistemas tolerantes a falhas são:

- Aplicações críticas de sistemas em tempo real, como medicina, controle de processos e transporte aéreo;
- Aplicações seguras de tempo real, como transporte urbano;
- Aplicações em sistemas de tempo real de longo período de duração sem manutenção, como viagens espaciais, satélites e sondas;
- Telefonia e telecomunicações;
- Aplicações comerciais de alta disponibilidade como sistemas de processamento de transações e servidores de redes.

Exigências de disponibilidade e confiabilidade são encontradas em todas as áreas, mas os sistemas tolerantes a falha são caros e portanto empregados apenas em situações em que a sua não-utilização acarretaria prejuízos irrecuperáveis[5].

2.4.1 Sistemas de tempo real

Sistemas de computação de tempo real são empregadas em aplicações de controle, de supervisão e de comunicação. Condições para aplicações desses sistemas são:

- Disponibilidade de curto intervalo de tempo para reconhecimento de erros que não prejudiquem o processamento do sistema;
- Impossibilidade de uso de recuperação por retorno já que eventos passados são descartados;
- Exigência de redundância passiva para garantir a continuidade do processamento em caso de falhas.

Exemplos de sistemas em tempo real tolerantes a falhas são os sistemas FTMP(*Fault Tolerant Multiprocessor*) e SIFT(*Software Implemente Fault Tolerance*)[3].

2.4.2 Sistemas digitais de telefonia

Sistemas para telefonia empregam técnicas de tolerância a falhas por apresentar requisitos estritos de disponibilidade e alta qualidade e longa vida ao uso dos componentes. Requisitos para aplicações nessa área são:

- Detecção e localização automática de erros(em software e hardware);
- Tratamento automáticos de erros(reconfiguração do sistema);
- Substituição de componentes faltosos durante o período de operação normal do sistema.

A principal técnica de tolerância a falhas empregada na construção deste tipo de sistema é a duplicação de componentes de hardware. Um exemplo de sistemas de telefonia tolerante a falhas são os sistemas ESS 1A[3].

2.4.3 Sistemas de Processamento de Transações

Sistemas de processamento de transações necessitam da existência de uma base comum de dados usada interativamente e concorrentemente com vários usuários em maquinas diferentes. Como estes sistemas são muito usados em transações financeiras, alguns requisitos para essas aplicações são:

- Integridade e garantia dos dados em sua base de dados;
- Alta disponibilidade para processamento contínuo;
- Tratamento de erros sem interrupção do sistema.

Integridade e consistência dos dados são os requisitos mais importantes para este tipo de aplicação(propriedades de *safety*), por isso suas operações são baseadas no modelo *fail-stop*. Caso ocorra um erro, o sistema pára sem propagar este erro. Os sistemas Tandem e Stratus são dois exemplos de sistemas comerciais de transação tolerantes a falhas[5].

Tandem foi o primeiro sistema tolerante a falhas proposto para o uso geral para aplicações comerciais. O Tandem é um sistema contínuo (*non stop*) para aplicações *on-line* de transações em banco de dados. Tandem adotou a estratégia de “pares de processos”, onde cada processo executando em um programa pode ter um processo de *backup* que é executado em um separado módulo de processamento.

Stratus foi construído para competir com o Tandem na tolerância a falhas em processamento de transações *on-line*. Cada modulo compara resultados fornecidos por elementos duplicados do sistema, quando a comparação indica erro, nenhum resultado é fornecido como saída.

Capítulo 3

Infra-estruturas de *Middleware*

Neste capítulo são descritos os conceitos das infra-estruturas de *middleware* escolhidas, enfatizando os mecanismos de tolerância a falhas disponíveis em cada uma. Por fim, faremos uma comparação entre os mecanismos igualmente fornecidos pelas infra-estruturas e suas variações de mecanismos.

3.1 Sprint

3.1.1 Introdução

Sprint é uma infra-estrutura de middleware para alto desempenho e alta disponibilidade de gestão de dados, pois ele fornece a funcionalidade de um banco de dados em memória principal (IMDB – *in-memory database*) que geralmente são limitados pela capacidade de memória da máquina que executa o IMDB[12]. Sprint não necessita de uma forte detecção de defeitos para garantir consistência, e permitir uma reação rápida a falhas, pois experimentos realizados em um *cluster* usando TPC-C e um micro-benchmark mostrou que Sprint fornece um bom desempenho e escalabilidade. Projetado para arquiteturas de *middleware* onde as transações são pré-definidas e parametrizadas antes da execução. Sua arquitetura é divida em servidores físicos (*physical servers*), parte da infra-estrutura de hardware, e servidores lógicos (*logical servers*), componentes de software que compõem o sistema. Há três tipos de servidores lógicos: servidores de borda (ES-*Edge Servers*), servidores de dados (DS-*Data Servers*) e servidores de durabilidade (XS-*Durability Servers*). O servidor físico pode solicitar qualquer quantidade de servidores lógicos. Por exemplo, apenas um DS, um DS e um XS ou duas instâncias diferentes de DS.
DS operam um IMDB locais e executam transações sem acesso ao disco. XS garantem a persistência das transações e a recuperação de defeitos.
Sprint garante as propriedades ACID (atomicidade, consistência, isolamento e durabilidade) para transações: todas as mudanças resultante de uma transação são aplicados por inteiro ou não são aplicados (atomicidade); transações bem-sucessidas não levam um banco de dados de um estado correto para um estado incorreto (consistência); execuções concorrentes são equivalentes a

execuções seriais das mesmas transações usando uma única cópia do banco de dados (isolamento ou serialização de uma cópia); os efeitos das transações sobrevivem a defeitos de banco de dados (durabilidade).

3.1.2 Arquitetura Sprint

A Figura 7 ilustra a arquitetura Sprint. ES permitem que clientes iniciem transações. Solicitações provindas de uma mesma transação podem ser operadas por um mesmo ES. Novas transações podem ser iniciadas em ES diferentes. ES são iniciados e desligados de acordo com a quantidade de carga ou exigências de tolerância a falhas.
Tabelas de banco de dados são particionadas entre os DS. Tarefas realizadas no banco de dados para DS e o mapeamento de DS para servidores físicos são chamados de configuração de banco de dados (*database configuration*). Para permitir a execução paralela de operações de leitura, os dados são réplicados para múltiplos DS. Isso prejudica as operações de escrita, pois precisam modificar todas as réplicas de forma consistente, por isso as declarações vindas dos clientes são divididas em sub-declarações de acordo com a configuração do banco de dados. A configuração de banco de dados muda na ocorrência de qualquer defeito, então uma nova instância é criada em um servidor diferente.
Todos os estados permanentes são gravados pelo servidor de durabilidade (*durability servers*-XS), incluindo o estado do banco de dados e a configuração de banco de dados. O componente *Log manager* informa ao ES e ao *execution manager* sobre os estados de terminação de transação de atualização (*update*). A recuperação de defeitos é feita pelo *Recovery Manager*, que reconstrói o estado de um servidor de dados defeituoso a partir de um registro (*log*) aramazenado no *Log Manager*.

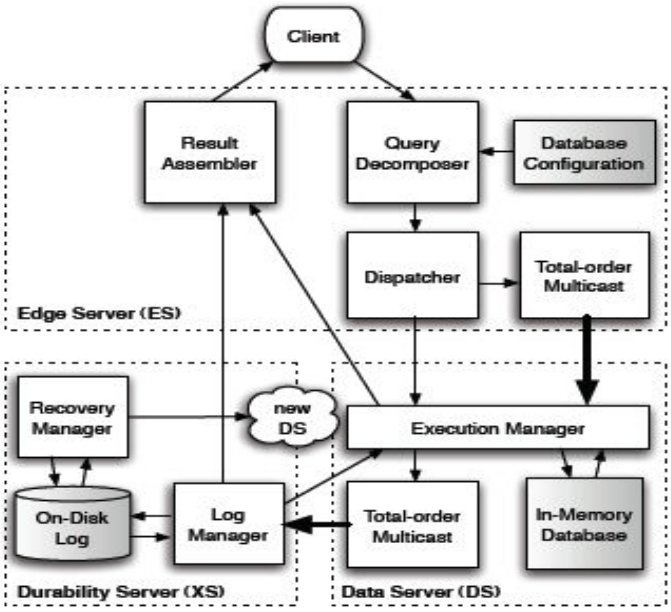

Figura 7. Arquitetura Sprint[12].

3.1.3 Transações

- **Execução de Transações**

ES mantém duas estruturas de dados, *server* e *status*, para cada transação executada. Estas estruturas de dados são mantidas por toda a execução da transação e são descartadas (*garbage*

collected) assim que a transação é abortada ou sucedida. A estrutura *server* mantêm uma lista dos servidores de dados (DS) acessado pela transação e a estrutura *status* registra o tipo atual das transações, locais ou globais.
Na execução de transações locais, declarações SQL recebidas do cliente são passadas ao DS correspondente para serem processadas e a resposta ser repassada ao cliente. Uma transação torna-se global se ela executa uma operação mapeada para mais de um DS ou para um DS diferente do que foi utilizado em uma operação anterior. Transações globais utilizam *multicast* totalmente ordenado para sincronizar suas execuções, evitando *deadlocks*. Cada transação global é utiliza *multicast* apenas uma vez para enviar a identificação da transação, quando o ES percebe que a transação é global, as próximas requisições são enviadas para os DS ustilizando comunicação ponto-a-ponto.
Diz-se que duas transações globais entram em conflito (*conflict*) se elas acessam dados em um mesmo DS e ao menos uma delas altera o dado. Um DS *recebe transações T* quando recebe pela primeira vez uma operação de T.
A execução das transações globais é ordenada por uma seqüência de números. A serialização é garantida por um escalonador local em cada DS e pela certeza de que dois DS diferentes não ordenam a mesma transação com uma seqüência de números diferentes. *Deadlocks* locais são solucionados pela execução da transação pelo IMDB. *Deadlocks* distribuídos são resolvidos evitando ciclos nos escalonadores das transações. caso o IMDB detecte *deadlock* entre duas transações, uma delas será abortada para que se execute apenas uma delas.

- **Terminação de Transações**

Transações apenas de leitura terminam com sucesso (commit) com mensagens do ES para os DS envolvidos na transação. A transação é terminada quando ES recebe um reconhecimento de cada DS. Se o DS falha e não pode enviar o reconhecimento, o ES irá suspeitar de um DS defeituoso então abortar a transação. Reconhecimentos são necessários para assegurar a corretude, apesar de DS defeituosos.
Terminação de transações que modificam o banco de dados (*update*) é mais complexa, pois o sucesso de transações de atualização envolve XS para garantir a sobrevivência de estados da execução em caso de defeitos no servidor. As terminações de transações de atualização são baseadas em *multicast* totalmente ordenado para prover aos servidores de dados uma maneira simples de manter seus estados persistentes (no XS).

- **Terminação com Suspeita de Defeito**

A complexidade da terminação de transações com suspeita de defeito se deve à possibilidade de suspeitas erradas na participação de DS. Para assegurar que todos os servidores envolvidos irão encaminhar a mesma resposta após o termino de transações *update*, o procedimento de terminação é conduzido da seguinte maneira: se o ES suspeita do defeito de um DS durante a terminação da transação, via *multicast* são enviados votos de aborto em relação ao DS. Os votos a ser considerados são os primeiros votos enviados por cada DS. Para DS não suspeitos apenas, um voto será entregue, enquanto para DS suspeitos haverá a possibilidade de vários votos. O *multicast* totalmente ordenado garante que todos os servidores destino entregarão os votos de transação na mesma ordem e, conseqüentemente, chegarão a uma mesma decisão.

3.1.4 Tolerância a Falhas

- **Detecção de Defeitos**

Servidores físicos se comunicam apenas por troca de mensagens, isto é, não existe memória compartilhada. Servidores lógicos podem usar comunicação ponto-a-ponto e *multicast* totalmente ordenado.
Servidores físicos podem falhar por queda (*crash*) mas não por comportamento malicioso (defeitos bizantino). O servidor pode se recuperar após um defeito mas todas as informações contidas na memória principal, antes da queda, serão perdidas. Defeitos nos servidores físicos implicam em defeitos no servidores lógicos.
Sprint utiliza detecção de defeitos não-confiável; um servidor defeituoso possivelmente será detectado por servidores operacionais, mas o servidor operacional pode erroneamente suspeitar que o sistema esta defeituoso caso ele esteja apenas atrasado.

- **Recuperação de defeitos**

A recuperação de defeitos no Sprint é feita individualmente por cada servidor lógico.

- *Edge Server*

Se ocorrer um defeito em um ES durante a execução de uma transação, os DS envolvidos conseguirão detectar o defeito e abortar a transação. Se a falha ocorre durante a execução do protocolo de terminação, a transação finaliza com sucesso ou aborta, dependendo de quando o defeito ocorreu. Se a requisição de um ES para terminar uma transação alcançar todos os DS participantes, eles estão pronto para terminar a transação com sucesso, e seus votos serem entregues, então a resposta será finalizada.
Uma nova instância de ES será, imediatamente, criada em qualquer servidor físico. Durante a inicialização, o ES manda mensagens para um dos XS, perguntando pela configuração atual do banco de dados. O ES estará pronto para processar requisições assim que for recebida a configuração do banco de dados.

- *Data Server*

Recuperação de DS defeituosos é simples pois é apenas necessário criar uma outra instância do servidor em um servidor físico operacional. Com um DS configurado para evitar acesso ao disco, não existe imagem de banco de dados para ser restabelecida de um disco local após a queda. Em conseqüência disso, uma nova cópia do DS defeituoso será implantada em um servidor físico usando o estado armazenado por um XS.

- **Evitando Inconsistências no Processo de Recuperação**

Sprint evita inconsistências para assegurar que transações só podem ser executadas se os DS acessados não são substituídos durante a execução. O *middleware* garante essa propriedade usando *incarnation numbers* e *incarnation numbers vector*.
Incarnation numbers são identificadores únicos para cada instância de um DS. Eles podem ser implementados por uma contagem simples de quantas vezes o DS foi substituído ou “encarnado”. *Incarnation number vectors* contém um *incarnation number* por DS no sistema. No momento da terminação, o *incarnation number* de cada DS envolvido na transação é comparada com o vetor para checar se a transação pode executar.
Quando uma transação é iniciada, é nomeado um vetor com o máximo de dados de *incarnation numbers* percebidos pelo ES. O ES hospeda este vetor e o nomeia como a visão atual do vetor de transações. O vetor designado pela transição será enviada pelo ES como parte do procedimento de terminação da transação.

Quando da suspeita de um DS defeituoso, o ES envia mensagens *multicast* de mudança de DS(*change-DS*) para todos os servidores junto com o identificador de um servidor físico onde a nova instância de DS estará localizada. Após a entrega dessa mensagens, todos os servidores consistentemente aumentam o *incarnation number* de um DS particular e atualizam a configuração do banco de dados.
Mensagens de reconhecimento são enviadas aos DS como parte da execução de uma transição *read-only* global retornando o valor atual do *incarnation number* do DS. O reconhecimento permite ao ES identificar possíveis estados inconsistentes.

- **Reconstruindo o estado de DS defeituosos**

Caso um DS seja replicado, é possível recuperar seu estado caso ele esteja defeituoso. Mas de qualquer forma o estado do banco de dados do DS pode ser recuperado por registros mantidos pelo XS. Assim o DS recuperado necessita da imagem inicial do banco de dados obtido no XS e as atualizações perdidas para se atualizar a imagem. Após esta recuperação as entradas de dados podem ser armazenadas localmente.
A recuperação rápida do DS defeituoso é importante para a disponibilidade, pois as transações que requisitam dados armazenados em um DS defeituoso não podem ser executadas até que o servidor seja substituído. Os DS são réplicados se for exigido uma alta disponibilidade. Transações de atualização falharão apenas se todas as réplicas que estiverem disponíveis para a transação falharem.

- *Durability Server*(XS)

A entrega de mensagens perdidas para a recuperação do XS podem ser feita por um XS operacional. XS também implementam regras para recuperação de DS defeituosos. Cada XS cria, periodicamente, uma imagem no disco, do estado atual do banco de dados. Esse estado é construído a partir das mensagens entregues pelos XS, como parte do protocolo de terminação das transações de atualização.

3.2 Horus

Horus é um sistema que oferece um extenso e flexível modelo de comunicação de grupos. A necessidade de um ambiente de grupos de processo (conjunto de processos que se comunicam utilizando um mesmo endereço de grupo) para computação distribuída representa um grande passo para robustez em aplicações distribuídas com eventos críticos (como perdas financeiras ou de vidas humanas). Grupos de processos podem ser usados para dar suporte a domínios de segurança de alta disponibilidade e uma boa execução de mecanismos de grupo na criação de uma rede inteligente. O sistema Horus fornece um flexível modelo de comunicação de grupos. Essa flexibilidade aplica-se às interfaces do sistema, às propriedades fornecidas pelo protocolo de pilha (camadas de protocolos que podem ser empilhadas um sobre a outra de formas variadas durante a execução) e à própria configuração do Horus, que pode rodar em um espaço de usuário, em um *kernel* de sistema operacional ou microkernel, ou ser dividido entre eles. Horus pode ser utilizado por diversas interfaces de aplicações, até mesmos as que possuem oculta funcionalidades de grupos por trás de sistemas de comunicação UNIX.
Horus fornece suporte eficiente para modelos de execução virtualmente síncronos (*virtually synchronous*), esse modelo cria a ilusão à aplicação de que ela está executando em um ambiente onde cada processo falho será detectado, e se algum processo for suspeito de falha, então certamente esse processo falhou. Seu funcionamento baseia-se em grupos de processos (*group membership*) com mecanismos de entrada en grupo e obtenção de estado, saída de grupo (um processo falho é automaticamente retirado do grupo a que ele pertencia), e comunicação com

grupo usando multicast ordenado. Essas funções primitivas são usadas para dar suporte a ferramentas de tolerância a falhas, tais como execução de requisições de carga balanceada, computação tolerante a falhas, dados replicados coerentes e segurança. Propriedades como sincronização virtual podem, em certos momentos, ser indesejadas, por introduzir *overheads* desnecessários ou conflitos com outros objetos, como para garantia de tempo-real (*real-time*). Além disso a implementação ideal em um ambiente inseguro pode aceitar *overhead* de dados criptografados, mas irá evitar esse custo quando executado dentro de um *firewall*.
Na arquitetura do Horus, protocolos de suporte de grupos podem variar, durante a execução, para corresponder com a especificação requisitada pela aplicação ou ambiente. Mas substituições em comunicação ponto-a-ponto com grupos de comunicação são uma abstração essencial.

3.2.1 Arquitetura Horus

Arquitetura Horus é baseada em camadas de grupos de processos similarmente a uma caixa de blocos Lego. Horus possui uma arquitetura de comunicação que trata protocolo como instancias de um tipo de dado abstrato fazendo com que os desenvolvedores particionem protocolos complexos em microprotocolos simples. Cada bloco implementa um microprotocolo (Figura 8) que fornece características diferentes de comunicação. Para fornecer a combinação desses blocos dentro de macroprotocolos com propriedades desejáveis, os blocos foram padronizados em interfaces *top* e *bottom* que os permite ser empilhados uns sobre os outros para serem executados de modos variados. Na prática, algumas combinações de blocos não fazem sentido, por exemplo, a camada de comunicação com protocolos de transporte receber chamadas da camada de entrega de mensagens totalmente ordenada. Apesar disso, valor conceitual da arquitetura é que, se permitido, seja simples criar novos protocolos empilhamento com o arranjo dos blocos existentes. Cada bloco do protocolo é um módulo de software com um conjunto de pontos de entrada para chamadas de procedimentos (camada acima ou abaixo). Por exemplo, existem chamadas para camadas mais abaixo para envio de mensagens e para camadas mais acima para receber mensagens. Cada camada é identificada por um nome ASCII e registra todas as chamadas ocorridas entre as camadas no momento da inicialização[13].

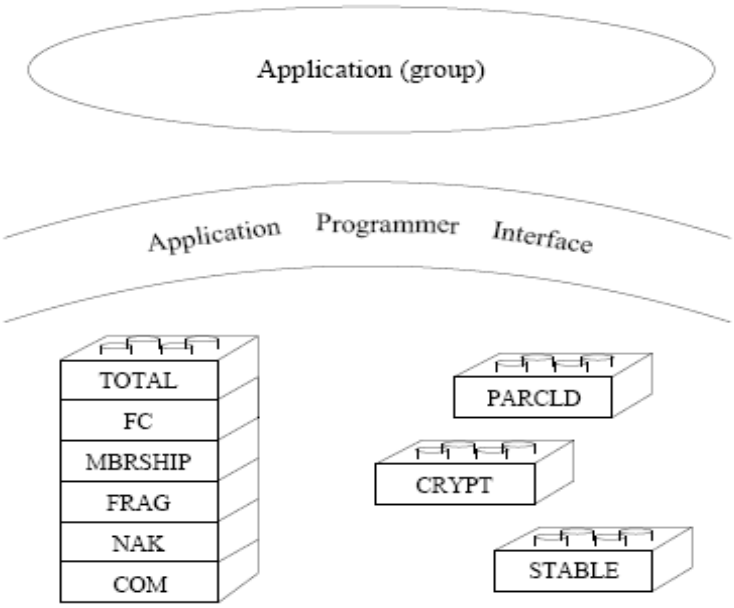

Figura 8. Arquitetura Horus em camadas de grupos de protocolo[13]

A camada mais acima é a única que se desvia da interface padrão do Horus: ela converte a abstração do protocolo do Horus em uma que combina com as necessidades e expectativas do usuário. Assim quando Horus é utilizada em uma interface *socket*, a camada mais acima converte as operações *sendto* e *recvfrom* do *socket* para o paradigma Horus.
As camadas especifícas atualmente suportadas pelo Horus resolve problemas como a interface do sistema para vários mecanismos comunicações da camada de transporte, superação de pacotes perdidos, criptografia e descriptografia, fluxo de controle, etc. Algumas das mais importantes camadas são descritas a seguir.
A Arquitetura Horus implementa os seguintes microprotocolos:

- COM: fornece aos grupos uma interface para protocolos de baixo nível, como IP, UDP e algumas interfaces ATM.
- NAK: implementa protocolos de reconhecimento negativo baseado em mensagens de retransmissão.
- CYCLE: disseminação de mensagens multimídia
- PARCLD: disseminação de mensagens hierárquicas.
- FRAG: fragmentação/remontagem
- MBRSHIP: cada membro com uma lista de pontos finais(*endpoints*) que pode ser acessado. Então ele executa um protocolo de consenso para fornecer aos usuários um modelo de sincronização virtual.
- FC: controle de fluxo
- TOTAL: entrega de mensagens totalmente ordenadas
- STABLE: detecta quando a mensagem foi entregue para todos os destinatários finais e descartadas(*garbage colleccted*).
- CRYPT: criptografia/descriptografia
- MERGE: localização e agrupamento de instâncias de grupos múltiplos

3.2.2 Tolerância a Falhas

Cada pilha lida com outros 3 tipos de objetos: ponto finais(endpoints), grupos(*groups*) e mensagens(*message*).
O objeto *endpoint* modela a entidade de comunicação. Possuem um endereço e enviam e recebem mensagens. Mensagens não possuem endereços para *endpoints*, mas para grupos. O endereço do *endpoint* é usado na associação. O objeto grupo é usado para manter o estado do protocolo local no *endpoint*. Um processo pode ter múltiplos *endpoints*, cada um com sua pilha de protocolo. Um objeto grupo possui ainda um endereço de grupo(*group adress*), onde cada mensagem é enviada e uma visão(*view*), que é uma lista de endereços dos destinatários finais que possam ser membros acessíveis do grupo. O objeto mensagem é uma estrutura local armazenada que possui operações de inclusão(*push*) e retirada(*pop*) de cabeçalhos de protocolos[13].
Horus permite que diferentes *endpoints* tenham diferentes visões de um mesmo grupo. Um *endpoint* pode possuir múltiplos grupos de objeto, permitindo que ele se comunique com diferentes grupos e visões. O usuário pode instalar diferentes novas visões quando os processos caem ou recupera-se e usa um dos vários protocolos de associação para atingir alguma forma de acordo de visões entre múltiplos objetos grupo de um mesmo grupos[13].
As mensagens que chegam na pilha de protocolos entram pela camada mais alta, que invoca a função da camada mais abaixo e pode adicionar um cabeçalho. Isto acontecem em todas as camadas até que se chegue à camada mais baixa da pilha, que então chama o dispositivo específico para realmente mandar executar a ação.

- **Detecção de Defeitos**

Quando se cria um *endpoint*, o processo descreve quais protocolos de pilha são necessários, e a base de *endpoint* para que ela seja construída. É permitido a um processo colocar múltiplos *endpoints* em uma única base de *endpoint*. Dado um *endpoint* e um endereço de grupo, um processo pode entrar em um grupo de *endpoints*. Isso resulta em uma chamada de uma visão que descreva o conjunto de *endpoints* com que os processos podem se comunicar. No caso da camada MBRSHIP pertencer a pilha, cada *endpoint* na visão estará garantido de ter sido enviado a mesma visão.

Tais camadas especificadas resolvem diversos problemas como produzir uma interface para um sistema com vários protocolos de comunicação de transporte, tolerar pacotes perdidos, criptografar e descriptografar, gerenciar grupos de processos, ajudar processos que entraram no grupo a obter o seu estado atual, agrupar grupos que foram particionados, controlar fluxo de dados, etc. Horus também adiciona ferramentas para ajudar no desenvolvimento e depuração de novas camadas[13].

O protocolo de associação, MBRSHIP, simula um ambiente para os membros de um grupo de comunicação onde um membro pode apenas falhar(não pode estar lento ou desconectado) e as mensagens não podem ser perdidas. Cada membro tem noção da visão(*view*) atual, a qual possui uma lista ordenada dos membros. Cada membro na visão atual é garantido pela aceitação da mesma visão ou será removido da visão atual[15]. Mensagens enviadas na visão atual são entregues aos membros sobreviventes da visão atual, e as mensagens recebidas na visão atual são recebidas por todos os membros sobreviventes da visão. A camada TOTAL fornece apenas um tempo de entrega para os membros sobreviventes da visão, e a camada fornece informações de defeitos a partir das atualizações das visões. Isso é chamado sincronização virtual(*virtual synchrony*) porque todos os membros que aparecem na comunicação enxergam defeitos no mesmo instante lógico, reduzindo significantemente o número de cenários de defeitos[15]. Sincronização virtual é melhor entendido como uma simulação do comportamento *fail-stop* (membros excluídos de uma visão ainda podem estar vivos). Quando a comunicação é restabelecida, visões podem se juntar à comunicação chamando o protocolo MERGE. Apenas se MBRSHIP foi usado como um detector de defeitos perfeito essa simulação pode ser “exata”[15].

A camada NAK coloca uma sequencia de numeros em cada mensagem enviada que é analisada pelo receptor, se o receptor detectar mensagens perdidas é enviado uma mensagem de reconhecimento negativo (*nak*), para que a camada NAK retransmita a mensagem se ela ainda estiver armazenada. Se não estiver armazenada, cada *endpoint* enviará via *multicast* seu estado para que mensagens armazenadas possam ser descartadas(*flush*).

- **Recuperação de Defeitos**

Na camada MBRSHIP, o protocolo principal é o protocolo *flush* que é executado quando é detectado a queda de um dos membros ou uma visão se junta a comunicação[15].

Um dos membros é denominado de coordenador do *flush*. O coordenador transmiti mensagens *flush* aos membros sobreviventes da visão. Primeiro todos os membros retornam as mensagens dos membros falhos que não se sabia que foram entregues. Essas chamadas são chamadas instáveis(*unstable*). Por fim, cada membro envia uma mensagem de resposta *flush_ok*. Então, os membros irão ignorar as mensagens enviadas pelos supostos membros defeituosos, e esperam por uma nova instalação da visão[15].

Assim que recebido todas as mensagens *flush_ok*, o coordenador transmite as mensagens do membro defeituoso que ainda estão instáveis. Neste ponto uma nova visão será instalada. Quando

todas as mensagens estabilizarem, *flush* estará completada. Se algum processo falhar durante este processo,uma nova rodada do protocolo *flush* poderá ser iniciada imediatamente[15].
Cada pilha de blocos são cuidadosamente protegido por outras pilhas. Elas possuem suas próprias *threads* priorizadas e controlam o acesso de memória disponível com um mecanismo chamado canal de memória(*memory channel*). Horus possui um escalonador de memória que especifica dinamicamente a taxa que cada pilha pode acessar a memória, dependendo da disponibilidade e prioridade, então não há possibilidade de uma pilha monopolizar a disponibilidade da memória.
Quando múltiplas mensagens chegam simultaneamente, é importante impor um ordenamento na entrega das mensagens. Horus numera as mensagens e usa variáveis de sincronização *event count* para reconstruir a ordem quando necessário.

3.2.3 Protocolos de pilhas

A arquitetura de microprotocolos do Horus não seria de grande valor se não fosse pela variedade de classes de protocolos de grupos de processos que se pode suportar como, bom desempenho, compartilhamento significativo de funcionalidades e simplificação pela implementação de pilhas em camadas. Na Figura 9 mostramos todas as ferramentas de grupos de processos virtualmente síncronos.
A pilha 1 fornece totalmente ordenado, comunicação de controle de fluxo sobre a abstração dos grupos de associação. As camadas FRAG, NAK e COM fornecem respectivamente quebra de mensagens muito extensas em mensagens menores, suporte a perda de pacotes usando reconhecimento negativo, e interface Horus para protocolos básicos de transporte. A pilha 2 é bem parecida, mas fornece fraca ordenação e inclui uma camada para suportar estado de conexão(*state transfer*) para processos que entram em um grupo ou grupo que se juntam após uma partição da rede. A pilha 3 suporta subida por toda a estrutura hierárquica onde cada processo "pai" é responsável por um conjunto de processos "filho". A pilha dividida mostrado neste caso representa uma característica na qual a mensagem pode ser roteada para diferentes pilhas, dependendo do tipo de processamento exigido.

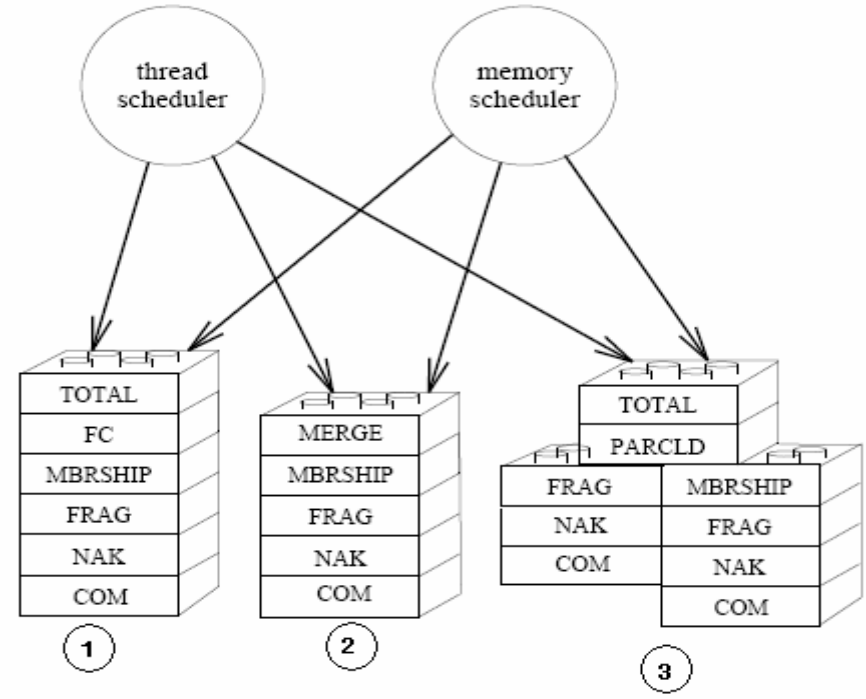

Figura 9. Protocolos de pilhas[13]

HCPI(*Horus Common Interface Protocol*) fornece uma extensa interface que suporta todas as operações comuns em um sistema de comunicação de grupo, indo além das funcionalidades do sistemas em camadas. HCPI é projetado para multiprocessamento e é completamente assíncrono e reincidente.
Geralmente, interfaces HCPI recaem em 2 categorias: as do primeiro grupo que se preocupam com o envio e o recebimento de mensagens e com a estabilidade das mensagens, quando o processamento foi completo e as informações associadas podem ser descartadas(*garbage collected*). E as do segundo grupo que se preocupam com a associação dos membros. De cima para baixo, que permite as aplicações ou a camada controlar o grupo de associação usada pela camada abaixo, de baixo para cima, relatam as mudanças de associação, problemas de comunicação e outros eventos relacionados a aplicações.
Com o suporte de um mesmo HCPI, cada camada Horus executa um protocolo diferente. Embora seja possível que as camadas sejam empilhadas em qualquer ordem, muitas das camadas impõem certas semânticas para as camadas abaixo dela, impondo uma ordem parcial de empilhamento. Essas restrições podem ser tabeladas, fornecendo informações sobre as propriedades fornecida pela aplicação, às vezes é possível gerar automaticamente um protocolo de pilha mínimo que consiga as propriedades desejáveis.

3.3 TAO(The ACE ORB)

ADAPTIVE Communication Environment(ACE) é um kit de ferramentas orientado a objetos usado por desenvolvedores em serviços de comunicação de alto desempenho. ACE automatiza configurações e reconfigurações por serviços dinamicamente conectados em aplicações em tempo de execução e executa esses serviços em um ou mais processos ou *threads*[16].
TAO(The ACE ORB) foi desenvolvido com o padrão e os componentes do *framework* ACE. TAO é um ORB(*Object Request Brokers*) de alto desempenho para sistemas de tempo real baseado no padrão CORBA(*Common Object Request Broker Architecture*) que permite a clientes invocar operações de objetos distribuídos sem se preocupar com a localização do objeto, linguagem de programação, sistemas operacionais, protocolos de comunicação e interconexões e hardware.
Vários domínios de aplicação, como comando e controle de sistemas, telecomunicações, serviços financeiros e simulações interativas distribuídas exigem garantias de tempo real para redes básicas, sistemas operacionais e componentes de *middleware* para satisfazer seus requisitos de Qualidade de Serviço(QoS)[16].
As garantias de QoS às quais TAO da suporte podem ser divididas em determinísticas, como aplicações em tempo real de controle de avião, onde decisões criticas são cruciais e estatísticas, como aplicações de teleconferência, onde pequenas flutuações de entrega e garantia de confiabilidade são toleráveis.

3.3.1 Arquitetura TAO

Como TAO é uma implementação de tempo real de CORBA, sua arquitetura é baseada na arquitetura CORBA que será apresentada a seguir.

- **Componentes do Modelo CORBA**

Um modelo de objetos é um conjunto de definições sobre propriedades das entidades computacionais, tal qual os tipos disponíveis e suas semânticas, regras para compatibilidade de tipos, comportamento em caso de erros, etc. Os modelos de plataformas de *middleware* são muito similares, porém os detalhes em suas diferenças são fatores que tem maior impacto no projeto e desempenho do sistema.
Para um melhor entendimento dos modelo de ORB, apresentaremos, na Figura 10, os componentes do modelo CORBA, que dá suporte a vários níveis de transparência e permite aos clientes invocar operações nos objetos-alvo sem se preocupar onde estão os objetos, em que linguagens de programação estão escritos, em que plataformas de *hardware*/sistema operacional eles rodam ou que protocolos de comunicação e de rede são usados para interconectar os objetos.

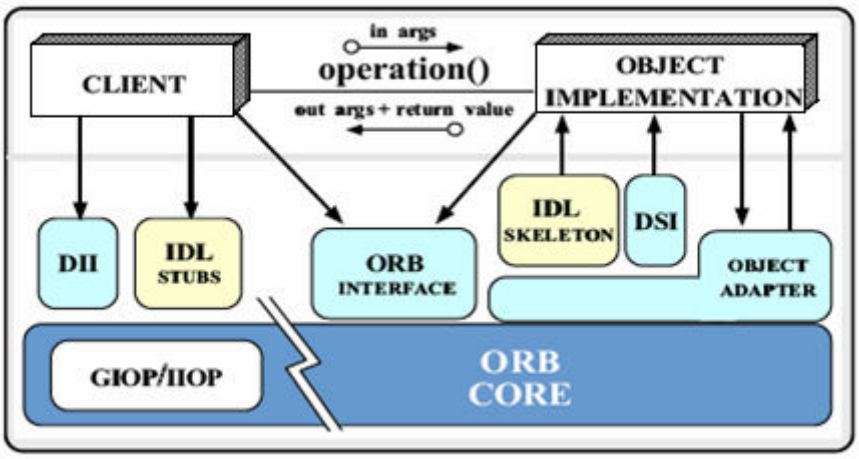

Figura 10. Componentes no modelo de referência CORBA[16].

Os componentes do modelo de referência CORBA são os seguintes:

- **Implementação dos Objetos (*Object Implementation*)**: definem operações que implementam uma interface especificada usando CORBA *Interface Definition Language*(IDL).
- **Cliente (*Client*)**: entidade que invoca uma operação em uma implementação de objeto. O acesso aos serviços do objeto remoto podem ser transparentes, pois o cliente não conhece os detalhes da implementação dos objetos, somente a interface IDL.
- **Núcleo do ORB (ORB *Core*)**: responsável por achar o objeto de implementação, entregando a requisição ao objeto e retornando ao cliente uma resposta, se existir. O Protocolo Geral Inter-ORB(GIOP) define a seqüência de mensagens do protocolo que implementa comunicação inter-ORB, isto é, ele é utilizado para transportar os dados entre ORBs distintos. Protocolo de Internet Inter-ORB(IIOP) é um mapeamento de GIOP para os protocolos TCP/IP. IIOP nasceu do protocolo de comunicação básica (cliente/servidor) para computação de objetos distribuídos sobre a Internet.
- **Interface do ORB (*ORB Interface*)**: interface abstrata para ocultar das aplicações detalhes de implementação. É a única interface que possui uma interação direta com ORB e por isso, é compartilhada pelo lado cliente e pelo lado servidor.
- **IDL *Stubs* e *Skeletons***: servem como intermediários entre aplicações cliente e servidor, respectivamente, e ORB. São interfaces estáticas geradas a partir da compilação da interface IDL. A interface IDL contém as interfaces dos objetos e dos métodos implementados no cliente e no servidor, *stub* e *skeleton* respectivamente. A transformação entre a definição IDL e a linguagem de programação alvo é automática devido oa compilador IDL, o que reduz o

potencial de inconsistências entre clientes *stub* e servidores *skeleton* e aumenta a otimização das implementações *stub* e do *skeleton*.

- **Interface de invocação dinâmica (DII)**: permite aos clientes acesso direto aos mecanismos básicos de requisição fornecidas pela ORB. Invocação dinâmica só ocorre quando em tempo de execução um cliente chama um método de um objeto que não possui uma interface conhecida.
- **Interface *skeleton* dinâmica (DSI)**: é a parte do servidor análoga ao DII no cliente. Ela permite a ORB entregar requisições a implementações de objetos cujo o tipo a ORB não conhece em tempo de execução.
- **Adaptadores de objetos (*Object Adapters*)**: possibilitam que a implementação dos objetos acesse o máximo possível de funções do ORB. Ajuda ORB com as requisições de demultiplexação para os objetos-alvo e enviando operações de chamadas do objeto. Esses objetos associam as implementação de objetos ao ORB. *Basic Object Adapters*(BOA) possuem acesso a diversas funções, como invocação, ativação e desativação de métodos, geração e interpretação de referências para objetos. Alem disso, o modelo CORBA define um adaptador de objetos portátil(POA – *Portable Object Adapter*), que permite maior flexibilidade na interação entre objetos do lado servidor e os ORBs de diversos desenvolvedores.

- **Componentes do modelo TAO**

TAO é um ORB de tempo-real e alto desempenho que tem por finalidade dar suporte à construção de aplicações com exigências de QoS determinísticas e estatísticas. O ORB TAO contém uma interface de rede, Sistema Operacional, protocolo de comunicação e componentes e serviços CORBA concordantes[17]. Na Figura 11 apresentamos os componentes no modelo TAO.

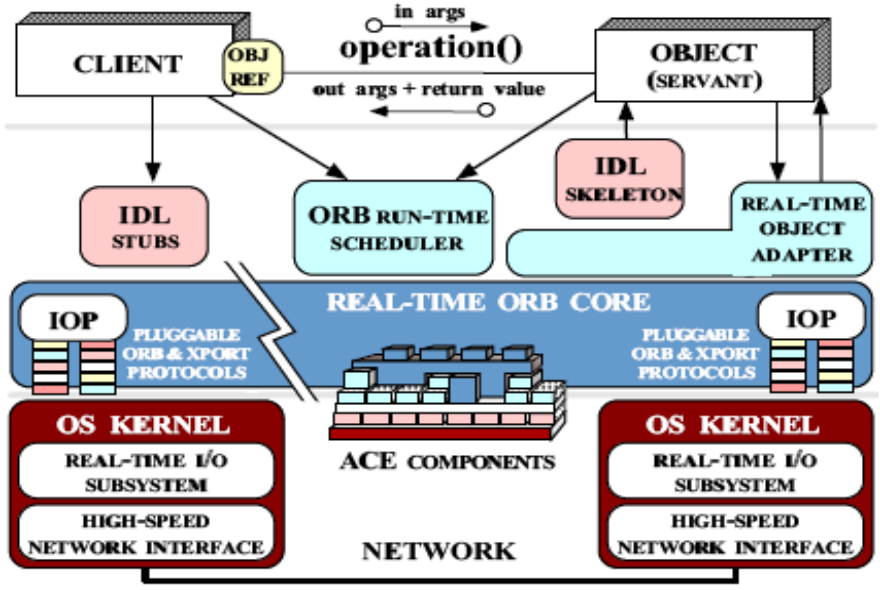

Figura 11. Componentes do modelo TAO[17]

- **IDL *stub* e *skeleton* otimizadas**: IDL *stub* e *skeleton* realizam *marshaling* e *demarshaling* nos parâmetros de aplicação da operação, respectivamente. O compilador IDL do TAO gera *stubs/skeletons* que podem seletivamente usar compilações altamente otimizadas e/ou interpretativo *(de)marslhing*. Essa flexibilidade permite aos desenvolvedores de aplicação seletivamente diminuir tempo e espaço, o que é crucial para alto desempenho de sistemas em tempo-real.

- **Adaptadores de objeto de tempo real**: no TAO, esses objetos utilizam perfeitamente *hashing* e otimização de demultiplexação ativa para envio de operações criadas em tempo constante, independente do número de conexões ativas criadas e operações definidas na interface IDL.
- **Escalonador de tempo real**: mapeia requisições QoS de aplicações, tais como latência fim-a-fim, reunindo periodicamente *deadlines* de escalonamento, para recursos de um sistema final/rede ORB, como CPU, memória, conexões de rede e dispositivos de armazenamento. TAO dá suporte a escalonadores de tempo real estáticos e dinâmicos.
- **Núcleo do ORB de tempo real**: entrega a requisição do cliente ao objeto adaptador e retorna uma resposta ao cliente, se houver alguma. O núcleo do ORB do TAO permite que protocolos customizados possam ser ligados ao ORB sem afetar o modelo de programação padrão das aplicações CORBA.
- **Interface de rede de alta velocidade**: no núcleo o subsistema de I/O do TAO, há uma interface de rede "*daisy-chained*" que consiste em um ou mais chips APIC(ATM *Port Interconnect Controller*). APIC foi desenvolvido para suportar uma taxa bidirecional agregada de 2.4 Gbps.
- **Componente interno**: TAO é desenvolvido usando um *middleware* de baixo-nível denominado ACE, que implementa núcleos concorrentes e padrões distribuídos para comunicações de software. ACE fornece componentes que dão suporte a requisições de QoS para alto desempenho, aplicações em tempo real e *middlewares* de alto nível, como TAO.

3.3.2 Tolerância a Falhas

TAO fornece tolerância a falhas a objetos CORBA utilizando três componentes da especificação FTCORBA, Detector de falhas(monitora processos e hospedeiros), Notificador de falhas(recebe relatórios do detector de falhas) e Gerenciador de Réplicas(gerencia grupos de objeto), junto com o DOORS, serviço de tolerância a falhas para entrega de aplicações, e a adição de replicações semi-ativas.

- **Detecção de Defeitos**

  - **FTCORBA**

FT CORBA é uma especificação que define serviços e estratégias para aumentar a confiabilidade das aplicações CORBA. Os mecanismos de tolerância a falhas usados pelo FT CORBA para detecção e recuperação de defeitos são baseados na redundância de entidades(*entity redundancy*)[18]. Para introduzir tolerância a falhas nos padrões CORBA foram definidas algumas especificações com um conjunto de serviços essenciais para o desenvolvimento de aplicações confiáveis.

- **Gerenciador de Réplicas**: serviços de tolerância a falhas interagem com o gerenciador de réplicas para criar, gerenciar propriedades e controlar associação em grupos de objetos. O gerenciador de réplicas também é responsável pela criação e manutenção de um referência de interoperabilidade entre grupos de objeto(IOGR– *Interoperable Object Group Reference*). As operações do gerenciador de réplicas são divididas em três interfaces separadas:
  - **Gerenciador de propriedades**: define operações para as configurações de propriedades.
  - **Gerenciador de grupos de objeto**: define operações de entrada e saída de membros de grupos de objeto, especifica ou descobre a localização dos membros do grupo de objetos e descobre o valor atual da referência do grupo de objeto e do identificador do grupo de objeto.

- **Fábrica genérica**: define operações para criar ou remover objetos. O objeto fábrica genérica negocia com os objetos Fábrica Local para criar ou remover réplicas de grupos de objetos. No processo de criação de réplicas são utilizados serviços de *logging* e *checkpoint*.

- **Gerenciador de Falhas**: responsável pela detecção, notificação, análise e diagnóstico de falhas em objetos. O detector de falhas(*fault detector*) periodicamente emite requisições CORBA para objetos monitorados e relata falhas sobre objetos que falharam em resposta. Detectores de falha são criados por fábricas de detectores de falhas e são criados nos mesmos processos que as fábricas de detectores de falhas. Notificador de falhas(*fault notifier*) reúne relatórios de falhas dos detectores de falha, aplicações ou plataformas especificas de detecção de falhas.
- **Domínios de tolerância a falhas**: geralmente possuem vários *hosts* e grupos de objeto, e um único *host* pode suportar vários domínios de tolerância a falhas. A existência de políticas de segurança e mecanismos pode ser mantida pela certeza de que um domínio de tolerância a falhas está totalmente contido em um único domínio de segurança. Todos os grupos de objetos de um domínio de tolerância a falhas são criados e gerenciados por um único gerenciador de réplicas, mas eles podem invocar e ser invocados por objetos de outros domínios de tolerância a falhas.
- **Gerenciamento de recuperação e *logging***: responsável pela consistência de estados das réplicas. O mecanismo de *logging* é responsável pela registro do estado atual do objeto primário nos objetos secundários em cada intervalo de *checkpoint* definido no gerenciador de propriedades. O mecanismo de recuperação é responsável pela recuperação de réplicas, do objeto ou do grupo após a ocorrência de falhas.

Na Figura 12, mostramos uma representação da arquitetura do FTCORBA, de como acontece um gerenciamento de tolerância a falhas em um único domínio de tolerância a falhas. Na figura temos um *host*, um cliente, o ORB cliente, dois *host* cada um com uma réplica do objeto servidor, uma fábrica, um detector de falhas, um ORB servidor e, em cada ORB, um mecanismo *logging* e um mecanismo de recuperação.

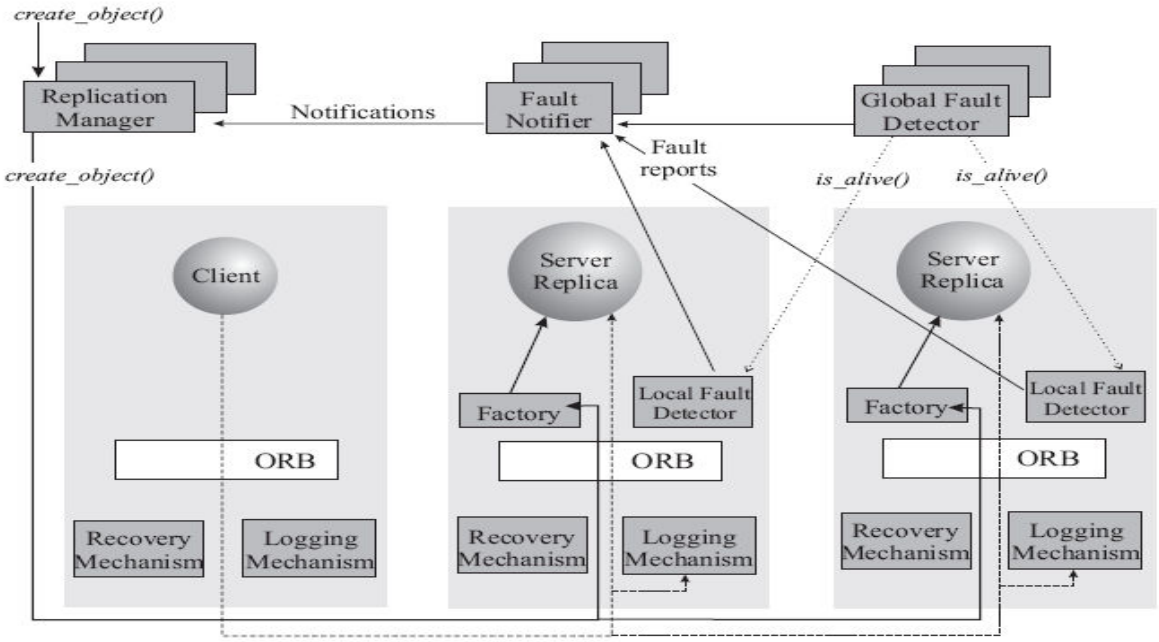

Figura 12. Arquitetura FTCORBA

Uma falha é detectada por objetos do Detector de Falhas, que está localizado em cada servidor *host* e pode ser supervisionado por um objeto adicional do Detector de Falhas. Quando a falha for detectada, ela será transmitida pelo Notificador de Falhas aos consumidores da falha, por exemplo, o gerenciador de replicas. Cada servidor possui o seu próprio mecanismo de recuperação e *logging* para se recuperar de falhas.
FTCORBA utiliza todas as modalidades disponíveis de replicação, no caso *stateless*, *cold passive*, *warm passive*, *active* e *active with voting*.

- **DOORS**

DOORS(*Distributed Object-Oriented Reliable System*) é um serviço de estratégia do CORBA para tolerar falhas que fornece serviços de políticas e mecanismos tolerantes a falhas para entregas de aplicações e foi incorporado ao padrão FT-CORBA em Janeiro de 2000.
A Figura 13 mostra a interação de protocolos entre os componentes do *framework* DOORS quando as aplicações usam o esquema de réplicação *warm passive*.

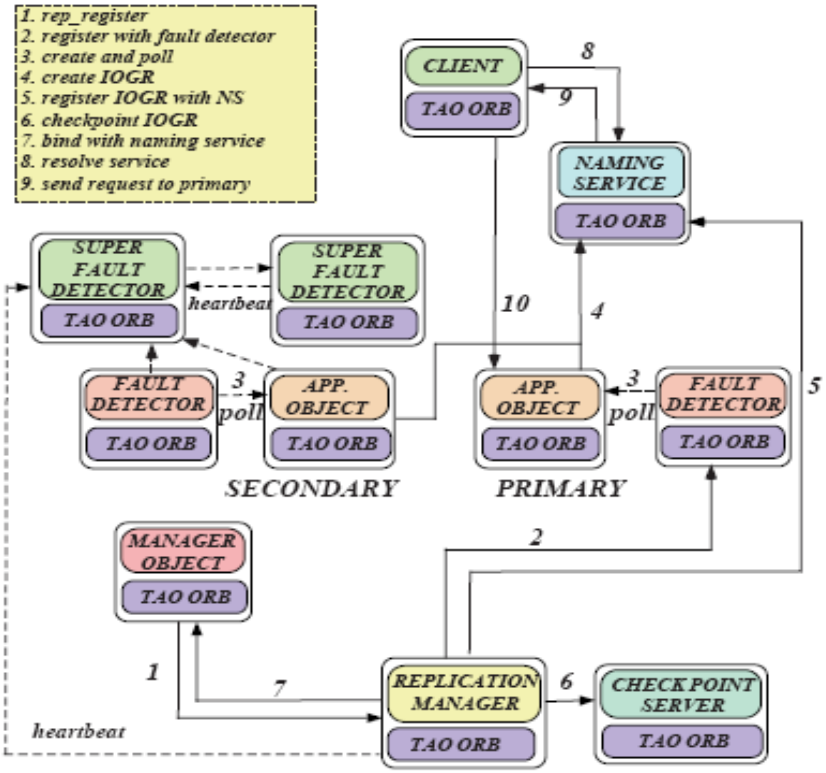

Figura 13. Interação de componentes FTCORBA[18]

1. O gerenciador de aplicações requisita ao gerenciador de réplicas que crie um grupo de réplicas usando a operação *()create_object* da interface fábrica genérica do FTCORBA e passa um conjunto de propriedades de tolerância a falhas para o grupo de réplicas.
2. O gerenciador de réplicas permite ao *task* a criação de réplicas individuais para a fábrica de objetos local baseado na propriedade de localização de objeto. As fábricas locais retornam referências individuais de objetos criados pelo gerenciador de réplicas.
3. Neste ponto, o gerenciador de réplicas informa ao detector de falhas para iniciar o monitoramento das réplicas.
4. O gerenciador de réplicas coleta todos os IORs das réplicas individuais, cria um IOGR para o grupo, e estabelece uma das réplicas como primária. No esquema de réplicação ativa, todas as réplicas são primárias.

5. Então o gerenciador de réplicas registra o IOGR com o serviço de nomes, o qual publica o registro em outras aplicações e serviços CORBA.
6. O gerenciador de réplicas recupera o ponto dos IOGR e outros estados.
7. O cliente interessado no serviço contata o serviço de nomeação(serviço que fornece um mapeamento de nomes, organizados hierarquicamente, para o IOR).
8. O serviço de nomeação responde com o IOGR.
9. Finalmente, o cliente faz a requisição e o cliente ORB garante que a requisição será entregue a réplica primaria.
Dependendo do estilo de monitoração escolhido(*heartbeat*, *polling*), o detector de falhas continuará monitorando as réplicas nos períodos de intervalo depois que o grupo de réplicas está estabelecido. O detector de falhas e o gerenciador de réplicas enviam mensagens de *heartbeat* ao "super" detector de falhas primário nos períodos de intervalo. Como o FTCORBA não permite ponto único de falhas, o "super" detector de falhas também é réplicado e as réplicas enviam umas às outras mensagens de *heartbeat* no período de intervalos. Uma das réplicas é denominada de primária e as outras servem como *backup*. Se a primária falha, as *backups* elegem uma nova réplica primária.

TAO dá suporte a especificação FTCORBA. Porém, um grupo de desenvolvedores do Instituto para Sistemas Intensivo de Software(ISIS), projetou e implementou melhorias no núcleo ORB do TAO para dar melhor suporte à introdução de tolerância a falhas em aplicações, incluindo a implementação de características a nível do componente núcleo do ORB definidas na especificação FTCORBA. Algumas dessas melhorias foram:
- adição de replicação semi_ativa(*semi_active*).
- separando interfaces e tipos de definição comuns as especificações múltiplas em módulo de grupo portátil(*Portable Group*).
- adição de registração de fábricas e interfaces de fábricas de detecctor de falhas.

- **Replicação semi-ativa**

A abordagem de replicação semi-ativa foi projetada com a evolução da réplicação ativa e da réplicação passiva para dar suporte à execução previsível de programas sem acrescentar *overhead*, imprevisibilidade e não-determinismo a estratégia padrão do FTCORBA. A replicação semi-ativa tenta combinar a velocidade das propriedades de recuperação de defeitos da replicação ativa com a habilidade de replicar aplicações com comportamento de não-determinismo da replicação passiva. Estende a noção de líder e seguidor, enquanto o processamento da requisição está sendo realizada em todas as replicas, é responsabilidade do líder realizar as partes não-determinisicas do procesamento e informar a resposta aos seguidores. Em caso de defeito no líder, uma eleição deve ser realizada para a escolha de um novo líder. A Figura 16 ilustra como as réplicas estão arrumadas para tolerar falhas na replicação semi-ativa. É criada uma lista com todas as réplicas que, exceto o objeto primário, são interligadas por conexões em nível de transporte orientado a conexão a cabeça da fila e adicionadas a uma fila onde a réplica que estiver na cabeça da fila será nomeada como objeto primário na ocorrência de falhas. Em caso de falha no objeto primário, o defeito é detectado pela próxima réplica da lista quando a conexão em nível de transporte for fechada. O termino da conexão é usado para detectar defeitos, isso é feito fazendo com que os objetos secundários esperem a abertura de conexões usando mecanismos de demultiplexação.
Réplicas consistentes podem ser mantidas por mensagens de sincronização de estados em multicast confiável para todas as réplicas. O protocolo *multicast* utilizado para invocação de requisições ou transferência de estados precisa forçar um ordenamento das mensagens. Para isso,

são passadas ao cliente uma lista ordenada de referências o qual garante a ordem da lista. Não são usados *threads* adicionais no envio de mensagens de *heartbeat* ou monitoramento dos objetos servidores.

Replicação semi-ativa soluciona diversas questões existentes nas demais replicações(ativa e passiva), como: uma eficiente e previsível detecção de defeitos que garante um tempo de recuperação mais rápido e previsível comparado as replicações passiva fria e passiva quente. Reduziu mensagens de *heartbeat* e de votação(*polling*) na rede desde que não são mais usadas na detecção de defeitos. Utilizam mais facilmente mecanismo RTCORBA (R*eal-Time CORBA*) e de policiamento. A replicação semi-ativa distribui o *overhead* de detecção de defeitos e de recuperação de erros entre as réplicas.

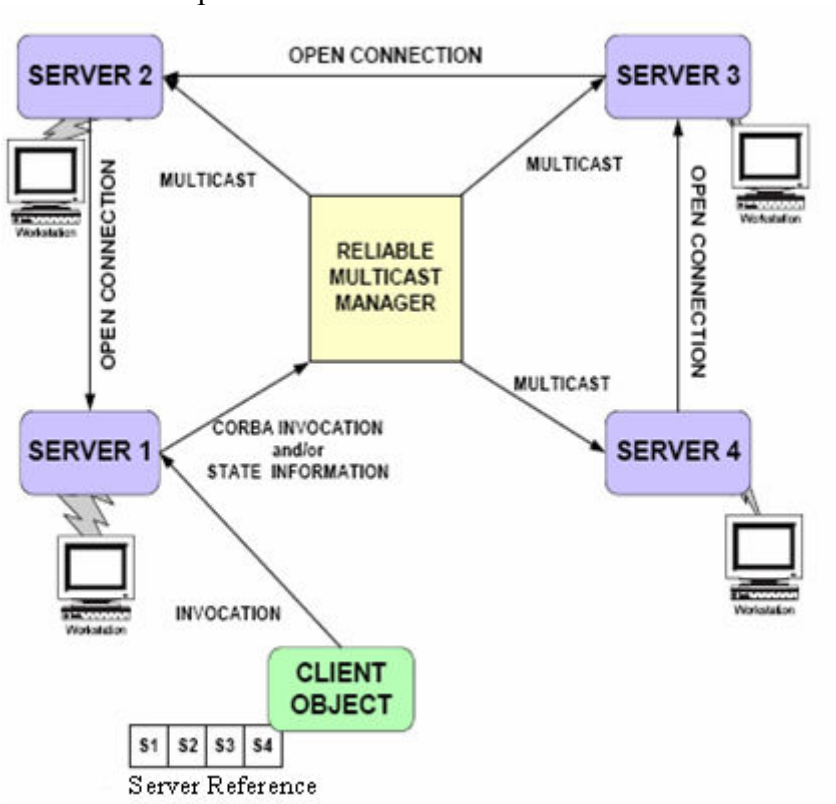

Figura 14. Arquitetura da replicação semi-ativa

- **Recuperação de Defeitos**
  - **Gerenciamento de recuperação e *logging***:
    responsável pela consistência de estados das réplicas. O mecanismo de *logging* é responsável pela registro do estado atual do objeto primário nos objetos secundários em cada intervalo de *checkpoint* definido no gerenciador de propriedades. O mecanismo de recuperação é responsável pela recuperação de réplicas, do objeto ou do grupo após a ocorrência de falhas.

3.4 Internet Communications Engine(ICE)

Internet Communications Engine(ICE) é uma plataforma de middleware orientada a objetos. Isso significa que ela fornece ferrementas, API e bibliotecas para a construção de aplicações cliente/servidor orientadas a objetos. Suas aplicações são apropriadas para o uso em ambientes

heterogêneos[19]. ICE representa uma nova abordagem de *middleware* que desenvolve os pontos fortes do CORBA, sem cometer seus erros. Clientes e servidores podem ser escritos em diferentes linguagens de programação e rodar em diferentes sistemas operacionais e mesmo assim podem se comunicar utilizando várias tecnologias de rede[20].
ICE fornece um modelo de objetos que é ao mesmo tempo simples e poderoso. Os principais objetivos de seu desenvolvimento foram[19]: (*i*) fornecer um middleware orientado a objetos apropriado para o uso em ambientes heterogêneos, um conjunto completo de características que dêem suporte ao desenvolvimento de aplicações distribuídas realistas; (*ii*) evitar complexidades desnecessárias criando uma plataforma de fácil aprendizado, fornecendo uma implementação eficiente em largura de banda da rede, uso de memória e *overhead* de CPU; e (*iii*) fornecer uma implementação de desenvolvimento seguro.
ICE atualmente é compatível com diversas linguagens de programação que podem ser empregadas na construção de aplicações como, Java, C++, Visual Basic, Ruby, Python, C# e PHP[19].

3.4.1 Arquitetura ICE

A arquitetura baseada em modelos de objetos do ICE melhora o modelo de objetos de CORBA, por isso os componentes da arquitetura ICE são compatíveis com os componentes do modelo CORBA.

- **Clientes e Servidores**

Clientes são entidades ativas. Eles emitem requisições para serviços dos servidores. Servidores são entidades reativas. Eles fornecem serviços solicitados pelos clientes. Geralmente os servidores não trabalham somente recebendo requisições e podem também trabalhar como clientes enviando requisições para outros servidores. Neste caso, porém, com a finalidade de responder a requisição do cliente inicial[19].

- **Objeto ICE**

Objetos ICE são entidades de espaço de endereço local ou remoto que respondem a requisições de clientes. Um Objeto ICE único pode ser instânciado em um único servidor, ou redundantemente, em servidores múltiplos. Objetos ICE possuem um ou mais interfaces(coleção de operações implementadas pelo objeto). Uma operação possui um ou mais parâmetros assim como valores de retorno. Parâmetros e valores de retorno possuem um tipo específico. Objetos ICE possuem uma interface distinta conhecida como interface principal(*main interface*), além de poder fornecer uma ou mais interfaces alternativas, conhecidas como *facets*. Cada objeto ICE possui uma única identidade de objeto(*object identity*, valor de identificação que distingue o objeto de todos os outros objetos). O modelo de objetos do ICE assume que a identidade do objeto é globalmente única, ou seja, dois objetos em um mesmo domínio de comunicação ICE não podem ter a mesma identidade[19].

- ***Proxies***

Proxies são tratados por objetos remotos, fornecendo conhecimento sobre a máquina e o número da porta onde o servidor está rodando e conhecimento sobre a identidade do objeto ao cliente[20]. Clientes utilizam *proxies* para conseguir se comunicar com os objetos ICE. O *proxy* atua como um representante local para o objeto ICE, quando os clientes invocam uma operação. Um *proxy* contém[19]: informações de endereçamento que permitem que o lado cliente contate o servidor correto; um objeto identidade que identifica no servidor qual é o objeto destino; e um identificador *facet* opcional que determina que interface *facet* particular de um objeto o *proxy* referencia. Aplicações podem comparar os *proxies* por igualdade e igualdade de *proxies* equivale a igualdade de objetos

ICE fornece não só uma herança de interfaces, como também uma agregação de interfaces. Clientes podem perguntar ao *proxy* por uma interface diferente da qual o objeto está representando. Embora um objeto forneça múltiplas interfaces, existe apenas um único objeto e conseqüentemente uma única identidade de objeto. Interfaces de agregação resolvem os problemas das versões, pois um único objeto pode ter múltiplas interfaces não relacionadas enquanto possui apenas uma identidade objeto. Desenvolvedores podem então adicionar novas interfaces a objetos pré-existentes sem violar o contrato cliente-servidor. Isso faz com que os desenvolvedores adicionem novas versões aos sistemas existentes sem impacto aos clientes implantados[20].

- ***Slice***

Tal como o IDL CORBA, ICE fornece uma linguagem de especificação. *Slice*(*Specification Language for ICE*) é um mecanismo de abstração fundamental para separar as interfaces dos objetos de suas implementações[19]. *Slice* estabelece contrato entre o cliente e o servidor e descreve os tipos e as interfaces de objeto utilizadas na aplicação, independente das linguagens de programação em que o cliente e o servidor foram implementados. *Slice* fornece um número mínimo de tipos primitivos internos(*short, int, long, float, double, byte, string, object e bool*). Além disso, dá suporte a vários tipos definidos pelos usuários, como constantes, listas, seqüências, estruturas e modelos, e ainda fornece novos construtores. A definição do *Slice* é focada nas interfaces de objetos, nas operações fornecidas por estas interfaces e pelas exceções que podem surgir por destas operações.

3.4.2 Réplicações

As replicações fazem com que os adaptadores de objeto(e seus objetos) fiquem disponíveis em múltiplos endereços. O objetivo da réplicação é de prover redundância pela execução de um mesmo servidor em computadores diferentes. Se ocorrer a falha em algum computador, o servidor continuará disponível em outros computadores. Um cliente pode acessar um objeto via um endereço e obter a mesma resposta como se fosse de qualquer outro endereço. Esses objetos não possuem estado, ou seja, sua implementação é projetada para sincronizar com o banco de dados ao invés de manter uma visão consistente de cada estado do objeto. Se compararmos essa estratégia com os que são implementados por FT-CORBA, pode-se dizer que o ICE usa apenas a primeira, replicação sem estado.

ICE fornece uma forma limitada de réplicação quando o *proxy* especifica endereços múltiplos para um objeto. A execução do ICE escolhe aleatoriamente um dos endereços para iniciar a tentativa de conexão e continua tentando em todos eles no caso de um defeito.

ICE fornece uma forma mais útil de réplicação conhecida como réplicação de grupos que exige o uso de um serviço de localização. Um grupo réplicado possui um identificador único e consiste em um valor qualquer de adaptadores de objeto[19]. O adaptador de objeto deve ser membro de no máximo um grupo réplicado. O grupo réplicado é tratado por um serviço de localização como um “adaptador de objetos virtual”. O comportamento de um serviço de localização quando resolvendo um proxy indireto contendo o identificador do grupo réplicado é apenas um detalhe de implementação. Por exemplo, o serviço de localização pode decidir retornar os endereços de todos os adaptadores de objeto do grupo, assim o cliente pode selecionar aleatoriamente um dos endereços e implementar novos serviços sobre mecanismo de replicação fornecido pelo ICE.

3.4.3 Transações

- **Modelos de Invocação**

ICE suporta, como o CORBA, invocações com limite de tempo, para configurações globais ou invocações individuais, onde as operações que não terminaram a execução no tempo especificado retornam com uma exceção de limite de tempo. ICE fornece invocações síncronas e assíncronas, além de datagramas e capacidade de executar requisições em lote(*batching capability*). CORBA também oferece *time outs* por *thread*, mas ICE não fornece devido ao alto custo ao acesso específico de *threads* armazenados. Então, o ICE fornece apenas um esquema simples de *time out* como forma de detecção de defeitos.
Requisições ICE possuem semântica *at-most-once*: a infra-estrutura de tempo de execução do ICE faz o melhor para entregar a requisição ao destino correto e, dependendo das exatas circunstancias, pode reenviar a requisições falhas. ICE garante que a requisição será entregue. Se não for possível entregar a requisição, informa ao cliente com um a exceção especifica. Requisições têm semântica *at-most-once*, o que garante que operações que não são *idempotent* sejam usadas com segurança.
Sem semântica *at-most-once*, podem ser construídos sistemas distribuídos que são mais robustos na presença de defeitos na rede. Sistemas realístas exigem operações não *idempotentes*, o que exige a semântica *at-most-once*, tornando o sistema menos robusto na presença de defeitos na rede. ICE permite o desenvolvedor marcar operações como *idempotent*. Para essas operações a execução ICE utiliza mecanismos mais rigorosos de recuperação de erros do que as operações não *idempotent*[19].

- **Invocações Sincronas**

Para invocações síncronas ICE fornece a semântica *at-most-once*, que exige um tratamento de erros conservador. Caso ocorra uma falha de invocação enquanto a resposta para a requisição está pendente, o lado cliente não tem outra alternativa a não ser propagar o defeito para a aplicação. A semântica *at-most-once* do ICE é complementada pelos modificadores de operações *nonmutating* e *idempotent*. O primeiro indica que uma operação não modifica estado e o segundo fixa o estado em um valor definido independente do estado anterior para onde a execução possa reenviar com segurança mensagens na presença de um defeito na rede. Isso torna a execução de ICE transparentemente recuperável com relação a falhas da rede[20].

- **Invocações Dinâmicas**

Nas aplicaçoes ICE, é mais comum o uso do modelo de invocação estática, onde a aplicação invoca uma operação Slice chamando uma função membro de uma classe gerada no proxy. No servidor, o modelo de envio estático comporta-se como de forma análoga: a requisição é enviada ao empregado como uma chamada do tipo estática a uma função membro no proxy. As execuções do ICE no cliente e no servidor trocam sequencias de bytes que representam o código dos argumentos e respostas da requisição(Figura 15).

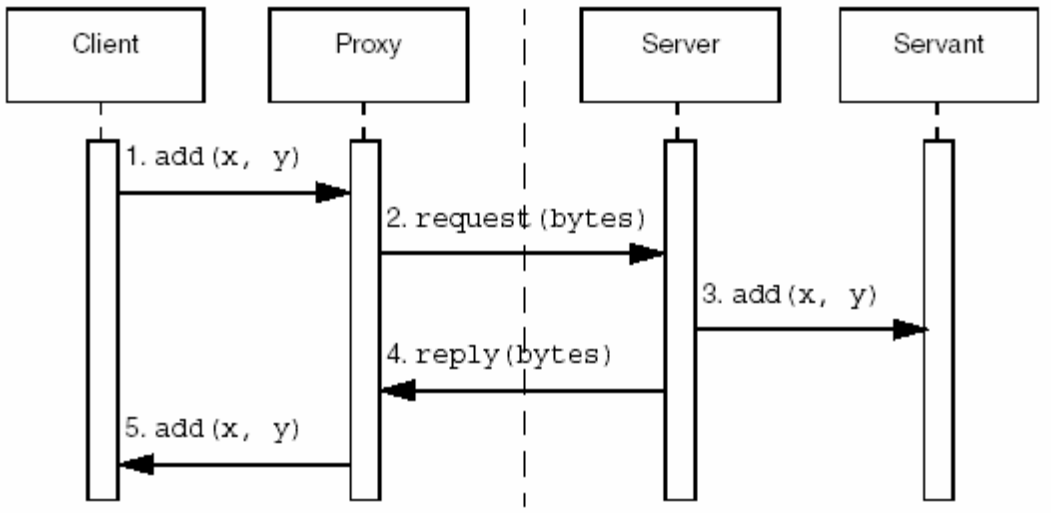

Figura 15. Interações na Invocação Estática[19].

1. O cliente inicia a chamada com a operação Slice *add* chamando a função *add* do membro no proxy.
2. A classe proxy gerada codifica os argumentos em uma sequencia de bytes para ser transmitida ao servidor.
3. No servidor, a classe empregado gerada decodifica os argumentos e chama a funcao *add* na subclasse.
4. O empregado codifica os resultados e transmiti-os para o cliente.
5. O proxy cliente decodificaos resultados e retorna-os ao chamador original da requição.

A aplicação é totalmente inconsciente desta negociação de baixo-nível, e na maioria das vezes isto é bastante vantagoso. No entanto, em algumas situações a aplicação pode utilizar esse tipo de negociação para conseguir tarefas que não seriam possíveis num ambiente do tipo estático. ICE da suporte a serviços de envio e invocação dinamica para essas situações, permitindo aplicações enviar e receber requisições como sequencia de códigos ao invés de utilizar argumento do tipo estático. No ICE as sequencias de bytes podem ser enviadas sem a necessidade de codificar e decodificar os argumentos. Além de ser mais eficiente que o implementação do tipo estático, ele permite aos serviços intermediarios desconhecer o tipo Slice usados pelos destinatários e remetentes.

A utilização de invocacão dinamica deve ser realizada com cuidado devido aos riscos e a complexidade. Por exemplo, uma aplicação que utiliza interfaces *streaming* estão exposto a um alto risco de defeitos em codificar e decodificar argumentos de requisições manualmente a assinatura do argumento de uma operação se mudar. Ao contrario, esse risco é altamente reduzido se usado o modelo de envio e invocação estática porque os erros em linguagens de tipo forte são encontrados cedo, durante a compilação. Por isso, só reconmendavel utilizar invocações dinamicas se as vantagens superarem de forma significativa os riscos.

3.4.4 Protocolos e Transporte

Protocolo ICE pode ser rodado em uma variedade de protocolos de transporte. ICE dá suporte a TCP/IP, SSL e UDP. Para transporte orientado a conexão, ICE oferece uma característica

opcional de controle de conexão ativa que automaticamente recupera conexões que tenham sido perdidas após um espaço tempo pré-estabelecido.

3.4.5 Tolerância a Falhas

- **Detecção de Defeitos**

ICE suporta, como o CORBA, invocações com limite de tempo, para configurações globais ou invocações individuais, onde as operações que não terminaram a execução no tempo especificado retornam com uma exceção de limite de tempo.
ICE não possui compromisso com as características das aplicações: com ICE podemos conseguir o mesmo que podemos conseguir com CORBA com menos esforço, menos código e menor complexidade.

- **Recuperação de Defeitos**

Slice fornece duas qualidades de operações, *nonmutating* e *idempotent*. *Nonmutating* indica que a implementação de uma operação não modifica o estado do objeto final. *Idempotent* indica que o efeito de duas ou mais invocações sucessivas de operações, é igual a uma única invocação. Com operações *idempotent*, se a primeira tentativa de invocar uma operação falhar, o lado cliente tenta reestabelecer a conexão com o servidor e com segurança enviar a requisição falha uma segunda vez. Se o servidor puder ser alcançado na segunda tentativa, o sistema continua normalmente e nunca anunciará um defeito(temporário). Apenas se houver falha na segunda tentativa é que à execução informa um retorno de erro a aplicação. *Idempotent* ajusta o estado a um valor definido(independente do estado anterior) pra que a execução possa enviar mensagens novamente com segurança após a presença das falhas de rede. [20]. Estas operações fornecem uma poderosa ferramenta de recuperação de erros porque, para as operações, a repetição após um erro nunca deve violar a semântica *at-most-once*(semântica do melhor esforço para entrega de requisões aos destinatários corretos).

3.5 JBoss

JBoss é um servidor, *open-source*, de aplicações compatível com Java EE(*Java Enterprise Edition*). Por ser *open-source*, é possível que desenvolvedores ampliem os serviços de *middleware* implementando dinamicamente novos componentes para o servidor. Possui suporte a serviços web Java EE, arquitetura orientada a serviços e modelo de programação orientada a aspectos para o desenvolvimento de soluções *middlewares*[21]. JBoss possui diversos aspectos pré-empacotados dar suporte a segurança, transações e *threads* assíncronos. Essa orientação a aspectos é um diferenciador que habilita os desenvolvedores a adicionar comportamento e competência a qualquer objeto. Essa tecnologia oferece uma ótima flexibilidade para customizar comportamento de servidores de aplicação específicos implantados em ambientes de requisição. JBoss é um *microkernel* com um *framework* orientado a aspectos que usa essa base para criar servidores de aplicação Java EE com um conjunto completo de APIs.

3.5.1 Arquitetura JBoss

A arquitetura de servidores de aplicação JBoss é dividida em quatro camadas principais(Figura 16): camada *microkernel*, camada de serviços, camada de aspectos e camada de aplicação.

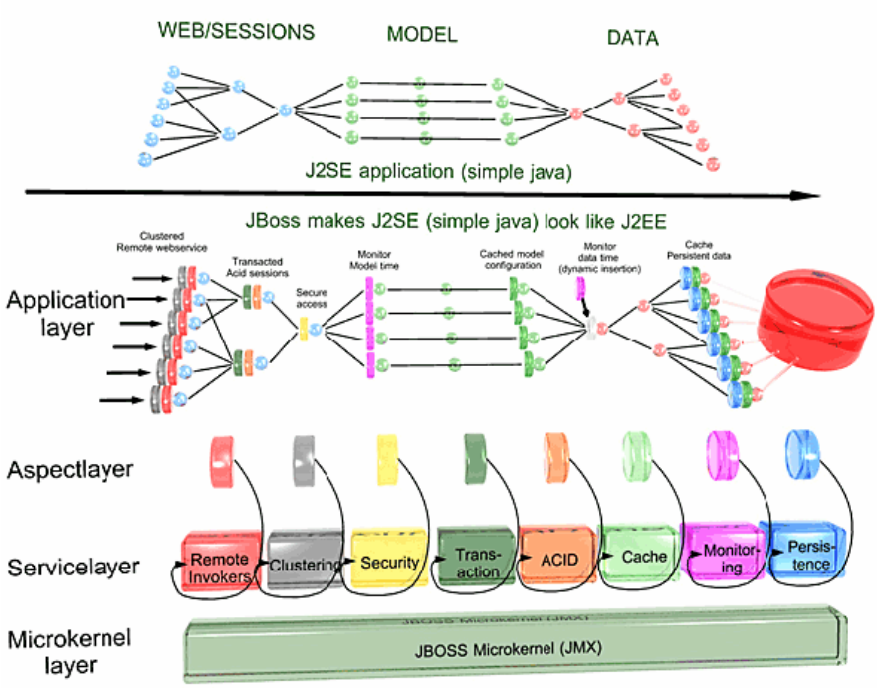

Figura 16. Arquitetura de servidores de aplicação Jboss

- **Camada *microkernel***: no seu núcleo tem um servidor baseado em *microkernel*. Utilizando JMX(*Java Management Extensions*), o *microkernel* entrega um modelo de componente leve mas que oferece avançadas características de carregamento de classes e uma forte introdução (reintrodução) de aplicações sem precisar reiniciar as aplicações para serem atualizadas e em todo gerenciamento do ciclo de vida. Essa é a base para a implantação flexível de componentes e a arquitetura orientada a aspectos.
- **Camada de serviço**: acima da camada *microkernel* existe a camada de serviço que consiste em uma série de serviços onde cada um é cuidadosamente empacotado e fortemente introduzido. Serviços de implantação no JBoss podem alcançar qualquer lugar, desde transações e serviço de mensagens até serviços de segurança. Cada serviço é empacotado em um arquivo de serviços(SAR) onde cada SAR é individualmente implantado, facilitando a extensão do JBoss. Assim, os desenvolvedores podem facilmente incluir/remover serviços ou construir seus próprios serviços e implantá-los como SARs dentro de um servidor de aplicação JBoss.
- **Camada de aspectos**: baseada no modelo de programação orientada a aspectos(AOP). JBoss utiliza o conceito de interceptores, que permitem ao sistema adicionar transparentemente o comportamento fornecido pelos serviços em todos os objetos. Desenvolvedores podem adicionar ou remover interceptadores de acordo com as suas especificações. É a camada de aspectos que permite aos desenvolvedores acrescentar capacidades como transações ou serviços de aglomerados.
- **Camada de aplicação**: é nesta camada onde as aplicações são hospedadas. As aplicações elevam as capacidades da infra-estrutura JBoss caso utilizem diretamente o serviço de *container* ou se empregam a camada de aspectos e etiquetam os aspectos para adicionar comportamentos aos objetos.

3.5.2 Transações

ArjunaCore é um sistema de programação orientado a objeto que fornece um conjunto de ferramentas para a construção de aplicações tolerantes a falha usando objetos e transações. ArjunaCore é utilizado apenas para transações locais, quando são necessárias transações distribuídas o ArjunaCore fornece o gancho necessário para que a informação seja transmitida. Por isso, é JBoss utiliza o JBossTS para transações distribuídas tendo o ArjunaCore como nucléo para o serviço de transações.

- **Arquitetura *ArjunaCore***

ArjunaCore foi projetado e implementado para fornecer uma forma de construir aplicações distribuídas tolerantes a falhas. Para isso foram levadas em consideração três propriedades de sistemas consideradas de extrema importância:

- **Modularidade**: o sistema deve ser fácil de se instalar e executar. Ou seja, é possível substituir um componente do ArjunCore por um já existente na base do sistema.
- **Integração de mecanismos**: um sistema tolerante a falhas precisa de uma variedade de funções para controle de concorrência, detecção de defeitos, recuperação de erros, etc. Estes mecanismos podem ser fornecidos de uma maneira integrada para que seu uso seja fácil e natural.
- **Flexibilidade**: os mecanismos devem ser flexíveis, permitindo às aplicações especificar características que melhor se adequam aos seus objetivos, por exemplo, tipo específico de concorrência e controle de recuperação.

- Objetos e Transações

Consideramos um modelo de computação onde cada programa de aplicação manipula objetos persistentes sob o controle de transações atômicas. As operações de um objeto têm acesso a instâncias de variáveis e podem modificar o estado deste objeto. Todas as operações de invocação são controladas pelo uso de transações atômicas que fornecem as propriedades ACID.
Os modelos de objetos e transações fornecem um arcabouço natural para o desenvolvimento de sistemas tolerantes a falhas com objetos persistentes. Quando não se usa objetos persistentes, se assume estar em um estado passivo em um objeto de armazenamento(*object store*) e ativado por demanda pelo carregamento dos estados e métodos de um armazenador de objeto persistente para um armazenador não-estavel(volátil), e associando-o com um objeto *containner*.

- Arquitetura do Sistema

ArjunaCore possui dois módulos principais para dar suporte a objetos persistentes: módulo de transações atômicas e módulo de armazenamento de objetos.

- **Módulo de transações atômicas**: fornece suporte de transações atômicas aos programas de aplicação na forma de operações para inicializaçãoo, sucesso(*commit*), e cancelamento de transações.
- **Módulo de armazenamento de objetos**: fornece um repositório de armazenamento estável. Esses objetos são registrados com um identificador únicos(UID) para seu nomeamento.

A estrutura ArjunaCore é altamente modular: pelo encapsulamento de propriedades persistente, recuperabilidade, compartilhamento, seqüenciabilidade nos módulos de transações atômicas,

interfaces bem definidas para o suporte a ambientes, ArjunaCore consegue um decrescimento significativo de modularidade além da portabilidade.

Serviços de Armazenamento de Objetos

- **Salvando estados de objetos**: ArjunaCore possui a capacidade de lembrar do estado do objeto, por vários motivos: inclusão de recuperação (o estado representa algum estado passado do objeto), persistente (o estado representa o estado final do objeto em uma aplicação de terminação), e para propósito de distribuição (o estado representa o estado atual de um objeto que deve ser levado ao local remoto).
- **Armazenador de objetos** (*Object Store*): fornece uma interface bastante restrita que pode ser implementada de vários maneiras. Por exemplo, armazenadores de objetos são implementados em memória compartilhada. Quando um objeto transacional está finalizando com sucesso (*commiting*), é necessário para ele tornar persistentes algumas mudanças de estado para que possa recuperar na ocorrência de um defeito e continuar com a finalização da transação, ou fazer um *rollback*. Para garantir as propriedades ACID, essas mudanças de estado podem ser descartadas (*flushed*) pela implementação do estado persistente antes da transação proceder com a ação de sucesso. Senão a aplicação pode presumir que a transação foi um sucesso quando de fato a mudança de estado pode ainda residir dentro da *cache* do sistema operacional e pode ser perdida por um subseqüente defeito da maquina. Como padrão, ArjunaCore garante que tais mudanças de estados sempre serão descartadas.

Serviços de Transações Atômicas

Recuperação e Persistência: fornece a ativação e desativação de objetos, recuperação de objetos e também mantém o nome dos objetos (na forma de objetos UID). Os objetos são divididos em três tipos. Objetos recuperáveis são geradas e mantidas informações de recuperação apropriadas do objeto. Estes objetos possuem um tempo de vida que não ultrapassa o programa de aplicação que o criou. Objetos recuperáveis e persistentes, por sua vez, têm tempo de vida maior. Por fim, objetos podem não possuir nenhum dos tipos citados. Neste caso, nenhuma informação de recuperação é mantida. O objeto não pode ganhar ou perder capacidade de recuperação em um ponto qualquer durante seu tempo de vida.

- **JBossTS**

JBossTS (*JBoss Transaction Service*) é a solução de *middleware* que dá suporte a aplicações criticas em ambientes de computação distribuída. ArjunaCore é o núcleo para o JBossTS. JBossTS é uma extensão dos serviços de transação do ArjunaCore fornecendo alto desmpenho, alta confiabilidade nos processos de transação e fornece JTA (*Java Transactions API*), JTS (*Java Transaction Service*) e o padrão dos serviços *Web*. JBossTS desempenha um papel crítico na construção confiável de aplicações sofisticadas de *e-business*, garantindo absoluta conclusão e exatidão dos processos de negócios.

Há diversos participantes em uma transação distribuída JBossTS, incluindo:

- **Gerenciador de transação**: está distribuído no sistema de transações. Gerencia e ordena o trabalho envolvido na transação.
- **Gerenciador de contexto**: identifica uma transação particular.
- **Cliente transacional**: um cliente transacional pode invocar operações de um ou mais objetos transacionais em uma única transação. O cliente transacional que iniciou a transação é chamado de originador de transação.

- **Objeto transacional**: objeto cujo comportamento é afetado por operações ocorridas dentro de um contexto transacional. Um objeto transacional também pode ser um cliente transacional.
- **Recurso recuperável**: um recurso recuperável é um objeto transacional cujo estado é salvo em um armazenamento estável se a transação for bem-sucedida e cujo estado pode ser reajustado para o que era no começo da transação se a transação é restaurada a um ponto de recuperação.

3.5.3 Tolerância a Falhas

- **Detecção de Defeitos**

JBoss utiliza o JBoss Remoting para detectar defeitos entre o lado cliente e o lado servidor da aplicação. JBoss Remoting fornece uma única API para a maioria da redes cliente/servidor baseadas em invocações e serviços que usam inclusão de protocolos de transporte e codificação de dados. A API JBoss Remoting fornece a capacidade de fazer chamadas remotas síncronas e assíncronas, envio e recepção de mensagens *callback*, e descoberta automática de servidores remotos. A intenção é permitir o uso de diferentes protocolos de transportes para servir para diferentes propósitos, mantendo a mesma API para fazer invocações remotas e apenas precisando de mudanças de configuração sem mudanças de código.

Algumas características disponíveis no JBoss Remoting são:

- **Identificação de servidor**: um identificador simples baseado em uma URL que permite os servidores remotos serem identificados e chamados. Isso é feito via *InvokerLocator*, que pode ser representado por uma simples *string* com um formato baseado em URL. Isso é tudo que se precisa para criar um servidor remoto ou fazer uma chamada a um servidor remoto.
- **Codificação de dados conectável**: diferentes codificadores e decodificadores de dados podem ser usados para converter a carga útil invocadas no formato de dados desejável para a linha de transferência.
- **Descoberta automática**: descobrir servidores remotos enquanto estão *on/off line*. Multicast ou JNDI.
- **Grupos de servidores**: capacidade de agrupar servidores em domínios lógicos, portanto a comunicação entre servidores só se dará dentro de um domínio especifico.
- ***Callbacks***: pode receber e enviar *callbacks* pelos modelos *push/pull*. O modelo *push* permite armazenamento persistente e gerenciamento de memória.
- **Notificação de conexões falhas**: notificação se o cliente ou o servidor falhou
- **Compressão de dados**: pode se usar compressão (*de*)*marshaling* para a compressão de grandes cargas úteis.

Detecção Automática

Para adicionar detecção automática, o detector *remoting* precisará habilitar os lados cliente e servidor assim como o registro de rede(*NetworkRegistry*) no lado cliente(Figura 18).
Quando o detector no lado servidor é criado e iniciado, ele irá periodicamente por em *InvokerRegistry* todos os invocadores servidor que foram criados. O detector irá usar a informação para publicar

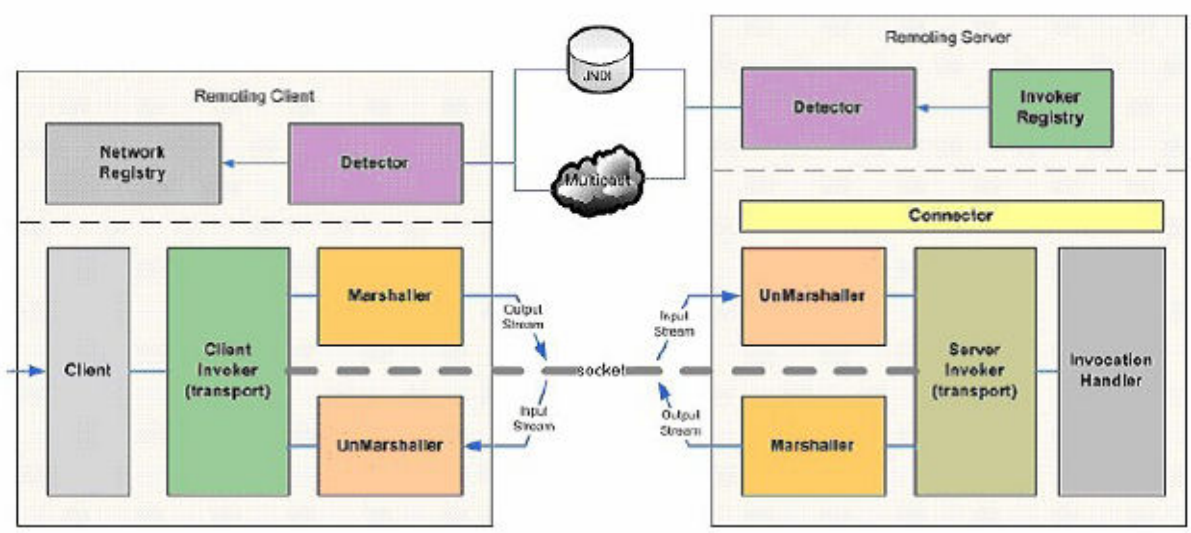

Figura 17. Inclusão de detecção automática

uma mensagem de detecção contendo o localizador e o subsistema suportado por cada invocador servidor. Um subsistema é um identificador de que camada superior do sistema o suportador de invocações está associado. A publicação da mensagem de detecção será enviada via mensagens *multicast* ou ligadas a um servidor JNDI. No lado cliente, o detector receberá as mensagens ou a poll do servidor JNDI para detecção de mensagens, se o detector determinar que a mensagem de detecção é para um servidor remoto que está *online* ele o registrará em *NetworkRegistry*. *NetworkRegistry* passará a informação de detecção para todos os servidores remotos descobertos. A mudança no *NetworkRegistry* também pode ser feita quando o detector descobre que o servidor remoto não está muito disponível e remove-o do registro.

- **Recuperação de Defeitos**

A Figura 19 mostra a arquitetura principal dos componentes dentro da recuperação de defeitos.

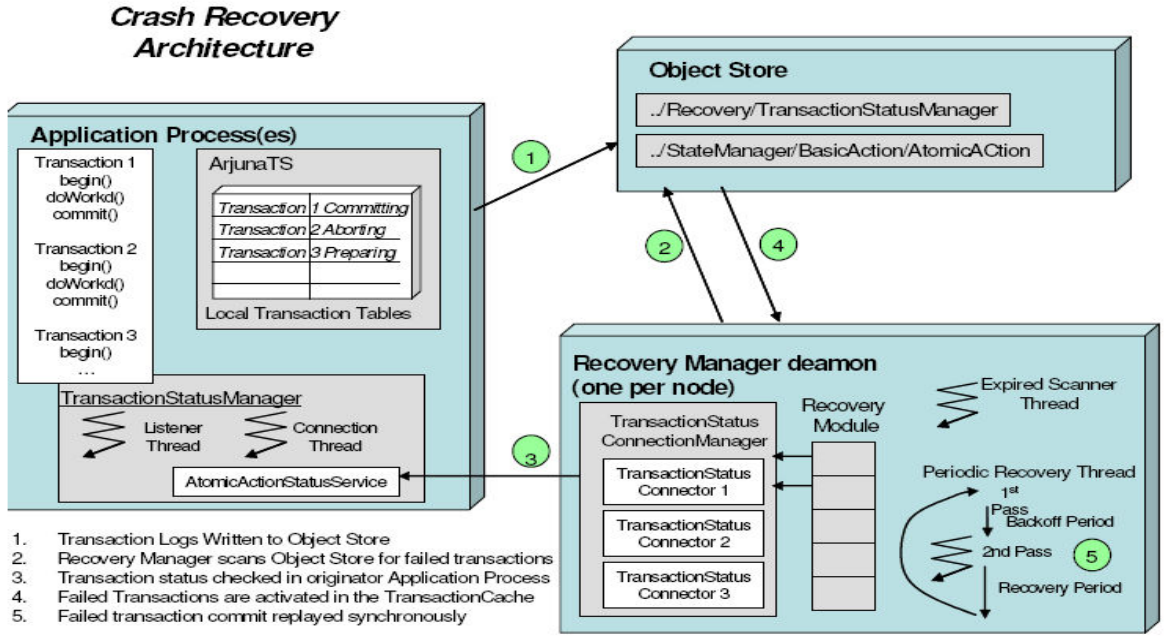

Figura 18. Arquitetura da recuperação de defeitos[21].

O gerenciador de recuperação(*recovery manager*) é o processo central responsável pela realização da recuperação de defeitos. Apenas um gerenciador de recuperação é executado por nó. O armazenador de objetos fornece armazenamento persistente de dados para transações de registro de dados. Durante o processamento de transação normal, cada transação registrará os dados persistentes necessários para que ela possa ser finalizada com sucesso(*commit*). Na caracterização do sucesso da transação, os dados são removidos. Porém se a transação falhar, os dados permanecem no armazenador de objeto.
As funções do gerenciador de recuperação são:
- Procurar periodicamente no armazenador de objetos por transações que possam ter falhado. Transições falhas são indicadas pela presença de registro de dados depois do período de tempo que a transação normalmente já deveria ter terminado.
- Verificar junto ao processo de aplicação que originou a transação se a transação ainda está em andamento ou não.
- Recuperar a transação pela reativação da transação e então repetir a fase 2 do protocolo de sucesso(*commit*).

3.6 Comparação de Mecanismos

A comparação feita neste trabalho é baseada nos mecanismos de detecção e recuperação de defeitos conhecidos na literatura de tolerância a falhas. Neste trabalho estudamos os mecanismos de mensagens de *heartbeat*, consenso, detecção de defeitos não confiavéis e *multicast* totalmente ordenado como mecanismos de detecção de defeitos e os mecanismos de recuperação por retorno e recuperação por avanço como mecanismos de recuperação de defeitos. As infra-estruturas escolhidas para tal estudo foram desenvolvidas tanto no meio acadêmico quanto no meio comercial. Sendo elas Sprint, Horus, TAO, ICE e JBoss.

3.6.1 Sprint

XS(*Durability Server*) são replicados para uma alta disponibilidade. Se houver a queda de XS, DS(*Data Servers*) não precisam esperar o XS se recuperar para o sucesso de suas transações de atualização. DS são replicados para melhor desempenho e disponibilidade. Quando há a queda de um DS, outra instância é iniciada em um servidor físico operacional. A nova instância estará pronta para processar trasnsações após receber ao dados do banco de dados que estão armazenados no XS.
Servidores fisicos podem falhar por queda(*crash*) mas não comportam-se maliciosamente(falhas Bizantinas). O servidor pode se recuperar após um defeito, mas perde toda a informação armazenada na memória principal antes da queda. Cada servidor tem acesso ao armazenador estável local(i.e. disco) cujo conteúdo sobrevive a quedas. O defeito em um servidor físico implica no defeito de todos os servidores locais nele hospedados.
O sistema aplica detecção de defeitos não-confiavel: (a) servidores falhos serão detectados por servidores operacionais, mas (b) um servidor operacional pode suspeitar erroneamente que um servidor tenha falhado, caso ele esteja muito lento.
Nos ES, Se um defeito ocorre durante a execução do protocolo de terminação, a transação finaliza com sucesso ou aborta, dependendo de quando o defeito ocorrer. Se a requisição do ES para a terminação da transação alcançar todos os DS participantes, eles estarão prontos para dar sucesso à transação, e seus votos entregues antes de qualquer outro voto para a transação, então a resposta será de sucesso.

Uma nova instância de ES será, imediatamente, criada em qualquer servidor físico. Durante a inicialização, o ES manda mensagens para um dos XS, perguntando pela configuração atual do banco de dados. O ES estará pronto para processar requisições assim que for recebida a configuração do banco de dados.
Nos DS a recuperação de servidores defeituosos é simples, pois é necessário apenas criar uma outra instância do servidor em um servidor físico operacional. Com um DS configurado para evitar acesso ao disco, não existe imagem de banco de dados para ser restabelecida de um disco local após a queda. Em conseqüência, uma nova cópia do DS defeituoso será implantada em um servidor físico usando o estado armazenado pelo servidor de durabilidade. São necessários alguns cuidados para evitar inconsistência das estratégias de recuperação.
Nos XS's mensagens entregues perdidas pela recuperação do XS podem ser recuperadas por um XS operacional. Cada XS cria, periodicamente, uma imagem no disco, do estado atual do banco de dados. Esse estado é construído a partir das mensagens entregues pelos XS, como parte do protocolo de terminação das transações de atualização.
Assim, concluimos que no Sprint os servidores físicos utilizam detectores de defeitos não-confiáveis. Isto se aplica também aos servidores lógicos. Já na recuperação de defeitos cada servidor cria uma nova instância do servidor falho e se reconfigura a partir de um ponto de recuperação que é uma imagem de banco de dados armazenada no servidor de durabilidade(XS), que se auto-recupera com a utilização de replicação.

3.6.2 Horus

Horus fornece suporte eficiente para o modelo de execução virtualmente síncrono. Este modelo é baseado em grupos de processos e primitivas de comunicação e dá suporte a uma variedade de ferramentas de tolerância a falhas, por exemplo, para execução de requisições com o balanceamento de cargas, computação tolerante a falhas, dados replicados coerentemente e segurança. Horus não fornece operações de controle de propósito geral, e possui apenas um formato de endereçamento. Protocolos de camadas distintas podem ser misturados e conectados livremente.
Horus tolera falhas do tipo *fail-stop*, pois a utilização do modelo de sincronização virtual os processos garante que é possível distinguir um processo falho de um que está lento. adicionalmente os membros de um mesma visão tomam conhecimento do defeito em um mesmo instante lógico. A camada TOTAL fornece entrega totalmente ordenada para mensagens *multicast* dentro de um grupo. A camada TOTAL fornece um tempo de entrega para as visões sobreviventes. A camada NAK fornece uma sequencia de numeros que são analisados pelo receptor como forma de detectar pacotes perdidos. Os defeitos são detectados pelos *endpoints*.
Visões(*views*) são utilizados para recuperação de defeitos. Um endereço de grupo está associado a um objeto grupo que mantem o estado do protocolo local. Um *endpoint* pode possuir múltiplos objetos e diferentes visões de um mesmo grupo. Então quando um processo falha, será criada uma nova visão em um mesmo grupo para recuperação e serão utilizados protocolos de associação para alcançar alguma concordância de visões entre múltiplos objetos grupo de um mesmo grupo.

3.6.3 TAO

TAO é uma infra-estrutura que estende o padrão e os componentes do *framework* ACE e que foi desenvolvido para prover um alto desempenho no padrão CORBA às aplicações de tempo real.
Sua arquitetura é baseada em uma melhoria dos componentes da arquitetura CORBA, principalmente no protocolo IIOP. As classes de falhas toleradas pelo TAO são falhas por

temporizacao, pois suas aplicações possuem um *time-out* de duração decidida pelo cliente e controlado em tempo de execução pelo objeto servidor.
TAO usa replicação como mecanismo de tolerância a falhas. Utilizam mensagens de *heartbeat* como mecanismo de detecção de defeitos pela extensão do protocolo DOORS do CORBA. TAO também utiliza mecanismos de *polling*(votação) como mecanismos de detecção de defeitos. Esses mecanismos são utilizados apenas nas aplicações que usam replicação ativa ou passiva. TAO inclui também um tipo de replicação híbrida denominada replicação semi-ativa onde as mensagens de *heartbeat* e de *polling* não são mais utilizadas para fornecer uma diminuição do *overhead* no sistema. Na replicação semi-ativa, são usadas mensagens confiável de *multicast* para a detecção de defeitos. As mensagens são enviadas por *multicast*, mas sua ordenação é forçada pelo envio de uma lista ordenada de referências do lado cliente.
A recuperação de erros é feita a partir de arquivos *log* que registram o estado atual do objeto primário, para que sejam recuperados na ocorrência de alguma falha. O mecanismo de recuperação de erros no TAO é baseado na redundância das entidades; um dos objetos é definido como primário e, caso este objeto falhe, um dos objetos redundantes será eleito o novo primário e o sistema continuará sendo executado com o registro arquivado em *log* do último estado ideal do objeto primário falho. TAO também pode usar uma forma simples de consenso baseado no modelo de replicação ativa com votação de FT-CORBA.

3.6.4 ICE

ICE detecta defeitos usando temporizadores, pois as operações que não terminaram a execução no tempo especificado lançam uma exceção de limite de tempo. ICE não possui compromisso com as características das aplicações: com ICE podemos conseguir o mesmo que podemos conseguir com CORBA com menos esforço, menos código e menor complexidade.
A literatura sobre ICE não especifica os mecanismos de detecção de defeitos usados para gerar a exceções, porém ICE foi criado para utilizar alguns mecanismos do padrão CORBA, por exemplo, replicação e temporizadores.
A linguagem Slice fornece duas qualificações(*idempotente* e *nonmutating*) para operações como garantia de utilização da semântica *at-most-once*. Assim, ele fornece um recuperação de erros de redes falhas porque com essas qualificações o reenvio após um erro não pode violar a semântica *at-most-once*. A utilização das operações *idempotent* e *nonmutating* na recuperação de defeitos é garantida pelo retorno ao ultimo ponto livre de falhas. Pois a operação *idempotent* garante o melhor esforço para a entrega da requisição e a operação *nonmutating* garante que o estado final não será modificado. A operação *idempotent* ajusta o estado a um valor definido para que a infra-estrutura de tempo de execução do ICE possa enviar mensagens novamente na presença das falhas de rede.

3.6.5 JBoss

JBoss é um *middleware* com uma arquitetura em camadas. Entre essas camadas, especificamente na camada de *microkernel*, existe uma extensão que serve como integração entre os componentes do JBoss e os serviços da aplicação.
As transações no JBoss são monitoradas por um componente chamado JBossTS, que utiliza mecanismos de *rollback* na recuperação de transações falhas de acordo com um ponto de recuperação armazenado em um dos componentes da transação denominado *recovery resources*.
O *JBoss Remoting* fornece serviços de chamadas remotas síncronas e assíncronas, chamadas *callback* e *multicast* confiável para detectar se um objeto e está ativo, inativo ou falhou. Além de um detector automático que envia mensagens periódicas aos clientes/servidores e armazenam a

informação de seu estado em um *log*(*NetworkRegistry*) para descobrir quando algum cliente/servidor falhou.
Além disso, existe uma arquitetura denominada *ArjunaCore* que foi desenvolvida para fornecer melhores mecanimos de tolerância a falhas às aplicações do JBoss, introduzindo objetos persistentes e transações atômicas. Combinados esses armazenar os estados dos objetos e persistir esses estados, de modo que eles possam ser recuperados no evento de uma falha.

3.6.6 Serviços Fornecidos pelas Infra-Estruturas de *Middleware*

Com base no que foi discutido sobre as infra-estruturas estudadas, apresentamos na Tabela 3 uma comparação dos mecanismo de tolerância a falhas e os serviços fornecidos em cada infra-estrutura.

Tabela 3. Comparação dos mecanismos de tolerância a falhas e as infra-estruturas de *middleware* estudados

		Infra-Estrutura de *Middleware*				
		Sprint	Horus	TAO	ICE	JBoss
Detecção de Defeitos	Mensagens de *heartbeat*		X	X		X
	Multicast Confiável	X	X	X		X
	Consenso	X			X	
	Detecção de Defeitos Não-Confiavel	X				
Recuperação de Defeitos	Recuperação por Retorno	X		X	X	X
	Recuperação por Avanço		X	X	X	

Como podemos observar na tabela 3, o Sprint e o TAO fornecem detecção de defeito não confiáveis, onde no Sprint os defeitos serão detectados por servidores que estejam operando enquanto que no TAO serão detectados por réplicas ativas devido o termino de *time-outs*. Já no Estas duas infra-estruturas também coincidem no mecanismo de recuperação que junto com o JBoss implementam recuperação por retorno. O TAO, e também o Jboss, armazena as informações em um arquivo *log*, já o Sprint armazena periodicamente a imagem do banco de dados no servidor de durabilidade(XS), assim ele recupera todo o banco de dados e precisa apenas atualizar as informações perdidas após o defeito.
Horus, TAO e JBoss utilizam *multicast* confiável como detecção de defeitos. A camada TOTAL do Horus garante a ordenação total na entrega de mensagens *multicast* em um grupo e a camada MBRSHIP detecta os defeitos pelos membros vizinhos. No TAO o a ordenação do *multicast* é forçado, para isso é necessário o envio ao cliente de uma lista ordenada de referencias, já o JBoss utiliza *multicast* para detectar o estado dos objetos e descobrir se eles estão vivos.
Apenas a infra-estrutura TAO utiliza mensagens comuns de *heartbeat*, essas mensagens são enviadas periodicamente ao detector de falhas e entre as replicas a fim de se detectar algum elemento falho. O Horus, o ICE e o Jboss utilizam temporizadores de mensagens como mecanismo de detecção.
Horus e Ice utilizam recuperação de erros por avanço. Ice utiliza a operação *idempotent*(que garante a semântica *at-most-once*) pois ele ajusta o estado a um valor definido, independente do

estado anterior, para que o tempo de execução possa enviar novamente com segurança, na presença de falhas da rede. No Horus, depois da utilização do protocolo *flush*, é criado uma nova instância da visão, assim que estiver pronta as mensagens instáveis do membro defeituoso são enviadas e o sistema está pronto para continuar com sua execução.

3.6.7 Análise dos Mecanismos

As mensagens por *multicast* confiavél devem ser utilizadas para aumentar a garantia ao usúario de entregas confiavéis aos receptores. Como sistemas distribuídos trabalham com um numero grande de receptores e clientes, esse tipo de mecanismo deve ser empregado em sistemas que não suportem perdas de pacotes. Por exemplo, nas transações bancarias os processos não podem falhar durante as transações, ou o processo se completa ou é cancelado.
Mecanismos de consenso, os nós participantes não precisam saber o resultado final decido pelos membros, apenas o membro líder conhece a decisão final da votação. Consenso pode ser empregado em sistemas que possuam falhas arbitrárias (ou bizantinas), para que mensagens maliciosas enviadas por membros da votação não influenciem na consistencia dos resultados.
Detectores de defeitos não-confiável é utilizado por serviços que têm a necessidade de saber quem está falho ou não no sistema. O detector de defeitos não-confiavél pode gerar informações incorretas, isto é, ele pode marcar que uma entidade está falha, quando na realidade ela está ativa e vice-versa, o que pode acarretar em gastos computacionais já que as entidades podem ser marcadas e desmarcadas como falho variás vezes durante a execução do sistema. A análise de falhas nas entidades se dá pelo envio periódico de mensagens aos participantes, sistemas que não tenham compromissos com tempo podem utilizar esse mecanismo, por exemplo, sistemas que necessitem de implementações de Qualidade de Serviço(QoS).
Recuperação por retorno o sistema deve guardar informações de estados consistentes por um determinado intervalo e tempo, para em caso de defeitos no sistema está informação sirva como ponto de retorno. Essa técnica deve ser usada por sistemas que necessitem de uma execução completa da aplicação para que o estado final esteja livre de falhas. Recuperação por retorno é comumente utilizada em sistemas seguros, sistemas em que a seguança é mais importante que a disponibilidade e devem apresentar comportamentos *fail-safe*. Por exemplo, sistemas de transporte urbanos utilizados na Europa para evitar colisões de trens, bondes.
Recuperação por avanço o sistema passa a um novo estado ainda não ocorrido após a ocorrencia de uma falha para continuar operando normalmente. Essa técnica é usada quando não há tempo de voltar para o estado anterior para retomar a execução, ou quando as ações podem ser desfeitas, por exemplo, sistemas em tempo real.

Capítulo 4

Conclusões e Trabalhos Futuros

Este trabalho procurou abordar uma visão geral de tolerância a falhas em *middleware* e seus mecanismos de detecção de defeitos e recuperação de erros, as infra-estruturas de *middleware* bastante conhecidas e de diferentes ambientes já que foram abordadas infra-estruturas tanto do âmbito acadêmico como do âmbito comercial. A tolerância a falhas fornece uma abstração ao usuário dos problemas ocorridos no sistema. A tolerância a falhas é um atributo importante para diversas aplicações na atualidade devido ao grande uso de aplicações via Internet e pelo alto grau de complexidade e de utilização de algumas aplicações, onde uma falha pode causar eventos catastróficos com perda de vidas humanas ou grandes perdas financeiras.

4.1 Contribuições

Com o objetivo de analisar as infra-estruturas de *middleware* existentes, este trabalho apresentou os mecanismos de detecção (mensagens de *heartbeat*, consenso, detectores de defeitos não-confiavéis e *multicast* confiavel) e recuperação de defeitos (recuperação por retorno e recuperação por avanço) e as infra-estruturas (Sprint, Horus, TAO, ICE e JBoss) que foram escolhidas para nossa análise. A comparação se deu com base nos mecanismos apresentados e nos serviços apresentados pelas infra-estrturas para o desenvolvimento de aplicações.
O objetivo da comparação entre os mecanismos de tolerância a falhas e infra-estruturas de *middleware* é bastante importante para que os desenvolvedores possuam alguma base prévia de informação quando da escolha de uma infra-estrutura para o desenvolvimento de sua aplicação. Assim evita-se um desperdício econômico caso uma aplicação seja desenvolvida em um ambiente impróprio ou uma grande catástrofe se o sistema desenvolvido não tolerar as falhas da forma esperada pelo desenvolvedor devido a infra-estrutura utilizada.
A organização dos resultados de acordo com os quesitos de detecção e recuperação facilitará o entendimento da comparação e permitirá que os desenvolvedores avaliem esses critérios de forma separada. Pois, essas duas funcionalidades são essenciais a construção de sistemas distribuídos tolerantes a falha.

4.2 Trabalhos Futuros

Este trabalho foi apenas um primeiro passo na análise dos mecanismos de tolerância a falhas e das infra-estruturas de *middleware* da literatura. Para que haja uma quantidade maior de informações aos desenvolvedores é necessário que as abordagens deste trabalho sejam ampliadas.
Os mecanismos de detecção de defeitos e recuperação de erros são muito importantes no desenvolvimento de sistemas distribuídos, porém existem outros mecanismos de tolerância a falhas que podem ser estudados. Além disso, existem outras maneiras de se detectar defeitos que podem ser analisadas e adicionadas a este estudo. São conhecidas na literatura uma gama de infra-estrutras de *middleware*, por isso, este estudo pode ser ampliado a um numero maior de infra-estruturas.
Para uma melhor avaliação dos desenvolvedores, seria necessário um estudo que envolve um número maior de mecanismos de tolerância a falhas, detecção de defeitos, recuperação de erros e de infra-estruturas de *middleware*. Assim seria feita uma analise minuciosa dos requisitos da aplicação e de qual infra-estrutura de *middleware* a ser utilizada.

Bibliografia

[1] Tanembaum, A e Steen, M. Sistemas Distribuídos: princípios e paradigmas. São Paulo: Prentice Hall, 2. ed., 2007

[2] Lamprort, L. 1977. Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. 3, 2 (Mar), 125±143.

[3] Anderson, T e Lee, P. Fault-Tolerance – Principles and Practice. New York: Springer-Verlag, 2. ed., 1990.

[4] Pradhan, D. K., *Fault Tolerant System* Design. Prentice Hall, New Jersey, 1996.

[5] Roteiro para exploração de conceitos básicos de tolerância a falhas. Disponível em:<http://www.inf.lasalle.tche.br/~barreto/disciplinas/tf/artigos/Taisy_ConceitosDependabilidade.pdf>. Acesso em 23 de março de 2008.

[6] Schneider, F. B. (1993). What good are models and what models are good? In Mullender, S., editor, *Distributed Systems*, páginas 17–26. Addison-Wesley, Workingham, 2ª Ed.

[7] Cristian, F. (1991). Understanding fault-tolerand distributed systems. *Communications of the* ACM, 34(2):56-78.

[8] Gärtner, Felix C., Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. ACM Comput. Surv. 31(1): 1-26 (1999)

[9] J. C. Laprie. *Dependable computing and fault tolerance : Concepts and terminology*. 15th International Symposium on Fault-Tolerant Computing Systems, Junho 1985

[10] A. Avizienis, J.-C. Laprie, e B. Randell. *Fundamental Concepts of Dependability*. Technical Report 739. Department of Computing Science. University of Newcastle upon Tyne. 2001.

[11] BIRMAN, K. *Building secure and reliable network applications*, 1996.

[12] L. Camargos, F. Pedone e M. Wieloch Sprint: A Middleware for High-Performance Transaction Processing 2nd European Conference on Systems Research (EuroSys'2007)

[13] Robbert van Renesse, Kenneth P. Birman e Silvano Maffeis, Horus, a flexible Group Communication System, Communications of the ACM, Abril 1996.

[14] D. D. Clark and D. L. Tennenhouse. *Architectural considerations for a new generation of protocols*. In Proceedings of the ACM SIGCOMM Conference on Communications, Architectures, Protocols and Applications, pg 200--208, Philadelphia, September 1990. ACM.

[15] Van Renesse, R., Birman, K.P., Friedman, R., Hayden, M., e Karr, D.A. A framework protocol composition in Horus. *Proceeding of the 14 Symposium on the Principles of Distributed Computing* ACM (Ottawa, 1995). pp. 88-89

[16] Schmidt, D. C.; Gokhale, A.; Harrison, T. H.; Levine, D.; & Cleeland, C. "*TAO: a High-Performance Endsystem Architecture for RealTime CORBA*." RFI response to the OMG Special Interest Group on Real-time CORBA, 1997.

[17] D. C. Schmidt and C. Cleeland, "*Applying Patterns to Develop Extensible and Maintainable ORB Middleware*," Communications of the ACM, Dec. 1997.

[18] B. Natarajan, A. Gokhale, D. C. Schmidt, e S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the 2*nd *International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Set. 2000

[19] M. Henning et al., *Distributed Programming with Ice,* ZeroC, 2003; disponível em: <www.zeroc.com/Ice-Manual.pdf>. Acesso em 20 de março de 2008.

[20] Michi Henning: A New Approach to Object-Oriented Middleware. IEEE Internet Computing 8(1): 66-75 (2004)

[21] JBoss Transactions Failure Recovery Guide; disponível em: <http://www.jboss.org/jbosstm/docs/4.2.3/manuals/pdf/core/FailureRecoveryGuide.pdf >. Acesso em Abril de 2008.

[22] Jboss Transactions programmers Guide; disponível em <http://www.jboss.org/jbosstm/docs/4.2.3/manuals/pdf/core/ProgrammersGuide.pdf> Acesso em Abril de 2008.