Instituto de Computação
Universidade Estadual de Campinas

Ações Atômicas Distribuídas para Xchart

Luciana de Paula Brito

Banca Examinadora:

- Alcides Calsavara
  Departamento de Informática — PUC — PR
- Edmundo R. M. Madeira
  Instituto de Computação — UNICAMP
- Hans K. E. Liesenberg
  Instituto de Computação — UNICAMP (Orientador)
- Luiz Eduardo Buzato
  Instituto de Computação — UNICAMP (Co-orientador)
- Maria Beatriz Felgar de Toledo
  Instituto de Computação — UNICAMP (Suplente)

Ações Atômicas Distribuídas para Xchart

Este Exemplar corresponde a redação final da tese devidamente corrigida e defendida pela Srta. **Luciana de Paula Brito** e aprovada pela Comissão Julgadora.

Campinas, 19 de dezembro de 1996.

Prof. Dr. Hans E. K. Liesenberg
Orientador

Dissertação apresentada ao **Instituto de Computação** — UNICAMP, como requisito parcial para a obtenção do título de **Mestre em Ciência da Computação.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Brito, Luciana de Paula

B777a Ações atômicas distribuídas para Xchart / Luciana de Paula Brito -- Campinas, [S.P. :s.n.], 1996.

Orientador : Hans E. K. Liesenberg

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de comunicação. 2. Programação orientada a objetos. 3. Sistemas operacionais distribuídos (Computadores). 4. Interfaces de usuários (Sistema de computador). I. Liesenberg, Hans E. K. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Tese de Mestrado defendida e aprovada em 17 de dezembro de 1996 pela Banca Examinadora composta pelos Professores Doutores

Profº. Drº. Alcides Calsavara

Profº Drº. Edmundo R. M. Madeira

Profº. Drº. Luiz Eduardo Buzato

A meus pais, Clarice e Hitler.

Resumo

Neste trabalho desenvolve-se um *sistema de ações atômicas distribuídas* com o propósito de fornecer os mecanismos necessários à implementação do *sistema de execução* de Xchart. Xchart é um ambiente integrado para a construção de interfaces homem-computador de sistemas interativos distribuídos. O sistema de execução de Xchart é o componente responsável pela gerência dos objetos que implementam o controle do *diálogo* entre homem e computador. Este sistema requer a construção de um componente de *software* que forneça mecanismos de atomicidade, acesso, localização, comunicação e controle de concorrência para os objetos do diálogo. Nesta dissertação, defende-se que o sistema de ações atômicas distribuídas contruído é adequado a Xchart porque satisfaz as suas necessidades.

O sistema de ações atômicas distribuídas fundamenta-se no emprego de objetos e ações atômicas para a construção de aplicações distribuídas, e em sua implementação utilizam-se os conceitos da programação orientada a objetos. O conjunto de classes que compõem este sistema implementa as funcionalidades necessárias ao processamento de ações atômicas distribuídas, fornecendo uma interface de programação transparente à distribuição de objetos. O emprego do mecanismo de herança do paradigma de objetos permite que o comportamento das classes do sistema de ações atômicas distribuídas seja selecionado de acordo com os requisitos específicos de cada aplicação distribuída, tornando-o flexível e fácil de usar.

Abstract

A *distributed atomic actions system* has been developed to provide the mechanisms required for the implementation of the Xchart *execution system.* Xchart is an integrated environment that has been created to aid in the building of computer-human interfaces for distributed systems. The Xchart *execution system* is responsible for the management of the objects that implement the control of the *dialog* between human and computer. Distributed object management requires the provision of mechanisms for atomicity, access, location, communication, and concurrency control. These mechanisms guarantee that the execution system will be capable of managing the objects of the dialog. This dissertation argues that the distributed atomic action system developed fulfills the needs of the execution system of Xchart.

The distributed atomic actions system is based on the use of objects and atomic actions for the building of distributed applications (dialog controllers). Its design and implementation is object-oriented. The class hierarchy of the system implements the funcionality necessary to provide distributed atomic actions and distribution transparency through the use of a stub generator. Selective use of inheritance allows programmers to flexibly assign properties of atomic actions (concurrency control and atomicity) to classes of their applications.

Agradecimentos

Ao professor Luiz E. Buzato, pela inestimável ajuda na realização deste trabalho. Agradeço, sinceramente, por ter indicado os caminhos a seguir, como também pela compreensão e generosidade em relação ao tempo.

Ao amigo Edilmar, pela amizade, companheirismo e incentivo durante todos os momentos.

Ao amigo Fábio Lucena, pela constante prontidão em solucinar minhas dúvias a respeito do projeto Xchart.

Aos professores Hans Liesenberg e Ricardo Anido, pela colaboração e atenção prestadas no início deste trabalho.

Aos professores, funcionários e colegas do IC, pela convivência agradável durante os anos que passei aqui. Em especial, ao saudoso amigo Luiz Claudio Rosa que resolvia todas as burocracias com dinamismo e simpatia.

Ao Conselho Nacional de Pesquisa (CNPq), processo nº 131771/94-9, e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 94/04177-0, pelo apoio financeiro.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O sucesso de programas interativos amigáveis tem confirmado a importância de se dedicar atenção especial à construção de interfaces entre o homem e o computador. A interface homem-computador, ou simplesmente a *interface*, é a porção de um sistema interativo responsável pela coordenação do diálogo entre o homem e o computador, traduzindo as acões do homem em ativações das funções do sistema e permitindo a visualização dos resultados produzidos pela computação destas funções. Embora não existam regras que garantam a qualidade de uma interface, programas que são complicados ou confusos de utilizar, ou que oferecem poucas possibilidades de entrada de dados e de visualização de resultados são, sob o ponto de vista do usuário, muito menos amigáveis e interessantes. Ao contrário, os sistemas que dispõem de recursos adequados de interação, principalmente interfaces gráficas, são mais atrativos porque proporcionam maior satisfação e facilidade de uso, resultando, muitas vezes, em um aumento da produtividade do usuário. Sendo assim, grande parte do custo de desenvolvimento de programas interativos amigáveis corresponde ao custo do projeto e implementação das interfaces homem-computador desses programas [38].

A interface de um sistema interativo pode ser decomposta em *apresentação* e *controle de diálogo* [48]. A *apresentação* é o componente que define a aparência de uma interface e é responsável pela captura de eventos gerados por um usuário, através de dispositivos de entrada, e apresentação dos resultados produzidos pela aplicação. O *controle de diálogo*, ou simplesmente diálogo, é o componente de uma interface que define o protocolo de comunicação entre a apresentação e a *aplicação*.

Como fruto de pesquisas que buscam soluções para reduzir os custos de desenvolvimento e melhorar a qualidade das interfaces, vários conceitos, métodos e técnicas têm sido empregados em ferramentas que auxiliam na especificação, projeto, construção de protótipos, implementação, execução e manutenção de interfaces [38]. De acordo com Morse [44], nos últimos anos houve um progresso significativo nas ferramentas que possuem recursos de apoio à criação da *apresentação* de interfaces. Exemplos de ferramentas de apoio ao desenvolvimento de apresentações são *toolkits* tais como Tcl/Tk [52], Interviews [36], Fresco [12, 14], etc. No entanto, o mesmo não ocorreu em relação ao *controle de diálogo*. O número de ferramentas que provêem recursos para a descrição de diálogos é reduzido [40]. Em algumas

dessas ferramentas verifica-se o uso de uma notação formal para a especificação do componente de diálogo de interfaces, como por exemplo, diagramas de transição de estados e, mais recentemente, Statecharts [30]. Um exemplo de ferramenta é o Statemate [28], que pode ser utilizado para a especificação de diálogos, apesar de ter sido desenvolvido para a especificação de sistemas reativos em geral. Contudo, essas notações possuem algumas limitações que restringem seu emprego na descrição de diálogos caracterizados, principalmente, pela concorrência de operações e por apresentarem funcionalidade dependente do contexto de execução do sistema interativo [40]. Alia-se a esse fato a necessidade de criação de interfaces que traduzem aspectos comportamentais de aplicações mais complexas, tais como de sistemas de bolsas de valores, de editoração eletrônica, etc. Muitos desses sistemas são intrinsecamente distribuídos e compostos por aplicações concorrentes (paralelas). O desenvolvimento de interfaces amigáveis para sistemas desse porte tem estimulado a investigação de novas linguagens para a especificação de diálogos [46]. Idealmente, o uso de uma linguagem para a especificação de diálogos deve ser apoiado por ferramentas que integrem essa linguagem, de forma vantajosa, ao processo de desenvolvimento de sistemas interativos.

Com o propósito de desenvolver uma ferramenta adequada ao projeto e implementação de interfaces como as dos sistemas interativos descritos acima, está sendo construído um ambiente integrado de desenvolvimento de interfaces denominado **Xchart** [41]. O ambiente possui recursos que apóiam a criação tanto da *apresentação*, como do *diálogo* de uma interface. A partir das especificações da apresentação e do diálogo, **Xchart** produz automaticamente o código que implementa esses componentes, reduzindo a complexidade de implementação da interface. A especificação dos diálogos é feita utilizando-se uma linguagem gráfica formal especialmente desenvolvida para esse ambiente, também denominada *Xchart*. Fundamentada nos conceitos de estados, eventos, transições e ações, *Xchart* possui mecanismos próprios que permitem a descrição de diálogos através da representação, em um diagrama, da hierarquia, comunicação e concorrência entre estados destes diálogos. A linguagem *Xchart* também fornece recursos que permitem especificar diálogos de interfaces de aplicações distribuídas. Os diálogos deste tipo são decompostos em sub-diálogos que representam unidades de distribuição. De acordo com a notação de *Xchart*, cada sub-diálogo é um Xchart[1].

A funcionalidade de um diálogo é implementada por um conjunto de objetos cooperantes, produzidos pelo ambiente **Xchart**. *A integração entre esses objetos é coordenada por um sistema de execução que mantém a configuração do diálogo durante sua execução, reagindo à ocorrência de eventos de acordo com a semântica da linguagem* Xchart. *Em uma configuração distribuída do diálogo, tais objetos estão espalhados entre as máquinas, ou nós, do ambiente distribuído. A distribuição dos objetos acrescenta uma série de problemas à implementação do sistema de execução. Alguns desses problemas, tais como o acesso e a localização dos objetos, a comunicação entre os objetos e o controle da concorrência de*

[1] As notações **Xchart**, *Xchart* e Xchart são utilizadas no texto para designar, respectivamente, o ambiente, a linguagem e um sub-diálogo (ou diagrama) Xchart.

*execução de operações dos objetos, decorrem diretamente da distribuição dos objetos. Além disso, configurações distribuídas do sistema interativo estão sujeitas à ocorrência de falhas parciais de comunicação (corrupção, troca e perda de mensagens) e/ou de processadores (*fail-silent*). Portanto, a principal motivação para o desenvolvimento deste trabalho é baseada na necessidade de construção de um conjunto de mecanismos computacionais que solucionem adequadamente os problemas decorrentes da concorrência e da distribuição.*

1.1 Objetivos

A fim de fornecer o suporte adequado à construção do sistema de execução distribuído do ambiente **Xchart**, faz-se necessário um conjunto de mecanismos que implementem funções para a identificação, localização e acesso a objetos, comunicação entre objetos, prevenção do aparecimento de inconsistências decorrentes de acessos concorrentes, e detecção de falhas que possam ocorrer em conseqüência da distribuição dos objetos. Para simplificar os esforços de programação do sistema de execução, é conveniente que os mecanismos tornem a distribuição dos objetos o mais *transparente* possível, de modo que os programadores não tenham que se preocupar com as dificuldades impostas pela distribuição. A transparência é provavelmente a característica mais importante a ser considerada quando se projeta um sistema de computação distribuído, pois fornece aos usuários a ilusão de que estejam trabalhando em um ambiente de computação centralizado [43].

Várias pesquisas na área de sistemas distribuídos têm produzido uma série de modelos computacionais e técnicas alternativas para a solução dos diferentes problemas inerentes ao desenvolvimento de uma aplicação distribuída (Capítulo 2). A dificuldade de implementação da aplicação é inversamente proporcional ao nível de transparência oferecido por cada uma das alternativas. Quanto menor o envolvimento do programador com as questões impostas pela distribuição, como por exemplo o tipo de protocolo utilizado para a comunicação entre objetos distribuídos, maior é a transparência oferecida. A solução adotada nesse trabalho para fornecer os mecanismos que tratam da distribuição é obtida a partir da integração de alguns desses modelos e técnicas.

No modelo *cliente-servidor* de computação, uma aplicação é organizada como um conjunto de objetos cooperantes, onde objetos *servidores* realizam tarefas requisitadas por objetos *clientes*. O protocolo de comunicação é simples: um cliente envia uma mensagem ao servidor, possivelmente remoto, requisitando a realização de uma determinada tarefa. Posteriormente, o servidor executa a tarefa e envia os resultados ao cliente. Esse modelo oferece uma abstração que simplifica a utilização dos protocolos envolvidos na comunicação entre os objetos [64]. De fato, essa vantagem tem sido comprovada pelo grande número de sistemas distribuídos que adotam o modelo cliente-servidor de computação.

Apesar de o modelo cliente-servidor constituir-se numa boa alternativa para a construção de aplicações distribuídas, ele permite apenas a comunicação transparente entre objetos. No entanto, em aplicações desse tipo é bastante comum a realização de acessos e atualizações

concorrentes aos objetos, podendo comprometer a integridade dos objetos. Além da concorrência de operações, falhas parciais de componentes da aplicação também podem causar inconsistências nos objetos. Neste contexto, o conceito de *ações atômicas* tem emergido como uma das principais técnicas para o controle do compartilhamento de objetos em aplicações distribuídas tolerantes a falhas[20].

Uma ação atômica é um conjunto de operações que sempre termina com um resultado bem definido — sucesso ou falha — e satisfaz as propriedades de atomicidade, consistência, isolamento e persistência de resultados. Resumidamente, a *atomicidade* assegura que cada ação atômica é executada por completo, produzindo os resultados esperados, ou, se isso não for possível, nenhuma de suas operações é executada e nenhum resultado é obtido. A *consistência* garante a integridade dos objetos envolvidos na ação atômica. A propriedade de *isolamento* assegura que, apesar das ações atômicas serem executadas concorrentemente, os resultados serão obtidos como se as ações atômicas tivessem sido executadas em alguma ordem serial. A *persistência* garante que os resultados produzidos por uma ação atômica são armazenados em memória estável, de maneira que possam ser recuperados após a ocorrência de uma falha parcial.

O objetivo deste trabalho é desenvolver um sistema integrado de programação e execução de ações atômicas distribuídas. O projeto do conjunto de mecanismos que garantem as propriedades de ações atômicas deste sistema é baseado no modelo de programação orientada a objetos. Aplicando-se os conceitos da programação orientada a objetos na construção do sistema de ações atômicas, as funções de gerenciamento da concorrência, recuperação e comunicação dos objetos são transferidas dos "processos gerentes" tradicionalmente utilizados em sistemas de bases de dados, onde o conceito de ações atômicas foi inicialmente aplicado, para os próprios objetos, tornando-os autônomos. Esta é uma vantagem que permite aplicar seletivamente as propriedades de ações atômicas, possibilitando funcionalidades particulares a classes de acordo com as necessidades específicas de aplicações construídas com base neste sistema.

O propósito da construção do sistema de ações atômicas é fornecer os mecanismos adequados à implementação do sistema de execução distribuído do ambiente **Xchart**, permitindo a criação de interfaces de sistemas interativos distribuídos. O sistema de ações atômicas é composto de uma coleção de subsistemas que realizam os serviços de identificação, localização, armazenamento e comunicação entre objetos, necessários à semântica do processamento de ações atômicas; mais um conjunto de classes que fornecem as propriedades de ações atômicas para a construção de aplicações distribuídas baseadas nos conceitos de objeto e ações. Através do mecanismo de herança, tais classes fornecem uma interface de programação transparente à distribuição dos objetos. Por exemplo, o acesso a objetos construídos a partir destas classes é uniforme, independentemente de suas localizações.

O sistema de ações atômicas é baseado no modelo cliente-servidor, e a comunicação entre os objetos é realizada através de chamadas de procedimentos remotos[2] [7]. Durante a

[2]Em Inglês: *Remote Procedure Call.*

execução de chamadas de procedimentos remotos, as mensagens entre clientes e servidores são interceptadas por procedimentos de apoio (*stubs*) que efetuam o empacotamento e passagem de parâmetros, início e término de execução dos servidores. O sistema de ações atômicas também possui um *gerador de stubs* que, a partir da definição de classes, produz automaticamente os *stubs* para o cliente e para o servidor.

1.2 Contribuições

As contribuições deste trabalho são:

- O desenvolvimento de um *sistema de ações atômicas distribuídas* que forneça mecanismos adequados para execução de operações atômicas e para acesso, localização, comunicação e controle de concorrência de objetos distribuídos. Este sistema é baseado no modelo computacional de ações atômicas e objetos, porque

  - ações atômicas garantem a atomicidade, consistência, isolamento e persistência dos resultados de um conjunto de operações, propriedades fundamentais em um ambiente distribuído, e

  - o emprego dos conceitos do paradigma de objetos permite a construção de programas modulares e flexíveis, a partir de modelos de um universo constituído de objetos que interagem entre si e, principalmente porque, através do mecanismo de herança, a utilização das propriedades de ações atômicas distribuídas oferecidas pelo sistema pode ser feita de forma *seletiva* pelos programadores, de acordo com as necessidades específicas de suas aplicações.

- A integração do sistema de ações atômicas distribuídas com o ambiente **Xchart**. Embora o sistema de ações atômicas distribuídas possa ser utilizado genericamente por aplicações distribuídas não tolerantes a falhas, seu projeto e implementação foram fundamentados em um conjunto de requisitos elaborados a partir dos problemas advindos da interpretação de diálogos em **Xchart** (Seção 4.1). Estes problemas — atomicidade de operações, acesso, localização, comunicação e controle de concorrência — são decorrentes, basicamente, da distribuição dos objetos dos diálogos. Nesse sentido, o sistema proposto fornece um componente de *software* que é útil à construção do sistema de execução de **Xchart**.

1.3 Organização do Texto

No Capítulo 2 apresentam-se conceitos fundamentais sobre sistemas distribuídos, orientação a objetos e ações atômicas com o objetivo de tornar o texto auto-contido, facilitando a

compreensão dos próximos capítulos. Também são descritos os principais mecanismos necessários à implementação de um sistema de ações atômicas e algumas técnicas e algoritmos utilizados na implementação destes mecanismos.

No Capítulo 3, descreve-se em detalhes o ambiente **Xchart**. Apresenta-se inicialmente a arquitetura de *software* do ambiente através da definição da estrutura e da funcionalidade de cada um de seus componentes, a fim de esclarecer como o processo de desenvolvimento de interfaces é apoiado por **Xchart**. A seguir, o sistema de execução da linguagem *Xchart* é então comentado, evidenciando-se a necessidade e a aplicação do sistema de ações atômicas desenvolvido neste trabalho.

No Capítulo 4 descrevem-se o projeto e a implementação do sistema de ações atômicas distribuídas desenvolvido para **Xchart**, incluindo a descrição de todos os seus componentes.

No Capítulo 5 há alguns exemplos de utilização do sistema de ações atômicas distribuídas. A partir de fragmentos de código que os implementam, demonstra-se o uso do sistema de ações atômicas pelo ambiente **Xchart** e por aplicações distribuídas.

O Capítulo 6 contém descrições concisas de alguns sistemas que implementam ações atômicas e que, de alguma forma, influenciaram nas decisões deste projeto. Faz-se, também, uma comparação entre os sistemas descritos.

Finalmente, no Capítulo 7 apresentam-se os resultados obtidos com o desenvolvimento deste trabalho e algumas sugestões de possíveis melhorias e extensões.

Capítulo 2

Fundamentos

Neste Capítulo, são definidos e analisados os principais conceitos associados à construção de sistemas distribuídos, com o objetivo de tornar a dissertação um trabalho quase auto-contido e didático.

2.1 Sistemas Distribuídos

A crescente evolução tecnológica dos sistemas de computação, com a criação de processadores e de meios de comunicação cada vez mais potentes e com melhor relação custo/benefício, tem propiciado uma mudança nos ambientes de processamento de dados. Os sistemas centralizados baseados em *mainframes* vem sendo substituídos por sistemas distribuídos, onde vários computadores autônomos interligados por uma rede de comunicação cooperam na execução de tarefas através da troca de mensagens [3]. A principal característica de um sistema distribuído é a oferta para seus usuários de uma interface homem-computador e de programação que se comporta como a de um sistema centralizado, muito embora o processamento dos serviços seja executado em uma coleção de computadores interligados. Atualmente, o progresso dos sistemas distribuídos é tal que permite até mesmo a interconexão de máquinas de arquiteturas diferentes com relativa facilidade.

A construção de sistemas distribuídos envolve aspectos não só dos meios físicos de comunicação e da interconexão das máquinas. No que diz respeito ao *software*, a programação de sistemas distribuídos é mais complexa que a de sistemas centralizados. Os benefícios resultantes da distribuição, como um aumento na disponibilidade de recursos computacionais, na confiabilidade e no desempenho do sistema, resultam no aumento da complexidade de implementação [65]. Na ausência de mecanismos para o gerenciamento confiável do compartilhamento dos dados e da distribuição de código e de dados, bem como para o tratamento de falhas, tais tarefas se constituem em trabalho a ser realizado pelos programadores e é fundamental para alcançar a confiabilidade. Muitas vezes, esta é uma das principais dificuldades encontradas na construção dessas aplicações.

- **Compartilhamento de dados.** Com a utilização de computadores em uma variedade de aplicações, tem se tornado comum a necessidade de compartilhamento de dados, ao invés de pertencerem exclusivamente a uma aplicação. A consistência de dados compartilhados é um requisito primordial de aplicações confiáveis. Em aplicações distribuídas, bem como em centralizadas, os acessos concorrentes a dados compartilhados devem ser controlados para evitar que tais dados se tornem inconsistentes. O controle da concorrência de acessos a dados compartilhados adiciona complexidades à programação, além de introduzir uma fonte potencial de erros na programação. Portanto, os programadores necessitam de uma maneira sistemática para controlar interações entre usos concorrentes de dados compartilhados.

- **Distribuição de dados e de código.** Uma aplicação distribuída é composta de um conjunto de componentes (código e dados) espalhados pelas máquinas que constituem a rede de um sistema distribuído. Na programação de aplicações deste tipo não há somente os problemas de localização e acesso a componentes remotos, há também a possibilidade de ocorrência de falhas quando o acesso a tais componentes envolver a comunicação entre computadores. Sendo assim, os programadores necessitam de uma maneira confiável de localização e acesso a componentes remotos.

- **Ocorrência de falhas.** Aplicações distribuídas são constituídas de muitos componentes que podem falhar, podendo levar a uma redução da confiabilidade de tais aplicações. Uma maneira de ampliar a confiabilidade de uma aplicação distribuída é, por exemplo, através da especificação de mecanismos especiais para a *prevenção* da ocorrência de falhas. No entanto, em alguns casos isto não é o suficiente pois nem todas as falhas podem ser previstas. Uma alternativa é fornecer mecanismos para a *detecção* de falhas ocorridas. O principal problema, neste caso, é a impossibilidade de recuperação das aplicações a um estado consistente. Isto só é possível através de mecanismos de *tolerância* a falhas, os quais permitem o armazenamento das informações necessárias à recuperação dos estados das aplicações como medida de precaução. Para facilitar a recuperação de uma aplicação após a ocorrência de uma falha, o programador necessita de mecanismos que permitam definir quais informações são importantes, e que portanto, devem ser armazenadas, e como elas devem ser recuperadas após a ocorrência de uma falha.

Várias pesquisas na área de sistemas de gerência de bases de dados, sistemas distribuídos e linguagens de programação têm produzido uma variedade de técnicas e ferramentas para satisfazer os quesitos comentados anteriormente. A seguir, apresentam-se algumas destas técnicas.

2.1.1 Transparência

Idealmente, a programação e uso de um sistema distribuído deveriam ser idênticos à programação de um sistema centralizado. Entretanto, nem sempre é possível atingir esta identidade de abstrações de programação. Por exemplo, em um sistema centralizado, a chamada de um procedimento ocorre sem que o programador tenha que se preocupar com a localização do procedimento. Em um sistema distribuído, o programador precisa primeiro localizar o procedimento para depois invocá-lo. Assim, diz-se que um sistema distribuído é *transparente* com relação a alguma abstração de programação, se ele implementa mecanismos que o tornam similar a um sistema centralizado para esta abstração. As principais transparências implementadas por sistemas distribuídos são comentadas a seguir [35].

A *transparência de acesso* está relacionada com mecanismos de chamada de procedimento de linguagens de programação. Mecanismos de transparência de acesso mascaram as diferenças dos componentes do sistema distribuído, permitindo que o acesso a seus objetos seja feito de maneira uniforme com respeito às representações de dados e às interações entre componentes. O mecanismo de chamada de procedimento remoto (Seção 2.1.2) tem sido bastante utilizado para dar suporte à transparência de acesso em linguagens de programação de sistemas distribuídos. Geralmente, a transparência de acesso é obtida através do uso de um pré-processador de chamadas de procedimentos remotos (gerador de *stubs*) durante o desenvolvimento de programas distribuídos.

A *transparência de localização* permite o acesso a um objeto sem a necessidade de se conhecer sua localização física. Em sistemas que fornecem este tipo de transparência, o acesso a um objeto pode ser feito a partir, unicamente, de sua identificação. A localização do objeto pode, de alguma maneira, ser obtida a partir de sua identificação através, por exemplo, de servidores de nome e de localização.

A *transparência de migração* permite a movimentação de objetos pelas máquinas de um ambiente distribuído, sem afetar seus comportamentos. A movimentação de objetos é utilizada para melhorar o desempenho, a segurança, a confiabilidade e a disponibilidade de sistemas distribuídos. A implementação de mecanismos de transparência de migração requer transparências de acesso e de localização. Se as localizações dos objetos não forem especificadas em um programa, e a sintaxe e a semântica de operações locais e remotas são idênticas, então os objetos podem ser movidos de forma transparente de um nó para outro.

A *transparência de concorrência* permite o acesso concorrente a objetos compartilhados, garantindo a consistência dos mesmos.

A *transparência de replicação*[1] permite a criação de réplicas de objetos, sem a necessidade de se conhecer como estas réplicas são criadas e mantidas, ou quais objetos são replicados. A replicação de objetos é utilizada para se aumentar a confiabilidade e disponibilidade destes, ou até mesmo a tolerância a falhas de todo um sistema. Nos mecanismos de transparência de replicação, os objetos são gerenciados por protocolos de consistência de réplica apropri-

[1] O termo replicação é utilizado como um sinônimo de duplicação (cópia).

ados, que garantem a consistência das cópias. Em sistemas transparentes à replicação, o programador especifica o nível de disponibilidade e confiabilidade para um objeto, deixando as tarefas de criação e gerência das réplicas ao sistema distribuído. Tal como a transparência de migração, esta transparência requer transparências de acesso e de localização.

A *transparência a falhas* oculta, quando possível, a ocorrência de falhas no sistema. Mecanismos transparentes a falhas são particularmente importantes em sistemas distribuídos, nos quais as possibilidades de ocorrência de falhas são significativamente ampliadas em relação a um sistema centralizado. Quando uma falha ocorre em um ambiente transparente a falhas, devem ser acionados mecanismos providos pelo sistema para recuperar, de forma transparente, os estados dos objetos afetados.

2.1.2 Comunicação

Devido a ausência de memória compartilhada em sistemas distribuídos, a comunicação entre objetos se dá através da troca de mensagens transportadas por sistemas de comunicação.

O Modelo Cliente-Servidor

No modelo *cliente-servidor* de comunicação, um sistema distribuído é organizado como um conjunto de objetos cooperantes. Alguns dos objetos realizam a função de fornecedores de serviços, os *servidores*, enquanto outros utilizam-se destes serviços, os *clientes*. Uma mesma máquina pode conter clientes, servidores, ou ambos.

O modelo cliente-servidor é baseado no protocolo de troca de mensagens *pedido/resposta* [66]. O cliente envia uma mensagem ao servidor requisitando alguma tarefa. O servidor realiza tal tarefa e envia os resultados ao cliente, ou um código de erro indicando porque a tarefa não pôde ser realizada. A comunicação entre clientes e servidores pode ser síncrona ou assíncrona. O estilo mais comum de interação cliente-servidor, no entanto, é a comunicação síncrona, a qual envolve o bloqueio do cliente até que o servidor complete sua tarefa e envie a resposta. A comunicação síncrona é mais comumente utilizada porque fornece um nível de abstração análoga àquela proporcionada pela execução de tarefas nas linguagens de programação convencionais, tipicamente baseadas em mecanismos de *chamadas de procedimentos*.

Chamada de Procedimento Remoto

Em 1984, Birrel e Nelson [7] propuseram uma solução para o problema da comunicação síncrona em sistemas distribuídos, com a introdução do mecanismo de *Chamada de Procedimento Remoto*. Em resumo, o que Birrel e Nelson sugeriram foi permitir que programas chamassem procedimentos localizados em outras máquinas. Quando um processo em uma máquina A chama um procedimento em uma máquina B, o processo da máquina A é suspenso, e a execução do procedimento na máquina B é iniciada. Informações podem ser

transportadas, na forma de parâmetros, do processo que faz a chamada para os processos chamados, e podem retornar como resultados do procedimento.

Embora a idéia básica se pareça simples e elegante, existem alguns problemas. Inicialmente, como os procedimentos, tanto o que faz a chamada como o que a recebe, são executados em máquinas diferentes, seus espaços de endereçamento também são diferentes. Parâmetros e resultados devem ser transportados, o que pode ser complicado, principalmente se as arquiteturas de *hardware* das máquinas não forem idênticas. Finalmente, ambas as máquinas podem falhar, e cada uma das falhas possíveis podem causar problemas diferentes. Ainda assim, a maioria destes problemas podem ser resolvidos e chamadas de procedimentos remotos são amplamente utilizadas em sistemas distribuídos.

Quando é feita uma chamada de procedimento remoto (Figura 2.1), uma versão diferente do procedimento remoto é chamada, o procedimento *stub* do cliente, que empacota os parâmetros em uma mensagem e a envia ao servidor através da rede de comunicação. Após o envio da mensagem, o cliente fica bloqueado aguardando a resposta. Quando a mensagem chega no servidor, ela é passada ao *stub* do servidor, que é ligado ao servidor. O *stub* do servidor desempacota os parâmetros da mensagem e então realiza uma chamada local ao procedimento no servidor. Para o servidor tudo se dá como se o cliente fosse local. O servidor realiza seu trabalho e retorna os resultados ao seu chamador de maneira usual. O *stub* servidor obtém de volta o controle quando a chamada é completada pelo término do procedimento. Então, empacota os resultados em uma mensagem e a envia de volta ao cliente. Quando a mensagem chega no cliente, o *stub* cliente é desbloqueado, inspeciona a mensagem, desempacota o resultado e o devolve ao processo chamador do procedimento. Quando o processo que fez a chamada recebe o resultado, não tem conhecimento de que o trabalho foi realizado remotamente e não localmente.

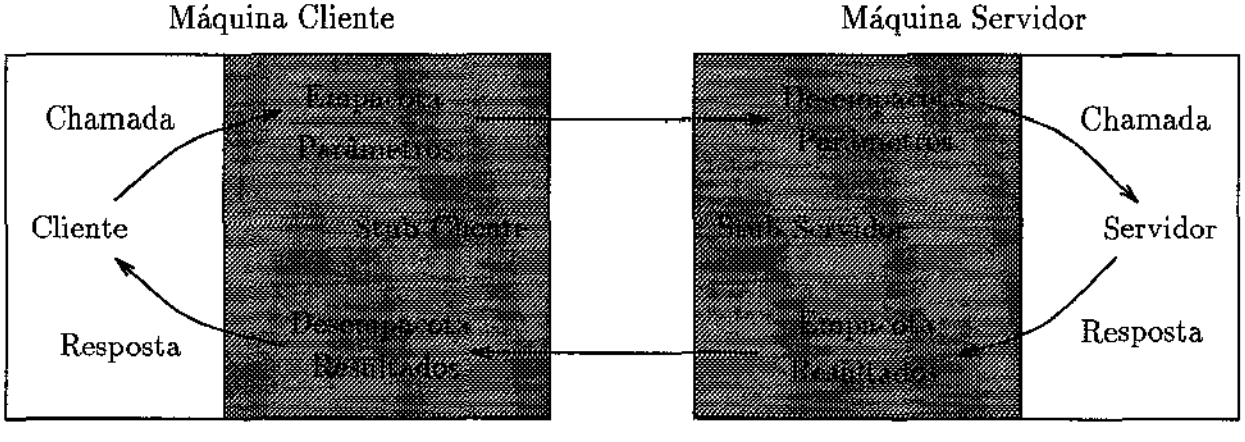

Figura 2.1: Chamadas e mensagens em uma chamada de procedimento remoto.

Em chamadas de procedimentos remotos, procedimentos remotos são chamados tal como se fossem locais, e sem a utilização direta de primitivas como `send` e `receive`. Todos os

detalhes de trocas de mensagens são "escondidos" pelos *stubs*. Resumindo, a função de todo o mecanismo de chamada de procedimento remoto é converter a chamada local feita pelo procedimento cliente em uma chamada local no procedimento servidor, sem que cliente e servidor tenham conhecimento dos passos intermediários.

Em ambientes de programação para computação distribuída, os procedimentos *stubs* são geralmente produzidos automaticamente por um *gerador de stubs*, que traduz a descrição de interfaces entre clientes e servidores em código dos *stubs* de cliente e servidor. Uma *interface* contém uma lista de assinaturas de procedimentos, isto é, seus nomes e tipos de parâmetros.

2.1.3 Interfaces de Programação

O desenvolvimento de aplicações distribuídas pode se tornar uma tarefa extremamente complexa se os programadores não dispuserem de uma linguagem de programação que integre, adequadamente, um conjunto de abstrações que tornem "transparentes" os vários mecanismos para controle do compartilhamento e da distribuição dos dados. Ou seja, os programadores devem ter acesso a uma linguagem com interface de programação que permita acesso uniforme aos dados, sejam eles privados ou compartilhados, locais ou remotos. Nesse sentido, para se obter uma total uniformidade, também é necessário que o programador não tenha que lidar com questões que envolvam o controle dos efeitos de falhas decorrentes do conflito de acessos concorrentes e da distribuição dos dados. Uma boa interface de programação deve fornecer uso flexível, extensível e seletivo dos mecanismos de transparência, e esconder do programador os detalhes de como são resolvidos os problemas de compartilhamento, distribuição e falhas de um sistema distribuído.

Pesquisas em linguagens de programação para sistemas distribuídos tem resultado na criação de uma diversidade de linguagens de programação [3]. Algumas são linguagens inteiramente construídas a este propósito, como Ada [10] e Emerald [8, 56], outras, são extensões de linguagens já existentes para a programação de sistemas não distribuídos, através de pré-processamento ou de mudanças no compilador. Exemplos destas últimas são: Concurrent C [25] e Distributed Smalltalk [4].

Muitas destas linguagens empregam o paradigma de orientação a objetos, que possibilita a construção de linguagens com interfaces flexíveis e com relativa facilidade de extensão.

2.2 Programação Orientada a Objetos

A orientação a objetos, segundo Khoshafian [31], é a "tecnologia de modelagem e desenvolvimento de sistemas que facilita a construção de programas complexos a partir de componentes individuais," chamados *objetos*. Através do emprego de ferramentas e conceitos que permitem modelar o mundo real tão próximo possível da visão que o ser humano tem desse mundo, a orientação a objetos encoraja o reaproveitamento e a extensão de código existente, com redução dos esforços de implementação e manutenção. De fato, o emprego de uma linguagem

de programação orientada a objetos permite a criação de sistemas que são mais facilmente compreendidos e compartilhados com outras pessoas. O conceito de objetos foi primeiramente empregado em Simula67 [16, 47], a primeira linguagem orientada a objetos, em 1967. No entanto, a programação orientada a objetos se disseminou somente após o aparecimento de Smaltalk [26], em 1980. Desde então, a programação orientada a objetos tem se difundido largamente em meios comerciais e acadêmicos.

2.2.1 O que são Objetos?

Um *objeto* é definido como um conjunto de dados que representa o *estado* interno de uma entidade concreta ou abstrata, cujos valores podem ser modificados ao longo da atividade do objeto, mais um conjunto de operações pré-definidas que acessam tais dados e respondem pelo *comportamento* do objeto em relação a *mensagens*. De acordo com a terminologia da programação orientada a objetos, os dados são representados por *variáveis de instância*, e as operações são denominadas *métodos*.

O estado de um objeto é completamente protegido, isto é, só pode ser consultado ou alterado através de seus métodos. A invocação de métodos se dá pelo envio de uma *mensagem* ao objeto. O conjunto de mensagens às quais um objeto responde estabelece a sua *interface*.

Uma mensagem contém a identificação do método a ser ativado (invocado) e carrega os parâmetros deste método. A recepção de uma mensagem válida faz com que o objeto receptor execute o método pedido e devolva os resultados para o objeto que originou a mensagem.

Em um programa orientado a objetos, a computação é definida em função da execução dos métodos correspondentes às mensagens enviadas aos objetos (acoplamento mensagem-método). A seguir, apresentam-se os principais conceitos de orientação a objetos.

2.2.2 Tipos Abstratos de Dados

A *abstração* é um dos mecanismos mais importantes utilizados pela mente humana para compreender o mundo real. Ao analisar um problema complexo, o homem naturalmente separa objetos que, na realidade, nunca se encontram isolados, caracterizando-os a partir de certas propriedades que definem aproximadamente seus principais aspectos estruturais e comportamentais. Baseando-se na observação das características relevantes para sua análise, o homem estabelece semelhanças e diferenças entre objetos, atributos e interações presentes no problema, para que esses elementos possam ser descritos, analisados e entendidos.

A fomalização de abstrações resultantes do estudo de um problema para a identificação de seus componentes e processos é feita, pelos programadores, utilizando-se ferramentas formais, como por exemplo, linguagens de programação. O paradigma de objetos busca, através de mecanismos como abstração de dados e herança, aproximar as entidades do mundo real das abstrações oferecidas por uma linguagem de programação.

Idealmente, tanto na análise quanto na construção de um sistema complexo, o comportamento de um componente do sistema não deve depender da maneira como os demais com-

ponentes são estruturados internamente. Tais componentes são descritos, em programação orientada a objetos, por tipos abstratos de dados, uma unidade de encapsulamento que "esconde" as informações sobre a estrutura de um objeto de observadores externos. Com isso, as abstrações de um sistema são definidas, na prática, por duas partes: uma *interface* que descreve quais operações podem ser executadas; e uma *implementação* que determina como as operações são executadas.

Classe é a construção de linguagem mais comumente utilizada para a implementação de tipos abstratos de dados em linguagens de programação orientada a objetos. Uma classe é uma definição de tipo abstrato que descreve as propriedades do estado interno e do comportamento de um conjunto de objetos, através das especificações de variáveis de instância e métodos próprios. Objetos com a mesma estrutura e o mesmo comportamento pertencem a uma mesma classe e são denominados *intâncias da classe.*

2.2.3 Herança

Classes são organizadas hierarquicamente através do uso do mecanismo de *herança*, de tal forma que uma classe mais especializada herda todas as propriedades da classe mais geral a qual está subordinada na hierarquia. A clásse mais geral é denominada *classe base* e a classe mais especializada é denominada *classe derivada.* Uma classe base representa, em relação às suas classes derivadas, uma abstração de *generalização*, já que a classe base fatora semelhanças existentes entre as classes derivadas, definindo uma abstração mais geral. Inversamente, uma classe derivada é obtida por *especialização* de uma classe base (ou de várias classes bases, no caso de *herança múltipla*), na medida que representa uma abstração com estrutura e comportamento mais específicos. Quando a herança envolve apenas uma classe base, tem-se *herança simples.*

O mecanismos de herança permitem a reutilização de código. Através deste mecanismo, uma classe derivada herda a estrutura e comportamento de outras classes.

2.2.4 Polimorfismo

Polimorfismo é a capacidade que um método apresenta de aceitar muitas formas (tipos) de parâmetros, ou seja, um método é aplicável a instâncias de diferentes de classes. Por exemplo, o método "+" pode denotar soma de inteiros, soma de reais, união de conjuntos ou concatenação de cadeias de caracteres. O mecanismo utilizado para implementar polimorfismo é conhecido como *mecanismo de sobrecarga.* Durante o acoplamento de uma mensagem (chamada) a um método, uma assinatura compatível com a assinatura da mensagem é procurada em um conjunto de assinaturas. Este conjunto de assinaturas é determinado através do código do programa (acoplamento estático) ou em uma tabela de nomes de métodos (acoplamento dinâmico). O acoplamento estático ocorre durante a compilação e o acoplamento dinâmico durante a ligação/execução de um programa. A determinação do conjunto de assinaturas, a ser explorado em busca de um acoplamento, leva em conta o relacionamento de

herança entre classes. Assim, é possível que um método assuma *formas* diferentes ao longo da cadeia de derivação de tipos formada pela hierarquia de classes.

2.2.5 Identidade

Identidade é a propriedade de um objeto que o distingue de todos os demais objetos de um sistema. O conceito de identidade permite a criação de objetos complexos que são contruídos a partir de outros objetos ou de referências a outros objetos.

A implementação mais simples de identidade de objetos em linguagens de programação convencionais é o uso do endereço de memória onde o objeto é armazenado, como é o caso de C++. No entanto, este tipo de implementação de identidade não é adequada para sistemas distribuídos, pois os objetos estão espalhados entre computadores e, conseqüentemente, em espaços de endereçamento diferentes. Da mesma forma, para objetos persistentes[2], a utilização de um endereço como identidade não é adequada porque esse endereço perde o significado ao término da execução do programa. Uma alternativa, nesse caso, é identificar objetos através de *surrogates* [13, 33], identificadores únicos gerados automaticamente e completamente independentes do tipo, do estado e da localização do objeto em memória. A estratégia de implementação de *surrogates* acrescenta um nível de indireção na identificação de um objeto: dado um *surrogate*, deve-se encontrar o endereço de memória do objeto correspondente. Embora a utilização de *surrogates* seja considerada, segundo Khoshafian [32], "a mais poderosa técnica para implementação de identidade", estes identificadores globais únicos (cuja geração automática é tarefa não trivial em ambientes distribuídos) não são interessantes para utilização direta pelo programador. Ao invés disso, é mais conveniente identificar um objeto através da definição de um *nome* de alto nível, evidentemente mais significativo. Esta estratégia usa igualmente a indireção para identificação efetiva de um objeto. As duas técnicas, porém, não são excludentes: ao custo de uma indireção adicional, o programador pode identificar um objeto através de um nome que, posteriormente, será mapeado pelo sistema no *surrogate* correspondente e, finalmente, no endereço de memória do objeto.

2.3 Ação Atômica

O conceito de ação atômica, ou transação atômica, é originário de sistemas de gerência de base de dados e foi formalizado por Eswaran [24] e Lomet [37]. Uma ação atômica é um conjunto indivisível de operações que sempre termina com um resultado bem definido: sucesso ou falha. Ações atômicas possuem as seguintes propriedades [9, 27]:

- **Atomicidade.** Esta propriedade assegura que a computação de uma ação atômica é *finalizada* normalmente (*commit*) após a execução de todas as suas operações e que os

[2]Objetos persistentes são aqueles que sobrevivem à execução de um programa de aplicação, sendo armazenados em memória estável.

resultados esperados são produzidos, ou a computação das operações é *interrompida* (*abort*) e nenhum resultado é produzido, isto é, tudo se passa como se a ação nunca houvesse se iniciado.

- **Consistência.** Uma ação atômica deve ser um programa correto que realize transformações corretas de estados sobre objetos, não violando quaisquer restrições de integridade de estado.

- **Isolamento.** Esta propriedade assegura que, apesar das ações atômicas executarem concorrentemente, os resultados sempre serão como se fossem executadas seqüencialmente, em alguma ordem. Esta propriedade garante que a execução concorrente de ações atômicas não produzirá inconsistências.

- **Persistência.** Quando uma ação atômica é finalizada com sucesso, os resultados de suas operações devem ser armazenados em memória estável. Assim, em caso de falha, não se perderão os resultados da ação atômica.

As operações BEGIN_ATOMIC_ACTION e END_ATOMIC_ACTION são utilizadas para delimitar o corpo de uma ação atômica, isto é, a seqüência de operações realizadas entre o início e o final da ação é atômica. Se acaso uma ação atômica não puder ser completada em decorrência de algum tipo de falha durante seu processamento, pode-se desfazê-la através da operação ABORT_ATOMIC_ACTION.

2.3.1 Ações Atômicas Encadeadas

Ações atômicas encadeadas foram introduzidas por Moss [45], para aumentar a flexibilidade na estruturação de aplicações distribuídas. Como o próprio nome sugere, no modelo de ações atômicas encadeadas, uma ação atômica é estruturada por um conjunto de outras ações atômicas, chamadas de sub-ações. Sub-ações também podem ser decompostas em várias outras sub-ações, formando uma hierarquia. A estrutura da hierarquia de ações atômicas forma uma floresta, ou seja, um grafo com um número de nós raízes (representando as ações atômicas de nível superior), cada um dos quais com vários filhos (representando as sub-ações atômicas). Cada sub-ação pode realizar o *commit* ou o *abort*, no entanto, a operação de *commit* só terá efeito se todas as sub-ações ancestrais a ela na hierarquia também realizarem o *commit*. Conseqüentemente, qualquer sub-ação só realizará o *commit* se a ação atômica de nível superior (raiz) realizar o *commit*. Para que uma ação atômica de nível superior (raiz) possa realizar a operação de *commit*, todas as sub-ações abaixo na hierarquia (filhos) devem também realizar o *commit*. Se acaso uma das sub-ações realizar o *abort*, então a ação atômica de nível superior também podem realizar o *abort*. Neste caso, todas as sub-ações irão realizar o *abort*, mesmo aquelas que poderiam realizar um *commit*.

Ações atômicas encadeadas são úteis principalmente por duas razões:

- Elas permitem o aumento do nível de concorrência de operações em objetos compartilhados, uma vez que uma ação atômica pode executar várias sub-ações atômicas concorrentemente sem violar a propriedade de isolamento.

- Elas fornecem uma maneira de proteção contra falhas mais flexível, permitindo recuperação encadeada dentro de uma ação atômica. Quando uma sub-ação realizar o *abort*, seus ancestrais podem continuar sua execução e possivelmente até completar suas tarefas se executarem uma sub-ação alternativa no lugar da que abortou.

2.3.2 Controle de Concorrência

O texto contido nesta e nas próximas Seções deste Capítulo é baseado na obra de Bernstein [5]. Um dos algoritmos mais antigos e mais utilizados para resolver o problema de interferência de acesso a dados compartilhados é baseado no conceito de *locking*. De maneira simples, quando uma ação atômica precisa efetuar algum acesso a um objeto compartilhado, ela deve bloqueá-lo antes de realizar o acesso. Se duas ou mais ações atômicas concorrentes tentam fazer acesso ao objeto compartilhado, elas o farão de forma *intercalada*, isto é, operações de uma ação podem ser executadas entre duas operações de outra ação. Esta intercalação pode gerar inconsistências nos estados dos objetos envolvidos nas ações atômicas. O objetivo do *controle de concorrência* é evitar interferências entre operações de ações diferentes que *lêem* e *escrevem* (alteram) estados de objetos compartilhados.

As ações atômicas fazem acesso a objetos por meio de operações de *leitura* e de *escrita*. Pode-se diferenciar os *locks* em *locks de escrita* e *locks de leitura*. Se um *lock* de leitura é atribuído a um objeto, então são permitidos outros *locks* de leitura. Os *locks* de leitura são feitos para se assegurar que o objeto não seja alterado, permitindo que várias operações de leitura sejam feitas em um objeto concorrentemente. Em contrapartida, somente um *lock* de escrita pode ser obtido para um objeto de cada vez. Então, *locks* de leitura são compartilhados e *locks* de escrita são exclusivos.

A Tabela 2.1 define a compatibilidade entre operações de leitura e escrita, e indica que somente *locks* de leitura são compatíveis. Isto significa que se em uma ação atômica T1 efetua-se um *lock* de leitura num objeto O e em outra ação atômica T2 solicita-se um *lock* em O, então o *lock* será efetuado se, e somente se, for um pedido de *lock* de leitura. Se na ação atômica T1 efetuou-se um *lock* de escrita num objeto O, nenhum outro *lock* (de leitura ou de escrita) poderá ser efetuado neste objeto por outra ação atômica.

Two-Phase Locking

A atribuição e liberação de *locks* em objetos, antes e imediatamente depois da realização de operações, respectivamente, pode levar ao surgimento de inconsistências. A fim de evitar que isto ocorra, o controle de concorrência pode ser implementado através do uso de um protocolo chamado *two-phase locking*, onde uma ação atômica primeiro bloqueia todos os

Compatibilidade		Modo de *lock*	
		Leitura	Escrita
Modo do Pedido	Leitura	Sim	Não
	Escrita	Não	Não

Tabela 2.1: Tabela de compatibilidade de *locks*.

objetos a que faz acesso, atribuindo-os *locks* de acordo com as operações a serem realizadas sobre eles, e os libera somente em seu término. A fase de atribuição de *locks* é chamada de *growing phase* e a de liberação, de *shrinking phase*.

Deadlocks

O *two-phase locking* pode levar à ocorrência de *deadlocks*. Por exemplo, se uma ação atômica A obter um *lock* de escrita em um objeto X e uma outra ação B obter um *lock* de leitura em um objeto Y, supondo-se que o *lock* de leitura em Y não tenha sido liberado por B, A ficará bloqueada na tentativa de obter um *lock* de escrita em Y. Se B tentar obter um *lock* de leitura em X, enquanto A estiver bloqueado, ocorrerá um *deadlock*, ou seja, as ações A e B ficam presas na espera da liberação dos *locks* de Y e de X, respectivamente.

Deadlocks podem ser resolvidos utilizando-se técnicas que incluem *timeout* e grafos de dependência. No caso de *timeout*, quando se sabe com antecedência que um mesmo tipo de *lock* nunca será mantido por mais que T segundos sobre o mesmo objeto, e isso ocorrer, então deve haver um *deadlock* e a ação atômica é interrompida.

A outra técnica para se resolver *deadlocks* é detectá-los através de grafos de dependência que mantêm informação de quais ações atômicas possuem quais *locks* e espera por quais *locks*, e então verifica-se a existência de ciclos nestes grafos. Quando existir um ciclo, uma das ações atômicas pertencentes a ele é interrompida e o *deadlock* é desfeito. A detecção possui a desvantagem de exigir muito tempo de processamento, especialmente em sistemas distribuídos, onde a informação necessária à construção dos grafos está espalhada em várias máquinas.

Variações do *Two-Phase Locking*

É possível implementar-se o algoritmo de *two-phase locking* de tal forma que uma ação atômica nunca seja interrompida em decorrência de um *deadlock*. Tal implementação, chamada de *conservative two-phase locking*, requer que todos os *locks* de uma ação atômica sejam obtidos antes que qualquer outra operação seja realizada.

Uma outra implementação do *two-phase locking*, chamada *strict two-phase locking*, requer que todos os *locks* sejam liberados de uma só vez, quando a ação atômica termina, ou seja, depois que a ação atômica efetuar *commit* ou *abort*. Desta forma, garante-se que nenhuma

ação atômica pode realizar operações de escrita ou de leitura sobre objetos alterados por alguma outra ação atômica que ainda não realizou *commit* ou *abort*.

2.3.3 Atomicidade

Antes do *commit* de uma ação atômica, é preciso verificar se todos os objetos envolvidos na ação estejam aptos a realizá-la, a fim de garantir a atomicidade. Caso contrário, uma ação atômica poderia terminar deixando objetos em estados inconsistentes. Por exemplo, considerando-se uma ação atômica distribuída A, cuja execução envolve objetos armazenados nos computadores $C_1, C_2, \ldots, C_n$, e que o processo P, responsável pela execução de A, esteja em C_1. Antes que P possa enviar operações de *commit* a C_1, C_2, ..., C_n, é preciso verificar se todos os objetos envolvidos na ação A estão aptos a realizar tal operação.

São duas as condições que um objeto deve satisfazer para que esteja apto a realizar o *commit*:

- As operações de leitura realizadas no objeto devem ter sido feitas em um estado consistente deste objeto, ou seja, alterado exclusivamente por uma ação atômica que realizou *commit* e não *abort*. Se o protocolo de controle de concorrência utilizado por este objeto for o *strict two-phase locking*, esta condição será verdadeira; sendo assim, não precisa ser testada antes da realização de uma operação de *commit*.

- O estado do objeto, alterado pela ação atômica, deve estar armazenado de forma que possa ser recuperado.

P poderá enviar operações de *commit* a C_1, C_2, ..., C_n somente após a confirmação de que os objetos armazenados nestes computadores estão aptos a realizarem o *commit*. Essencialmente, este é o protocolo *two-phase commit*, descrito a seguir.

Para facilitar a descrição do protocolo *two-phase commit*, será assumido que para toda ação atômica distribuída A, haverá um processo em execução em cada computador C_i que armazena objetos envolvidos em A. Tal processo será responsável pela realização do protocolo de finalização atômica de A. O processo no computador onde A é criada será chamado de *coordenador*, enquanto que os demais processos serão chamados de *participantes*. O coordenador conhece a identificação de todos os participantes, tal que possa enviar mensagens a eles. Todo participante conhece a identificação do coordenador, mas não conhece, necessariamente, a identificação dos outros participantes. É importante notar que o coordenador e os participantes são somente abstrações adotadas para explicar o funcionamento do protocolo *two-phase commit*.

Um protocolo de finalização atômica é um algoritmo onde todos os processos que gerenciam objetos acessados por uma ação atômica devem decidir sobre a realização de uma das duas operações: *commit* ou *abort*. A escolha da operação é feita a partir de uma resposta (sim ou não) que todo processo deve emitir, respondendo se está apto a realizar o *commit*, e deve ser tal que:

- Condição 1. Todos os processos devem escolher a mesma opção de finalização.
- Condição 2. Um processo não pode reverter a sua escolha depois que a tenha feito.
- Condição 3. A escolha por *commit* só pode ser feita, se todos os processos responderem sim.
- Condição 4. Se não há falhas e todos os processos responderam sim, então a escolha será *commit*.
- Condição 5. Considerando-se apenas as falhas que este algoritmo possa prever, em qualquer ponto de uma execução, se todas as falhas ocorridas forem recuperadas e nenhuma outra ocorrer, então todos os processos farão a mesma escolha.

Protocolo *Two Phase Commit*

O mais simples e popular protocolo de finalização atômica é o protocolo *two-phase commit*. Assumindo a ausência de falhas, o protocolo se comporta da seguinte maneira:

- Passo 1. O coordenador envia uma mensagem *VOTE-REQ* (um pedido de voto, que será respondido com sim ou não) a todos os participantes.
- Passo 2. Quando um participante recebe uma mensagem *VOTE-REQ*, envia ao coordenador uma mensagem sim ou não, de acordo com o seu voto. Se votar não, realiza a operação de *abort*.
- Passo 3. O coordenador coleta os votos de todos os participantes. Se todos forem sim e o do coordenador também foi sim, então o coordenador decide por realizar a operação de *commit* e envia a todos os participantes uma mensagem de *COMMIT*. Caso contrário, o coordenador escolhe a operação de *abort* e envia uma mensagem de *ABORT* a todos os participantes que votaram sim (aqueles que votaram não já realizaram o *abort* no passo 2). Em ambos os casos, o coordenador finaliza sua execução.
- Passo 4. Cada participante que tenha votado sim espera por uma mensagem de *COMMIT* ou *ABORT* do coordenador. Quando recebe a mensagem, realiza uma operação (*commit* ou *abort*), de acordo com a mensagem recebida.

As duas fases do protocolo são a *voting phase* (passos 1 e 2) e a *decision phase* (passos 3 e 4). O período entre o momento em que um processo responde sim e o momento em que recebe informação suficiente para saber qual escolha será feita é chamado de "período de incerteza". Durante esse período, o processo não tem como escolher *commit* ou *abort*, e nem pode escolher, ele próprio, o *abort*. Todo participante inicia seu período de incerteza quando envia um sim ao coordenador (passo 2) e o termina quando recebe uma mensagem de *COMMIT* ou *ABORT* (passo 4).

O protocolo satisfaz as condições 1 a 4, mas não satisfaz a condição 5 por duas razões. Em primeiro lugar, em vários pontos do protocolo, os processos devem esperar por mensagens antes que continuem suas execuções. Contudo, tais mensagens podem não chegar, devido à ocorrência de falhas, e os processos podem ficar esperando por tempo indeterminado. Para evitar que esta situação ocorra, *timeouts* são utilizados. Quando um processo que está esperando por uma mensagem é interrompido por um *timeout*, o processo deve realizar uma ação especial chamada de "ação de *timeout*". Então, para satisfazer a condição 5, deve-se oferecer ações de *timeout* adequadas a cada passo do protocolo em que um processo espera por uma mensagem. Em segundo lugar, quando um processo recupera-se de alguma falha, torna-se necessário que o processo faça uma escolha consistente com a que outros processos fizeram durante o tempo decorrido na recuperação, para que a condição 5 seja satisfeita. De acordo com o que Bernstein [5] propõe, esta informação pode ser armazenada no *log* de ação atômica, no qual coordenadores e participantes podem registrar informações sobre ações atômicas. O *log* de ação atômica deve ser armazenado em meio estável, tal que possa sobreviver a ocorrência de falhas.

Ações de *Timeout*

Há três lugares no protocolo *two-phase commit* onde um processo está esperando por uma mensagem: no início dos passos 2, 3, e 4. No passo 2, um participante espera por uma mensagem *VOTE-REQ* do coordenador. Isto acontece antes que o participante tenha votado sim ou não. Como qualquer processo pode escolher realizar o *abort* antes de responder sim, se um participante excede o tempo de *timeout* na espera por uma mensagem *VOTE-REQ*, ele pode simplesmente escolher o *abort* e finalizar sua execução.

No passo 3, o coordenador está esperando por mensagens de sim ou não de todos os participantes. Neste ponto, o coordenador ainda não escolheu nenhuma operação, e nenhum participante escolheu *commit*. O coordenador pode escolher *abort* e, então, enviar *ABORT* a todos os participantes que responderam sim.

No passo 4, um participante p, que tenha votado sim, está esperando por uma mensagem *COMMIT* ou *ABORT* do coordenador. Neste momento, p está em seu período de incerteza. Diferentemente dos outros casos onde um processo pode escolher ele próprio uma operação, neste caso, o participante deve consultar outros processos para saber qual operação realizar. Esta consulta é chamada de "protocolo de terminação" para o protocolo *two-phase commit*.

No protocolo de terminação, para que nenhum processo fique bloqueado desnecessariamente até que possa restabelecer comunicação com o coordenador, é necessário que os participantes possuam a identificação uns dos outros. Na implementação do protocolo, o coordenador deve adicionar uma lista contendo a identificação dos participantes à mensagem *VOTE-REQ*, enviada a cada participante.

Bernstein [5] propõem um *protocolo de terminação cooperativo* onde: um participante p que consumir todo o tempo de *timeout*, enquanto estiver em seu período de incerteza, envia uma mensagem *DECISION-REQ* a cada um dos outros processos, q, para perguntar se q

conhece a escolha, ou pode, ele próprio, optar por uma. Há três casos:

- Caso 1. q já escolheu *commit* (ou *abort*): q simplesmente envia uma mensagem *COMMIT* (ou *ABORT*) a p, e p escolhe de acordo.
- Caso 2. q ainda não respondeu sim ou não: q pode, por conta própria, escolher *abort*. Então q envia um *ABORT* a p, que escolhe *abort*.
- Caso 3. q respondeu sim, mas ainda não escolheu uma operação: q também está no período de incerteza e, então, não pode ajudar p na escolha de uma operação.

Com este protocolo, se p pode se comunicar com algum q que satisfaz o caso 1 ou o caso 2, então p pode chegar a uma escolha sem ficar bloqueado. De outro lado, se o caso 3 é válido para todos os processos com os quais p pode se comunicar, então p está bloqueado. Isto persistirá até que as falhas sejam recuperadas para habilitar p a se comunicar com um q para o qual o caso 1 ou o caso 2 se aplica. Pelo menos um destes processos existe e é o coordenador. Então, este protocolo satisfaz a condição 5.

Recuperação

Considerando um processo p, que se recupera de um falha, para que possa satisfazer a condição 5, p deve escolher uma operação compatível com a escolha de outros processos — se não imediatamente após a sua recuperação, então algum tempo após a recuperação de falhas ocorridas em outros processos.

Supondo-se que quando p recuperar-se, obtenha seu estado no instante em que tenha falhado, então, se falhou antes de ter votado sim ao coordenador (passo 2 do protocolo *two-phase commit*), pode escolher, sozinho, o *abort*. Se p falhou após ter recebido uma mensagem *COMMIT* ou *ABORT* do coordenador ou depois de ter escolhido o *abort*, então já fez sua escolha. Nestes casos, p pode se recuperar independentemente.

Se p falhou enquanto estava em seu período de incerteza, quando se recuperar não pode fazer uma escolha por conta própria. Se tiver votado sim, antes que tenha ocorrido a falha, é possível que todos os outros processos também tenham respondido e, portanto, escolhido *commit* enquanto p estava parado. Também é possível que algum processo tenha respondido não, ou tenha escolhido *abort*. O processo p não pode escolher entre estas duas possibilidades baseado-se na informação disponível localmente e deve, então, consultar outros processos para fazer sua escolha.

Neste caso, p está exatamente no mesmo estado em que estaria se seu tempo de *timeout* tivesse se esgotado enquanto estivesse esperando por um *COMMIT* ou *ABORT* do coordenador. O processo p pode realizar sua escolha utilizando o protocolo de terminação. Nota-se que p pode estar bloqueado, desde que possa se comunicar apenas com processos que estão em seus períodos de incerteza.

Para retornar ao estado em que estava quando falhou, cada processo deve armazenar informações no *log* de ação atômica de sua máquina. Cada processo tem acesso somente

a seu *log* de ação atômica local. Considerando a utilização do protocolo de terminação cooperativo, o *log* de ação atômica deve ser gerenciado da seguinte forma:

- Quando o coordenador envia uma mensagem *VOTE-REQ*, escreve um registro *start-2PC* no *log* de ação atômica. Este registro possui a identificação dos participantes e pode ser escrito antes ou depois do envio das mensagens.

- Se um participante votou sim, escreve um registro de sim no *log* de ação atômica, antes de enviá-lo ao coordenador. Este registro contém o nome do coordenador e uma lista de todos os outros participantes (fornecida na mensagem *VOTE-REQ*, pelo coordenador). Se o participante votar não, escreve um registro de *abort* antes ou depois de enviar uma mensagem de não ao coordenador.

- Antes que o coordenador envie um *COMMIT* aos participantes, escreve um registro de *commit* no *log* de ação atômica.

- Quando o coordenador envia *ABORT* aos participantes, escreve um registro de *abort* no *log* de ação atômica. Este registro pode ser escrito antes ou depois do envio das mensagens.

- Depois de ter recebido um *COMMIT* (ou *ABORT*), um participante escreve um registro de *commit* (ou *abort*) no *log* de ação atômica.

A escrita de um registro de *commit* ou de *abort* no *log* de ação atômica é a atividade na qual um processo escolhe *commit* ou *abort*.

Quando uma máquina S se recupera de uma falha, a execução de uma ação atômica distribuída pode ser realizada examinando-se seu *log* de ação atômica:

- Se o *log* de ação atômica contiver um registro de *start-2PC*, então S é a máquina do coordenador. Se também possuir um registro de *commit* ou *abort*, então o coordenador escolheu sua operação antes da ocorrência da falha. Se nenhum registro é encontrado, o coordenador pode escolher por conta própria o *abort*, inserindo um registro de *abort* no *log* de ação atômica.

- Se o *log* de ação atômica não contiver um registro de *start-2PC*, então S é a máquina de um participante. Há três casos a se considerar:

  - Caso 1. O *log* de ação atômica contém um registro de *commit* ou *abort*. Então o participante tinha feito sua escolha antes da falha.

  - Caso 2. O *log* de ação atômica não contém um registro de sim. Neste caso, o participante falhou antes que tenha votado, ou votou não (mas não escreveu um registro de *abort* antes da falha). Ele pode então escolher, por ele próprio, o *abort*, inserindo um registro de *abort* no *log* de ação atômica.

- Caso 3. O *log* de ação atômica contém um sim, mas não um registro de *commit* ou *abort*. Neste caso, a falha ocorreu enquanto o participante estava em seu período de incerteza. Ele pode tentar fazer sua escolha utilizando o protocolo de terminação 2PC.

2.3.4 Recuperação de Estado

Para satisfazer a propriedade de atomicidade, o mecanismo que implementa uma ação atômica deve possuir alguma forma de recuperação dos estados de objetos que foram alterados durante a sua execução. A técnica de recuperação de estados de objetos em ações atômicas é chamada de *backward error recovery* (recuperação por retrocesso). Os objetos são recuperados ao estado imediatamente anterior ao início da ação atômica. A recuperação é feita alterando-se o estado final de cada objeto por um estado previamente armazenado (recuperação baseada em estado), ou registrando-se todas as operações realizadas a partir do início da ação atômica, e desfazendo-se seus efeitos através das operações inversas (recuperação baseada em operação).

2.4 Sumário

Neste Capítulo, apresentou-se um resumo das principais técnicas normalmente empregadas na construção de sistemas distribuídos, com o objetivo de apresentar os fundamentos nos quais se baseou o desenvolvimento do sistema de ações atômicas distribuídas, detalhado no Capítulo 4.

Inicialmente, na Seção 2.1 foram discutidos os principais problemas a serem tratados na criação de aplicações distribuídas confiáveis — compartilhamento, distribuição e possibilidade de ocorrência de falhas. Para facilitar a programação de aplicações distribuídas, o conceito de transparência no contexto de sistemas distribuídos foi então apresentado, e alguns mecanismos que implementam a transparência de distribuição, discutidos. A seguir, apresentaram-se modelos de comunicação entre objetos distribuídos, e a interface de programação necessária à programação destas aplicações. Linguagens de programação distribuída devem ter um poder de expressão suficiente para descrever os componentes de uma aplicação distribuída, bem como os processos envolvidos. Neste sentido, linguagens orientadas a objetos possuem características que as tornam potencialmente adequadas para a programação distribuída. Foram descritos, na Seção 2.2 os principais conceitos relacionados ao paradigma de orientação a objetos.

Finalmente, na Seção 2.3, foram descritos os principais mecanismos necessários à implementação de um sistema de ações atômicas e algumas técnicas e algoritmos utilizados na implementação destes mecanismos.

Capítulo 3

Xchart

O propósito do projeto Xchart é desenvolver um ambiente computacional, denominado **Xchart**, adequado à definição, implementação e execução de interfaces homem-computador. Apenas por curiosidade, o nome **Xchart** foi obtido da união da 22ª letra do alfabeto grego (**X** e χ), identificada em Inglês pela palavra CHI, que coincide com o acrônimo de *Computer-Human Interface*, com a palavra *chart*, tradução para o Inglês do termo *diagrama*.

Ferramentas de apoio à construção de interfaces são conhecidas como Sistemas de Desenvolvimento de Interfaces (SDIs) ou ainda Sistemas de Gerenciamento de Interfaces (SGIs). Os termos empregados em Inglês são, respectivamente, *User Interface Development System* (UIDS) e *User Interface Management System* (UIMS) [55]. Tanto SGIs quanto SDIs compreendem notações e técnicas voltados para a especificação e a implementação de interfaces. Contudo, um SGI difere de um SDI por apoiar a execução de uma interface. **Xchart** é um SGI [41] que apóia a construção e execução de interfaces de sistemas interativos distribuídos. O ambiente integra mecanismos que tornam transparentes para o projetista a distribuição dos componentes da interface. Uma vez que estes recursos são inexistentes na grande maioria dos SGIs, **Xchart** é chamado nesse texto de Sistema de Gerenciamento de Interfaces Distribuídas (SGID), ou, em Inglês, *Distributed User Interface Management System* (DUIMS).

Neste Capítulo, descreve-se a arquitetura do SGID **Xchart** e o processo de desenvolvimento de interfaces apoiado por esse ambiente. A seguir, o sistema de execução da linguagem *Xchart* é brevemente comentado. Detalhes a respeito de todo o projeto Xchart podem ser encontrados no endereço URL: `http://www.dcc.unicamp/proj-xchart/`.

3.1 A Arquitetura do Ambiente Xchart

Em muitos trabalhos destinados à construção de sistemas interativos verifica-se o emprego de arquiteturas de *software* que atendem aos requisitos considerados importantes pelas atuais metodologias de engenharia de *software* para o desenvolvimento de sistemas computacionais, como facilidade de uso, modularização e outros. Algumas arquiteturas estruturam um sistema interativo em componentes autônomos e mais fáceis de serem mantidos. Além de

fornecer uma melhor estruturação, esta subdivisão amplia a produtividade durante a fase de implementação, pois partes diferentes do sistema podem ser desenvolvidas em paralelo. Não há, no entanto, uma arquitetura ideal. No SGID **Xchart** adota-se a arquitetura que divide logicamente um sistema interativo em três componentes funcionais (Figura 3.1): (1) o componente de *apresentação* define a aparência da interface do sistema interativo e é responsável pela captura de eventos gerados pelo usuário e exibição dos resultados produzidos pela aplicação; (2) o *controle de diálogo*, que implementa protocolos de comunicação entre a apresentação e a aplicação e define o comportamento da aplicação coordenando a interação do usuário com o sistema interativo e (3) a *aplicação*, que implementa a funcionalidade do sistema interativo. A interface do sistema interativo corresponde aos dois primeiros componentes.

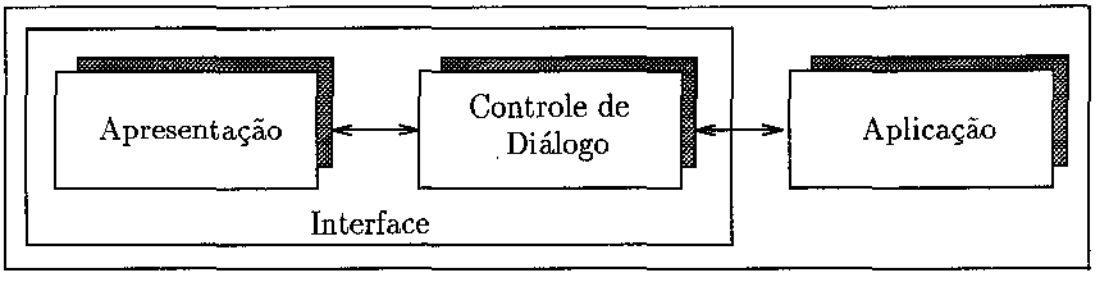

Figura 3.1: Arquitetura de um sistema interativo em **Xchart**.

A construção de uma interface no ambiente *Xchart* compreende a criação da apresentação e do controle de diálogo da interface. A Figura 3.2 mostra a composição do ambiente **Xchart**. A apresentação é criada em um sistema construtor de apresentações, denominado **Grace**. Grace possui um editor gráfico, baseado em um *toolkit* de um sistema de janelas, que permite a definição gráfica da apresentação. A partir desta definição, Grace produz o código (definições de classes) `C++` que implementa a apresentação. O controle de diálogo é produzido em um sistema construtor de diálogos, denominado **Smart**, onde a especificação dos diálogos é feita em termos da linguagem gráfica *Xchart*. A partir destas especificações, Smart produz automaticamente o código que implementa o diálogo, livrando o usuário de **Xchart** das dificuldades encontradas na implementação manual deste componente.

As especificações dos diálogos são construídas no editor gráfico Smart, que possibilita criar, modificar e armazenar diagramas que modelam o funcionamento dos diálogos. A versão em desenvolvimento [58] permite a criação dos diagramas visualizados na forma de "árvores", representando a hierarquia de estados do diagrama, e de um grafo, representando estados e transições. As especificações em *Xchart* são armazenadas em forma de texto, através de uma linguagem textual, denominada TeXchart (*Textual Xchart*), equivalente a *Xchart* e independente de linguagem de programação. O código que implementa o diálogo é produzido por um compilador de TeXchart. Além das ferramentas Grace e Smart, o ambiente

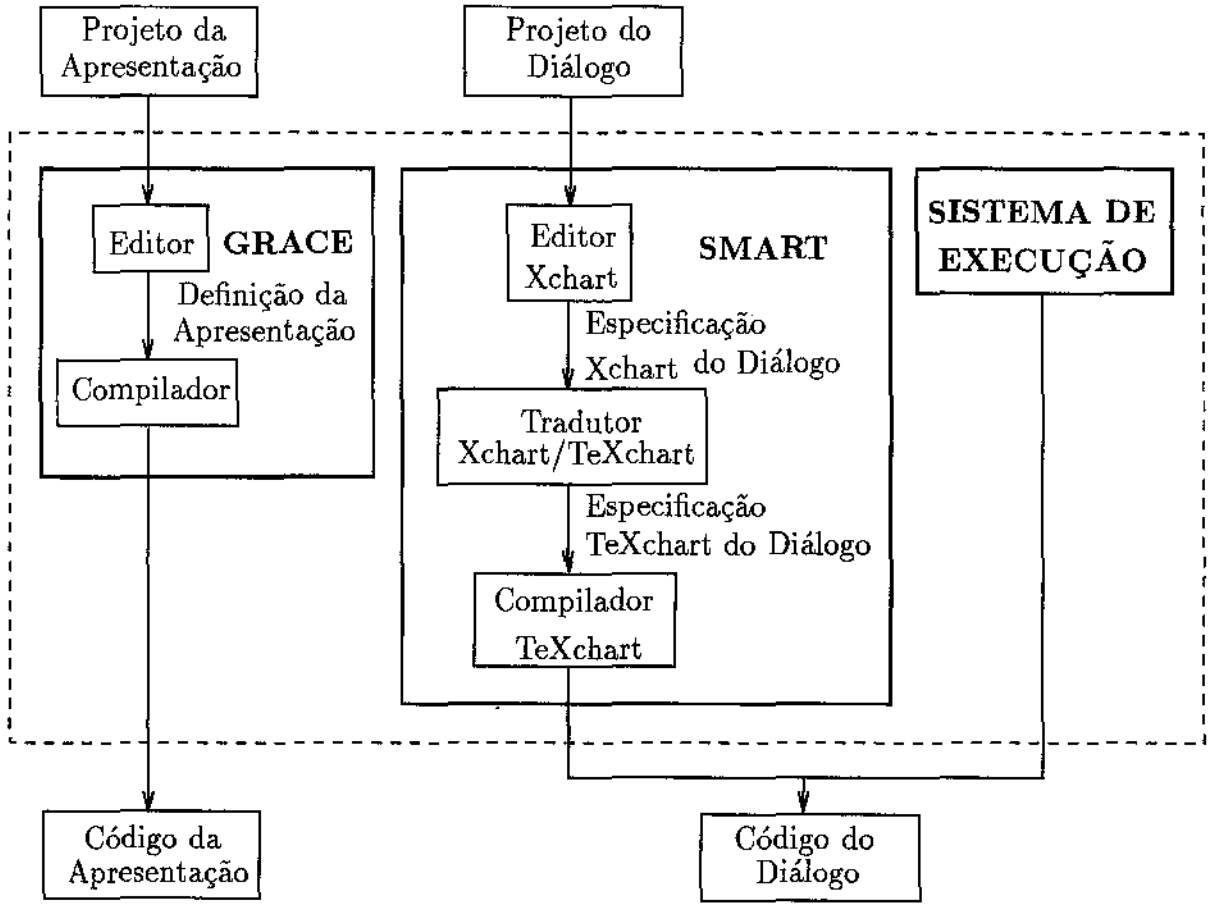

Figura 3.2: Componentes do Ambiente **Xchart**.

Xchart também possui um sistema de execução que implementa os mecanismos necessários à execução de diálogos.

A linguagem *Xchart* ainda se encontra em processo de definição. No Apêndice A apresentam-se informalmente alguns dos principais elementos e recursos da linguagem; uma descrição formal da linguagem pode ser encontrada em *Lucena et al* [42].

3.2 O Sistema de Execução de Xchart

O sistema de execução é o componente do ambiente **Xchart** responsável pela coordenação da interação entre os objetos que compõem o diálogo de uma interface produzida neste sistema. Em uma configuração distribuída de um diálogo, os objetos estão espalhados pelos nós do ambiente distribuído. A Figura 3.3 apresenta a estruturação do Sistema de Execução nas camadas *Núcleo Reativo*, *Gerente de Distribuição* e *Sistema Operacional*, descritas a seguir. As funções do sistema de execução são desempenhadas pela cooperação entre o núcleo reativo

e o gerente de distribuição, ambos construídos com base nas funções fornecidas pelo sistema operacional.

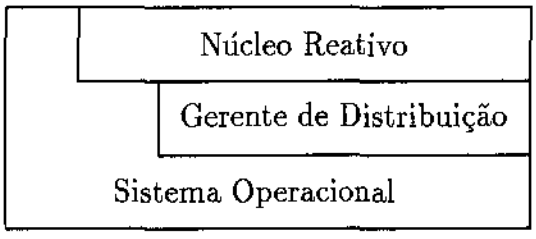

Figura 3.3: O sistema de execução de **Xchart**.

3.2.1 O Núcleo Reativo

O núcleo reativo interpreta diálogos especificados em *Xchart*. Durante a interpretação de um diálogo, o núcleo reativo mantém a configuração do(s) Xchart(s), coordenando a ativação, desativação e transição de estados em reação à ocorrência de eventos. Os eventos são gerados pelo usuário e capturados pela apresentação da interface, ou gerados por ações internas ao diálogo. A interação entre diálogos localizados em máquinas distintas de um ambiente distribuído é feita através dos recursos oferecidos pelo gerente de distribuição.

A interface de programação do núcleo reativo é orientada a objetos. Esta interface exporta métodos que são utilizados na implementação do sistema interativo.

3.2.2 O Gerente de Distribuição

O gerente de distribuição possibilita a execução concorrente e distribuída das tarefas realizadas pelo núcleo reativo, através de um conjunto de serviços fornecidos por:

- Um **sistema de ações atômicas distribuídas.** Este sistema é o componente de *software* responsável pela execução das tarefas de identificação, localização, armazenamento, comunicação e controle de concorrência de objetos distribuídos, necessárias à implementação do processamento de ações atômicas requeridas pelo núcleo reativo. A comunicação entre os objetos é feita através de um mecanismo de chamada de procedimento remoto, que utiliza as operações de comunicação ponto-a-ponto do sistema de comunicação em grupo para efetuar a transmissão de mensagens entre objetos. O servidor de nomes do sistema de comunicação em grupo é utilizado para o acesso e localização destes objetos. O projeto e a implementação do sistema de ações atômicas distribuídas, incluindo o mecanismo de chamada de procedimentos remotos, é pertinente a este trabalho.

- Um sistema de **comunicação em grupo**, o *Port System*. Este sistema já se encontra disponível no ambiente **Xchart**, e é composto pelos seguintes módulos [2]: (1) um *subsistema de comunicação*, que efetua a troca de mensagens entre objetos; (2) um *servidor de grupos*, responsável pela manutenção de grupos de objetos; (3) um *servidor de nomes*, que mantém a identificação e localização dos objetos, e (4) um *subsistema de detecção de falhas*, que detecta falhas na comunicação entre objetos.

3.2.3 O Sistema Operacional

O núcleo reativo e o gerente de distribuição do sistema de execução de **Xchart** são construídos sobre o sistema operacional Windows NT (versão 3.51) [15, 57].

3.3 Sumário

O ambiente **Xchart** emprega uma abordagem distinta das atuais ferramentas existentes para auxiliar na implementação de interfaces homem-computador de sistemas interativos. De fato, estas ferramentas têm sido caracterizadas principalmente por disporem de mecanismos que apoiam apenas a construção do componente de apresentação da interface, enquanto que o controle de diálogo é, em geral, construído manualmente pelos programadores de interfaces a partir de abstrações oferecidas por sistemas operacionais e linguagens de programação convencionais. **Xchart**, no entanto, possui mecanismos para a construção não só da apresentação, mas também do controle de diálogo. O ambiente produz automaticamente o código que implementa tais componentes, facilitando as tarefas dos programadores de interfaces. Uma outra vantagem do ambiente **Xchart** em relação às outras ferramentas, é que também possui mecanismos especiais que permitem a construção de interfaces distribuídas.

Neste capítulo foram descritos o processo de desenvolvimento de interfaces apoiado pelo ambiente **Xchart** e seu sistema de execução. A construção e execução de interfaces de sistemas distribuídos neste ambiente é possível através dos serviços oferecidos pelo gerente de distribuição, fundamentalmente, ações atômicas distribuídas.

Capítulo 4

Sistema de Ações Atômicas Distribuídas

No Capítulo anterior foi apresentada a arquitetura do sistema de execução do ambiente **Xchart** e verificou-se que ela contém dois componentes principais: um núcleo reativo e um gerente de distribuição, composto por um sistema de ações atômicas distribuídas e um sistema de comunicação em grupo. Neste Capítulo detalha-se a arquitetura de *software* do sistema de ações atômicas distribuídas. Na Seção 4.1, apresentam-se os módulos que implementam a funcionalidade desse sistema. Na Seção 4.2, descreve-se a hierarquia de classes do módulo de ações atômicas distribuídas. O gerador de *stubs* e a implementação do mecanismo de chamada de métodos remotos utilizado pelo sistema proposto são descritos na Seção 4.3.

4.1 A Arquitetura do Sistema

O projeto do sistema de ações atômicas distribuídas proposto neste trabalho foi elaborado em função de necessidades específicas requeridas para a implementação do sistema de execução do ambiente **Xchart**, basicamente:

(1) Atomicidade de operações. A execução de ações/reações em diálogos, efetuada por um núcleo reativo durante a interpretação destes diálogos, deve ser atômica.

(2) Controle de concorrência. Em uma configuração distribuída de diálogo, podem existir várias instâncias de sub-diálogos que estão espalhadas pelos computadores do ambiente distribuído. Durante a interpretação de um diálogo, os núcleos reativos distribuídos acessam concorrentemente estes sub-diálogos, o que requer a implementação de uma estratégia de controle de concorrência.

(3) Comunicação. A execução correta de uma ação/reação durante a interpretação de um diálogo requer que os núcleos reativos distribuídos trabalhem de forma cooperante, o que exige a implementação de mecanismos de comunicação entre diálogos.

O sistema de ações atômicas distribuídas implementa os mecanismos que garantem a atomicidade de operações e o controle de concorrência e a comunicação de objetos, necessários à implementação do sistema de execução de **Xchart**. O projeto e a implementação do sistema de ações atômicas distribuídas são orientados a objetos, isto é, encapsulamento (classes), herança e polimorfismo foram utilizados para criar uma arquitetura de *software* modular e flexível. Classes de aplicações distribuídas, em geral, e a hierarquia de classes do ambiente **Xchart**, em particular, podem herdar seletivamente e de forma transparente os mecanismos que garantem atomicidade, controle de concorrência e compartilhamento de objetos (localização, acesso e distribuição). A linguagem de programação orientada a objetos utilizada na programação de todo o sistema de ações atômicas distribuídas foi `C++`. O sistema proposto é constituído dos seguintes módulos:

- **Módulo de Ações Atômicas Distribuídas.** É o componente do sistema que implementa os mecanismos de ações atômicas distribuídas e satisfaz as necessidades (1) e (2) apresentadas acima, através de uma coleção de classes que fornece um protocolo básico para a execução das tarefas de *Begin*, *Commit* e *Abort* de uma ação atômica. A execução destas tarefas requer a implementação de estratégias de identificação e localização de objetos e, para o caso de ocorrência de alguma falha (operação *Abort*), de estratégias de armazenamento e recuperação de estados de objetos. Por causa dessa possibilidade, os estados dos objetos são armazenados em *repositórios distribuídos de objetos*, responsáveis pelo armazenamento do estado consistente de um objeto. Associado a cada estado de objeto, um repositório mantém uma lista de *locks* (Seção 4.2.2) necessária para a realização dos serviços de controle de concorrência requeridos pelo sistema. Na Seção 4.2 são apresentadas, em detalhes, as classes que implementam o módulo de ações atômicas distribuídas proposto nesse trabalho.

- **Módulo de Comunicação.** A solução adotada para a comunicação entre objetos baseia-se na adaptação de um mecanismo de chamada de procedimentos remotos ao modelo de orientação a objetos. Como procedimentos são pertinentes a linguagens não orientadas a objetos, ao invés de "chamada a procedimentos remotos", utiliza-se "chamada a métodos remotos". O mecanismo de chamada a métodos remotos fornece acesso transparente a objetos distribuídos e satisfaz a necessidade (3), apontada anteriormente. Esta transparência é garantida a partir da geração automática dos *stubs* de cliente e de servidor, efetuada por um *gerador de stubs*, cuja funcionalidade é detalhada na Seção 4.3. A transmissão de mensagens entre objetos no mecanismo de chamada a métodos remotos é implementada a partir de operações de comunicação do tipo ponto-a-ponto fornecido pelo sistema de comunicação em grupo [2]. A execução efetiva das tarefas realizadas pelo módulo de comunicação requer, como no caso do módulo de ações atômicas distribuídas, a implementação de mecanismos de identificação e localização de objetos distribuídos. Estes serviços são prestados pelo componente do sistema descrito a seguir.

- **Servidor de Nomes.** O servidor de nomes é o componente de *software* responsável pelos serviços de identificação e localização transparente de objetos no ambiente distribuído, serviços essenciais para a implementação dos módulos de ações atômicas e de comunicação do sistema proposto. Basicamente, o servidor de nomes é um dicionário centralizado cujas associações contém a identidade e a localização de um objeto. A identidade é definida por um nome atribuído pelo programador e pelo *surrogate* correspondente; a localização é definida pelo endereço da porta de comunicação do repositório de objetos que armazena o objeto. A principal funcionalidade do servidor de nomes é fornecer a localização do objeto, a partir de sua identidade [2].

A Figura 4.1 ilustra os módulos que definem a arquitetura do sistema de ações atômicas distribuídas. O nível mais alto de abstração é fornecido pelo módulo de ações atômicas distribuídas, o qual utiliza os serviços oferecidos pelo servidor de nomes e pelo módulo de comunicação do sistema. Os programadores de aplicações construídas a partir das classes do sistema de ações atômicas distribuídas, portanto, normalmente *devem* utilizar os serviços fornecidos pelo módulo de ações atômicas distribuídas, justamente pelos benefícios proporcionados pela abstração. Essa conveniência, evidentemente, não restringe o uso direto das tarefas executadas pelo servidor de nomes, pelo módulo de comunicação ou até mesmo pelo sistema operacional.

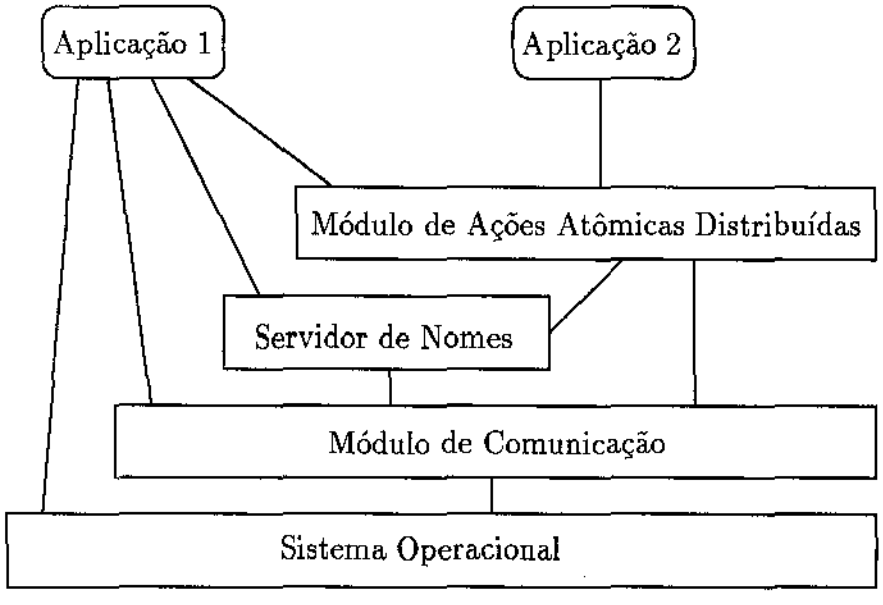

Figura 4.1: Arquitetura do Sistema de Ações Atômicas Distribuídas.

A Figura 4.2 ilustra uma possível instância da arquitetura do sistema de ações atômicas distribuídas. Neste exemplo, uma ação atômica está sendo executada no computador `M1`. A execução distribuída desta ação atômica é realizada invocando-se operações no objeto local

`Objeto 1` (armazenado no repositório de objetos de `M1`) e nos objetos remotos `Objeto 2` e `Objeto 3` (armazenados no repositório de objetos do computador `M2`). A invocação de operações em objetos remotos (chamadas a métodos remotos) é executada pelo módulo de comunicação. Observa-se, na figura, a instância única do servidor de nomes e as instâncias múltiplas do módulo de comunicação e do repositório de objetos.

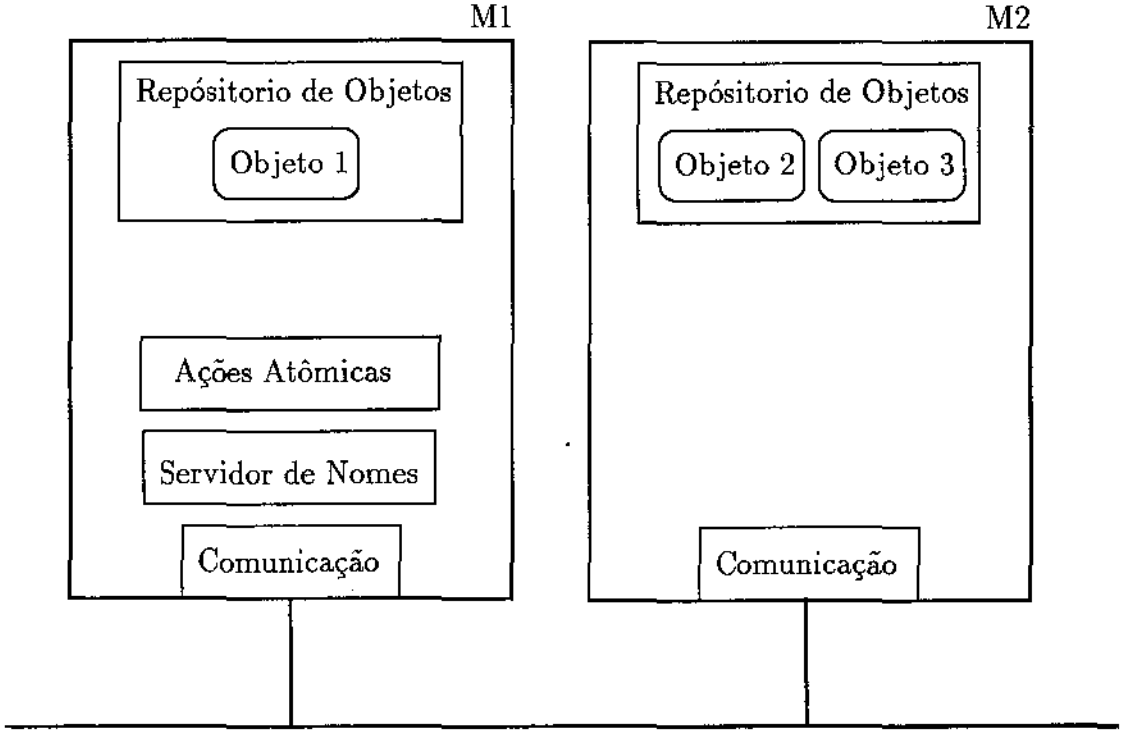

Figura 4.2: Uma instância da arquitetura do sistema.

4.2 Implementação do Módulo de Ações Atômicas Distribuídas

A Figura 4.3 ilustra a hierarquia de classes que implementa o módulo de ações atômicas distribuídas. As classes são representadas por retângulos contento o nome da classe e especializações são representadas por setas com linhas cheias que interligam os retângulos. As setas com linhas tracejadas representam como classes definidas pelo usuário utilizam os mecanismos fornecidos pelo sistema de ações atômicas através do mecanismo de herança.

A seguir, comenta-se como estas classes implementam os mecanismos de ações atômicas. A descrição de cada classe é feita de forma tabular. Cada tabela contém o tipo, o nome e a descrição dos principais atributos e métodos da classe. Opcionalmente, o identificador da

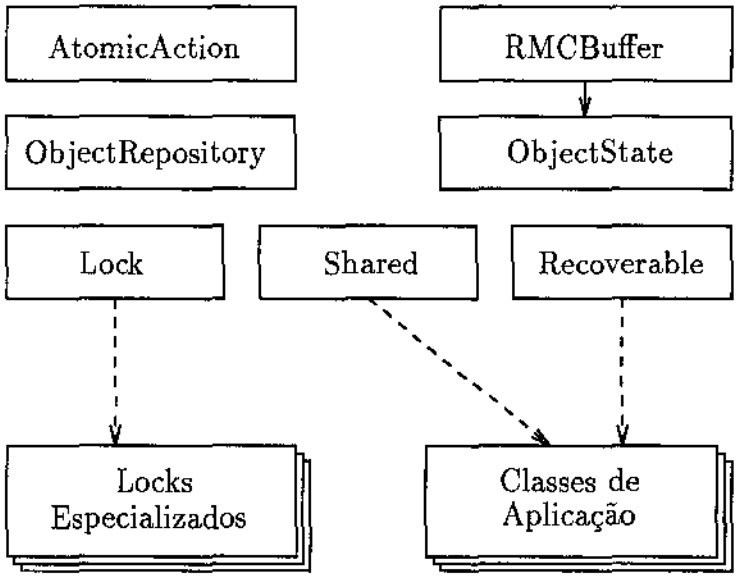

Figura 4.3: Classes do módulo de ações atômicas.

classe é seguido pela lista de suas classes bases; o símbolo ⇐ separa o nome da classe da sua lista de classes bases.

As classes que implementam o sistema de ações atômicas distribuídas serão descritas partindo das que realizam os mecanismos básicos do sistema (linearização e repositório de objetos) e chegando finalmente às classes que implementam o mecanismo de controle de concorrência e o protocolo de finalização de ações atômicas. Esta descrição "*bottom-up*" da arquitetura das classes do sistema foi escolhida por duas razões: (1) segue a seqüência de construção de níveis de abstração sucessivamente mais altos utilizados no projeto do sistema de ações atômicas distribuídas e (2) permite que a descrição da interface de programação do sistema fique próxima dos exemplos de sua utilização, apresentados no Capítulo 5.

4.2.1 Recuperação do Estado de Objetos

No sistema de ações atômicas distribuídas é utilizada a técnica de recuperação de estado baseada no retrocesso do estado dos objetos envolvidos em uma ação atômica a um estado anterior ao início da ação. A recuperação de estado por retrocesso é feita armazenando-se em memória volátil uma cópia do estado original, denominada *shadow copy*, de cada objeto que participa de uma ação atômica. Se ocorrer uma operação de *abort*, esta cópia será utilizada para restaurar os estados dos objetos, atribuindo-lhes os valores anteriores ao início da ação atômica.

No sistema de ações atômicas distribuídas, os objetos são identificados por um nome único; o compartilhamento de estados de objetos é feito através da utilização de sua identidade única. Quando um objeto é criado pela primeira vez, ou seja, possui identidade distinta

de qualquer outro objeto, o seu estado é armazenado no repositório de objetos. A fim de compartilhar este estado, deve-se criar um objeto com a mesma identidade, ou seja, uma cópia do estado do objeto.

Antes que um objeto possa ter suas operações controladas por uma ação atômica, o seu estado é obtido do repositório de objetos através de seu nome. Similarmente, se a ação atômica for realizada com sucesso, ou seja, ocorreu o *commit*, este estado é atualizado no repositório. Tanto o armazenamento quanto a recuperação de estados de objetos são realizados por objetos da classe `Recoverable`. Portanto, quaisquer classes cujas instâncias podem ter suas operações controladas por uma ação atômica devem ser derivadas de `Recoverable`.

Instâncias da classe `ObjectState` são utilizadas para a transferência de estados de objetos executados em processos distintos. Objetos `ObjectState` mantêm um *buffer*, no qual os valores de atributos que constituem o estado de um objeto podem ser armazenados, e possuem uma interface que permite que objetos `C++` sejam convertidos para instâncias de `ObjectState`. As unidades de armazenamento do repositório de objetos são instâncias da classe `ObjectState`.

A classe `ObjectRepository`

A classe `ObjectRepository` (Tabela 4.1) fornece as operações necessárias ao armazenamento de estados de objetos, sua função principal é gerenciar instâncias da classe `ObjectState`. Os estados dos objetos são armazenados em uma tabela de espalhamento (*hash table*) que possui como chave de busca a identidade do objeto. Cada elemento desta tabela é composto por uma instância da classe `ObjectState` e uma lista contendo os *locks* associados ao objeto.

Um repositório de objetos é um objeto existente durante todo o tempo de execução do sistema de ações atômicas, que aguarda por mensagens enviadas por objetos `Shared` (Seção 4.2.2) e/ou `Recoverable`. Tais mensagens são de requisições para a realização de tarefas pertinentes ao repositório de objetos. Após o recebimento de uma mensagem, a tarefa requerida é executada e os resultados produzidos são enviados ao objeto solicitante. As tarefas executadas por um repositório de objetos são:

1. *Armazenamento de estados de objetos.* O armazenamento do estado de um objeto é efetuado em reposta à mensagem `StoreObjState()`, parametrizada com um objeto `ObjectState`.

2. *Armazenamento de um lock.* A execução do método privado `StoreLock()` adiciona o novo *lock*, na lista de *locks* de um objeto.

3. *Liberação de um lock de um objeto.* A liberação de um *lock* associado a um objeto é feita em resposta à mensagem `ClearLock()`, parametrizada com a identidade do objeto para o qual se deseja liberar um *lock* (`ObjectID`) e com o *lock* a ser liberado (`LockT`).

4. *Obtenção do estado de um objeto.* Em resposta à mensagem `GetObjState()` enviada a um repositório de objetos, obtém-se o estado de um objeto armazenado em um `ObjectState`. `GetObjState()` é parametrizada com a identidade do objeto do qual se deseja obter o estado (`ObjectID`).

5. *Obtenção dos locks.* Obtém-se os *lock* associados a um objeto a partir da execução do método `GetLock()`. Tal método é parametrizado com a identidade do objeto do qual se deseja obter os *locks* (`ObjectID`).

Classe `ObjectRepository`			
Atributo		Acesso	Descrição
`HashTable`	`ObjectTable`	privado	Tabela de objetos
Método		Acesso	Descrição
`ObjectRepository(ObjTableSize)`		público	Construtor
`~ObjectRepository()`		público	Destrutor
`void`	`StoreObjState(ObjState&)`	público	Armazena `ObjectState`
`void`	`StoreLock(ObjectID, LockT)`	público	Armazena *lock*
`void`	`ClearLock(ObjectID, LockT)`	público	Libera *lock*
`ObjectState&`	`GetObjState(ObjectID)`	público	Obtém `ObjectState`
`LockList&`	`GetLock(ObjectID)`	público	Obtém lista de *locks*

Tabela 4.1: Classe `ObjectRepository`.

A classe `RMCBuffer`

A classe `RMCBuffer` (Tabela 4.2) fornece um protocolo para o empacotamento e desempacotamento dos tipos básicos de `C++`. Esta classe também encapsula as funções necessárias para o envio e recebimento de mensagens entre objetos remotos, utilizando funções do sistema de comunicação [2]. A Tabela 4.2 apresenta apenas os principais atributos e métodos desta classe.

A classe `ObjectState`

A classe `ObjectState` (Tabela 4.3) é derivada da classe `RMCBuffer`. Esta classe possui toda a funcionalidade da classe `RMCBuffer` (empacotamento e desempacotamento de tipos primitivos `C++`) e adiciona um novo atributo (`ObjectId`). O atributo `ObjectId` é utilizado pelo repositório de objetos tanto no armazenamento, quanto na recuperação do estado do objeto. As tarefas executadas por um objeto `ObjectState` são:

- *Atribuição da identidade do objeto.* A atribuição da identidade de um objeto é feita em resposta ao envio da mensagem `SetObjectID()`, parametrizada com a identidade do objeto.

Classe `RMCBuffer`			
Atributo		Acesso	Descrição
`BYTE*`	`Buffer`	privado	Início do buffer
`unsigned`	`Count`	privado	Tamanho do buffer
`unsigned`	`Itens`	privado	Número de itens empacotados
Método		Acesso	Descrição
`RMCBuffer(unsigned, unsigned)`		público	Construtor
`RMCBuffer(RMCBuffer&)`		público	Construtor
`~RMCBuffer()`		público	Destrutor
`unsigned short`	`CreateConnection(char*, char*)`	público	Conexão para comunicação
`int`	`Send(char*, char*)`	público	Envia Buffer
`int`	`Send(Addr*)`	público	Envia Buffer
`int`	`Receive(unsigned long)`	público	Recebe Buffer
`RMCBuffer`	`operator <<(char)`	público	Empacota char
`RMCBuffer`	`operator <<(unsigned char)`	público	Empacota unsigned char
`RMCBuffer`	`operator <<(short)`	público	Empacota short
`RMCBuffer`	`operator <<(unsigned short)`	público	Empacota unsigned short
`RMCBuffer`	`operator <<(int)`	público	Empacota int
`RMCBuffer`	`operator <<(unsigned int)`	público	Empacota unsigned int
`RMCBuffer`	`operator <<(long)`	público	Empacota long
`RMCBuffer`	`operator <<(unsigned long)`	público	Empacota unsigned long
`RMCBuffer`	`operator <<(float)`	público	Empacota float
`RMCBuffer`	`operator <<(double)`	público	Empacota double
`RMCBuffer`	`operator <<(long double)`	público	Empacota long double
`RMCBuffer`	`operator <<(char*)`	público	Empacota char*
`RMCBuffer`	`operator <<(RMCBuffer&)`	público	Empacota outro RMCBuffer
`RMCBuffer`	`operator >>(char&)`	público	Desempacota char&
`RMCBuffer`	`operator >>(unsigned char&)`	público	Desempacota unsigned char&
`RMCBuffer`	`operator >>(short&)`	público	Desempacota short&
`RMCBuffer`	`operator >>(unsigned short&)`	público	Desempacota unsigned short&
`RMCBuffer`	`operator >>(int&)`	público	Desempacota int&
`RMCBuffer`	`operator >>(unsigned int&)`	público	Desempacota unsigned int&
`RMCBuffer`	`operator >>(long&)`	público	Desempacota long&
`RMCBuffer`	`operator >>(unsigned long&)`	público	Desempacota unsigned long&
`RMCBuffer`	`operator >>(float&)`	público	Desempacota float&
`RMCBuffer`	`operator >>(double&)`	público	Desempacota double&
`RMCBuffer`	`operator >>(long double&)`	público	Desempacota long double&
`RMCBuffer`	`operator >>(char*&)`	público	Desempacota char*&

Tabela 4.2: Classe `RMCBuffer`.

- *Obtenção da identidade do objeto.* Obtém-se a identidade de um objeto, em resposta ao envio da mensagem `GetObjectID()` a um objeto `ObjectState`.

Classe `ObjectState` ⇐ `RMCBuffer`		
Atributo	Acesso	Descrição
`ObjectID ObjectId`	privado	Identidade do objeto
Método	**Acesso**	**Descrição**
`ObjectState(ObjectID)`	público	Construtor
`ObjectState()`	público	Construtor
`ObjectState(ObjectState&)`	público	Construtor de cópia
`void SetObjectID(ObjectID)`	público	Atribui identidade ao objeto
`ObjectID GetObjectID()`	público	Obtém identidade do objeto

Tabela 4.3: Classe `ObjectState`.

A classe `Recoverable`

A classe `Recoverable` (Tabela 4.4) encapsula três dados a respeito de um objeto controlado pelo sistema de ações atômicas distribuídas: (1) o identificador único do objeto (atributo `ObjectId`), (2) o endereço do repositório de objetos (atributo `ObjRepAddr`) que armazena o estado do objeto [2] e (3) a informação de se o objeto já existe (atributo `HasObjectState`), ou seja, se o estado do objeto está armazenado no repositório.

Nao é possível prever a estrutura de classes definidas pelo usuário do sistema de ações atômicas. Portanto, as classes derivadas de `Recoverable` devem possuir algumas operações particulares, a fim de cooperarem na recuperação do estado de objetos. Especificamente, devem implementar os métodos `SaveState()` e `RestoreState()` declarados como virtuais na classe `Recoverable`. Para que o estado de uma instância de uma classe definida pelo usuário possa ser gerenciado pelo sistema de ações atômicas distribuídas, ele deve informar ao sistema a estrutura de atributos de suas classes, indicando como localizá-las. Os métodos `SaveState()` e `RestoreState()` servem para este fim, ou seja, permitem que um usuário programe os métodos que são ativados pelo sistema de ações atômicas distribuídas para ativar/desativar objetos da aplicação.

Objetos da classe `Recoverable` executam as seguintes tarefas:

1. *Obtenção do estado do objeto.* Toda classe derivada de `Recoverable` deve possuir um construtor especial que toma como parâmetro a identidade de um objeto. Durante a construção do objeto, este parâmetro é transferido para o construtor da classe base. O construtor de `Recoverable` utiliza a identidade do objeto para consultar o servidor de nomes e verificar se o objeto já existe. Se ele existir, então o construtor obtém o seu estado do repositório de objetos. Depois, este estado é utilizado como parâmetro

do método `RestoreState()`. Quando um objeto do tipo `Recoverable` não é utilizado em uma ação atômica, a recuperação de seu estado deve ser feita pelo programador. Para verificar se o objeto já existe, utiliza-se o método `HasObjectState()`. Se existir, obtém-se o seu estado através do método `GetObjectState()`.

2. *Armazenamento do estado do objeto.* O estado de um objeto é armazenado no repositório de objetos, quando uma ação atômica realiza a operação de *commit*. Em resposta à mensagem `UpdateState()` enviada por uma ação atômica a um objeto `Recoverable`, o estado mais recente do objeto é armazenado no repositório de objetos. `UpdateState()` executa o método `SaveState()` que transfere os valores dos atributos do objeto para uma instância de `ObjectState` e, a seguir, a envia ao repositório.

3. *Certifica uma ação atômica de que o objeto está ativo.* Em resposta a uma mensagem `CanCommit()` enviada por uma ação atômica, um objeto `Recoverable` certifica a esta ação atômica de que está ativo.

Classe `Recoverable`			
Atributo		Acesso	Descrição
`ObjectID`	`ObjectId`	privado	Identidade do objeto
`Addr`	`ObjRepAddr`	privado	Repositório de objetos
`BOOL`	`HasObjectState`	privado	Objeto já existe?
Método		Acesso	Descrição
`Recoverable(ObjectName)`		público	Construtor
`BOOL`	`HasObjState()`	público	Objeto já existe?
`BOOL`	`UpdateState()`	público	Atualiza estado de objeto
`BOOL`	`CanCommit()`	público	Commit?
`ObjectState&`	`GetObjectState()`	público	Obtém estado do objeto
`virtual BOOL`	`SaveState(ObjectState&)`	público	Armazena estado de objeto
`virtual BOLL`	`RestoreState(ObjectState&)`	público	Restaura estado de objeto

Tabela 4.4: Classe `Recoverable`.

4.2.2 Controle de Concorrência

O sistema de ações atômicas utiliza o protocolo *strict two-phase locking* para realizar o controle de concorrência dos objetos. A aquisição de *locks* é controlada pelo programador, que indica o tipo de *lock* necessário a uma operação. A liberação dos *locks* (`unlock`) é feita por uma ação atômica, facilitando a ação do programador e garantindo a propriedade de *strict two-phase locking*.

As funções de *lock* e *unlock* estão encapsuladas na classe `Shared` (Tabela 4.6). Toda classe definida pelo programador cujos objetos podem participar de ações atômicas deve ser

derivada da classe `Shared`. Objetos desta classe não conhecem diretamente a política de alocação de *locks* definida para o controle de concorrência. A classe `Lock` (Tabela 4.5) é responsável pela implementação da política de *locks* utilizada pelo sistema de ações atômicas distribuídas.

Os tipos de *lock* definidos na classe `Lock` podem ser *write* (exclusivo) ou *read* (compartilhado). No entanto, o programador pode definir outros tipos e semânticas de *lock* para classes particulares de objetos através da construção de classes derivadas da classe `Lock`. As novas semânticas de *lock* devem ser compatíveis com o protocolo de controle de concorrência *strict two-phase locking.*

A versão inicial do sistema de ações atômicas não possui nenhum esquema de detecção de *deadlocks.* Será utilizada a técnica de *timeout* para evitá-los. Há um número pré-estabelecido de tentativas que são realizadas quando um pedido de *lock* não puder ser imediatamente aceito. Se finalmente o *lock* não puder ser estabelecido, então fica a cargo do programador decidir se a ação atômica será interrompida, ou se outras tentativas deverão ser feitas.

Um mesmo estado de objeto pode ser compartilhado por diversos processos distintos, possivelmente distribuídos. Portanto, a informação sobre os *locks* de um objeto deve ser compartilhada por todos estes processos. Cada objeto presente no repositório de objetos possui uma lista com os *locks* associados a ele. Esta lista é consultada no instante em que um novo *lock* deve ser atribuído a um objeto.

A classe `Lock`

A classe `Lock` (Tabela 4.5) fornece *locks* do tipos *read* (`ReadLock`) e *write* (`WriteLock`). A fim de atender às necessidades específicas de *lock* em uma aplicação, outros tipos de *lock* podem ser derivados da classe `Lock` utilizando-se o mecanismo de herança. Embora o mecanismo de controle de concorrência utilizado seja o *strict two-phase locking*, pode-se definir qualquer tipo de *lock.* A classe `Lock` encapsula a informação do tipo (atributo `LockType`) de *lock* e possui um protocolo de operações que auxiliam na aquisição de um *lock* para um objeto qualquer. As tarefas executadas por um objeto da classe `Lock` são:

1. *Verificação de compatibilidade entre locks.* Verifica-se a compatibilidade entre dois objetos *locks* enviando-se uma mensagem `Conflicts()` a um objeto `Lock`, parametrizada com o *lock* que deve ser verificado.

2. *Atribuição de lock.* A atribuição do *lock* é efetuada em resposta ao envio da mensagem `SetLock()`, parametrizada com o tipo de *lock*, ao um objeto `Lock`.

3. *Obtenção do tipo de lock corrente.* Pode-se obter o tipo de *lock* de um objeto `Lock` enviando-o a mensagem `GetLock()`.

Classe Lock			
Atributo		Acesso	Descrição
LockT	LockType	privado	Tipo de lock
Método		**Acesso**	**Descrição**
Lock(LockT)		público	Construtor
Lock(Lock)		público	Construtor de cópia
Lock()		público	Construtor
virtual BOOL	Conflicts(LockT)	público	Compara locks
void	SetLock(LockT)	público	Atribui lock
LockT	GetLock()	público	Obtém lock

Tabela 4.5: Classe Lock.

A classe Shared

A classe Shared (Tabela 4.6) encapsula a identidade de um objeto do sistema de ações atômicas (atributo ObjectId) e o endereço do repositório de objetos (atributo ObjRepAddr) que armazena o estado do objeto [2]. Além disso, fornece um protocolo para as operações necessárias ao controle de concorrência, descritas a seguir:

Classe Shared			
Atributo		Acesso	Descrição
ObjectID	ObjectId	privado	Identidade do objeto
Addr	ObjRepAddr	privado	Repositório de objetos
Método		**Acesso**	**Descrição**
Shared(ObjectName)		público	Construtor
ErrorCode	SetLock(Lock&, AtomicAction&, Tout)	público	Atribui *lock* a objeto
ErrorCode	SetLock(Lock&, Tout)	público	Atribui *lock* a objeto
ErrorCode	ReleaseLock(Lock&)	público	Libera *lock* de objeto

Tabela 4.6: Classe Shared.

1. *Aquisisão de locks.* A operação de *lock* é executada em resposta à mensagem SetLock(), que possui duas sintaxes distintas. Em uma, deve ser parametrizada com um objeto da classe Lock para indicar o *lock* necessário, a ação atômica na qual o *lock* é requerido e um número de *timeout*. O segundo parâmetro é utilizado para atualizar a lista de objetos da ação atômica, se o *lock* for fornecido, o terceiro parâmetro indica o número máximo de tentativas a serem realizadas até que o pedido de *lock* seja aceito. Na segunda sintaxe do método SetLock, são necessários apenas os parâmetros Lock e *timeout*. O método SetLock pode ser utilzado com esta sintaxe quando o objeto ao

qual se deseja atribuir um *lock* não fizer parte de uma ação atômica. Na implementação de `SetLock()` verifica-se a possibilidade de realizar o *lock* do objeto comparando o *lock* requerido com os outros *locks* de objetos armazenados no repositório de objetos. Se o *lock* for fornecido, o novo objeto `Lock` é adicionado à lista de *locks*. A comparação é feita em resposta à mensagem `Conflicts()` enviada ao objeto `Lock`.

2. *Liberação de locks.* O *Unlock* de um objeto é feito por uma ação atômica através do envio da mensagem `ReleaseLock()`, parametrizada com um objeto da classe `Lock`. A operação `ReleaseLock()` elimina o objeto `Lock`, inserido na lista de objetos `Lock` (atributo `LockList`) pelo método `SetLock()`, e o remove da lista de *locks* mantida pelo repositório de objetos.

4.2.3 Execução de Ações Atômicas

A versão inicial do sistema de ações atômicas fornece mecanismos para o controle e a realização de ações atômicas simples, ou seja, não são permitidas ações atômicas encadeadas. O sistema gerencia o controle de concorrência e a atomicidade das operações de objetos distribuídos. A interface de programação do sistema fornece os métodos necessários à construção de operações atômicas, isto é *begin*, *commit* e *abort*, para iniciar, terminar e cancelar ações atômicas, respectivamente. Estes métodos implementam o protocolo de finalização atômica *two-phase commit*, descrito na Seção 2.3.3. Como a recuperação do estado de objetos é feita apenas em memória volátil, os mecanismos de recuperação a falhas do protocolo *two-phase commit* não são implementados.

A classe `AtomicAction`

A classe `AtomicAction` (Tabela 4.7) fornece um protocolo para a execução das tarefas pertinentes a uma ação atômica e encapsula as informações necessárias à sua correta execução. As tarefas executadas por um objeto da classe `AtomicAction` são:

Classe `AtomicAction`		
Atributo	Acesso	Descrição
`Participator ParticipatorList`	privado	Lista de participantes
Método	**Acesso**	**Descrição**
`AtomicAction()`	público	Construtor
`virtual ErrorCode Begin()`	público	Inicia a ação atômica
`virtual ErrorCode Commit()`	público	Efetua *commit* da ação atômica
`virtual ErrorCode Abort()`	público	Efetua *abort* da ação atômica

Tabela 4.7: Classe `AtomicAction`.

1. *Manutenção da lista de objetos da ação atômica.* Uma ação atômica gerencia a realização de operações sobre uma coleção de objetos. Estes objetos são adicionados a uma lista (atributo `ParticipatorList`) mantida por `AtomicAction`.

2. *Begin.* A operação *begin*, efetuada em resposta à mensagem `Begin()`, delimita o início de uma ação atômica.

3. *Commit.* A operação de *commit* de uma ação atômica é realizada pelo método `Commit()` que implementa o protocolo *two-phase commit.* Primeiramente `Commit()` envia uma mensagem `CanCommit()` a todos os elementos da lista `ObjectList` para verificar se o *commit* pode ser realizado. Em caso positivo, envia as mensagens `ReleaseLock` e `UpdateState` a todos os elementos de `ObjectList`. Caso contrário, realiza o *abort.*

4. *Abort.* O *abort* de uma ação atômica é feito em resposta à mensagem `Abort()` enviada a um objeto `AtomicAction`. `Abort()` envia uma mensagem `ReleaseLock` a cada elemento da lista `ObjectList`.

Utilização de Ações Atômicas

Com a finalidade de ilustrar a utilização da classe `AtomicAction` e a relação entre os outros componentes da arquitetura, a Figura 4.4 mostra um fragmento de código `C++` que utiliza o sistema de ações atômicas distribuídas. A seguir, são explicadas as ações executadas por cada linha do programa.

```
1: Window w("Exemplo1");
2: AtomicAction a;
3:
4: a.Begin();
5: if (w.SetLock(Lock(WRITELOCK), a) == LOCKED)
6:         w.DoAnything();
7: ...
8: if (...)
9:     a.Abort();
10:else
11:    a.Commit();
```

Figura 4.4: Exemplo da utilização de ações atômicas.

- Linha 1: A classe `Window` é derivada de `Recoverable` e de `Shared`. Supondo-se que um objeto da classe `Window` já tenha sido criado anteriormente com a identidade `Exemplo1`, durante a construção do objeto `w`, o construtor de `Recoverable` é executado e seu estado é obtido do repositório de objetos.

- Linha 2: Criação de uma instância a da classe `AtomicAction`.
- Linha 4: Chamada da operação `Begin()` para iniciar a ação atômica a.
- Linha 5: De acordo com o protocolo *two phase locking*, realiza-se o *lock* sobre o(s) objeto(s) pertencentes à ação atômica, neste caso, sobre `Exemplo1`. A mensagem `SetLock` é enviada ao objeto, que realiza o *lock* e adiciona a localização do objeto na lista de objetos da ação atômica.
- Linha 6: Se o *lock* foi efetuado, a operação `DoAnything()` é realizada sobre o objeto `Exemplo1`.
- Linha 9: Sob o controle do programa, a ação de `A` pode ser desfeita através da operação de `Abort()`.
- Linha 11: Caso contrário, a ação atômica é finalizada pela operação `Commit()`, que efetua todas as alterações feitas ao estado do objeto, percorrendo sua lista de objetos e atualizando o repositório de objetos.

4.3 O gerador de *Stubs*

A finalidade do gerador de *stubs* é: a partir de definições de classe, produzir como saída o código `C++` dos *stubs* tanto do cliente quanto do servidor, permitindo que instâncias destas classes sejam acessadas remotamente. O código produzido automatiza o início e o término da execução de servidores, realiza o empacotamento dos parâmetros e executa todas as outras funções necessárias para se fazer as chamadas a métodos remotos.

Assume-se que a entrada para o gerador de *stubs* seja um arquivo com definições de uma ou mais classes `C++` sintaticamente corretas. A interface fornecida por objetos distribuídos é a definida pelo conjunto de métodos públicos de cada classe. Como este arquivo pode conter algumas diretivas para o pré-processador (por exemplo `#include`), deve-se primeiramente pré-processá-lo para removê-las. Isto é feito automaticamente pelo gerador de *stubs*, através da compilação das classes, instruindo-se o compilador de `C++` a apenas pré-processar o arquivo de definições. Após este pré-processamento, o gerador de *stubs* então produz o código dos *stubs* de cliente e de servidor.

4.3.1 Implementação do Mecanismo de Chamadas de Métodos Remotos

Com a utilização do mecanismo de chamada de métodos remotos, um cliente pode invocar métodos em um servidor. Tal como a Figura 4.5 mostra, a troca de mensagens entre os *stubs* cliente e servidor é realizada por objetos `RMCClient` e `RMCServer` respectivamente. As setas indicam as seqüências de chamadas necessárias à execução de uma chamada de

método remoto. As setas de número 1 a 7 correspondem ao envio da chamada do cliente ao servidor, enquanto que as de número 8 a 14 correspondem ao envio das respostas do servidor ao cliente.

Quando uma aplicação envia uma mensagem a um objeto remoto, o método correspondente é invocado no objeto *stub* cliente. Ao invés de conter o código que implementa o método remoto, o *stub* cliente:

1. Empacota em uma mensagem os parâmetros necessários para que o método apropriado seja invocado no servidor.

2. Envia a mensagem ao *stub* servidor através de um objeto `RMCClient` e aguarda até que os resultados do método remoto sejam recebidos. `RMCClient` utiliza-se das operações de envio e recebimento de mensagens implementadas pelo módulo de transporte representado na Figura 4.5 [2].

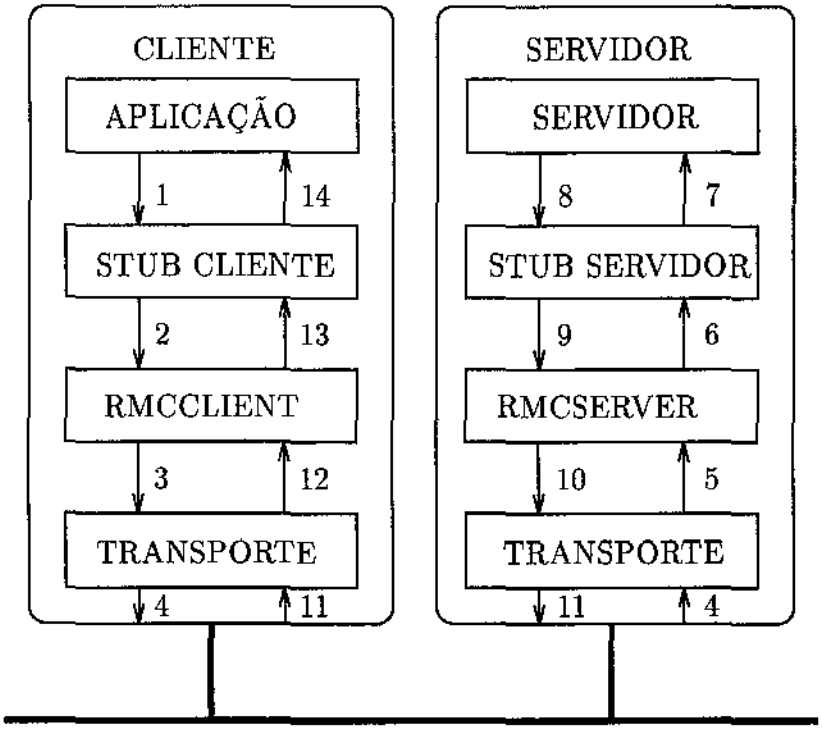

Figura 4.5: Arquitetura de uma Chamada de Método Remoto.

Na porção servidor são realizados os seguintes passos:

1. O objeto *stub* servidor recebe uma mensagem do *stub* cliente, através do módulo de transporte.

2. A mensagem recebida que contém um pedido de execução de método é desempacotada.

3. Se a mensagem é o construtor, então o objeto remoto é criado. A partir deste instante, o *stub* servidor redireciona as mensagens recebidas ao objeto remoto associado.

4. Os resultados são enviados ao *stub* cliente pelo *stub* servidor, através do módulo de transporte.

A criação e destruição de objetos *stub* servidores é feita por um *gerente de objetos* descrito na seção 4.3.2.

A seguir, descrevem-se as classes `RMCClient` e `RMCServer`. Estas classes são implementadas em uma biblioteca de *run time* que deve ser ligada ao código de clientes e servidores.

A classe `RMCClient`

A classe `RMCClient` (Tabela 4.8) é derivada da classe `RMCBuffer`. Essa classe fornece o protocolo de operações necessário à comunicação com objetos das classes `RMCServer` e `ObjectManager` (Seção 4.3.2). As tarefas executadas por um objeto da classe `RMCClient` são:

1. *Iniciar conexão com o servidor.* Em resposta à mensagem `Connect`, parametrizada com o nome do arquivo com o código executável do servidor, um objeto `RMCClient` consulta o servidor de nomes, através de `ClientNS` [2], para obter a identificação da porta do gerente de objetos presente no nó onde se encontra o servidor. Então, envia uma mensagem ao gerente de objetos (Seção 4.3.2) requerendo uma conexão com o servidor e permanece à espera de uma resposta por um tempo determinado, segundo parâmetro da mensagem `Connect`. A execução do método `Connected()` permite verificar se a conexão cliente/servidor está estabelecida.

2. *Enviar mensagens ao servidor.* Ao enviar mensagens `SendRequest` a objetos do tipo `RMCClient`, um cliente invoca um método no objeto servidor. O parâmetro especifica o tempo que `RMCClient` ficará à espera dos resultados enviados pelo servidor.

3. *Interromper conexão com o servidor.* A interrupção de uma conexão cliente/servidor é executada pelo método `Disconnect()`.

A classe `RMCServer`

A classe `RMCServer` (Tabela 4.9) é derivada da classe `RMCBuffer`. Essa classe encapsula informações necessárias à comunicação com objetos das classe `RMCClient` e fornece um protocolo para a execução das seguintes tarefas:

1. *Estabelecer conexão com o cliente.* Cada código servidor produzido pelo gerador de *stubs* possui uma instância da classe `RMCServer`. Quando o gerente de objetos cria

Classe RMCClient ⇐ RMCBuffer		
Atributo	Acesso	Descrição
Addr serverAddr	privado	Porta do servidor
BOOL connected	privado	Conectado?
Método	**Acesso**	**Descrição**
RMCClient()	público	Construtor
RMCClient(char*, unsigned long)	público	Construtor
~RMCClient()	público	Destrutor
BOOL Connect(char*, unsigned long)	público	Inicia conexão
void Disconnect()	público	Termina conexão
BOOL Connected()	público	Conectado?
BOOL SendRequest(unsigned long)	público	Envia mensagem

Tabela 4.8: Classe RMCClient.

o processo que executa o servidor, envia-lhe a identificação de uma porta de entrada do cliente com a qual deve se conectar. Ao enviar a mensagem Connect a um objeto RMCServer, este envia uma mensagem ao cliente, através da porta indicada pelo seu atributo clientAddr, contendo o endereço de uma porta de entrada na qual receberá solicitações. A partir do estabelecimento de conexão entre cliente e servidor, a conexão entre cliente e gerente de objetos é interrompida.

2. *Receber mensagens do cliente.* A captura de mensagens enviadas pelo cliente é realizada em resposta à mensagem ReceiveRequest(), parametrizada com o tempo que o servidor aguarda por uma mensagem do cliente.

3. *Enviar mensagens ao cliente.* O envio de resultados de retorno de um método executado no objeto servidor é feito em resposta à mensagem SendReply().

4.3.2 O Gerente de Objetos

O gerente de objetos é um objeto responsável pela criação de objetos servidores. Após a inicialização do servidor de nomes, tal objeto é ativado em cada nó da rede em que o sistema de ações atômicas é executado.

A classe ObjectManager (Tabela 4.10) fornece um protocolo para as operações necessárias à execução de um gerente de objetos, descritas a seguir:

1. *Receber mensagens de clientes contendo pedidos de conexão com um servidor.* Cada objeto ObjectManager possui uma porta de entrada associada (ManagerPortIN), utilizada para receber mensagens de clientes. A identificação de tal porta é composta pela palavra MANAGER adicionada do nome da máquina em que o gerente de objetos é criado. A criação de ManagerPortIn é realizada pelo método Server(). Cada

Classe RMCServer ⇐ RMCBuffer		
Atributo	Acesso	Descrição
Addr clientAddr	privado	Porta do cliente
BOOL connected	privado	Conectado?
Método	**Acesso**	**Descrição**
RMCServer(Addr&)	público	Construtor
~RMCServer()	público	Destrutor
void ReceiveRequest(unsigned long)	público	Recebe mensagem
BOOL SendReply()	público	Envia mensagem
BOOL Connected()	público	Conectado?
void Connect(Addr&)	privado	Conecta-se com cliente

Tabela 4.9: Classe RMCServer.

ObjectManager permanece em execução, em resposta à mensagem Serve(), à espera de mensagens de clientes.

2. *Execução de processos servidores.* Mensagens recebidas por um ObjectManager contém o nome de um arquivo executável de um servidor e o endereço de uma porta de entrada do cliente à qual o servidor deve se conectar. Em resposta à mensagem CreateServer() parametrizada com o conteúdo da mensagem recebida do cliente, o objeto efetua a criação de um novo processo que executa o código do servidor. Assume-se que o código que implementa cada servidor está em um diretório pré-determinado.

Classe ObjectManager		
Atributo	Acesso	Descrição
PortIn* ManagerPortIn	privado	Para recebimento de mensagens
Método	**Acesso**	**Descrição**
ObjectManager()	público	Construtor
~ObjectManager()	público	Destrutor
void Serve()	privado	Espera pedidos de RMCClient
BOOL StartServer(const char*, Addr&)	público	Executa servidor

Tabela 4.10: Classe ObjectManager.

4.3.3 Diretivas

A fim de assegurar que o arquivo utilizado como entrada permaneça aceitável por compiladores C++ padrão, as diretivas ao gerador de *stubs* são fornecidas na forma de comentários e

devem preceder as declarações de classe às quais se aplicam. Embora algumas sejam conflitantes, mais de uma diretiva pode ser estabelecida. O conjunto de diretivas é o seguinte:

- `@Remote`. Indica que objetos da classe seguinte serão acessados remotamente e, logo, o gerador de *stubs* deve produzir o código de cliente e servidor bem como as respectivas definições.

- `@NoRemote`. O inverso da diretiva acima. Neste caso, nenhum código de chamada de método remoto será produzido para a classe, embora sua interface pública seja acrescida de métodos de empacotamento dependendo das demais opções.

- `@AutoMarshall`. Indica que deve ser produzido automaticamente o código de empacotamento para a classe seguinte, habilitando que instâncias desta classe sejam passadas como argumentos em chamadas de método remoto.

- `@UserMarshall`. Adiciona à interface a assinatura dos métodos de empacotamento, mas não produz o código correspondente, que deverá ser fornecido pelo usuário.

- `@NoMarshall`. Instâncias da classe não podem ser empacotadas.

- `@FreeMemory`. Aplicável somente dentro da definição de uma classe e a métodos membros. Esta opção possibilita um controle explícito de liberação de memória no servidor.

Considerando algum método que retorna um ponteiro como resultado, este ponteiro pode ter várias semânticas de alocação de memória associadas. Ele pode ser simplesmente uma cópia de um ponteiro que é válido somente enquanto o objeto em que o método foi executado existir. Ou, pode ser uma cópia de um ponteiro para um objeto e, quando o ponteiro é retornado, a responsabilidade de liberar a memória do objeto para o qual ele aponta é delegada ao invocador (cliente). Tanto cliente quanto servidor devem conhecer qual alternativa está sendo empregada tal que a utilização de memória seja eficiente. Contudo, se ambos utilizam a mesma semântica, a separação de cliente e servidor em espaços de endereçamento diferentes pode levar ao emprego ineficiente de memória por parte do servidor, no caso em que não for indicado ao gerador de *stubs* qual a semântica a ser utilizada. A especificação da diretiva `@FreeMemory` previne tal problema.

As seguintes diretivas são aplicáveis na declaração de uma lista de parâmetros de métodos membros e modificam apropriadamente a maneira como estes são enviados.

- `@In`. Marca o parâmetro como somente de entrada. É utilizada para todos os parâmetros, exceto ponteiros, referências e *arrays*, a menos que sejam especificados como constantes (`const`).

- `@Out`. Marca o parâmetro como somente de saída. Pode ser utilizada em casos onde o parâmetro é inválido antes da chamada, mas é alterado pelo método invocado.

- `@InOut`. Utilizada para ponteiros, referências e *arrays*, faz com que tais parâmetros sejam enviados por valor (*copy-in, copy-out*) como uma forma de aproximação às chamadas por referências.

A diretivas utilizadas de forma padrão para uma classe são `@Remote` e `@NoMarshall`.

4.3.4 Arquivos de Saída

O gerador de *stubs* produz um conjunto de arquivos para cada classe definida em um arquivo de entrada. No entanto, este funcionamento pode ser alterado através da utilização de diretivas especiais no arquivo de entrada, descritas na Seção 4.3.3.

Seja um arquivo `input.h` que contém a definição de uma classe qualquer chamada `Test` (Figura 4.6). De maneira padrão, o gerador de *stubs* produzirá os seguintes arquivos:

- `input_stub.h` Este arquivo deve ser incluído no código fonte que implementa os métodos de `Test` em lugar de `input.h`. Seu conteúdo é similar ao arquivo de entrada original, exceto que as definições de classes são transferidas para outros arquivos; tais arquivos são incluídos em `input_stub.h`. Outras construções também podem ser removidas; em particular definições de métodos `inline`. Definições externas para os métodos de empacotamento podem ser adicionadas (Figura 4.6).

- `Test_stubclass.h` Este arquivo contém definições das novas classes criadas a partir de `Test`. São elas: `LocalTest`, `RemoteTest` e `ServerTest`. São utilizadas algumas diretivas para o processador a fim de esconder a alteração de nome da classe original.

- `Test_client.cpp` Porção cliente do código que fornece novas implementações de todas as operações públicas da classe original, como chamadas de método remotos (Figura 4.6).

- `Test_server.cpp` Porção servidor do código que decodifica as chamadas recebidas, invoca as operações na classe original e retorna os resultados ao cliente (Figura 4.6).

- `Test_servermain.cpp` Um simples programa principal que cria uma instância da classe servidor e a faz aguardar por pedidos do cliente. Este é o programa que é executado pelo *gerente de objetos.*

- `Test_marshall.cpp` Código de empacotamento/desempacotamento para instâncias da classe `Test`. Este arquivo pode conter código para o empacotamento de ponteiros e referências.

4.3.5 Empacotamento de Parâmetros

Para realizar a transferência de mensagens entre clientes e servidores, os parâmetros e resultados são empacotados em um *buffer* para a transmissão e desempacotados no nó receptor.

O gerador de *stubs* utiliza sempre que possível a mesma sintaxe para empacotamento/desempacotamento de parâmetros de todos os tipos. Utilizando a sobrecarga de operadores << e >> aplicados a *buffers*, fornece métodos (implementados pela classe `RMCBuffer`) para o empacotamento e o desempacotamento respectivamente. A classe `RMCBuffer` contém operações para empacotamento e desempacotamento de todos os tipos básicos `C++`.

Instâncias de Classes

O gerador de *stubs* produz alguns métodos novos para as classes marcadas com a diretiva `@AutoMarshall`. Primeiramente, produz definições apropriadas para os métodos << e >> que permitem o empacotamento/desempacotamento de instâncias da classe, ponteiros e referências para instâncias da classe. A seguir, adiciona à classe a definição de dois outros métodos públicos responsáveis por realizar o empacotamento e desempacotamento dos atributos individuais da classe. Tais métodos possuem a seguinte assinatura:

virtual void marshall(RMCBuffer&);
virtual void unmarshall(RMCBuffer&);

e são declarados como virtuais somente se a classe a ser empacotada é derivada de alguma(s) outra(s) classe(s) ou possui outros métodos virtuais.

Ponteiros e Referências

Estruturas de dados sofisticadas, tais como listas e árvores, podem ser empacotadas/desempacotadas pelos métodos da classe `RMCBuffer`.

Para assegurar que a semântica de referências `C++` sejam satisfeitas, o gerador de *stubs* as converte em ponteiros no cliente e os empacota. Esta conversão ocorre somente se a referência é para um tipo fundamental (`int`, `char`, instâncias de classe, etc.), outras (por exemplo, referência a um ponteiro) são enviadas sem qualquer modificação.

4.3.6 Utilização do Gerador de *Stubs*

A Figura 4.6 mostra um exemplo de como o gerador de *stubs* pode ser utilizado. Seja um arquivo `input.h` que contém a definição de uma classe qualquer chamada `Test`. A fim de que uma aplicação do usuário (`aplicação.cpp`) possa criar e enviar mensagens a objetos remotos de `Test` faz-se o seguinte:

1. O arquivo `input.h` deve ser processado pelo gerador de *stubs*, que produz os seguintes arquivos: `input_stub.h`, `Test_stubclass.h`, `Test_client.cpp`, `Test_server.cpp`, `Test_servermain.cpp`, e possivelmente `Test_marshall.cpp`. A Figura 4.6 não contém `Test_stubclass`, porque este arquivo é incluído em `input_stub.h`.

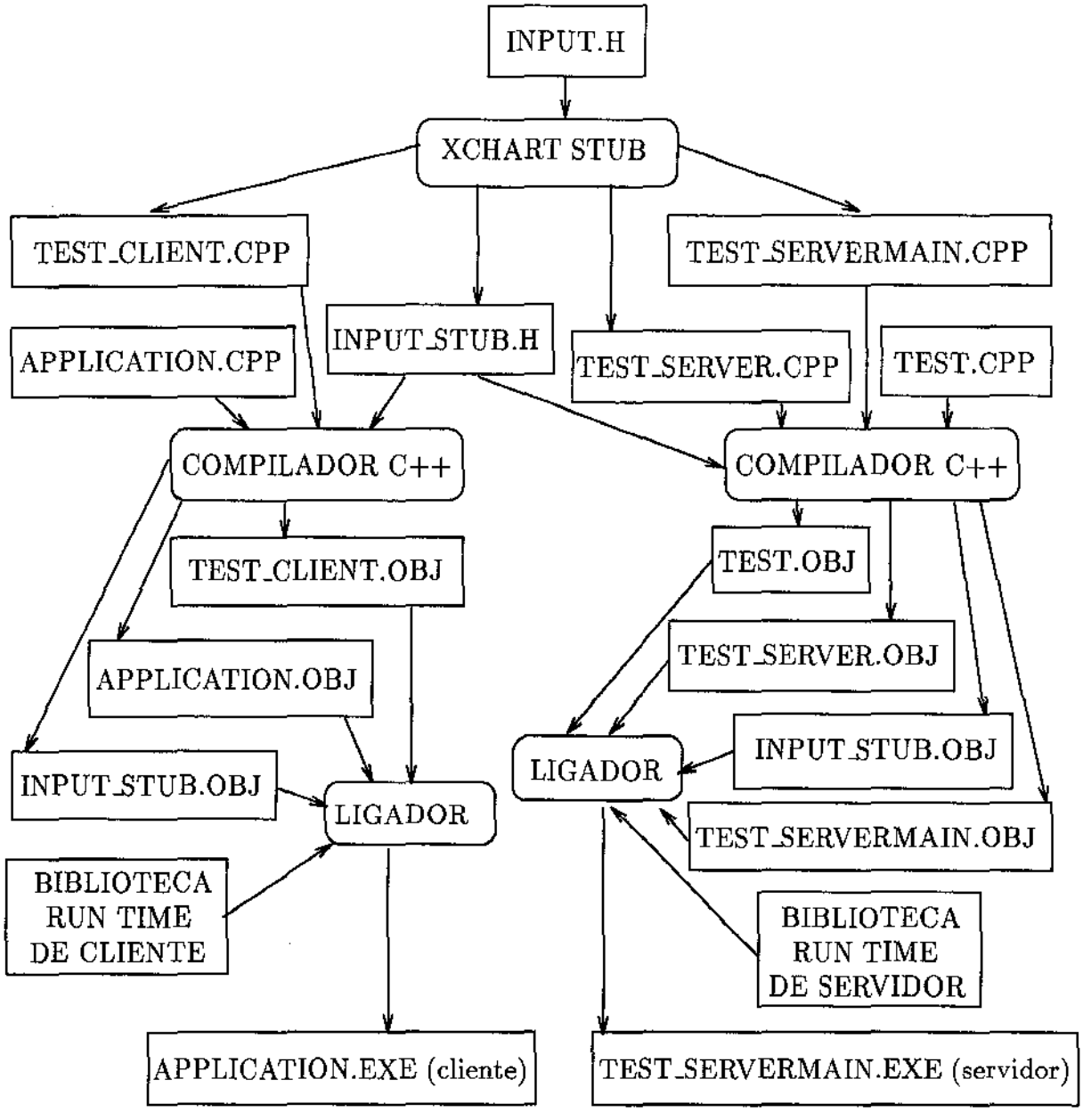

Figura 4.6: Exemplo de utilização do gerador de *stubs*.

2. O arquivo `input_stub.h` deve ser incluído na aplicação (`aplicação.cpp`) que utiliza `Test` e no arquivo que contém o código que implementa `Test` (`test.cpp`), ao invés de `input.h`. Para que a interface de classe correta seja utilizada pelo servidor, deve-se iniciar algumas variáveis para o pré-processador antes de incluir este arquivo. No código que implementa `Test`, troca-se

```
#include "input.h"
```

por:

```
#define TEST_SERVER
#define "input_stub.h"
```

3. A compilação dos arquivos `Test_server.cpp`, `Test_servermain.cpp` e `Test.cpp` produz o código do servidor e o código do cliente é obtido a partir da compilação dos arquivos `Test_client.cpp` e `aplicação.cpp`.

4. Finalmente, é feita a ligação dos códigos fontes com uma biblioteca (Seção 4.3.1) de *run-time* que realiza o transporte de mensagens entre clientes e servidores.

4.3.7 Limitações

O gerador de *stubs* possui as seguintes restrições para a forma das classes e métodos:

- Não se permite variáveis públicas e *friends*. A existência de variáveis públicas em uma definição de classe não satisfaz a propriedade de encapsulamento suposta pelo gerador de *stubs*. Tais variáveis são removidas automaticamente (com a produção de avisos), quando a definição de classe é analisada.

- Não se permite listas de argumentos com tamanhos variáveis. Estas não são permitidas porque o código *stub* precisa saber exatamente o número e tipo de argumentos de cada método para fins de empacotamento.

- Não se permite atributos estáticos. Classes não podem conter atributos estáticos pois o gerador de *stubs* não pode executar corretamente a semântica de tais atributos.

- Não se permite ponteiros para funções. A utilização de ponteiro para funções requeriria a comunicação do servidor com o cliente para executar o código necessário. A implementação atual do mecanismo de chamada de procedimento remoto não possui tal funcionalidade.

- Sobrecarga de operadores limitada. Métodos deste tipo geralmente possuem como parâmetro instâncias da mesma classe. Logo, necessitar-se-ia da mobilidade de objetos, pois estas instâncias poderiam residir em diferentes localidades remotas.

- Semântica de parâmetros. Argumentos dos tipos ponteiro e referência são implementados como valor/resultado. Isto possibilita o surgimento de alguns problemas. Por exemplo, a declaração de uma variável como ponteiro para algum tipo não implica que a variável aponta para uma única instância daquele tipo — ele pode apontar para um *array*. Isto pode ocorrer especialmente no caso de `char*`, ou seja, um ponteiro para um caractere que, por convenção, é utilizado para representar uma cadeia ou lista de caracteres. O gerador de *stubs* sempre considera que um ponteiro para um tipo aponta para somente uma instância daquele tipo — exceto os ponteiros para caracteres.

4.4 Sumário

Neste Capítulo apresentou-se o sistema de ações atômicas distribuídas desenvolvido nesse trabalho. O projeto orientado a objetos do sistema foi fundamentado em requisitos específicos para a implementação do sistema de execução do ambiente **Xchart**: atomicidade, controle de concorrência e comunicação. Na Seção 4.1 descreveu-se a arquitetura do sistema proposto, composta por um módulo de ações atômicas distribuídas, um módulo de comunicação e um servidor de nomes. A implementação orientada a objetos do módulo de ações atômicas distribuídas foi detalhada na Seção 4.2, evidenciando-se os algoritmos e estratégias utilizados para a recuperação de estados de objetos, controle de concorrência (protocolo *strict two-phase locking*) e execução de ações atômicas (protocolo de finalização atômica *two-phase commit*). Na Seção 4.3, tratou-se do gerador de *stubs*, evidenciando-se sua funcionalidade, utilização e limitações, bem como as implementações do mecanismo de chamada de métodos remotos e do gerente de objetos, responsável pela criação de objetos servidores.

Capítulo 5

Exemplos

Neste Capítulo, apresentam-se alguns exemplos de utilização do sistema de ações atômicas distribuídas. Na Seção 5.1, são analisadas diferentes implementações de uma fila de inteiros para mostrar o uso seletivo das propriedades de recuperação, controle de concorrência e atomicidade de operações. O objetivo é ilustrar a flexibilidade do sistema de ações atômicas distribuídas na construção de aplicações que são independentes do ambiente **Xchart**. Na Seção 5.2, descreve-se uma implementação distribuída do conhecido *jogo da velha* (*Tic-Tac-Toe*), cujo diálogo foi desenvolvido com o apoio de **Xchart**. O propósito deste exemplo é demonstrar como o sistema de ações atômicas distribuídas e **Xchart** serão integrados.

Estes exemplos foram escolhidos porque:

- são simples o suficiente para que seus comportamentos sejam facilmente compreendidos. De fato, filas são objetos bastante familiares em computação e o jogo da velha é uma brincadeira cujas regras são igualmente bem conhecidas. Vale salientar que o objetivo, neste Capítulo, *não* é o desenvolvimento de uma aplicação distribuída complexa, mas sim a demonstração dos recursos oferecidos pelo sistema de ações atômicas distribuídas proposto.

- são complexos o suficiente para que seja exibida a utilização *seletiva* das propriedades de recuperação, controle de concorrência e atomicidade de operações (Seção 5.1) do sistema de ações atômicas distribuídas, bem como sua integração com o ambiente **Xchart**.

5.1 Fila

Nesta Seção, descrevem-se exemplos simples de aplicações que fazem uso de uma fila de inteiros. A implementação da fila será feita por uma classe que mantém uma lista duplamente ligada e fornece as operações usuais de filas (inserção no final e remoção do início). Também é possível obter-se o número de nós da fila e trocar ou verificar o valor de seus elementos.

5.1.1 Construindo Filas Compartilhadas

A classe `SharedQueue` (Figura 5.1) implementa filas compartilhadas que podem ser acessadas concorrentemente. Os nós da fila são implementados pela classe `QueueLinks` (Figura 5.2), que encapsula os apontadores para nós da fila (atributos `next` e `prev`, linhas 8 e 9) e um apontador para o elemento da fila (atributo `data`, linha 10). Os elementos da fila são objetos da classe `QueueElement` (Figura 5.3), que encapsula um valor inteiro (linha 7). A classe

```
1: class SharedQueue: public Shared
2: {
3:     public:
4:         SharedQueue(ObjectName);
5:         ~SharedQueue();
6:
7:         void Enqueue(QueueElement&);
8:         QueueElement& Dequeue();
9:
10:        unsigned int QueueSize();
11:        BOOL InspectValue(unsigned int, QueueElement&);
12:        BOOL GetValue(unsigned int, QueueElement&);
13:    private:
14:        QueueLinks* headOfList;
15:        QueueLinks* tailOfList;
16:        unsigned int numberOfElements;
17:};
```

Figura 5.1: Classe `SharedQueue`.

```
1: class QueueLinks
2: {
3:     public:
4:         QueueLinks();
5:         ~QueueLinks();
6:
7:         QueueElement* data;
8:         QueueLinks* next;
9:         QueueLinks* prev;
10:};
```

Figura 5.2: Classe `QueueLinks`.

```
1: class QueueElement
2: {
3:     public:
4:         QueueElement(int);
5:         ~QueueElement();
6:
7:         int value;
8: };
```

Figura 5.3: Classe `QueueElement`.

`SharedQueue` é derivada da classe `Shared` (Figura 4.6), que possui operações para o controle de concorrência de operações em objetos. Através desta herança, objetos (filas) da classe `SharedQueue` podem ser compartilhados. A classe `SharedQueue` possui os seguintes métodos:

- `Enqueue` (linha 7). Adiciona um elemento (`QueueElement`) no final da fila.
- `Dequeue` (linha 8). Remove um elemento (`QueueElement`) do início da fila.
- `QueueSize` (linha 10). Retorna o número de elementos da fila.
- `InspectValue` (linha 11). Verifica qual elemento está em uma posição determinada da fila. O primeiro parâmetro deste método é a posição a ser verificada e o segundo parâmetro é o elemento retornado pelo método, caso seja encontrado.
- `SetValue` (linha 12). Troca o valor de um elemento localizado em uma determinada posição da fila. O primeiro parâmetro deste método é a posição do elemento na fila e o segundo parâmetro é o elemento contendo o novo valor.

O acesso a objetos compartilhados se dá através de suas identidades únicas (nomes únicos). Para que os objetos possam ser localizados, os construtores de classes derivadas da classe `Shared` devem possuir no mínimo um parâmetro (Figura 5.1, linha 4): o nome de um objeto (`ObjectName`). Como a localização de objetos compartilhados é feita no construtor da classe `Shared`, nos construtores de classes derivadas desta classe deve-se fazer uma chamada ao construtor de `Shared`, passando o nome do objeto (Figura 5.4, linha 2).

O controle de concorrência dos objetos pode ser implementado de duas formas: (1) na implementação dos métodos da classe `SharedQueue` ou (2) na aplicação que a utiliza. Em ambos os casos, o programador deverá garantir a obtenção de *locks* antes de efetuar qualquer operação em um objeto, pois não há como determinar se uma operação modifica ou não o estado de um objeto. O tipo de *lock* que é atribuído varia de acordo com as operações a serem realizadas. Quando não são utilizadas ações atômicas, a liberação dos *locks* também é de responsabilidade do programador.

```
1: SharedQueue::SharedQueue(ObjectName QueueName):
2:     Shared(QueueName),
3:     headOfList(0),
4:     tailOfList(0),
5:     numberOfElements(0)
6: {
7: }
```

Figura 5.4: Construtor da classe `SharedQueue`.

Com a implementação do método `InspectValue` da classe `SharedQueue` (Figura 5.5), exemplifica-se como o controle de concorrência de um objeto pode ser implementado em seus métodos. Efetua-se um *lock* de leitura (linha 5), pois não é alterado o estado do objeto, obtém-se o valor desejado (linha 14) e então libera-se o *lock* obtido (linha 18). A chamada ao método `SetLock` deve ser parametrizada com o tipo de *lock* a ser atribuído e, opcionalmente, com o número máximo de tentativas a serem realizadas na obtenção do *lock*. O valor *default* deste parâmetro é 10, ou seja, são realizadas no máximo dez tentativas de obtenção do *lock*. Se acaso o *lock* não puder ser obtido, então o valor retornado pelo método `SetLock` será `REFUSED`. Portanto, a garantia de que *locks* sejam efetivamente atribuídos antes da realização de operações em um objeto deve ser controlada pelo programador (linha 5).

A Figura 5.6 mostra um trecho de programa que implementa uma chamada ao método `InspectValue` (Figura 5.5) por um objeto da classe `SharedQueue` (Figura 5.1). Como o controle de concorrência é feito na implementação do método, não é necessária a obtenção do *lock* antes do envio da mensagem e, conseqüentemente, tão pouco sua liberação.

A outra forma de realização do controle de concorrência de acesso a objetos é fazê-la fora da implementação dos métodos das classes. Supondo-se que o método `Enqueue` tenha sido implementado sem o controle de concorrência, na Figura 5.7 exemplifica-se, através de um trecho de código, uma chamada a este método. Nota-se que antes da chamada a este método (linha 10), o *lock* de escrita é obtido (linha 7) e que após a chamada ele é liberado (linha 14).

5.1.2 Construindo Filas Recuperáveis

A classe `RecoverableQueue` (Figura 5.8) define filas cujas instâncias podem ter seus estados recuperados. A implementação desta classe é similar à implementação da classe `SharedQueue`. Os nós da fila também são implementados pela classe `QueueLinks` (Figura 5.2), que encapsula os ponteiros da fila e um ponteiro para o elemento da fila. Cada elemento é um objeto da classe `QueueElement` (Figura 5.3).

A classe `RecoverableQueue` é derivada da classe `Recoverable` (Figura 4.4), que implementa operações para a recuperação do estado de objetos. Através desta herança, objetos

```
1: BOOL SharedQueue::InspectValue(unsigned int index, QueueElement& element)
2: {
3:     BOOL found = FALSE;
4:
5:     if (SetLock(Lock(READLOCK)) == GRANTED)
6:     {
7:         if (numberOfElements ≥ index)
8:         {
9:             QueueLinks* tmp = headOfList;
10:
11:            for (int i = 0; i < index; i++)
12:                tmp = tmp→next;
13:
14:            element = tmp→data→value;
15:            found = TRUE;
16:        }
17:        found = FALSE;
18:        ReleaseLock(Lock(READLOCK));
19:    }
20:    return found;
21:}
```

Figura 5.5: Método `InspectValue` da classe `SharedQueue`.

```
    ...
SharedQueue queue("Queue1");
QueueElement element;
    ...
queue.InspectValue(3, element);
    ...
```

Figura 5.6: Chamada ao método `InspectValue` da classe `SharedQueue`.

```
1: #include <iostream.h>
2: #include "SharedQueue.h"
3: void main()
4: {
5:     SharedQueue q("Queue2");
6:
7:     if (q.SetLock(Lock(WRITELOCK)) == GRANTED)
8:     {
9:         cout ≪ "Write lock granted\n" ≪ endl;
10:        q.Enqueue(QueueElement(10));
11:    }
12:    else
13:        cout ≪ "Write lock not granted\n" ≪ endl;
14:    if (q.ReleaseLock(Lock(WRITELOCK)) == RELEASED)
15:        cout ≪ "Write Lock released\n" ≪ endl;
16:    else
17:        cout ≪ "Write Lock not released\n" ≪ endl;
18:}
```

Figura 5.7: Programa que utiliza objeto da classe `SharedQueue`.

(filas) da classe `RecoverableQueue` podem ter seus estados recuperados. Os construtores de classes derivadas da classe `Recoverable` devem possuir no mínimo um parâmetro (Figura 5.8, linha 4) — o nome do objeto (`ObjectName`) — e devem fazer uma chamada ao construtor de `Recoverable`, passando-lhe o nome do objeto (Figura 5.9, linha 2).

Com a especialização da classe `Recoverable`, faz-se necessária a definição de dois métodos na classe `RecoverableQueue` para a recuperação do estado de objetos desta classe: o método `SaveState` (Figura 5.8, linha 14) e o método `RestoreState` (Figura 5.8, linha 15), declarados como virtuais na classe `Recoverable`. Os outros métodos da classe `RecoverableQueue` (Figura 5.8, linhas 7 a 12), realizam as mesmas operações que os métodos da classe `SharedQueue` (Figura 5.1).

A implementação dos métodos `SaveState` (Figura 5.10) e `RestoreState` (Figura 5.11) é feita com a utilização das operações de empacotamento e desempacotamento de tipos primitivos de `C++`, fornecidas pela classe `RMCBuffer` (Figura 4.2). O parâmetro destes métodos é um objeto da classe `ObjectState` (Figura 4.3), derivada da classe `RMCBuffer`. No método `SaveState` (Figura 5.10) empacota-se o estado do objeto no objeto fornecido como parâmetro deste método. Primeiramente é empacotado o número de elementos da fila (linha 5), e a seguir, todos os elementos da fila (linhas 6 a 10). No método `RestoreState` (Figura 5.11), o estado do objeto é recuperado a partir do objeto fornecido como parâmetro deste método. Inicialmente, são descartados todos os objetos que eventualmente foram adicionados à fila (linhas 7 a 12) e atribui-se valor 0 aos atributos da fila (linha 13). A seguir, o número de

```
1: class RecoverableQueue: public Recoverable
2: {
3:     public:
4:         RecoverableQueue(ObjectName);
5:         ~RecoverableQueue();
6:
7:         void Enqueue(QueueElement&);
8:         QueueElement& Dequeue();
9:
10:        unsigned int QueueSize();
11:        BOOL InspectValue(unsigned int, QueueElement&);
12:        BOOL SetValue(unsigned int, QueueElement&);
13:
14:        virtual void SaveState(ObjectState&);
15:        virtual void RestoreState(ObjectState&);
16:    private:
17:        QueueLinks* headOfList;
18:        QueueLinks* tailOfList;
19:        unsigned int numberOfElements;
20:};
```

Figura 5.8: Classe `RecoverableQueue`.

```
1: RecoverableQueue::RecoverableQueue(ObjectName QueueName):
2:     Recoverable(QueueName),
3:     headOfList(0),
4:     tailOfList(0),
5:     numberOfElements(0)
6: {
7: }
```

Figura 5.9: Construtor da classe `RecoverableQueue`.

```
1: void RecoverableQueue::SaveState(ObjectState& objectState)
2: {
3:     QueueLinks* tmp = headOfList;
4:
5:     objectState << numberOfElements;
6:     for (int i = 0; i < numberOfElements; i++)
7:     {
8:         objectState << tmp->data->value;
9:         tmp = tmp->next;
10:    }
11:}
```

Figura 5.10: Método `SaveState` da classe `RecoverableQueue`.

elementos da fila é desempacotado (linha 14) e os valores que compõe o estado da fila são adicionados a ela (linhas 15 a 28).

Assim como na implementação do controle de concorrência (exemplo de filas compartilhadas, seção 5.1.1), a recuperação do estado de objetos poder ser feita tanto nos métodos de classes de objetos recuperáveis (derivadas de `Recoverable`), como nas aplicações que utilizam estes objetos. Na Figura 5.12 mostra-se a implementação do método `Enqueue` da classe `RecoverableQueue`. Antes da realização de quaisquer operações em objetos recuperáveis, deve-se obter seus estados mais recentes, e ao final destas operações, realizar a atualização dos estados dos objetos, caso tenham sido alterados. Através de uma chamada ao método `HasObjectState` (linha 5), verifica-se a existência de um objeto cujo nome foi fornecida ao construtor de `RecoverableQueue`. Se o objeto com este nome já existir, seu estado pode ser obtido com uma chamada ao método `GetObjectState` e recuperado por uma chamada ao método `RestoreState` (linha 6). A atualização do estado de um objeto pode ser feita com uma chamada ao método `UpdateState` (linha 16). Os métodos `HasObjectState`, `GetObjectState`, `RestoreState` e `UpdateState` são definidos na classe `Recoverable`.

A Figura 5.13 mostra um trecho de programa que implementa uma chamada ao método `Enqueue` (Figura 5.12) por um objeto da classe `RecoverableQueue` (Figura 5.8). Como a recuperação do estado do objeto é feita na implementação do método, não é necessária a realização de nenhuma operação para a recuperação do estado do objeto antes ou depois da chamada ao método. Neste exemplo é adicionado um elemento com valor 10 a uma fila chamada "Queue3".

Se acaso a recuperação do estado de objetos de uma classe não é implementada em seus métodos, então deve ser feita nas aplicações que utilizam tais objetos. Supondo-se que o método `Dequeue` tenha sido implementado sem a recuperação do estado do objeto, na Figura 5.14 exemplifica-se uma chamada a este método em uma aplicação qualquer. Nota-se que o estado do objeto é obtido antes da chamada aos métodos `QueueSize` e `Dequeue` (linha

```
1: void RecoverableQueue::RestoreState(ObjectState& objectState)
2: {
3:     QueueLinks* tmp = headOfList;
4:     int i;
5:     int value = 0;
6:
7:     for (i = 0; i < numberOfElements; i++)
8:     {
9:         tmp = tmp→next;
10:        delete headOfList;
11:        headOfList = tmp;
12:    }
13:    numberOfElements = headOfList = tailOfList = 0;
14:    objectState ≫ numberOfElements;
15:    for (i = 0; i < numberOfElements; i++)
16:    {
17:        objectState ≫ value;
18:        tmp = new QueueLinks;
19:        tmp→data = new QueueElement(value);
20:        tmp→prev = tailOfList;
21:        tmp→next = 0;
22:        if (tailOfList)
23:            tailOfList→next = tmp;
24:        tailOfList = tmp;
25:        if (headOfList == 0)
26:            headOfList = tailOfList;
27:    }
28:}
```

Figura 5.11: Método `RestoreState` da classe `RecoverableQueue`.

```
1: void RecoverableQueue::Enqueue(QueueElement& element)
2: {
3:     QueueLinks* tmp = new QueueLinks;
4:
5:     if (HasObjectState())
6:         RestoreState(GetObjectState());
7:     tmp→data = new QueueElement(element.value);
8:     tmp→prev = tailOfList;
9:     tmp→next = 0;
10:    if (tailOfList)
11:        tailOfList→next = tmp;
12:    tailOfList = tmp;
13:    if (headOfList == 0)
14:        headOfList = tailOfList;
15:    numberOfElements++;
16:    UpdateState();
17:}
```

Figura 5.12: Método `Enqueue` da classe `RecoverableQueue`.

```
    ...
RecoverableQueue queue("Queue3");
    ...
queue.Enqueue(QueueElement(10));
    ...
```

Figura 5.13: Chamada ao método `Enqueue` da classe `RecoverableQueue`.

9) e atualizado após a alteração do estado do objeto "Queue4" (linha 15). Este exemplo retira todos os elemento de uma fila chamada "Queue4".

```
1: #include <iostream.h>
2: #include "RecoverableQueue.h"
3: void main()
4: {
5:     RecoverableQueue queue("Queue4");
6:     QueueElement element;
7:     unsigned int elements;
8:
9:     if (queue.HasObjectState())
10:    {
11:        queue.RestoreState(queue.GetObjectState());
12:        elements = queue.QueueSize();
13:        for (int i = 0; i < elements; i++)
14:            cout≪"Value at index "≪i≪" is "≪(queue.Dequeue()).value≪endl;
15:        queue.UpdateState();
16:    }
17:}
```

Figura 5.14: Programa que utiliza objeto da classe `RecoverableQueue`.

5.1.3 Construindo Filas Atômicas

Objetos da classe `AtomicQueue` (Figura 5.15) controlam quando as operações das classes `Shared` (Figura 4.6) e `Recoverable` (Figura 4.4) devem ser executadas a fim de fornecer as propriedades de atomicidade, controle de concorrência e recuperação de estado de objetos. A utilização de ações atômicas requer que as classes de objetos, cujas operações serão controladas por acões atômicas, sejam derivadas das classe `Recoverable` e `Shared`.

A classe `AtomicQueue` é derivada das classes `Recoverable` e `Shared`. Por isso, possui como parâmetro de seu construtor o nome dos objetos desta classe (linha 4) e os métodos `SaveState` (linha 14) e `RestoreState` (linha 15). No construtor desta classe (Figura 5.16), faz-se uma chamada aos construtores das classes `Shared` e `Recoverable`, passando o nome dos objetos. A implementação dos métodos `SaveState` e `RestoreState` das classes `AtomicQueue` e `RecoverableQueue` é idêntica, pois o estado dos objetos destas classes é o mesmo.

Na Figura 5.17 exemplifica-se a utilização de ações atômicas na implementação do método `SetValue`. O método `Begin` (linha 6) delimita o início da ação atômica `action`, o método `Commit` a finalização de `action` (linha 21) e `Abort` a interrupção de `action` (linha 25).

```
1: class AtomicQueue: public Recoverable, public Shared
2: {
3:     public:
4:         AtomicQueue(ObjectName);
5:         ~AtomicQueue();
6:
7:         QueueStatus Enqueue(QueueElement&);
8:         QueueStatus Dequeue(QueueElement&);
9:
10:        unsigned int QueueSize();
11:        BOOL InspectValue(unsigned int, QueueElement&);
12:        BOOL SetValue(unsigned int, QueueElement&);
13:
14:        virtual void SaveState(ObjectState&);
15:        virtual void RestoreState(ObjectState&);
16:    private:
17:        QueueLinks* headOfList;
18:        QueueLinks* tailOfList;
19:        unsigned int numberOfElements;
20:};
```

Figura 5.15: Class `AtomicQueue`.

```
1: AtomicQueue::AtomicQueue(ObjectName QueueName):
2:     Recoverable(QueueName),
3:     Shared(QueueName),
4:     headOfList(0),
5:     tailOfList(0),
6:     numberOfElements(0)
7: {
8: }
```

Figura 5.16: Construtor da classe `AtomicQueue`.

Cada um destes métodos retorna um valor que pode ser utilizado para se determinar se uma operação foi realizada com sucesso (linha 21). Quando uma ação atômica é finalizada

```
1: BOOL AtomicQueue::SetValue(unsigned int index, QueueElement& element)
2:{
3:     BOOL result = FALSE;
4:     AtomicAction action;
5:
6:     action.Begin();
7:     if (SetLock(Lock(WRITELOCK), action) == LOCKED)
8:     {
9:         if (numberOfElements ≥ index)
10:        {
11:            QueueLinks* tmp = headOfList;
12:
13:            for (int i = 0; i < index; i++)
14:                tmp = tmp→next;
15:            tmp→data→value = element.value;
16:            result = TRUE;
17:        }
18:    }
19:    if (result == TRUE)
20:    {
21:        if (action.Commit() != COMMITTED)
22:            result = FALSE;
23:    }
24:    else
25:        action.Abort();
26:    return result;
27:}
```

Figura 5.17: Método `SetValue` da classe `AtomicQueue`.

(utilizando-se o método `Commit`), um protocolo de *Two-Phase Commit* é realizado. Se alguma falha for detectada durante a primeira fase deste protocolo, a ação atômica é interrompida durante a segunda fase. Nota-se que se a operação de `Commit` não retornou um resultado esperado (linha 21), não é necessária uma chamada ao método `Abort`. O mecanismo de controle de concorrência é integrado com o mecanismo de ação atômica, ou seja, quando *locks* são atribuídos a objetos, a informação necessária para a liberação destes, no instante correto, é armazenada pela ação atômica (linha 7). Com isso, a ação do programador é facilitada, pois não precisa liberar explicitamente os *locks*. Contudo, se *locks* forem utilizados fora do escopo de uma ação atômica, é da responsabilidade do programador liberá-los quando necessário,

utilizando-se a operação `ReleaseLock`. O programador necessita apenas realizar chamadas ao método `SetLock`, de acordo com as operações a serem realizadas no objeto. As operações para a recuperação do estado do objeto são realizadas pelos métodos de `AtomicQueue`.

A Figura 5.18 mostra um trecho de programa que implementa uma chamada ao método

```
...
AtomicQueue queue("Queue5");
...
queue.SetValue(3, QueueElement(10));
...
```

Figura 5.18: Chamada ao método `SetValue` da classe `AtomicQueue`.

`SetValue` (Figura 5.17) por um objeto da classe `AtomicQueue` (Figura 5.15). Ações atômicas também podem ser utilizadas fora da implementação dos métodos. Supondo-se que o método `Enqueue` não seja atômico, na Figura 5.19 exemplifica-se a utilização deste método por uma aplicação onde é adicionado à fila "`Queue6`" um elemento com valor 10.

5.2 Tic-Tac-Toe

Nesta Seção, descreve-se o desenvolvimento do diálogo da interface de um jogo da velha distribuído, no ambiente **Xchart**. Através de alguns trechos de código que implementam procedimentos do núcleo reativo, demonstra-se como o sistema de ações atômicas é integrado a este ambiente. Como o núcleo reativo encontra-se ainda em implementação, o jogo da velha representa o modo previsto para o acoplamento do sistema de ações atômicas distribuídas ao ambiente **Xchart**.

5.2.1 As Regras do Jogo

Neste Tic-Tac-Toe, vários *usuários* competem contra a *máquina*. As nove posições do jogo são chamadas de *células*. Inicialmente, todas as células estão vazias. A primeira jogada é sempre de um usuário. Após cada jogada válida de um usuário qualquer, a máquina joga. O jogo termina quando houver um empate ou um dos jogadores (usuário ou máquina) vencer. Todos os usuários tem sempre a mesma visão do jogo. A cada jogada efetuada, a nova configuração das células pode ser vista por todos os usuários. Quando um jogo termina, uma mensagem contendo o resultado do jogo pode ser vista por todos os usuários durante alguns segundos, acompanhada por um som de alarme. Após o alarme, um novo jogo é iniciado.

As jogadas dos usuários são representadas por um "**X**", enquanto as da máquina são representadas por um "**O**". Usuários realizam suas jogadas pressionando a tecla esquerda do

```
1: #include <iostream.h>
2: #include "AtomicQueue.h"
3: void main()
4: {
5:     AtomicQueue q("Queue6");
6:     QueueStatus status = NOTDONE;
7:     AtomicAction a;
8:
9:     a.Begin();
10:    if (q.SetLock(Lock(WRITELOCK), a))
11:        status = q.Enqueue(QueueElement(10));
12:    if (status == DONE)
13:    {
14:        if (a.Commit() == COMMITTED)
15:            cout << "Added successfully" << endl;
16:    }
17:    else
18:    {
19:        cout << "An error has occurred" << endl;
20:        a.Abort();
21:    }
22:}
```

Figura 5.19: Programa que utiliza ação atômica.

mouse sobre a célula em que deseja efetuar uma jogada. Se isto for feito sobre uma célula já preenchida, nenhum efeito é produzido. Caso contrário, existem duas possibilidades: uma jogada pode ou não ser efetuada. Como vários usuários podem escolher uma mesma célula em um determinado instante de tempo, apenas um terá sua jogada efetuada. O usuário que efetuou sua jogada terá sua célula marcada com um **X**. Os outros, apenas serão notificados com um som de alarme, avisando que sua jogada não foi efetuada. No entanto, como após qualquer jogada efetuada todos os usuários tem a mesma visão do jogo, aqueles que não efetuaram a jogada terão a célula preenchida, devido a jogada de um outro usuário ter sido aceita. Durante a jogada da máquina, nenhuma jogada de usuário será efetuada.

Um novo usuário pode entrar ou sair de um jogo em qualquer instante. Para entrar no jogo, um usuário apenas executa o programa `TicTacToe`, para sair, pressiona a tecla `Esc`.

5.2.2 Construindo o Diálogo

A construção de um diálogo no ambiente **Xchart** é feita com a definição de diagramas Xchart (Apêndice A). Na Figura 5.20, apresenta-se um diagrama, denominado **TicTacToe**, que descreve o diálogo do jogo. A regra presente no canto inferior direito do estado **TicTacToe** indica que sempre que este estado estiver ativo e ocorrer o evento *repaint*, a acão *Repaint()* deve ser executada (Figura 5.20).

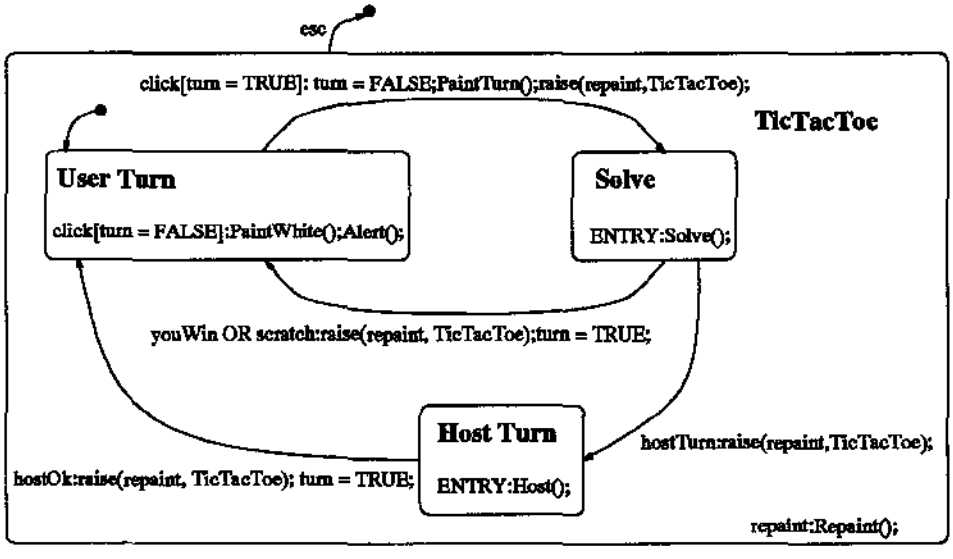

Figura 5.20: Xchart do Tic-Tac-Toe.

O estado **UserTurn** é o estado inicial e representa a espera por jogadas dos usuários. Quando um usuário pressiona o botão esquerdo do *mouse*, o evento *click* é produzido. A regra no interior deste estado determina que quando o evento *click* ocorrer e a variável de controle compartilhada *turn* contiver valor `FALSE`, as ações *PaintWhite()* e *Alert()* devem ser

executadas. A variável *turn* é utilizada para garantir que somente uma jogada de usuário será computada por vez. Quando seu valor for TRUE, o usuário pode efetuar uma jogada, quando for FALSE, não. No último caso, quando o usuário pressionar o botão do *mouse* sobre uma célula, sua jogada não será efetuada mesmo que a célula esteja vazia. O valor de *turn* é FALSE enquanto uma jogada de um usuário estiver sendo computada, ou quando a máquina estiver jogando. Esta variável contém valor TRUE enquanto se espera por uma jogada de um usuário. A aresta que representa a transição entre os estados **UserTurn** e **Solve** é rotulada com uma regra que indica que tal transição só é percorrida se o evento *click* ocorrer e o valor da variável *turn* for TRUE. Neste caso, o valor de *turn* será alterado para FALSE, a acão *PaintTurn()* será executada e o evento *repaint* será enviado para todas as instâncias existentes de **TicTacToe** (ação *raise(repaint, TicTacToe)*). Como a computação destas ações é atômica, garante-se, através desta regra, que quando vários jogadores pressionarem o botão do *mouse* ao mesmo tempo, sobre uma mesma célula vazia, somente um jogador terá sua jogada efetuada.

O estado **Solve** representa a computação de uma jogada de um usuário. A regra no interior deste estado indica que quando ele se tornar ativo, a ação *Solve()* deve ser executada. A execução desta ação pode resultar na produção de três eventos internos: *youWin*, *scratch* e *hostTurn*. O evento *youWin* indica que o jogo foi ganho pelos usuários, o evento *scratch* indica que a computação da última jogada dos usuários resultou em empate, e o evento *hostTurn* indica que nenhum destes resultados foi obtido e a próxima jogada será feita pela máquina. A aresta que representa a transição entre os estados **Solve** e **UserTurn** é rotulada com uma regra que indica que se o evento *youWin* ou o evento *scratch* ocorrer, o evento *repaint* será enviado a todas as instâncias existentes de **TicTacToe** (ação *raise(repaint, TicTacToe)*) e o valor da variável *turn* será modificado para TRUE. A regra que rotula a resta que representa a transição do estado **Solve** para o estado **HostTurn** indica que se o evento *hostTurn* ocorrer, o evento *repaint* deve ser enviado a todas as instâncias de **TicTacToe** (ação *raise(repaint, TicTacToe)*).

O estado **HostTurn** representa uma jogada da máquina. A regra no interior deste estado indica que quando se tornar ativo, a ação *Host()* deve ser executa. A execução desta ação resulta na produção do evento interno *hostOk*. A aresta que representa a transição entre os estados **HostTurn** e **UserTurn** é rotulada com uma regra que indica que tal transição é percorrida quando o evento *hostOK* ocorrer. A ação desta regra resulta no envio de um evento *repaint* a todas as instâncias de **TicTacToe** (ação *raise(repaint, TicTacToe)*) e na alteração do valor da variável *turn* para TRUE.

A saída de um usuário do jogo é representada pelo evento *esc* que rotula a transição que leva ao estado final do diagrama **TicTacToe**.

A partir da definição do(s) diagrama(s) que descreve(m) um diálogo, o ambiente **Xchart** produz um conjunto de classes C++, uma para cada diagrama. A identificação de cada classe corresponde à identificação do diagrama que representa. A hierarquia de classes produzida pela computação do diagrama **TicTacToe** está representada na Figura 5.21. As classes

`Recoverable`, `Shared` e `Xchart` são pré-definidas e fazem parte do ambiente. A classe `Xchart` encapsula métodos e atributos referentes à implementação da semântica do núcleo reativo. Essa classe é derivada das classes `Recoverable` e `Shared` pois alguns de seus métodos são implementados como ações atômicas.

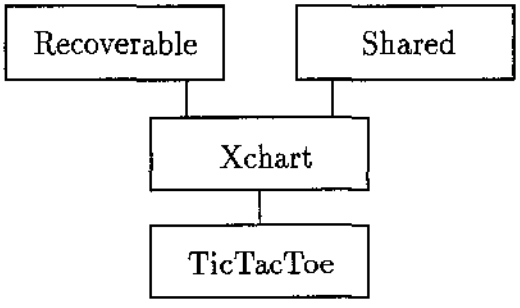

Figura 5.21: Classes do Tic-Tac-Toe.

Classes derivadas da classe `Xchart`, produzidas pelo ambiente, encapsulam atributos (armazenadas em forma de tabelas) específicos ao diagrama que representam, como por exemplo, os eventos a que reage, regras, conjunto de estados, estados iniciais e finais, bem como os métodos que implementam suas ações. Os métodos que acessam tais atributos são declarados como virtuais na classe `Xchart` e sobrecarregados pelas suas classes derivadas. Durante a execução de um diálogo, tais atributos são acessados e modificados de acordo com a configuração do **Xchart** que representam e com a semântica da linguagem *Xchart*.

A definição das assinaturas dos métodos que implementam ações de um diagrama é feita durante sua construção. A classe `TicTacToe` (Figura 5.22) encapsula, portanto, um conjunto de métodos que implementam as ações do diagrama **TicTacToe** (descritos a seguir), mais um conjunto de métodos e atributos necessários à execução do diálogo que representa.

- `Repaint()` Restaura a configuração das células após a realização de uma jogada. Se o jogo terminar, divulga o resultado em forma de uma mensagem acompanhada por um som de alarme durante alguns segundos, e a seguir, recomeça um jogo, onde todas as células estão vazias.

- `PaintTurn()` Preenche uma célula, com um **X** ou com um **O**, de acordo com a última jogada realizada.

- `Alert()` Emite um som de alarme.

- `Solve()` Efetua a computação de uma jogada, podendo produzir um dos eventos: *youWin*, *scratch* ou *hostTurn*. O valor de retorno deste método corresponde ao evento produzido.

```
class TicTacToe: public Xchart
{
   public:
      void Repaint();
      void PaintWhite();
      void PaintTurn();
      void Alert();
      Event Solve();
      Event Host();
         ...
};
```

Figura 5.22: Classe `TicTacToe`.

- `Host()` Realiza uma jogada da máquina, faz uma chamada ao método `Solve()` para computar tal jogada, a seguir, faz uma chamada ao método `PaintTurn()` para preencher a célula correspondente à jogada e, finalmente, produz o evento *hostOk* (valor de retorno do método).

Em diálogos de interfaces de aplicações distribuídas, instâncias de um diagrama Xchart podem estar espalhadas pela rede que compõe o ambiente distribuído. Para que o acesso a tais instâncias seja transparente, as classes que implementam os diagramas de um diálogo devem ser processadas pelo gerador de *stub* do sistema de ações atômicas distribuídas. Esta tarefa é realizada automaticamente pelo ambiente **Xchart**, logo após a compilação de um diálogo. Para cada classe de um diálogo, são produzidos os *stubs* de cliente e de servidor correspondentes. As classes pré-definidas do ambiente, como por exemplo, `Recoverable`, `Shared` e `Xchart`, são processadas pelo gerador de *stubs* uma única vez.

5.2.3 Implementação do Núcleo Reativo

Muitas das operações do núcleo reativo são implementadas como ações atômicas. Nesta Seção, apresentam-se apenas alguns exemplos que ilustram como são utilizadas as classes do sistema de ações atômicas distribuídas pelo núcleo reativo.

Variáveis de Controle Compartilhadas

Na definição dos diagramas de um diálogo, pode-se definir variáveis de controle compartilhadas e seus valores iniciais. Tais variáveis armazenam um valor lógico ou inteiro e podem ser acessadas por quaisquer instâncias destes diagramas. Como são compartilhadas entre as instâncias (distribuídas), as variáveis de controle são implementadas como objetos atômicos.

A classe `SharedVariable` (Figura 5.23) encapsula o valor de uma variável de controle compartilhada (atributo `value`), além de métodos para acesso (método `GetValue`) e alteração (método `SetValue`) deste valor.

```
class SharedVariable: public Recoverable, public Shared
{
   public:
      SharedVariable(ObjectName);
      ~SharedVariable;

      void SetValue(int);
      int GetValue();

      void SaveState(ObjectState&);
      void RestoreState(ObjectState&);
   private:
      int value;
};
```

Figura 5.23: Classe `SharedVariable`.

Como resultado da compilação dos diagramas de um diálogo, é produzida uma tabela contendo as identificações e valores iniciais das variáveis de controle compartilhadas. Na execução do diálogo, é criado inicialmente um conjunto de objetos da classe `SharedVariable` a partir da informação contida nesta tabela, sendo um objeto para cada variável. No decorrer da execução, o acesso e alteração do valor de tais objetos são controlados por ações atômicas. Na Figura 5.24 exemplifica-se, através de um trecho de código, a alteração do valor da variável de controle compartilhada "Xchart1" para 10.

Tabela de Instâncias

Na linguagem *Xchart* há algumas ações pré-definidas para a contrução de diálogos. Entre elas, a ação *raise*, a qual tem várias sintaxes. Na Figura 5.20, a sintaxe utilizada para a ação *raise* foi (*evento*, *diagrama*), significando a emisão do *evento* a todas as instâncias do *diagrama*. Para que a computação desta ação seja realizada pelo núcleo reativo, é utilizada uma tabela contendo, para cada diagrama de um diálogo, a identificação de todas as suas instâncias. Tal tabela, denominada "tabela de instâncias", é atualizada durante a execução de um diálogo. A tabela de instâncias é única para um mesmo diálogo e compartilhada por objetos da classe `Xchart`.

A classe `TableOfInstances` (Figura 5.25) encapsula uma tabela de instâncias e métodos para a alteração e obtenção de seu conteúdo. Uma tabela de instância é um objeto atômico. Portanto, a classe `TableOfInstances` é derivada das classes `Recoverable` e

```
        ...
SharedVariable xchart1("Xchart1");
AtomicAction action;
BOOL ok;
BOOL valueChanged = FALSE;

action.begin();
if (xchart1.SetLock(Lock(WRITELOCK), action) == LOCKED)
{
        xchart1.SetValue(10);
        ok = TRUE;
}
if (ok)
{
   if (action.Commit() == COMMITTED)
      valueChanged = TRUE;
}
        ...
```

Figura 5.24: Chamada ao método `SetValue` da classe `SharedVariable`.

```
class TableOfInstances: public Recoverable, public Shared
{
   public:
      TableOfInstances(ObjectName);
      ~TableOfInstances();

      BOOL AddEntry(Diagram, ObjectName);
      BOOL DetachEntry(ObjectName);
      void Flush();
      InstancesList& GetInstances(Diagram);

      void SaveState(ObjectState&);
      void RestoreState(ObjectState&);
   private:
      Table* tableOfInstances;
};
```

Figura 5.25: Classe `TableOfInstances`.

`Shared`. Métodos da classe `Xchart` que realizam operações em uma tabela de instâncias são implementados como ações atômicas.

O Método `Raise`

A seguir descreve-se a implementação de um dos métodos da classe `Xchart`. O método `Raise` (Figura 5.26) corresponde à implementação da ação *raise* com a sintaxe (*evento*, *diagrama*). O método `Raise` é implementado como uma ação atômica.

Inicialmente a ação atômica `action` é inicializada (linha 8). A lista de instâncias para as quais o evento `event`, especificado como parâmetro do método, será enviado, é obtida a partir de uma consulta à tabela de instâncias (atributo `tableOfInstances`) da classe `Xchart` (linha 11). Como a tabela de instâncias é um objeto atômico e só é realizada uma operação de leitura sobre ele, ou seja, não são alterados os seus valores, antes da obtenção da lista de instâncias, adquire-se um *lock* de leitura sobre a tabela (linha 9). Se acaso o *lock* não puder ser obtido, a variável lógica `raised` se torna falsa, conduzindo ao aborto da ação atômica `action` (linha 28). A lista de instâncias possui a identifição das instâncias. Para cada um dos elementos desta lista, é criado um objeto da classe `Xchart` com a mesma identificação das instâncias (linha 16). O conjunto destes objetos é armazenado na variável `xchart`, criada e inicializada nas linhas 12 e 13, respectivamente. Como tais instâncias já foram criadas em algum outro instante da execução do diálogo, com a criação de um objeto com a mesma identificação é possível compartilhar o estado das instâncias. Antes de se enviar um evento para a lista de eventos de cada instância (linha 18), alterando-se o valor de seu estado, obtém-se um *lock* de escrita para cada instância (linha 17). Se acaso algum *lock* não puder ser obtido antes da emissão do evento para alguma instância, então a variável `raised` se torna falsa (linha 19), impedindo a emissão do evento às instâncias restantes da lista (linha 14). Ao final do método, se a variável `raised` for verdadeira, efetua-se o *commit* de `action` (linha 25), caso contrário, o *abort* (linha 28). Finalmente, o conteúdo da variável `xchart` é eliminado (linhas 29 a 34) e o resultado do método, retornado (linha 35).

5.3 Sumário

Neste Capítulo apresentaram-se duas implementações de aplicações distribuídas que utilizam os recursos do sistema de ações atômicas distribuídas desenvolvido nesse trabalho. Na primeira aplicação, uma fila de inteiros, procurou-se demonstrar o emprego do sistema na construção de uma aplicação distribuída não tolerante a falhas, enfatizando-se a utilização seletiva das propriedades de ações atômicas — recuperação, controle de concorrência e atomicidade de operações — através de definições de novas classes derivadas das classes fornecidas pelo sistema (Seção 5.1). Na segunda aplicação, Seção 5.2, exemplificou-se, através da implementação de um jogo da velha distribuído, como o sistema pode ser integrado ao ambiente **Xchart**. É importante salientar que, em virtude do núcleo reativo encontrar-se ainda em

```
1:BOOL Xchart::Raise(Event& event, Diagram diagram)
2:{
3:    AtomicAction action;
4:    Xchart** xchart = 0;
5:    int instances = 0, i;
6:    BOOL raised = TRUE;
7:
8:    action.Begin();
9:    if (tableOfInstances→SetLock(Lock(READLOCK), action) == LOCKED)
10:    {
11:        InstancesList& list = tableOfInstances→GetInstances(diagram);
12:        xchart = new Xchart*[instances = list.ItemsInList()];
13:        for (i = 0; i < instances; i++) xchart[i] = 0;
14:        for (i = 0; i < instances && raised; i++)
15:        {
16:            xchart[i] = new Xchart(list.Current());
17:            if (xchart[i]→SetLock(Lock(WRITELOCK), action) == LOCKED)
18:                xchart[i]→AddEventQueue(event);
19:            else raised = FALSE;
20:            list++;
21:        }
22:    } else raised = FALSE;
23:    if (raised)
24:    {
25:        if (action.Commit() != COMMITED)
26:            raised = FALSE;
27:    }
28:    else action.Abort();
29:    if (xchart)
30:    {
31:        for (i = 0; i < instances; i++)
32:            if (xchart[i]) delete xchart[i];
33:        delete xchart;
34:    }
35:    return raised;
36:}
```

Figura 5.26: Implementação do método `Raise` da classe `Xchart`.

fase de implementação, ilustra-se no exemplo o modo *previsto* de acoplamento do sistema de ações atômicas distribuídas com o ambiente **Xchart**. Por isso, para a efetiva implementação do jogo da velha, implementou-se também, de forma específica aos propósitos do exemplo, algumas operações do núcleo reativo, encapsuladas na classe `Xchart`.

Capítulo 6

Trabalhos Correlatos

Este Capítulo apresenta, inicialmente, os sistemas Arjuna, Camelot e Clouds e como eles implementam ações atômicas. Na Seção 6.4, descreve-se como o padrão CORBA define os objetos que implementam o mecanismo de ação atômica. Na Seção 6.5, faz-se uma comparação entre estes sistemas e o sistema de ações atômicas distribuídas..

O conceito de ação atômica tem sido utilizado por uma diversidade de sistemas que fornecem suporte ao desenvolvimento de aplicações distribuídas confiáveis. Há basicamente duas abordagens distintas para a implementação de ações atômicas. Na primeira abordagem, o mecanismo de ação atômica é implementado como uma camada de *software* acima do sistema operacional, fornecendo ao usuário uma interface de programação que lhe permite estruturar suas aplicações utilizando acões atômicas. Sistemas como Arjuna e Camelot adotam esta abordagem. Na segunda abordagem, adotada por Clouds, o mecanismo de ações atômicas é implementado a nível de sistema operacional e as ações atômicas são executadas automaticamente, sem a intervenção do usuário. As várias implementações se diferenciam, também, pela forma como tratam questões relacionadas com a sua utilização, independência de plataforma, tolerância a falhas, persistência, protocolos de controle de concorrência e de validação atômica.

6.1 Arjuna

Arjuna [53] é um sistema de programação orientado a objetos que fornece um conjunto de ferramentas para a construção de aplicações distribuídas tolerantes a falhas. O modelo computacional utilizado por Arjuna para estruturar as aplicações é baseado no emprego de ações atômicas encadeadas que controlam operações realizadas em objetos persistentes — modelo de *objetos* e *ações*. Assegurando que objetos são acessados e modificados somente sob o controle de ações atômicas, Arjuna garante a consistência dos objetos mesmo perante a ocorrência de falhas como quebra de máquina e perda de mensagens na rede. Arjuna é constituído de uma hierarquia de classes C++ que implementam controle de concorrência, persistência e recuperação de objetos e controle de ação atômica. Alguns destes

recursos, como armazenamento de objetos e ações atômicas são implementados como objetos e podem ser criados e acessados como qualquer outro objeto C++; outros, como controle de concorrência [54] e recuperação de objetos são utilizados através de mecanismos de herança, próprios de sistemas orientados a objetos, que possibilitam, também, implementações particulares destes recursos de acordo com as necessidades e características específicas das aplicações. Toda entidade no sistema é um objeto. Os objetos podem ser agrupados dinamicamente para melhorar a performance das aplicações que os utilizam, através de técnicas de *clustering*.

6.1.1 Arquitetura

A Figura 6.1 mostra os principais módulos que compõe o sistema Arjuna. O módulo de **Chamadas de Procedimentos Remotos** é usado para a realização de operações em objetos remotos. O **Servidor de Nomes** mantém informações de identificação e localização dos objetos. O **Repositório de Objetos** armazena uma representação consistente dos estados dos objetos. O módulo de **Ação Atômica** possui uma interface que possibilita a utilização de ações atômicas pela aplicação. Estes módulos são descritos a seguir.

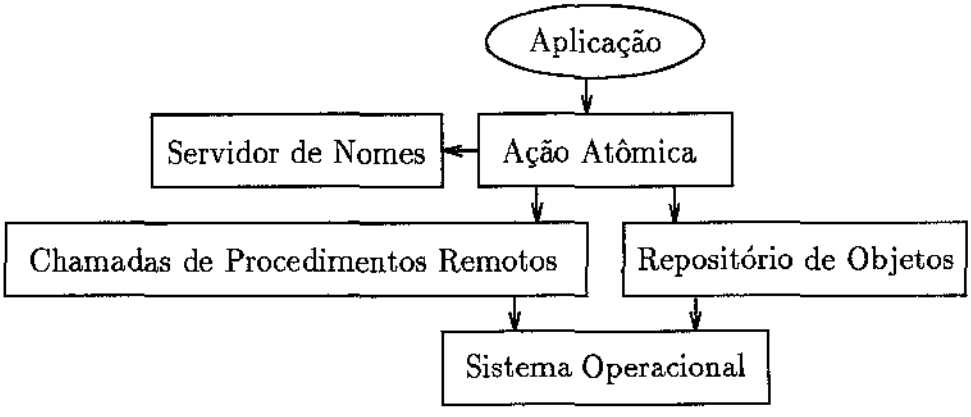

Figura 6.1: Arquitetura de Arjuna.

Módulo de Chamadas de Procedimentos Remotos

O sistema Arjuna adota o modelo cliente-servidor para acessar objetos. Um servidor gerencia o estado de um objeto, definindo e executando operações que são exportadas a clientes. Clientes realizam chamadas a estas operações para acessar estados de objetos mantidos pelo servidor. Se um objeto é remoto ao chamador (cliente), então as chamadas a operações são implementadas como chamadas de procedimentos remotos pelo módulo de chamadas de procedimentos remotos, que utiliza mensagens para transportar os parâmetros de uma

operação e retornar os resultados ao chamador (cliente). Um processo chamado *gerente de processos*, presente em cada máquina do ambiente de execução de Arjuna, é responsável pela criação de processos que são associados a objetos servidores para receberem solicitações de clientes e enviar os resultados de uma operação realizada pelo servidor. O módulo de chamadas de procedimentos remotos possui operações para a criação e destruição destes processos.

O ambiente de programação do Arjuna possui uma ferramenta chamada **Gerador de Stubs** (*Stub Generator*) que processa definições de classes `C++`, cujas instâncias são objetos que podem ser acessados remotamente, e produz o código correspondente aos *stubs* cliente e servidor. Fazendo uso deste gerador de *stubs*, Arjuna fornece as transparências de acesso e localização a quaisquer chamadas a operações em objetos *stub* clientes que realizam o transporte das chamadas até os objetos *stub* servidores (remotos) correspondentes, através do mecanismo de chamada de procedimento remoto. As chamadas a objetos remotos e locais possuem a mesma sintaxe.

Servidor de Nomes

O servidor de nomes mantém a identificação e a localização dos objetos no sistema Arjuna. A indentificação de objetos é útil para se ter acesso a objetos criados anteriormente. Em Arjuna, a identificação de um objeto é dada pelo usuário (programador de uma aplicação distribuída) e então é convertida a um identificador único que o diferencia de todos os demais objetos. Este identificador é mantido pelo servidor de nomes. A localização dos objetos, necessária ao acesso e comunicação de objetos, é obtida a partir de seus identificadores.

Repositório de Objetos

No sistema Arjuna, objetos acessados e modificados por uma aplicação podem se tornar persistentes para que possam ser recuperados após a ocorrência de falhas [19]. O repositório de objetos armazena o estado *passivo* de objetos persistentes em memória estável. A representação do estado passivo dos objetos é independente de arquiteturas de máquinas, permitindo que estados de objetos sejam transportados sem problemas entre memórias estável e volátil, bem como em mensagens de comunicação entre máquinas. Tal representação é implementada pela classe `ObjectState`, cujas instâncias armazenam de forma contínua os atributos que constituem o estado de um objeto. Esta classe possui um conjunto de operações que permitem que a representação de um objeto `C++` seja convertida de e para instâncias `ObjectState`. A função do repositório de objetos é gerenciar instâncias da classe `ObjectState`.

O conjunto de operações fornecidas pelo repositório de objetos, implementado pela classe `ObjectStore`, inclui: `read_state`, que retorna uma instância de `ObjectState` designada por um identificador único, e `write_state`, que armazena uma instância de `ObjectState`. A Figura 6.2 mostra o ciclo de vida e transições de estados de um estado persistente, juntamente

com as operações que produzem as transições. Antes que um objeto passivo seja acessado por uma aplicação, ele deve se tornar *ativo* novamente. O módulo de **ação atômica** ativa um objeto persistente pela operação `read_state` e então obtém o estado do objeto através de `restore_state`. A operação inversa compreende a execução de `save_state`, que empacota o estado do objeto, e de `write_state` que efetua a escrita do estado do objeto em meio estável, tornando-o passivo novamente.

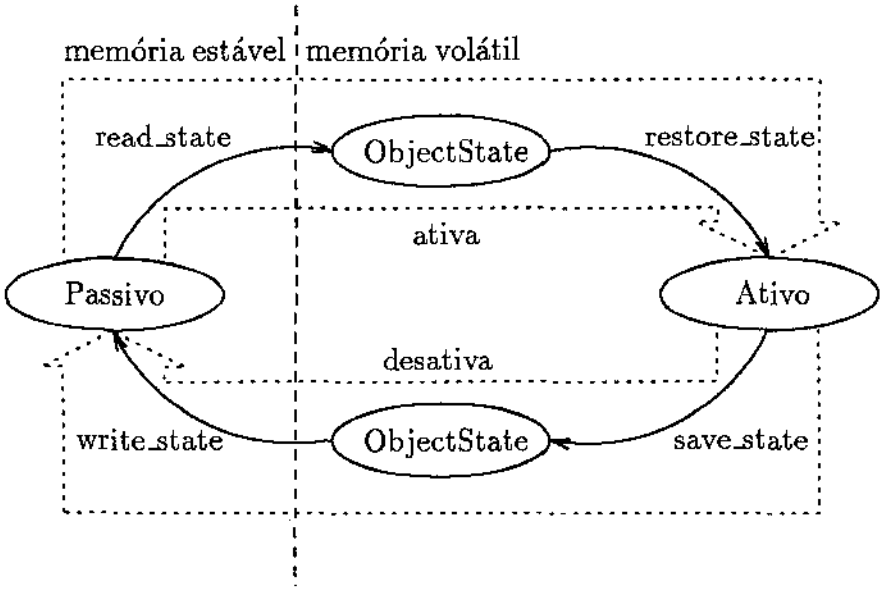

Figura 6.2: Transições de estados de objetos em Arjuna.

Módulo de Ação Atômica

O módulo de ação atômica fornece a interface de programação a uma aplicação. Os mecanismos necessários ao controle de concorrência, persistência, recuperação e controle de ação atômica são implementados pela hierarquia de classes mostrada na Figura 6.3. Para construir uma aplicação de acordo com o modelo de objetos e ações, o usuário declara em seu programa instâncias da classe `AtomicAction`, e então utiliza-se das operações (`begin`, `end` e `abort`) fornecidas por esta classe para organizar suas implementações de ações atômicas. Ações atômicas controlam somente objetos que são instâncias de classes do Arjuna ou de classes definidas pelo usuário e derivadas da classe `LockManager` — classes definidas pelo usuário se tornam membros da hierarquia quando derivadas de `LockManager`, como mostra a Figura 6.3.

A maioria das classes de Arjuna são derivadas de `StateManager`, que possui as propriedades básicas necessárias para a construção de objetos persistentes e recuperáveis. A classe `LockManager` utiliza as operações da classe `StateManager` e `Lock` para fornecer o controle de

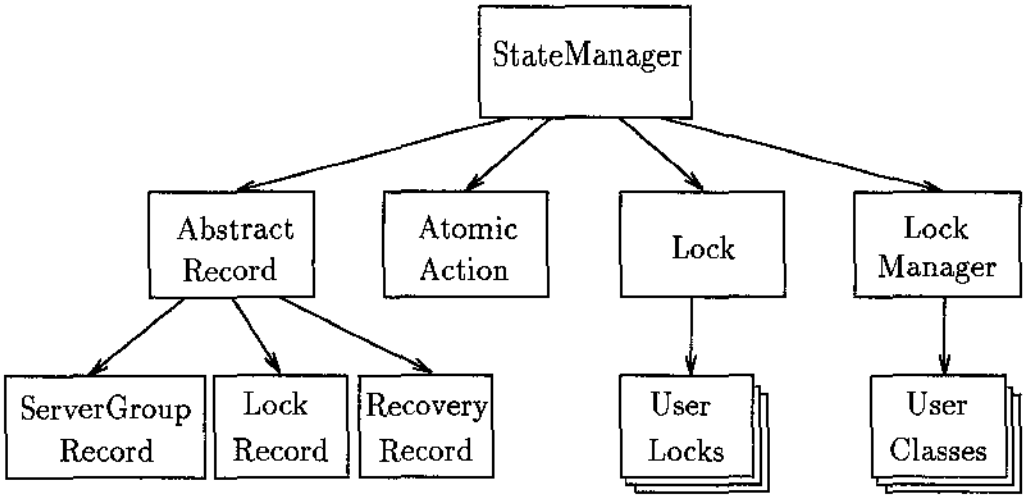

Figura 6.3: Hierarquia de classes do módulo de ação atômica de Arjuna.

concorrência, implementado pelo protocolo *strict two-phase locking* para operações de escrita exclusiva e leitura compartilhada em objetos. Os *locks* são atribuídos a objetos dentro de uma ação atômica e liberados somente quando esta termina (ou é abortada) [60].

As outras classes mostradas na Figura 6.3 implementam o suporte às operações do módulo de ações atômicas; mais detalhes sobre esta hierarquia de classes podem ser encontrados em [59].

6.1.2 Replicação de Objetos

Em Arjuna, cada objeto normalmente reside em somente um repositório de objetos. Contudo, a disponibilidade de um objeto pode ser ampliada replicando-se o seu estado por vários repositórios de objetos. Objetos replicados são gerenciados através de protocolos integrados ao sistema Arjuna que garantem que as cópias dos objetos permaneçam mutuamente consistentes. Mecanismos de transparência à replicação ocultam do usuário o número e o local das cópias de recursos e dos serviços implementados em objetos. Usuários de um objeto replicado geralmente não possuem conhecimento das várias cópias existentes de um objeto, para eles os objetos replicados podem ser identificados individualmente quando requisitam a realização de operações. Embora a solicitação de operações possa ser efetuada concorrentemente por todas as cópias de um serviço replicado, a transparência de replicação garante que os usuários de um serviço recebem somente um único conjunto de resultados. Na interface de programação de serviços replicados, cada recurso replicado é representado por um *grupo de réplicas.*

Os grupos são gerenciados pelo servidor de nomes com a ajuda de uma base de dados especializada chamada *GroupView Database.* Tanto o servidor de nomes quanto a base de dados *GroupView* mantém informações pertinentes aos objetos replicados. Para replicar um objeto, o programador deve criar um grupo de réplicas fornecendo informações a respeito

das réplicas nos dois sistemas.

Os mecanismos de replicação de objetos são flexíveis e dinâmicos, permitindo que mudanças no protocolo de replicação sejam feitas durante o ciclo de vida de um objeto de acordo com as alterações das necessidades do ambiente distribuído e da aplicação. Como todas estas reconfigurações de objetos são feitas utilizando-se ações atômicas, é garantida a consistência dos objetos acessados concorrentemente.

6.1.3 Tolerância a Falhas

Arjuna assume que os componentes de hardware são estações de trabalho (nós) conectados por um subsistema de comunicação (por exemplo, uma rede local). É assumido que um nó trabalha como especificado ou simplesmente para de trabalhar (quebra). Depois de uma quebra um nó é reparado em uma quantidade finita de tempo e se torna novamente ativo. Considera-se que um nó possui meios de armazenamento volátil e estável (não-volátil), que todos os dados armazenados em meio volátil são perdidos quando ocorre uma quebra, e que qualquer dado armazenado em meio estável permanece inalterado após uma quebra. Também é assumido que as falhas como perda e duplicação de mensagens são de responsabilidade do subsistema de comunicação. O módulo de chamadas de procedimentos remotos é responsável por tais falhas, utilizando-se de técnicas de nível de protocolo de rede. Ao suspeitar que um servidor não está respondendo a uma mensagem enviada por um cliente, o módulo de chamadas de procedimentos remotos retorna uma notificação de falha ao chamador (cliente).

6.2 Camelot/Avalon

Camelot [62] é um sistema de ações atômicas distribuído construído sobre o sistema operacional distribuído Mach [1]. Camelot [21] implementa mecanismos de recuperação, sincronização e comunicação necessários ao suporte a ações atômicas distribuídas e a definição de objetos compartilhados. Os serviços de Camelot são acessados através de uma biblioteca — uma coleção de procedimentos e macros escritos em linguagem `C`.

Avalon [29] é uma linguagem que estende os conceitos de `C++`, reduzindo o esforço necessário à construção de aplicações distribuídas confiáveis. Avalon fornece suporte linguístico a objetos atômicos e gerenciamento de ações atômicas com o uso das funcionalidades de Camelot. O compilador de Avalon produz automaticamente código que utiliza as operações de Camelot e Mach.

6.2.1 Arquitetura de Camelot

A estrutura de um nó de Camelot é mostrada na Figura 6.4 [22]. Camelot é executado acima do sistema operacional Mach, que fornece suporte flexível para comunicação entre processos, memória compartilhada, vários *threads* de controle, multiprocessadores e memória virtual.

O conjunto de processos de Camelot são vistos pelo usuário como uma camada de *software* acima do sistema operacional. Utilizando-se a biblioteca de Camelot, podem ser construídas duas classes de programas, os *servidores de dados* e as *aplicações*.

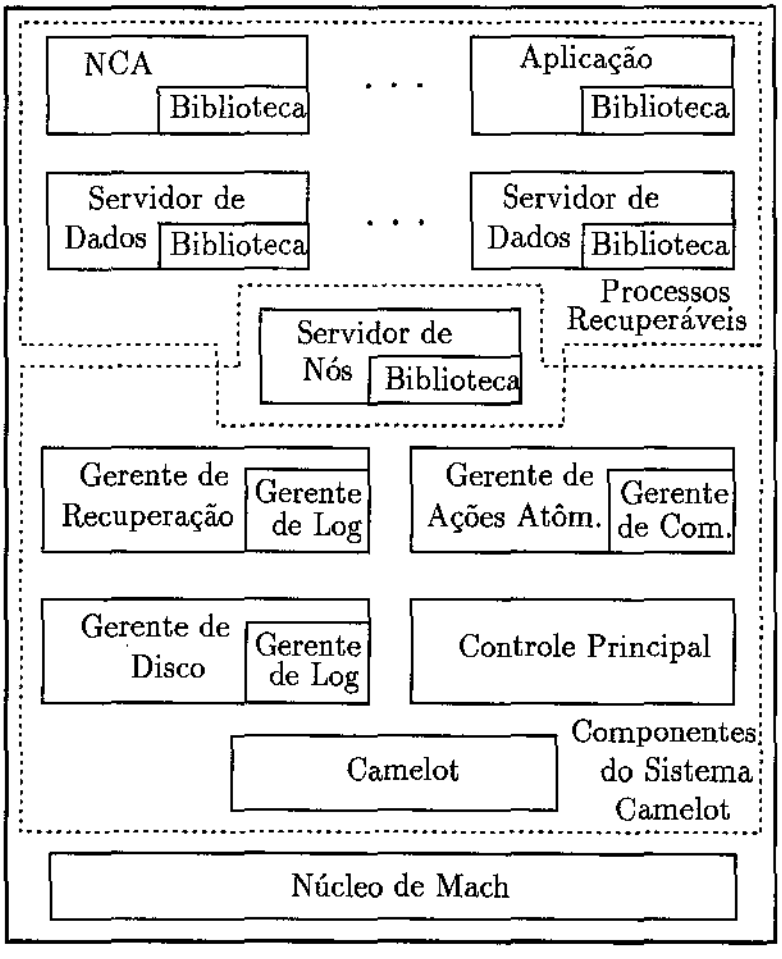

Figura 6.4: Estrutura de Camelot.

Os servidores de dados, ou gerentes de objetos, encapsulam código e dados em diferentes tipos abstratos de dados (classes). Servidores podem declarar e manipular *objetos recuperáveis* para manter dados persistentes em memória virtual recuperável, receber chamadas de procedimentos remotos, iniciar ações atômicas, e realizar chamadas de procedimentos remotos a outros servidores. Os *stubs* de chamadas de procedimentos remotos são produzidos por um gerador de *stubs* de Mach denominado MIG (*Mach Interface Generator*). Cada servidor de dados realiza seu próprio gerenciamento de concorrência tal que possa tirar vantagem de esquemas de obtenção de *locks* baseados em *timestamp* ou de tipos específicos de *lock*.

As aplicações atuam como clientes que podem iniciar ou terminar ações atômicas, e utilizam chamadas de procedimentos remotos para solicitarem a execução de métodos aos

servidores de dados. Os servidores de dados podem ser remotos a uma aplicação, podendo atuar tanto como clientes, quanto como servidores durante a execução de uma ação atômica.

Como mostra a Figura 6.4, a camada de Camelot é constituída dos seguintes componentes:

- **Servidor de Nós.** O servidor de nós é um servidor de dados especial utilizado para armazenar informações de configuração do sistema. Camelot mantém em cada nó uma coleção de dados de configuração para dar suporte à alocação e liberação de novos servidores de dados e do meio de armazenamento recuperável onde servidores de dados armazenam objetos persistentes.

- **Gerente de Ações Atômicas.** O gerente de ações atômicas realiza a execução de ações atômicas distribuídas. O modelo de ações atômicas adotado em Camelot fornece suporte a ações atômicas encadeadas e paralelismo dentro de uma mesma ação atômica através de sub-ações atômicas concorrentes. O gerente de ações atômicas é um processo existente em cada nó de Camelot que se comunica com servidores de dados e aplicações para controlar a realização de toda ação atômica ativa. Utilizando esta informação e comunicando-se com os gerentes de ações atômicas de outros nós, garante que todos os participantes envolvidos em uma transação produzam um resultado comum. São fornecidos dois protocolos de finalização atômica [11]: um bloqueante, baseado no protocolo *two-phase commit* e um não-bloqueante, que é uma combinação dos protocolos *three-phase* e *Byzantine* [63]. O protocolo não-bloqueante possui um *overhead* maior que o bloqueante; contudo a probabilidade de que um objeto permaneça bloqueado durante o protocolo é reduzida.

- **Gerente de Recuperação.** O gerente de recuperação é responsável por restaurar o estado consistente de objetos após a ocorrência de falhas. Camelot fornece servidores de dados com serviços de *logging* para armazenar modificações em objetos. Estes serviços permitem que sejam desfeitas ou refeitas modificações em meio de armazenamento recuperável após as falhas. As funções de recuperação de Camelot incluem a interrupção (*abort*) de ações atômicas e recuperação de servidores, nós e de falhas do meio (*media-failure*).

- **Gerente de Disco.** O gerente de disco realiza o gerenciamento de *buffers* e a manutenção de *log* comum.

- **Gerente de Comunicação.** O gerente de comunicação é responsável pela comunicação na rede e possui um servidor de nomes tal que aplicações possam localizar servidores de dados. Por questões de eficiência, o gerente de comunicação está embutido no gerente de ações atômicas.

- **Gerente de Log.** O gerente de *log* realiza a escrita e a leitura em memória estável. Por questões de eficiência, o gerente de log está embutido no gerente de disco e no gerente

de recuperação. São permitidas apenas operações de leitura nas cópias mantidas pelo gerente de recuperação.

- **Camelot e Controle Principal.** Estes componentes processam a inicialização do sistema e apresentam resultados de depuração.

- **Aplicação de Configuração de Nós** (*Node Configuration Application*)(NCA) [23]. Os usuários de Camelot podem executar o NCA para especificar as seguintes informações de configuração: quais servidores de dados executar, quando reiniciar servidores de dados, cotas de armazenamento para servidores de dados, tempo desejado para a recuperação de nós, a granularidade de alocação para o segmento de recuperação, quais usuários de Camelot têm permissão para criar servidores de dados, e quais usuários têm permissão para criar novas contas de usuário de Camelot. Esta informação de recuperação é armazenada em meio recuperável pelo servidor de nós. O NCA aceita comandos do usuário e realiza chamadas de procedimentos remotos ao servidor de nós para obter ou atualizar informações no banco de dados do servidor de nós.

6.2.2 Avalon

O ambiente de execução de Avalon é baseado nas operações fornecidas por Camelot para executar tarefas específicas do sistema operacional, gerenciamento de ações atômicas, comunicação entre os nós, protocolos de finalização (*commit*), e recuperação automática à falhas (*crash*).

O sistema Avalon/C++ consiste de um conjunto de *programas*, onde cada um é uma aplicação ou um servidor de dados. Um servidor em Avalon é bastante parecido com uma classe em C++. Tal como uma classe, servidores encapsulam alguns dados, e definem as operações que podem ser feitas sobre estes dados. Um cliente realiza chamadas a operações em servidores utilizando a mesma sintaxe de uma invocação a um método em uma classe. Há duas diferenças entre classes e servidores. Primeiro, um servidor possui suporte a concorrência: mais de um cliente pode fazer chamadas a operações em um servidor em um mesmo instante de tempo. Estas operações concorrentes são executadas como *threads* concorrentes dentro do servidor. Segundo, os dados de servidores são persistentes, ou seja, sobrevivem de maneira consistente a falhas (*crashes*). Objetos podem ser *persistentes* ou *voláteis*; objetos persistentes sobrevivem a falhas, enquanto voláteis não. Um servidor reside um único nó, mas cada nó pode conter vários servidores. Aplicações podem criar explicitamente um servidor em um nó chamando um de seus construtores. Ao invés de compartilhar dados diretamente, os servidores se comunicam através de chamadas de procedimentos remotos.

Avalon/C++ possui uma variedade de primitivas para a criação de ações atômicas em seqüência ou em paralelo, e para interromper (*abort*) e finalizar (*commit*) ações atômicas. Cada ação atômica é identificada com um processo. A semântica de ações atômicas é fornecida

por Avalon/C++ através de *objetos atômicos*. Avalon/C++ fornece uma coleção de tipos de objetos atômicos já construídos, e os usuários definem seus próprios tipos atômicos a partir destes objetos, pelo mecanismo de herança.

6.3 Clouds

Clouds [34] é um sistema operacional distribuído tolerante à falhas baseado no modelo de objetos e *threads* para a construção de aplicações confiáveis. O sistema é composto por um conjunto de módulos lógicos que formam, conceitualmente, um sistema centralizado integrado. Os módulos são *servidores de computação* — máquinas disponíveis para uso pelo usuário, *servidores de dados* — repositórios para dados persistentes, e *estações de trabalho do usuário* — que fornecem um ambiente para o desenvolvimento de aplicações e uma interface com os servidores de computação e de dados para a execução das aplicações nestes servidores [18].

O modelo de objetos e *threads* adotado em Clouds é uma adaptação do modelo de programação orientada a objetos. Em Clouds, os blocos básicos para a construção de aplicações são memória persistente (chamados *objetos*) e computações (chamadas *threads*). Os objetos fornecem uma abstração para o armazenamento e compartilhamento de dados, controle de concorrência e sincronização, e *threads*, para a implementação de computações de métodos [18]. Um objeto em Clouds é um espaço de endereçamento virtual persistente que pode conter um (ou mais) segmento(s) de dados e um (ou mais) segmento(s) de código. Um objeto é uma instância de uma classe, e uma classe é um módulo de programa compilado. Todos os dados, programas, dispositivos e recursos em Clouds são encapsulados em objetos. Objetos em Clouds respondem a *invocações* que resultam na execução de *threads*. Uma *thread* pode ser vista como uma unidade de controle que executa métodos dos objetos através de pontos de entrada definidos nestes objetos. Várias *threads* podem executar métodos simultaneamente em objetos, compartilhando o espaço de endereçamento destes objetos. No entanto, uma *thread* não pode fazer acesso a dados fora do espaço de endereçamento em que se encontra em execução. A transferência de controle entre espaços de endereçamento ocorre via invocações de objetos, e a transferência de dados via parâmetros. As invocações a objetos são equivalentes a chamadas de procedimentos a objetos persistentes que não compartilham dados globais. Invocações locais e remotas são diferenciadas apenas pelo sistema operacional.

A implementação de Clouds é dividida em três níveis. No nível mais baixo, há um núcleo, chamado Ra, que fornece os mecanismos para o gerenciamento de memória e escalonamento de processadores [17]. No nível acima há um conjunto de *objetos do sistema* — módulos que implementam os serviços essenciais do sistema (entre eles, o gerente de *threads*, o gerente de objetos do usuário, os clientes e servidores de memória compartilhada distribuída, o gerente de entrada e saída, e o gerente de comunicação). No último nível são implementados, como objetos do usuário, os serviços não críticos, como o servidor de nomes e de *spool*, para

completar a funcionalidade de Clouds.

Na primeira versão do sistema Clouds foi desenvolvida, em 1985, como parte do projeto Clouds, uma linguagem de programação chamada Aeolus para o acesso as funcionalidades de sincronização e recuperação de Clouds [34]. Mais tarde, em 1991, o projeto agrupou novos membros e a solução mudou para o uso de linguagens de programação já existentes. Conseqüentemente, foram desenvolvidos três ambientes de programação baseados em extensões de `C++`, `Eiffel`, e `Clide`; chamados `DC++`, `Distributed Eiffel`, e `CLiDe` respectivamente.

6.3.1 Atomicidade e Tolerância a Falhas em Clouds

A confiabilidade das aplicações distribuídas desenvolvidas em Clouds são garantidas por mecanismos que asseguram a consistência dos dados e progresso das computações mesmo perante a ocorrência de falhas.

A consistência dos objetos é implementada por um mecanismo chamado *controle de consistência baseado em invocação*, que fornece uma abstração na qual programadores especificam uma variedade de semânticas de atomicidade. Este mecanismo garante que a computação de uma *thread* ou é executada por completo ou não é executada, realizando automaticamente a atribuição de *locks* e a recuperação de dados persistentes. A atribuição de *locks* e a recuperação são realizadas a nível de segmento e não a nível de objetos. (Um objeto pode conter vários segmentos de dados. A disposição e número de segmentos são controlados pelo programador. Estes segmentos podem conter ponteiros entre os segmentos e os objetos suportam alocação dinâmica de memória em cada segmento.) Como os segmentos são definidos pelo programador, ele pode realizar o controle da granularidade dos *locks*.

As *threads* podem ser de dois tipos: *s-threads* e *cp-threads* (em Inglês *standard threads* e *consistency-preserving threads*, respectivamente). As *s-threads* não possuem nenhum tipo de controle de concorrência ou de recuperação, enquanto que *cp-threads* suportam propriedades de controle de concorrência e de recuperação bem definidas. Na execução de uma *cp-thread*, o sistema realiza o controle de concorrência automaticamente, em tempo de execução. Os segmentos atualizados são escritos por um mecanismo de *two-phase commit* quando uma *cp-thread* termina. Há duas variações de *cp-threads*: *gcp-thread* e *lcp-thread*. A semântica de *gcp-threads* fornece consistência global e a de *lcp-threads*, local. Quando criadas, todas as *threads* são *s-threads*. Cada operação possui um rótulo estático que declara suas necessidades de consistência.

Clouds adota uma técnica chamada *execução paralela de threads* (EPT) para computações tolerantes à falhas, ou recuperáveis. EPT tenta fornecer processamento ininterrupto face a presença de falhas pré-existentes (estáticas), bem como falhas do sistema e de aplicação que ocorrem quando uma computação recuperável está em progresso (falhas dinâmicas). O sistema EPT trabalha primeiramente replicando todos os objetos críticos em diferentes nós do sistema. O número de nós é outro parâmetro fornecido pelo usuário e reflete o grau de recuperação desejável. Os *threads* ou EPTs são executados independentemente, como se

não tivesse a replicação. Um *thread* invoca um objeto replicado escolhendo certas cópias do objeto. O algoritmo de seleção de réplicas tenta garantir que *threads* separados executam em diferentes nós para minimizar o número de *threads* afetados por uma falha.

6.4 CORBA

CORBA (*Common Object Request Broker Architecture*) [49, 50] foi proposto pela "*Object Management Group*" (OMG), uma organização que tem como objetivo estabelecer uma base para a integração de aplicações distribuídas baseadas em objetos. O padrão especifica os mecanismos utilizados para a comunicação transparente entre objetos envolvendo diferentes plataformas, sistemas e linguagens. O CORBA é composto apenas de especificação e não de *software*.

O padrão é dividido basicamente em cinco partes: o núcleo ORB (*ORB Core*), uma linguagem universal para definir interfaces (IDL — *Interface Definition Language*), uma interface para definir a chamada dinâmica de serviços (DII — *Dynamic Invocation Interface*), um banco de declarações de interfaces (IR — *Interface Repository*) e módulos para adaptação de diferentes modelos de implementação de objetos (OA — *Object Adapters*). Em conjunto, esses componentes possibilitam a integração de sistemas baseados em objetos desenvolvidos com alto grau de heterogeneidade.

O núcleo do padrão é o ORB (*Object Request Broker*), componente de *software* que prevê os serviços para a interação transparente entre objetos. De acordo com a nomenclatura da OMG, um sistema de programação baseado em objetos oferece serviços a clientes, onde um cliente é qualquer entidade que pode pedir a execução de determinado serviço. Um objeto é uma entidade com identidade única, e encapsula recursos, oferecidos através de serviços. Um cliente solicita a execução de um serviço através de uma requisição: a função do ORB é encontrar o objeto que deve responder a uma requisição e transportar os dados envolvidos. A maneira como os clientes e objetos interagem através do ORB é independente da localização destes, da linguagem de programação em que estão implementados, e de qual mecanismo de comunicação fazem uso.

Um ambiente de programação que queira compartilhar serviços através do padrão CORBA deve implementar o seu próprio ORB, com todos os seus serviços correspondentes e interfaces especificadas. Tais implementações podem variar bastante, dependendo das características dos ambientes onde estão baseadas; isso não é importante, desde que as interfaces sejam respeitadas.

A implementação de um objeto guarda os dados daquela instância e o código para executar os métodos. Os objetos são identificados através de uma referência, que é utilizada pelo ORB para localizar o destinatário de uma requisição. Um cliente pode executar serviços de um determinado objeto desde que tenha uma referência para ele.

A linguagem de definição de interfaces (IDL) define os tipos dos objetos através da descrição de suas interfaces. Uma interface é um conjunto de operações, identificadas por

seu nome e lista de parâmetros. O objetivo dessa linguagem é permitir que um cliente utilize as operações oferecidas por um determinado objeto conhecendo apenas sua interface.

Um cliente envia uma requisição a um determinado objeto através de funções *stub* ou através da interface de invocação dinâmica. As funções *stub* servem como intermediárias entre o mecanismo de ativação de métodos do cliente e as funções do ORB que fazem a transmissão de requisições. Os *stubs* são específicos para um determinado tipo de interface. Já o módulo de invocação permite que a chamada seja especificada em tempo de execução, quando o cliente monta uma requisição com a referência do objeto que vai executar a operação, qual a operação a ser executada, e a lista de parâmetros com os tipos correspondentes. A descrição da interface do objeto pode ser obtida através de uma consulta ao banco de interfaces, ou de qualquer outra forma disponível em tempo de execução.

Os adaptadores de objeto são utilizados para acomodar diferentes implementações de objetos, mantendo a representação comum do ORB genérica, ao tratar a heterogeneidade de implementações de maneira particular. Entre as funções de um adaptador, estão a geração e interpretação de referências, a chamada de métodos (através de esqueletos IDL), e a ativação e desativação de implementações.

Finalmente, o módulo de interface ORB é composto de um conjunto de funções que dão acesso direto a serviços do ORB. A maioria dos objetos componentes das aplicações interage com o ORB indiretamente, através dos módulos descritos anteriormente. As funções dessa interface devem ser necessariamente oferecidas por todas as implementações de ORB.

O Objeto de Serviço de Transações

Os objetos de serviço de transações (OTS — *Object Transaction Service*) adotado pela OMG definem interfaces que permitem que vários objetos distribuídos em múltiplos ORBs participem de transações mesmo na presença de falhas [51]. OTS suporta, opcionalmente, transações encadeadas. A função principal do OTS é combinar transações com objetos no nível do ORB. Com o OTS, ORBs fornecem um ambiente para a execução de componentes de atividades críticas.

O OTS de CORBA possui as seguintes características:

- Suporte a transações simples e encadeadas. Todas as implementações de OTS devem suportar transações simples; o suporte a transações encadeadas é opcional.

- Permite que tanto aplicações ORB quanto não-ORB participem de uma mesma transação. OTS permite a interação de objetos de transação com procedimentos de transações que se adaptam ao padrão X/Open DTP.

- Suporta transações que se estendem por ORBs heterogêneos. Objetos em vários ORBs podem participar de uma única transação. Além disso, um único ORB pode suportar vários serviços de transação.

- Suporta interfaces IDL já existentes. Uma única interface suporta tanto implementações baseadas em transações como não baseadas em transações. Classes de objetos que podem participar de transações são derivadas de uma classe abstrata OTS. Tais objetos são aqui chamados de *objetos transacionais*, como uma adaptação dos termos em inglês: *transactional objects.*

Objetos envolvidos em uma transação podem assumir um dos três papéis: *clientes de transações*, *servidores de transações*, ou *servidores recuperáveis.*

- Um cliente de transação realiza um conjunto de chamadas a métodos que são delimitados por demarcações de início e fim de transação. As chamadas podem ser feitas tanto para objetos transacionais, como para objetos não-transacionais. O ORB intercepta a chamada inicial e a direciona ao Serviço de Transação, que estabelece um contexto de transação associado ao cliente. O cliente então envia chamadas a métodos de objetos remotos. O ORB sinaliza, implicitamente, o contexto de transação e o propaga em todas as comunicações subseqüentes entre os participantes da transação. O ORB também é envolvido quando o cliente envia um *commit* ou *abort* e notifica o Serviço de Transação. O cliente não tem conhecimento de todas estas atividades; ele simplesmente inicia transações, realiza suas chamadas a métodos, e termina (*commit*) ou interrompe (*abort*) transações.

- Um servidor de transações é uma coleção de um ou mais objetos cujos comportamentos são afetados por transações, mas que não têm estados recuperáveis, ou recursos próprios. O ORB propaga, implicitamente, o contexto de transação quando estes objetos chamam um recurso recuperável. Um servidor de transações não participa da finalização da transação, mas pode forçar a transação a ser interrompida (*commit*).

- Um servidor que pode ser recuperado é uma coleção de um ou mais objetos cujos dados (ou estado) são afetados pela finalização (*commit*) ou interrupção (*abort*) de uma transação. Objetos recuperáveis são objetos transacionais com recursos a serem protegidos. Objetos recuperáveis realizam chamadas ao método *register_resource* para informar ao Serviço de Transações que um recurso que pode ser recuperado passou a pertencer à transação cujo contexto foi propagado na chamada do cliente. Além disso, objetos recuperáveis possuem métodos que são utilizados pelo coordenador (o coordenador é o serviço de transação) de uma transação para a execução do protocolo *two-phase commit.* O ORB é o mediador desta execução.

OTS é integrado com os mecanismos do ORB. OTS utiliza o ORB para a propagação automática de um contexto de transação. O escopo de uma transação é definido por um *contexto de transação*, que é compartilhado pelos objetos participantes. O contexto de transação é um pseudo-objeto manipulado pelo ORB. O contexto é nulo quando não há

transações associadas a *threads*. O serviço de transação gerencia e propaga o contexto com a ajuda do ORB.

OTS fornece interfaces no formato IDL para os objetos que constituem o serviço de transação. É possível que clientes e objetos transacionais se tornem mais intimamente envolvidos com os detalhes da propagação da transação através de chamadas explícitas a métodos. Contudo, a maioria das transações dependerá do ORB para realizar de forma transparente todas estas tarefas.

6.5 Comparação entre os Sistemas

Todos os sistemas apresentados nas Seções anteriores deste Capítulo são baseados no paradigma orientado a objetos. Em alguns sistemas, a orientação a objetos é aplicada somente no nível de interface com o usuário, em outros, é adotada no projeto e implementação de todos os níveis. Camelot [62] e Clouds [34] são exemplos do primeiro uso de orientação a objetos e Arjuna [53], do último. Em sistemas onde a orientação a objetos não é utilizada em todos os níveis, as vantagens oferecidas por uma interface orientada a objetos podem se tornar limitadas. Por exemplo, em Camelot, a extensão da recuperação de objetos utilizando-se o mecanismo de herança de Avalon/C++ pode se tornar complicada pois as propriedades de Camelot são implementadas como rotinas e não como objetos. O não emprego da orientação a objetos no projeto deste sistema também inibe o desenvolvimento de classes da aplicação via herança múltipla [43]. Ao contrário de Clouds e de Camelot, o *sistema de ações atômicas distribuídas* (Capítulo 4) é todo orientado a objetos. Esta é uma característica que permite a utilização de seus recursos através do mecanismo de herança (simples ou múltipla), além de possibilitar o acréscimo de novas funcionalidades ao sistema com a definição de novas classes, derivadas ou não das classes já existentes.

Arjuna também é todo orientado a objetos, no entanto, Arjuna não separa as características de recuperação de estados de objetos, controle de concorrência e persistência em classes distintas, preferindo fornecer declarações de tipos de dados "atômicos". Tais mecanismos são fornecidos por Arjuna através da interface da classe *LockManager*. Com isso, o usuário do sistema não pode optar pela utilização seletiva dos mecanismos que o sistema provê. O mesmo acontece com Clouds, que separa o controle de concorrência do mecanismo de recuperação de estado, mas não separa a recuperação de estado do mecanismo que fornece a persistência. Em Avalon, todos os objetos (servidores) possuem controle de concorrência, impedindo que o usuário opte ou não por esta funcionalidade. Diferentemente de Arjuna, Clouds e Avalon, o *sistema de ações atômicas distribuídas* permite a utilização seletiva de suas funcionalidades (controle de concorrência, recuperação de estado de objetos e atomicidade) de acordo com as características particulares de aplicações construídas com base neste sistema.

Tal como Arjuna e Avalon, o *sistema de ações atômicas distribuídas* possui uma interface de programação que permite ao programador estruturar suas aplicações utilizando operações

atômicas delimitadas por primitivas de início (*begin*), término (*commit*) e aborto (*abort*). Diferentemente, Clouds executa uma ação atômica automaticamente, isto é, a finalização (*commit*) de uma ação atômica ocorre em decorrência do término de execução de uma *thread*, restringindo o nível de controle que programadores podem exercer sobre o início e o fim de cada ação atômica.

Entre as semelhanças de Arjuna, Camelot/Avalon, Clouds e o *sistema de ações atômicas distribuídas* está a utilização do protocolo *two-phase commit* para a finalização de ações atômicas. Além deste protocolo, Camelot também fornece um protocolo de finalização atômica não bloqueante, que é uma combinação dos protocolos *three-phase commit* e *Bizantine*. Quanto ao protocolo utilizado para o controle de concorrência, o *sistema de ações atômicas distribuídas* utiliza o mesmo que Arjuna: *strict two-phase locking*. Em Camelot/Avalon, o controle de concorrência é efetuado pelos servidores: operações concorrentes são executadas como *threads* concorrentes dentro de um mesmo servidor. Em Clouds, *threads* podem ou não possuir controle de concorrência. Nas que possuem (*cp-threads*), o controle de concorrência é exercido automaticamente em tempo de execução.

Uma outra semelhança entre Arjuna, Camelot/Avalon, Clouds e o *sistema de ações atômicas distribuídas* é a utilização de chamadas de procedimentos remotos para a comunicação entre objetos. Arjuna, Camelot e o *sistema de ações atômicas distribuídas* empregam um gerador de *stubs* para a geração automática do código de clientes e de servidores.

O não fornecimento de mecanismos para a tolerância a falhas é uma das característica que distingue o *sistema de ações atômicas distribuídas* dos outros sistemas apresentados neste Capítulo. Arjuna, Camelot/Avalon e Clouds possuem mecanismos para o armazenamento e recuperação de objetos em memória estável, enquanto que o *sistema de ações atômicas distribuídas* realiza o armazenamento e recuperação do estado de objetos apenas em memória volátil.

Algumas características do *sistema de ações atômicas distribuídas* podem ser comparadas ao *Object Transaction Service* (OTS) definido por CORBA. Uma comparação precisa deveria levar em conta não só as propriedades que CORBA recomenda para o OTS, mas sim as característica particulares de implementações do OTS. Tanto o *sistema de ações atômicas distribuídas* como o padrão CORBA são baseados no modelo cliente-servidor. O *sistema de ações atômicas distribuídas* fornece suporte a transações atômicas simples, CORBA prevê que todas as implementações de OTS devem fornecer ações atômicas deste tipo; o suporte a ações atômicas encadeadas é opcional. Tanto no *sistema de ações atômicas distribuídas* como em CORBA a definição de interface de objetos servidores é feita utilizando-se uma linguagem de definição de interfaces. No *sistema de ações atômicas distribuídas* a linguagem utilizada para a definição de interfaces é a própria linguagem de programação utilizada na implementação do sistema (C++), enquanto que a recomendação CORBA utiliza uma linguagem de definição de interfaces (IDL - *interface Definition Language*) que é diferente das linguagens de programação adotadas para a implementação de protótipos CORBA.

6.6 Sumário

Neste Capítulo, apresentaram-se alguns sistemas que implementam o mecanismo de ação atômica. A seguir, descreveu-se como o padrão CORBA define os objetos que fornecem o serviços de ações atômicas. Finalmente, fez-se uma comparação entre estes sistemas e o sistema de ações atômicas distribuídas, sem o intuito de explorar os seus méritos, mas sim de ressaltar as semelhanças e diferença entre eles.

Capítulo 7

Conclusões

Neste trabalho desenvolveu-se um sistema de ações atômicas distribuídas com o propósito de fornecer os mecanismos necessários à implementação de interfaces distribuídas, construídas no ambiente **Xchart**. Na Seção 7.1, apresenta-se uma análise deste sistema, considerando-se suas principais características e funcionalidades. A seguir, na Seção 7.2, relacionam-se algumas sugestões de como este trabalho pode ser melhorado e ampliado.

7.1 Análise do Sistema

O ambiente **Xchart** está sendo construído para facilitar a construção de interfaces de sistemas interativos, inclusive sistemas interativos distribuídos. A funcionalidade do diálogo de uma interface é definida por objetos gerenciados por um sistema de execução. Em uma configuração distribuída do diálogo, os objetos encontram-se espalhados pelos computadores do sistema distribuído. Para facilitar a implementação do sistema de execução, faz-se necessária a criação de um componente de *software* que atenda aos requisitos advindos da interpretação de diálogos (Xcharts).

A solução proposta neste trabalho para o projeto e implementação do componente de *software* requerido por **Xchart** fundamenta-se no modelo computacional de ações atômicas distribuídas e objetos, no qual as operações de acesso, localização, comunicação e controle de concorrência de objetos são todas integradas de forma modular e flexível em um componente de *software* orientado a objetos.

O emprego de ações atômicas justifica-se devido as propriedades que uma ação atômica garante a um conjunto de operações — atomicidade, consistência, isolamento e persistência de resultados. De outro lado, a programação orientada a objetos vem sendo bastante utilizada no desenvolvimento de sistemas complexos de computação porque possibilita uma modelagem do mundo real de uma forma mais próxima da visão que o homem tem desse mundo. Os conceitos da programação orientada a objetos permitem que a abstração de um universo constituído de objetos que interagem através da troca de mensagens seja transportada para as fases de projeto e implementação dos sistemas. Como resultado, os programas

desenvolvidos utilizando-se a programação orientada a objetos são mais facilmente entendidos e compartilhados por outras pessoas, com redução dos esforços de programação e de manutenção. Em ambientes de programação distribuída, os componentes de um sistema distribuído podem ser natural e convenientemente definidos em termos de objetos.

O sistema de ações atômicas distribuídas é constituído por um conjunto de classes que implementam as funcionalidades necessárias ao processamento de ações atômicas. A interface de programação fornecida pelo sistema permite a definição de classes de objetos que possuem algum subconjunto das propriedades de ações atômicas. As classes foram projetadas e implementadas de tal forma que as propriedades podem ser adicionadas a classes de maneira seletiva, de acordo com as características particulares de cada aplicação, utilizando-se herança.

A linguagem de programação utilizada na implementação do sistema de ações atômicas distribuídas foi `C++`. Apesar de `C++` ser orientada a objetos, não é uma linguagem destinada à programação de aplicações distribuídas. As funcionalidades do sistema de ações atômicas distribuídas são fornecidas por classes de objetos e não por construções da linguagem. Em `C++` não há persistência, nem tão pouco o cónceito de identidade de objetos é implementado independentemente de suas localizações. Portanto, a correta utilização do sistema envolve algum esforço por parte dos programadores. Por exemplo, as operações `SaveState` e `RestoreState`, necessárias à recuperação de estados de objetos, devem ser implementadas pelo programador. Da mesma forma, uma ação atômica deve ser delimitada pelos métodos `Begin()` e `Commit()/Abort()`; por exemplo, é inadimissível que uma ação atômica inicie (*begin*) e não termine (*commit/abort*). Os exemplos da Seção 5.1 esclarecem como todos os mecanismos oferecidos pelo sistema podem ser utilizados.

O controle de concorrência oferecido pelo sistema de ações atômicas distribuídas é realizado por uma implementação do protocolo de *strict two-phase locking*, onde os tipos de *lock* podem ser compartilhados (leitura) ou exclusivos (escrita). Contudo, o usuário do sistema pode definir tipos particulares de *lock* de acordo com suas necessidades. A recuperação do estado de objetos é realizada por um mecanismo de recuperação baseado em estado, no qual as cópias dos estados dos objetos são armazenadas em memória volátil. O sistema também fornece uma interface que permite definir o escopo de uma ação atômica utilizando-se operações de *begin*, *commit* e *abort*. As ações atômicas são implementadas por um protocolo de *two-phase commit*. A comunicação entre objetos remotos é feita pelo envio de chamadas a procedimentos remotos. Os procedimentos necessários à comunicação transparente entre objetos são produzidos automaticamente por um gerador de *stubs*.

A partir dos requisitos de distribuição de **Xchart** e dos mecanismos fornecidos pelo sistema de ações atômicas distribuídas, pode-se concluir que os objetivos do trabalho foram satisfeitos; isto pode ser verificado pelos exemplos da Seção 5.2.

7.2 Extensões Futuras

O sistema de ações atômicas distribuídas, embora desenvolvido especificamente para o ambiente **Xchart**, pode ser utilizado como ferramenta de apoio à construção de várias outras aplicações distribuídas, como demonstrou-se com os exemplos da Seção 5.1. No entanto, a atual implementação deste sistema não possui mecanismos de tolerância a falhas, restringindo sua utilização em aplicações não-tolerantes a falhas.

Várias melhorias e extensões podem ser incorporadas ao sistema de ações atômicas distribuídas, de maneira a torná-lo mais genérico e, conseqüentemente, utilizável em um número maior de aplicações.

Uma das melhorias que podem ser adicionadas ao sistema de ações atômicas distribuídas seria a implementação de ações atômicas encadeadas. Isto poderia ser feito definindo-se uma nova classe derivada da classe `AtomicAction` (Figura 4.7), na qual os métodos `Begin`, `Commit` e `Abort` deveriam ser sobrecarregados para fornecerem esta funcionalidade específica. Para que a interface de utilização de ações atômicas não fosse alterada com a adição desta especialização, prevista no projeto da classe `AtomicAction`, tais métodos foram declarados como `virtuais` nesta classe.

O protocolo de finalização atômica *two-phase commit* implementado pelo sistema de ações atômicas distribuídas, através da classe *AtomicAction*, pode diminuir a concorrência de utilização dos objetos devido ao retardamento da liberação dos *locks*. Protocolos mais sofisticados, como o *tree-phase commit* [61], poderiam ser implementados com a adição de novas classes ao sistema.

A possibilidade de ocorrência de *deadlocks* é contornada por *timeouts* no sistema de ações atômicas distribuídas. Apesar desta técnica ser simples, ela é adequada a um grande número de aplicações. Contudo, o dimensionamento inadequado do tempo de *timeout* pode reduzir o desempenho de uma aplicação. Por exemplo, ele pode ser muito grande, retardando o término de uma operação de *lock*, ou muito pequeno, podendo causar o aborto de uma ação que não esteja realmente envolvida em um *deadlock*. Uma outra técnica para o controle de *deadlocks* que poderia ser adicionada à já existente no sistema de ações atômicas distribuídas seria a detecção de *deadlocks* através de grafos de dependência [6]. A implementação do mecanismo de detecção de *deadlocks* poderia ser feito com a criação de uma nova classe que forneceria as operações necessárias à manutenção dos grafos de dependência.

Para dar suporte à tolerância a falhas, seria necessária a implementação de mecanismos para o armazenamento e recuperação de estados de objetos em meio estável. Neste caso, pode-se optar por duas alternativas: fornecer uma interface de programação ao usuário que permita escolher se os objetos serão ou não persistentes, ou, então, uma em que todos os objetos são persistentes. A implementação da primeira alternativa pode ser feita criando-se uma nova classe, derivada de `Recoverable`, cujo construtor teria um parâmetro que permitiria ao usuário expressar a sua escolha. Neste caso, seria necessária também a criação de uma nova classe que realizasse as mesmas operações fornecidas pela classe `ObjectRepository`,

mas que ao invés de armazenar o estado do objeto em memória volátil, ela o armazenaria em memória estável. Como o estado de um objeto é representado por um objeto `ObjectState`, esta poderia ser a unidade de armazenamento do repositório não-volátil. Os arquivos contendo os estados dos objetos poderiam ser organizados de acordo com suas identidades, por exemplo, em uma estrutura de árvore-B. A implementação da segunda alternativa necessitaria apenas que as operações da classe `ObjectRepository` fossem realizadas em memória estável, como comentado anteriormente, e não em memória volátil.

Uma outra alternativa para a questão da tolerância a falhas, seria o acréscimo de mecanismos de replicação de objetos ao sistema de ações atômicas distribuídas. Neste caso, o armazenamento de objetos em memória estável poderia ser efetuado em *background*. Ao invés de armazenar uma cópia do estado de um objeto em meio estável, seriam feitas várias cópias do estado do objeto em memória volátil de diferentes máquinas. Esta solução poderia aumentar o desempenho do sistema, considerando-se que o tempo necessário nas operações de acesso às cópias armazenadas em meio volátil (memória principal) seria menor em relação ao tempo necessário às armazenadas em meio estável (disco).

O sistema de comunicação utilizado na construção do sistema de ações atômicas distribuídas também fornece mecanismos para a comunicação em grupo [2]. Uma possível melhoria na implementação do protocolo *two-phase commit* seria a utilização da comunicação em grupo. Com esta modificação, os participantes de uma ação atômica formariam um grupo e o protocolo de finalização da ação atômica poderia ser modificado para enviar uma mensagem para o *grupo*, ao invés de enviar mensagens para cada participante da ação atômica.

Finalmente, o sistema de ações atômicas distribuídas poderia ser utilizado como referência para a implementação do *Object Transaction Service* (OTS) previsto pelo padrão CORBA.

Apêndice A

A Linguagem Xchart

O projeto da linguagem *Xchart* decorre de experiências anteriores com a utilização da linguagem *Statecharts* para a especificação do controle de diálogo de interfaces [39]. Originalmente proposta para a descrição de sistemas reativos[1] em geral, *statecharts*, no entanto, possui algumas limitações que a tornam inadequada para a especificação de diálogos de interfaces [40]. *Xchart* é uma linguagem gráfica apropriada à descrição de diálogos de interfaces de sistemas interativos e apesar de ser sintaticamente similar à *Statecharts*, é bastante diferente em termos semânticos.

Xchart apresenta uma grande vantagem em relação às linguagens tradicionais para a descrição de interfaces, pois permite o desenvolvimento incremental e interativo do diálogo. A especificação de um diálogo em *Xchart* é formada por um conjunto de componentes modulares, os Xcharts, que se interagem através de eventos e de memória compartilhada. Um Xchart é descrito em termos da linguagem *Xchart* em um diagrama hierárquico, definido e modificado individualmente, composto de estados cooperantes sujeitos a transições disparadas por eventos e, possivelmente, de outros elementos que a linguagem oferece para a descrição de diálogos. Em uma execução distribuída de um diálogo, as instâncias dos Xcharts que o descrevem estão espalhados nos diferentes nós do ambiente distribuído.

Neste Apêndice apresentam-se informalmente os principais elementos definidos pela linguagem *Xchart*, descreve-se o conceito de instância de *Xchart*, e finalmente, alguns recursos definidos pela linguagem para a especificação de diálogos de interfaces.

A.1 Principais Elementos

A seguir, definem-se os principais elementos da linguagem *Xchart*.

- **Estado.** Um estado é representado graficamente por um retângulo de cantos arredondados e sua identificação escrita em negrito. Um estado identifica um contexto, ou seja, uma situação em que se encontra o sistema, ou parte dele, num determinado instante de tempo durante a sua execução. A Figura A.1 apresenta os três tipos de estados:

[1] Interfaces representam um subconjunto do universo de sistemas reativos.

Básico, Exclusivo e **Concorrente.** Estados do tipo Básico não são decompostos

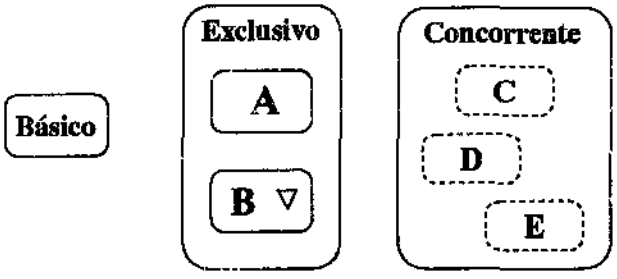

Figura A.1: Tipos de estados em *Xchart.*

em subestados e são representados por retângulos de contorno contínuo. Os estados **A**, **B**, **C**, **D** e **E** também são do tipo básico. Quando um estado não é básico e seus subestados não são descritos onde ele aparece, um pequeno triângulo, como o existente no interior do estado **B**, indica que a descrição completa deste estado encontra-se em outro lugar da especificação. Subestados de um estado **Exclusivo** são ativados por um critério de exclusão mútua e representados por retângulos de contorno contíguo. Subestados de um estado **Concorrente** são concorrentes, sendo ativados e desativados simultaneamente, e são representados por retângulos de contorno tracejado.

Evento	Quando ocorre
ch(v)	alteração do valor de v
en(s)	estado s se torna ativo
entry	*en(s)* para o estado corrente
ex(s)	estado s torna-se inativo
exit	*ex(s)* para o estado corrente
fs(c)	mudança para falso da condição c
tr(c)	mudança para verdadeiro da condição c

Tabela A.1: Eventos especiais de *Xchart.*

- **Evento.** Um evento corresponde a qualquer ocorrência a que o sistema modelado deve reagir, ou qualquer saída produzida pelo sistema. A ocorrência de um evento é denotada por um arco orientado denominado **aresta** e provoca a mudança de estados através de **transições**. Eventos são instantâneos e visíveis dentro de uma determinada configuração de estados ativos. Arestas com a origem representada por um ponto hachurado não possuem estado origem e são ditas *default.* Arestas *default* indicam o subestado a ser ativado quando da ativação do estado a que pertence, caso não haja candidato de ativação pré-determinado. Um estado destino de uma aresta *default* é

chamado **estado inicial**. A Tabela A.1 contém alguns eventos pré-determinados para atividades comuns em um diagrama Xchart.

- **Ação.** Uma ação é produzida em reação à ocorrência de um evento e pode estar associada à ativação ou desativação de um estado ou de um gatilho. Uma ação pode representar um evento que terá repercursão em outros estados, uma expressão que altera o valor de variáveis, um método que deverá ser executado, ou qualquer combinação destes. Ações são atômicas.

- **Atividade.** Uma atividade corresponde à computação de tarefas pertinentes à semântica de um estado. Atividades são executadas durante o período em que se encontra ativo o estado ao qual estão associadas. Conceitualmente, atividades equivalem a procedimentos, rotinas, métodos, ou funções em linguagens de programação convencionais.

- **Gatilho.** Gatilho é associado a uma transição e é constituído por uma referência a um evento, ou classe de eventos, seguido, opcionalmente, por uma condição. Diz-se que um gatilho é disparado quando o evento ocorre e a condição associada é verdadeira. Caso contrário, a ocorrência do evento não tem efeito e o gatilho não é disparado. Uma outra condição de disparo ocorre quando o(s) estado(s) onde a transição à qual está associado o gatilho se origina, está(ão) ativo(s).

- **Regra.** Uma regra é composta por um gatilho e uma ação. Quando seu gatilho é disparado, a regra é executada, realizando a execução de sua ação. Regras rotulam transições ou ocorrem no interior de estados. Quando rotula uma transição, a execução da regra é necessária para que a transição seja percorrida. Quando associada a um estado, uma regra é executada somente se o estado estiver ativo.

- **Transição.** Uma transição estabelece uma relação entre estados. Se uma transição ocorrer, o(s) estado(s) origem se torna(m) inativo(s) e o(s) estado(s) destino, se torna(m) ativo(s). Cada transição é rotulada por uma regra e só é percorrida se o gatilho da regra for disparado, e em conseqüência, a ação da regra executada. Uma transição representada por uma aresta partindo do estado mais externo de um Xchart para um pequeno círculo preto denota o fim da execução do Xchart.

- **Rótulo.** O rótulo é constituído por um gatilho que descreve as condições de disparo de uma transição e uma ação a ser executada por ocasião do disparo. Sua sintaxe é $\alpha[\gamma]/\beta$, onde:

  - α é o evento necessário para que a transição ocorra, se o estado origem da aresta estiver ativo;
  - γ é uma condição, também chamada de *guarda*, descrita por uma expressão lógica que produz um valor *verdadeiro* ou *falso*. Quando verdadeira, a guarda ativa, e quando falsa, inibe o disparo da transição;

– β é a ação executada quando ocorre o disparo da transição.

Os três elementos de um rótulo são opcionais. Se o rótulo é vazio, a transição é dita automática e é disparada quando concluída a execução da atividade associada ao estado origem da transição.

- **Variável de controle.** A variável de controle é um repositório para um valor inteiro ou lógico, possivelmente utilizado em expressões que envolvam operações matemáticas básicas. Uma variável de controle pode ser local ou global. Se for local, existe somente no contexto da instância a que faz parte, caso contrário, existe durante todo o tempo de execução de um Xchart. A memória pode ser compartilhada entre objetos através de variáveis de controle globais. Na Figura A.2, a variável de controle x é comum às instâncias **Entrada** e **Segurança**. Após o nome do usuário ser fornecido, a senha é verificada e, se for inválida, o evento *inválida* é gerado. Na quarta tentativa, o objeto **Segurança** detecta esta situação pelo gatilho $[x > 3]$ e a transição para **Procedimento especial** ocorre, independentemente dessas instâncias estarem ou não em um mesmo processador.

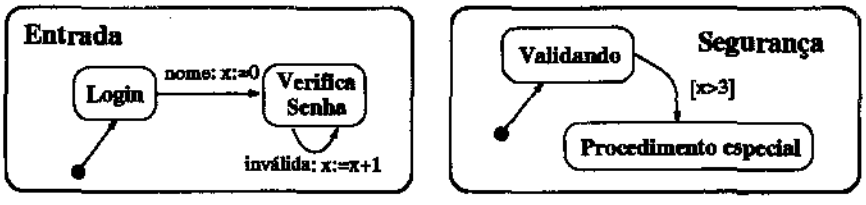

Figura A.2: Objetos Entrada e Segurança em *Xchart*.

A.2 Instância de Xchart

Uma instância de um Xchart é um Xchart em execução. Um Xchart é apenas um molde do qual é possível criar instâncias deste Xchart. Pode-se ter zero ou mais instâncias de um mesmo Xchart em execução durante um determinado instante de tempo. Esta relação entre um Xchart e sua instância é similar àquela existente entre *classes* e *objetos* em linguagens orientadas a objetos. Um Xchart é denominado pelo identificador do estado mais externo, ou raiz, da hierarquia de estados que o descreve. Cada instância possui um identificador único que a diferencia dentre todas as demais, inclusive aquelas instâncias de outros Xcharts.

Por simplicidade, o termo "instância de Xchart" será substituído por "instância", ou ainda, "Xchart", quando isto não comprometer a compreensão do texto. Graficamente, uma instância distingue-se de seu Xchart subjacente pelo contorno um pouco mais espesso do seu estado mais externo. A Figura A.3 contém um Xchart **E**, à esquerda, e uma instância deste Xchart identificada por **I**, à direita.

Figura A.3: Um Xchart e sua instância.

Toda instância existe durante um determinado período de tempo, que se inicia no instante em que a instância é criada e termina com a destruição da instância. Ao longo de sua existência, uma instância pode estar em uma das configurações possíveis: *ativa*, *inativa*, ou *inoperante*. Quando *ativa*, uma instância é sensível à ocorrência de eventos que são tratados normalmente. Quando *inoperante*, uma instância não percebe alterações no ambiente externo assim como não produz ações que podem atuar sobre este ambiente. Quando *inativa*, uma instância é insensível à ocorrência de eventos no ambiente e é "visível" para instâncias que estejam ativas. Estâncias *inativas* podem ser tornar *ativas*.

A Figura A.4 mostra um Xchart que descreve o ciclo de vida de uma instância. Após a confecção de uma descrição em Xchart, uma instância deste Xchart permanece em um estado virtual, suscetível a se tornar ativa. Uma vez ativa, a instância exerce algum tipo de função e a sua transição final pode ser percorrida, levando a destruição da instância ou a sua desativação, o que a conduz ao estado inoperante. Uma vez inoperante, uma instância jamais se retorna ativa novamente.

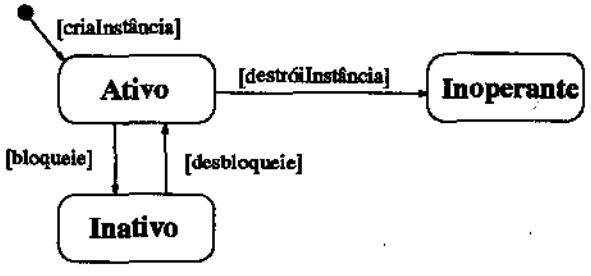

Figura A.4: Ciclo de vida de uma instância de um Xchart.

Principais Recursos

A partir do elementos relacionados anteriormente, *Xchart* permite a descrição de:

- **Hierarquia de estados.** Um estado pode ser decomposto em um ou mais estados denominados *subestados*. Um subestado também pode ser decomposto em subestados e assim sucessivamente. Uma hierarquia de estados é composta por estados e subestados encadeados e dispostos em níveis segundo a ordem de suas complexidades. Estados mais complexos são chamados de *ancestrais* de seus subestados *descendentes*. Esta

relação é transitiva, ou seja, o estado "mais externo" é ancestral de todos os demais estados de uma mesma hierarquia. O estado mais externo de uma hierarquia de estados é denominado *raiz*. Subestados não são compartilhados, isto é, para cada estado de uma hierarquia, exceto a raiz, há um único e bem definido ancestral.

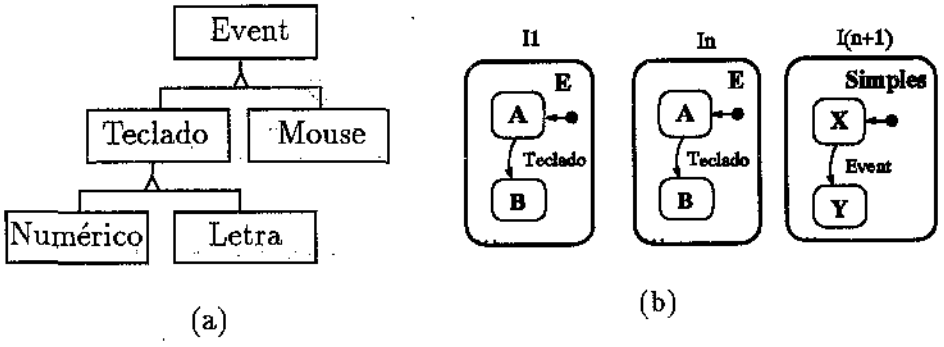

Figura A.5: Uma hierarquia de eventos e um exemplo em *Xchart*.

- **Hierarquia de eventos.** Eventos são organizados em classes que podem formar uma hierarquia. A Figura A.5 (a) exibe um exemplo de hierarquia de classes de eventos, onde a classe *Teclado* é especializada em *Numérico* e *Letra*. O rótulo de um gatilho pode conter o identificador de uma classe de eventos e não necessariamente o de uma instância daquela classe. Neste caso, qualquer instância desta classe ou de suas subclasses pode disparar o referido gatilho. Na Figura A.5 (b), por exemplo, a ocorrência de uma instância de **Event** (evento) pode provocar todas as transições exibidas, desde que **A** seja o estado ativo em todas as n instâncias de **E** e o subestado **X** do estado **Simples** esteja ativo.

- **Concorrência, sincronização e comunicação entre estados.** Os subestados de **Command** (Figura A.6) são estados concorrentes. Cada um destes subestados poderiam representar o comportamento de objetos distintos executados em diferentes processadores. A ocorrência do evento *cmd* provoca no estado **GetCmd** a mudança do subestado ativo **WaitC** para **Cmd**. A ocorrência de *arg* provoca um comportamento análogo no subestado **GetArg**. Assim que os estados **Arg** e **Cmd** tornarem-se ativos, duas outras transições são executadas. Uma delas desativa o estado **Arg** e torna ativo o estado **WaitA**. A outra desativa o estado **Cmd** e torna ativo o estado **WaitC**. Estas transições são sincronizadas através de gatilhos. Na linguagem *Xchart* não existe uma notação especial para representar a comunicação entre estados. Ela é implicitamente determinada através da maneira como o sistema é modelado. Nesse exemplo, há comunicação e sincronização implícitas entre os estados concorrentes **GetArg** e **GetCmd**. A Figura A.7 mostra um outro exemplo de comunicação entre estados, através da sinalização de eventos entre estados concorrentes e entre instâncias. Su-

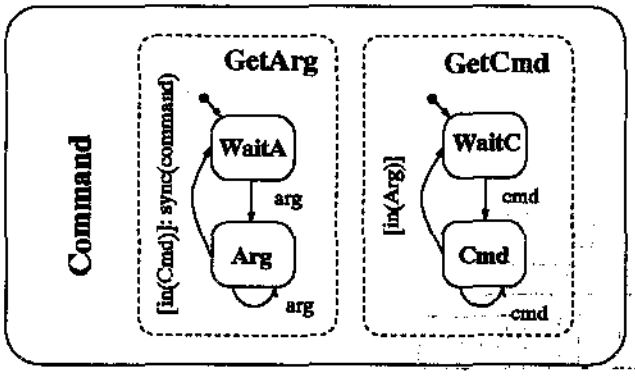

Figura A.6: Concorrência e comunicação entre estados em *Xchart*.

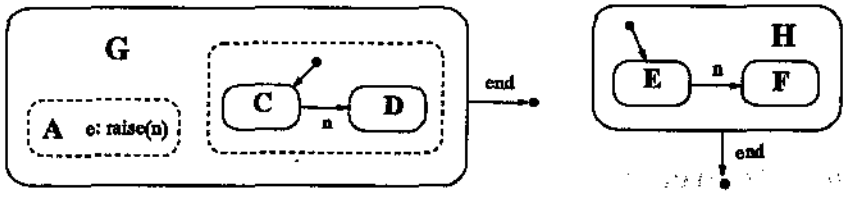

Figura A.7: Comunicação entre estados e instâncias em *Xchart*.

pondo que os estados **A**, **C** e **E** estejam ativos, a ocorrência do evento *e* faz com que a regra *e:raise(n)* seja executada. Em conseqüência, o evento *n* é gerado, provocando a transição do estado **C** para o **D**, na instância do Xchart **G**, e a transição do estado **E** para o estado **F**, na instância de **H**.

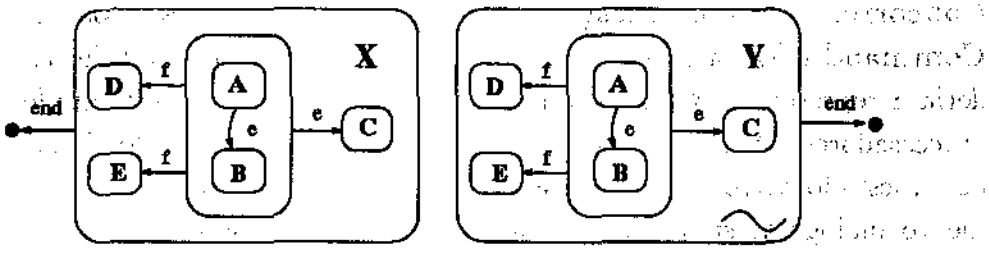

Figura A.8: Transições conflitantes e não determinismo em *Xchart*.

- **Determinismo e não determinismo de ações.** Conforme a Figura A.8, especificações podem ter interpretações distintas de acordo com seus atributos. Se o estado **A** do Xchart **X**, por exemplo, estiver ativo quando ocorrer o evento *e*, apenas a transição mais interna será disparada, pois tem prioridade sobre a outra transição rotulada com

o evento *e*. Contudo, se o evento *f* ocorrer, as transições com destino em **D** e **E** são conflitantes, e nenhuma delas é executada. No diagrama da direita, o Xchart **Y** é semelhante ao **X**, exceto quanto ao comportamento não determinístico, indicado no canto inferior direito pelo símbolo $\sim$. Neste caso, se **A** estiver ativo e o evento *e* ocorrer, tanto **B** quanto **C** são destinos possíveis. A ocorrência de *f* não provoca o mesmo efeito que no Xchart **X** e uma das transições é arbitrariamente executada.

- **Atomicidade de ações.** As ações disparadas pela ocorrência de eventos são executadas atomicamente. A Figura A.9 exibe duas instâncias de um comportamento descrito pelo estado **X**. Se o estado **A** estiver ativo nas instâncias **li** e **lk** e ocorrer o evento *e*, possivelmente sinalizado por outra instância derivada de outra especificação de comportamento, as transições exibidas serão percorridas. Como reação, as expressões que compartilham as variáveis x e y serão executadas. Logo, há um controle de concorrência implícito nas reações e os novos valores serão $x' = x - 4$ e $y' = y + 4$. Ao final das reações, ou os valores de x e y serão os citados anteriormente, ou $x' = x - 2$ e $y' = y + 2$ (uma das ações atômicas é abortada) ou ainda, não serão alterados (ambas abortadas). Nestes casos, a configuração resultante será a mesma anterior à ocorrência do evento que provocou a reação abortada.

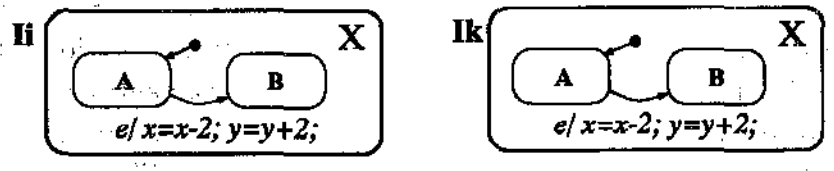

Figura A.9: Ação atômica em *Xchart*.

- **Operações de *Undo* e *Redo*.** O evento *undo* restaura o valor de expressões, condições e a última configuração de estados vigentes antes da mais recente reação do sistema. É permitida a especificação de uma ação como reação a este evento. O evento *redo* restabelece o contexto imediatamente anterior ao último *undo*. Como qualquer outro evento, *undo* e *redo* podem ser utilizados em regras. Um contexto pode ser alterado pela execução de uma regra.

- **Restrições temporais.** Especificações temporais podem estar associadas a estados e a eventos. Na Figura A.10 (a), após transcorridos 2 segundos de permanência no estado **A**, a regra *after(2s)* é executada, provocando a transição deste estado para o estado **B** e vice-versa, o que é denotado pela aresta bidirecional. Em (b), após a ativação do estado **C**, o evento *e* não provocará uma transição para **D** enquanto não transcorrerem 3 (três) segundos de permanência em **C**. Após os 3 segundos, a ocorrência do evento *e* habilita a transição para **D**. Eventos ocorridos nestes 3 segundos são descartados. Além

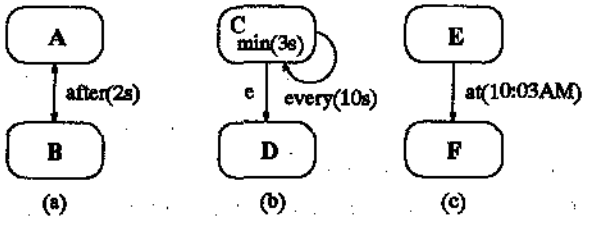

Figura A.10: Especificação de restrições temporais em *Xchart*.

disto, enquanto o estado **C** estiver ativo, a cada 10 (dez) segundos ele será desativado e ativado em seguida. Em (c), o evento *at(10:03AM)* provocará a mudança do estado de **E** para **F**, caso **E** esteja ativo no instante explicitado.

- **História**. História é um mecanismo especial de ativação de subestados. Se a ativação de um estado se der por história, o estado mais recentemente visitado é escolhido. Há dois tipos de história: simples e estrela. No tipo estrela (representada pelo símbolo H*), a ativação de subestados deverá ser independente do nível hierárquico onde o símbolo aparece. No tipo simples (representada por H), considera-se apenas subestados diretamente descendentes do estado que contém a indicação de história. Na Figura A.11 (a), a ativação do estado **K** é realizado por história simples. Se ocorrer o evento *e*, o subestado de **K** a ser ativado será o mais recentemente visitado entre **K1** e **K2**, ou seja, **B** ou **D** (subestados iniciais de **K1** e **K2**, respectivamente), pois o tipo de história simples não se aplica aos subestados de **K1** e **K2**. Se for a primeira vez que o estado **K** é visitado, o subestado *default* **D** de **K2** é ativado. A Figura A.11 (b) mostra um exemplo de utilização de história do tipo estrela. Com a ocorrência de *e*, o subestado de **K** a ser ativado é o estado mais recentemente visitado, independentemente do nível em que se encontrar. Assim, o estado ativado poderá ser **A**, **B**, **C** ou **D**, conforme o passado mais recente de ativação.

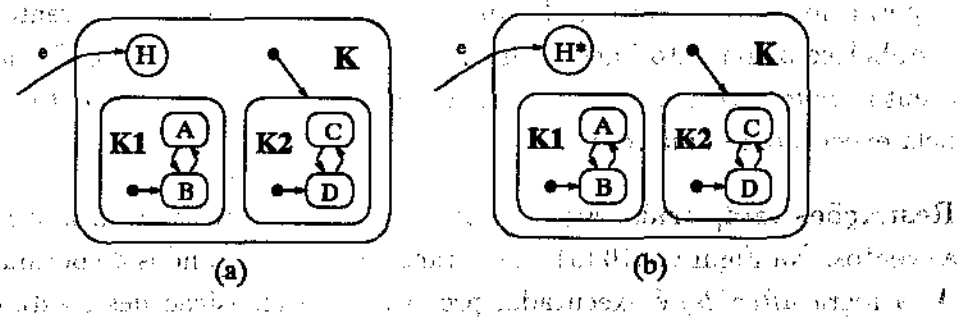

Figura A.11: Ativação de estados através de história em *Xchart*.

Referências Bibliográficas

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer USENIX Conference*, p. 93–112, Atlanta, GA, USA, 1986.

[2] E. L. Alves. Port System: Sistema de Comunicação em Grupo para o Ambiente Xchart. Dissertação de Mestrado, Universidade Estadual de Campinas, fevereiro 1996.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, setembro 1989.

[4] J. K. Bennett. The Design and Implementatio of Distributed Smalltalk. *ACM SIGPLAN notices*, 22(12):318–330, dezembro 1987.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Inc., 1987.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, capítulo 3, p. 79. Addison-Wesley Publishing Company, Inc., 1987.

[7] A. D. Birrel and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, fevereiro 1984.

[8] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. *ACM SIGPLAN notices*, 21(11):76–86, novembro 1986.

[9] A. P. Black. Understanding Transactions in the Operating System Context. *Operating Systems Review*, 25(1):73–76, janeiro 1991.

[10] A. Burns, A. M. Lister, and A. J. Welling. A Review of Ada Tasking. *Lecture Notes in Computer Science*, 282, 1987.

[11] R. S. Chin and S. T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, março 1991.

[12] S. Churchill. Understanding Fresco: A hands-on Tutorial to the Fresco User Interface System. Manual, The X Consortium, http://www/faslab.com/fresco/HomePage/html, 1994.

[13] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, dezembro 1989.

[14] X Consortium. Fresco Nov94 Reference Manual. Manual, The X Consortium, http://www/faslab.com/fresco/HomePage/html, 1994.

[15] H. Custer. *Windows NT*. McGraw-Hill Ltda, 1993.

[16] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula67 Common Base Language. Technical Report S-28, Norwegian Computing Centre, Oslo, outubro 1970.

[17] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, 3(1):11–46, 1990.

[18] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and Ramachandran U. The Clouds Distributed Operating System. *Computer*, 24(11):34–44, novembro 1991.

[19] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. The Treatment of Persistent Objects in Arjuna. *The Computer Journal*, 32(4):323–332, agosto 1989.

[20] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, capítulo 1, p. 1–23. Morgan Kaufmann Publishers, Inc., 1990.

[21] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*, capítulo 1, p. 7. Morgan Kaufmann Publishers, Inc, 1991.

[22] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*, capítulo 6, p. 81–84. Morgan Kaufmann Publishers, Inc, 1991.

[23] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*, capítulo 4, p. 57. Morgan Kaufmann Publishers, Inc, 1991.

[24] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, novembro 1976.

[25] N. H. Gehani and W. D. Roome. Concurrent C. *Software Practice and Experience*, 16(9):821–844, setembro 1986.

[26] A. Goldberg and D. Robson. *Smaltalk-80: The Language and its Implementation.* Reading, MA: Addison Wesley, 1983.

[27] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, capítulo 1, p. 6. Morgan Kaufmann Publishers, Inc., 1993.

[28] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, abril 1990.

[29] M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the 17th IEEE Fault-Tolerant Computing Symposium*, p. 89–74, Pittisburg, Pennsylvania, julho 1987. IEEE.

[30] R. J. K. Jacob. Using Formal Specifications in the Design of a Human-Computer Interface. *Communications of the ACM*, 26(4):259–264, abril 1983.

[31] S. Khoshafian and R. Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*, capítulo 9, p. 6. John Wiley & Sons, Inc., 1993.

[32] S. Khoshafian and R. Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*, capítulo 4, p. 168. John Wiley & Sons, Inc., 1993.

[33] S. N. Khoshafian and G. P. Copeland. ObjectIdentity. In *Proceedings of OOPSLA*, p. 406–416, setembro 1986.

[34] R. LeBlanc and C. T. Wilkes. Systems Programming with Objects and Actions. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, p. 132–139, MD, USA, maio 1985.

[35] Architecture Projects Management Limited. An Engineer's Introduction to the Architecture. Technical Report TR.03.02, ANSA, Cambridge (United Kingdom), março 1989.

[36] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8–22, fevereiro 1989.

[37] D. B. Lomet. Process Structuring Synchronization, and Recovery Using Atomic Actions. *ACM SIGPLAN notices*, 12(3):128–137, março 1977.

[38] F. N. Lucena. Construção de Interfaces Homem-Computador. Dissertação de Mestrado, Universidade Estadual de Campinas, março 1993.

[39] F. N. Lucena and H. K. E. Liesenberg. A Statechart Engine to Support Implementations of Complex Behaviour. In *Procedings of XII Semish - XIV SBC*, p. 177–191, Caxambu, MG, Brasil, agosto 1994.

[40] F. N. Lucena and H. K. E. Liesenberg. Reflections on Using Statecharts to Capture User Interface Behavior. In *Proceedings of XIV International Conference of the Chilean Computer Sience Society*, Chile, novembro 1994.

[41] F. N. Lucena, H. K. E. Liesenberg, and L. E. Buzato. Xchart: Um Sistema de Gerenciamento de Interfaces Homem-Computador. In *Workshop de Sistemas Hipermídia (WOSH'96)*, Fortaleza, CE, maio 1996.

[42] F. N. Lucena, H. K. E. Liesenberg, and M. Harada. Operational Semantics of Extended Statecharts (XCHART). In *Third Workshop on Logic, Language, Information and Computation (WOLLIC'96)*, Salvador, BA, maio 1996.

[43] D. L. McCue. *Selective Transparency in Distributed Transaction Processing.* PhD thesis, University of Newcastle upon Tyne, abril 1992.

[44] A. Morse and G. Reynolds. Overcome Current Growth Limits in User Interface Development. *Communications of the ACM*, 36(4):73–81, abril 1993.

[45] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Computing.* PhD thesis, MIT, Departament of Computer Science, 1981.

[46] Brad A. Myers, editor. *Languages for Developing User Interfaces.* J&B, 1992.

[47] K. Nygaard and O.-J. Dahl. The development of the Simula language. *ACM SIGPLAN notices*, 13(8), agosto 1978.

[48] D. R. Olsen. *User Interface Management Systems: Models and Algorithms*, capítulo 2, p. 15–16. Morgan Kaufmann, 1992.

[49] The Common Object Request Broker: Architecture and Specification, 1991. OMG Document Number 91.12.1.

[50] Object Management Architecture Guide, 1992. OMG Document Number 92.11.1.

[51] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*, capítulo 7, p. 124–134. John Wiley & Sons, Inc., 1996.

[52] J. K. Ousterholt. *Tcl and Tk Toolkit.* Addison-Wesley Publishing Company, Inc., primeira edição, 1993.

[53] G. D. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *Proceedings of the USENIX 2nd C++ Conference*, p. 37–50, San Francisco, abril 1990.

[54] G. D. Parrington and S. K. Shrivastava. Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems. In *Proceedings of the Second European Conference on Object-Oriented Programming (ECOOP'88)*, Norway, agosto 1988.

[55] G. Perlman. Software Tools for User Interface Development. In M. Helander, editor, *Handbook of Humam-Computer Interaction*, p. 819–833. North-Holland, 1988.

[56] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software Practice and Experience*, 21(1):91–118, janeiro 1991.

[57] J. Richter. *Advanced Windows NT*. Microsoft Press, 1994.

[58] O. Severino, Jr. SMART: Um Editor gráfico de Xchart. Dissertação de Mestrado, Universidade Estadual de Campinas, dezembro 1996.

[59] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the Arjuna Programming System. *IEEE Software*, 8(1):66–73, janeiro 1991.

[60] S. K. Shrivastava and S. Weather. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, p. 203–210, Paris, France, maio 1990.

[61] D. Skeen. Nonblocking Commit Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, p. 133–142, Ann Arbor, Michigan, New York, 1981.

[62] A. Z. Spector, R. F. Pausch, and G. Bruel. Camelot: A Flexible Distributed Transaction Processing System. In *Proceedings of IEEE Compcom*, San Francisco, março 1988.

[63] A. Z. Spector, D. S. Thonson, R. F. Paush, J. L. Eppinger, D. Duchamp, R. P. Dreaves, D. S. Daniels, and J. J Clock. Camelot: A Distributed Transaction Facility for Mach and the Internet: An Interim report. Technical Report CMU-CS-87-129, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1987.

[64] A. S. Tanenbaum. *Modern Operating Systems*, capítulo 10, p. 403. Prentice-Hall International, Inc., 1992.

[65] A. S. Tanenbaum. *Modern Operating Systems*, capítulo 9, p. 365. Prentice-Hall International, Inc., 1992.

[66] A. S. Tanenbaum. *Modern Operating Systems*, capítulo 10, p. 406. Prentice-Hall International, Inc., 1992.